# Assignment 4, 2048 Game Design

Tingyu Shi - shit19

2021 - 04 - 12

This MIS contains the specific information of modules for the implementation of game 2048. To make this game more general, the user can decide the size of the board at the beginning of the game. The following are the rules of the game:
The goal of this game is to combine numbered tiles to gain higher numbered tiles. Two numbered tiles can only be combined or merged when two numbers are the same. The result of combining two same numbered tiles is the sum of these two numbers. During the game, there are only four movements, which are left, right, up and down. When the player chooses a direction to move, all the numbered tiles will move at the same time. If the game board changes after a move, a random number will pop up in the game board.

An on-line version of 2048 can be found at:

https://play2048.co/

## 1 Likely changes

1. The size of the board may change
2. The printed message on screen may change
3. The data structure to store the game board may change
4. The random numbers showed in the game board after a move may change

## 2 Overview of the design

To design this game, MVC(Model View Control) design pattern is used. The following are specific information of each part:

- Model
  The model of 2048 game is obviously the game board. This corresponds to the `BoardT` class in the implementation. This class is responsible for storing data and states of a game board. The data is the numbered tiles of in the game board. The states include the score of the current game and the size of the game board. The data structure to store the game board is a 2D array. Also, in this class, there are methods to perform different move operations, determine if the game has failed.etc. For detailed information, please check MIS of `BoardT`.

- View
  View unit corresponds to `DisplayGame` code file. The role of this module is to display the game board and some other messages to players. Other messages include game instructions, welcoming message, ending messages.etc.

- Control
  The Control unit corresponds to `GameControl` of the code file. This is a link between model and view. Also, it handles user's inputs to modify the game board and select view correspondingly.

- Another java class is `MoveDirection`, this class is implemented as enum class, it contains all the movement options of game 2048.

## 3 MIS of each Module

## MoveDirection Module

### Module

MoveDirection

### Uses

None

### Syntax

#### Exported Constants

None

#### Exported Types

MoveDirection = {
LEFT, *#Move all the numbered tiles left*
RIGHT, *#Move all the numbered tiles right*
UP, *#Move all the numbered tiles up*
DOWN, *#Move all the numbered tiles down*
}

#### Exported Access Programs

None

### Semantics

#### State Variables

None

#### State Invariant

None

**Assumptions**

None

## Game Board Module

### Template Module

BoardT

### Uses

MoveDirection

### Syntax

#### Exported Constants

None

#### Exported Types

BoardT = ?

#### Exported Access Programs

| Routine name | In | Out | Exceptions |
|---|---|---|---|
| new BoardT | $\mathbb{N}$ | BoardT | IllegalArgumentException |
| move | MoveDirection | $\mathbb{B}$ | |
| put_one_random_number | $\mathbb{N}$ | | |
| is_end | | $\mathbb{B}$ | |
| set_board | $\mathbb{N}$, $\mathbb{N}$, $\mathbb{N}$ | | IllegalArgumentException |
| has_empty_cell | | $\mathbb{B}$ | |
| get_score | | $\mathbb{N}$ | |
| get_board | | seq of (seq of $\mathbb{N}$) | |
| get_size | | $\mathbb{N}$ | |
| equals | seq of (seq of $\mathbb{N}$) | $\mathbb{B}$ | IllegalArgumentException |

### Semantics

#### State Variables

$size : \mathbb{N}$
$score : \mathbb{N}$

*board*: seq[*size*] of (seq[*size*] of $\mathbb{N}$)
*# This is a 2D array to store all the data of the game board, the game board is a square board*

**State Invariant**

None

**Access Routine Semantics**

new BoardT(*board_size*):

- transition:
  $size := board\_size$
  $score := 0$

$$board := \langle \begin{array}{c} \langle 0_{00}, 0_{01}\ldots\ldots, 0_{0(size-2)}, 0_{0(size-1)}\rangle \\ \langle 0_{10}, 0_{11}\ldots\ldots, 0_{1(size-2)}, 0_{1(size-1)}\rangle \\ \langle 0_{20}, 0_{21}\ldots\ldots, 0_{2(size-2)}, 0_{2(size-1)}\rangle \\ \ldots \\ \ldots \\ \langle 0_{(size-3)0}, 0_{(size-3)1}\ldots\ldots, 0_{(size-3)(size-2)}, 0_{(size-3)(size-1)}\rangle \\ \langle 0_{(size-2)0}, 0_{(size-2)1}\ldots\ldots, 0_{(size-2)(size-2)}, 0_{(size-2)(size-1)}\rangle \\ \langle 0_{(size-1)0}, 0_{(size-1)1}\ldots\ldots, 0_{(size-1)(size-2)}, 0_{(size-1)(size-1)}\rangle \end{array} \rangle$$

  *#board is a 2D array which has size rows and for each row, it has size 0s. The subscript in the above graph is the index number of each 0.*

- output: $out :=$ self
- exception: exc := (($board\_size < 0 \vee board\_size = 0$) $\Rightarrow$ IllegalArgumentException)

move(*dir*):

- transition: transition will be made during the process of calculating the output
- output: $(dir = MoveDirection.LEFT \Rightarrow (move\_left(board))|$
  $dir = MoveDirection.RIGHT \Rightarrow (move\_right(board))|$
  $dir = MoveDirection.UP \Rightarrow (move\_up(board))|$
  $dir = MoveDirection.DOWN \Rightarrow (move\_down(board)))$
  *#At the same time of calculating the output, the state of of the game board is changed*
- exception: None

put_one_random_number($value$):

- transition: $board[i][j] := value$
  *#Assum we have the function random() to generate a number between 0 and 1.*
  Where
  $$\langle i, j\rangle := emptyPos[\lfloor random() * |emptyPos| - 1\rfloor]$$
  $$emptyPos := \{\forall i, j : \mathbb{N} | i \in [0..size - 1] \wedge j \in [0..size - 1] \wedge board[i][j] = 0 : \langle i, j\rangle\}$$

  *#emptyPos here is a set of pairs (i, j) such that board[i][j] = 0. Then we just random choose a pair and assign value to board[i][j].*

- output: none

- exception: none

is_end():

- transition: none

- output: $out := \neg(move(MoveDirection.LEFT) \vee move(MoveDirection.RIGHT) \vee move(MoveDirection.UP) \vee move(MoveDirection.DOWN))$
  *#If the game board can move left, right, up, or down, then game is not end.*

- exception: none

set_board($x$, $y$, $value$): *#This method is just used for testing.*

- transition: $board[x][y] := value$

- output: none

- exception: exc := $((x < 0 \vee y < 0 \vee x \geq size \vee y \geq size) \Rightarrow$ IllegalArgumentException)

has_empty_cell():

- transition: none

- output:

  $$out := \exists(i, j : \mathbb{N} | i \in [0..size - 1] \wedge j \in [0..size - 1] : board[i][j] = 0)$$

  *#Used to tell if board has 0*

- exception: none

get_score():

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

get_board():

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

get_size():

- transition: none
- output: $out := size$
- exception: none

equals($arr$): *#This method is just used for testing.*

- transition: none
- output:$out := \neg\exists(i, j : \mathbb{N} | i \in [0..size - 1] \wedge j \in [0..size - 1] : board[i][j] \neq arr[i][j])$
- exception: $exc := (\neg(|arr| = size) \vee \neg(|arr[0]| = size)) \Rightarrow IllegalArgumentException$

## Local Functions

*#For the following local functions, operational descriptions will be used since it is hard to express in mathematical expression.*
move_left : seq[$size$] of (seq[$size$] of $\mathbb{N}$) $\rightarrow \mathbb{B}$
The following are steps:

1. For each row, merge and sum up two adjacent positive numbers if they are the same. A number will always try to merge the number at the left side of it if the above condition is met. Also, if a number is produced by the above merging, it will not merge with other numbers again.
2. After merging, for each row, moving all the positive numbers toward left. The order between these positive numbers must not change.
3. After the above two steps, compare the current 2D array with the input 2D array, if they are the same, return true, otherwise return false.

move_right : seq[$size$] of (seq[$size$] of $\mathbb{N}$) $\rightarrow \mathbb{B}$
It is the same as move_left except in step2, moving all the positive numbers toward right.

move_up : seq[$size$] of (seq[$size$] of $\mathbb{N}$) $\rightarrow \mathbb{B}$
The following are steps:

1. For each column, merge and sum up two adjacent positive numbers if they are the same. A number will always try to merge the number at the top of it if the above condition is met. Also, if a number is produced by the above merging, it will not merge with other numbers again.
2. After merging, for each column, moving all the positive numbers up. The order between these positive numbers must not change.
3. After the above two steps, compare the current 2D array with the input 2D array, if they are the same, return true, otherwise return false.

move_down : seq[$size$] of (seq[$size$] of $\mathbb{N}$) $\rightarrow \mathbb{B}$
It is the same as move_up except in step 2, moving all the positive numbers down.

# DisplayGame Module

## DisplayGame Module

## Uses

None

## Syntax

### Exported Types

None

### Exported Constants

None

### Exported Access Programs

| **Routine name** | **In** | **Out** | **Exceptions** |
|---|---|---|---|
| print_board | BoardT | | |
| print_game_start | | | |
| print_game_instruction | | | |
| print_game_ending | | | |
| print_invlid_command | | | |
| print_game_fail | | | |
| print_wrong_size_input | | | |

## Semantics

## Environment Variables

window: A portion of computer screen to display the game and messages

### State Variables

None

### State Invariant

None

**Access Routine Semantics**

print_board(*board*):

- transition: window := Draws the game board onto the screen. Each elements in the board is accessed by using `get_board` method defined in `BoardT` class. Assume the size of the board is s, then the first row is from *board*[0][0] to *board*[0][s - 1], the second row will be from *board*[1][0] to *board*[1][s - 1]. Follow this pattern, the last row will be from *board*[s - 1][0] to *board*[s - 1][s - 1].

print_game_start():

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

print_game_instructions():

- transition: window := After game starts, this will print game instructions onto the screen. Instruction include how to control different movements and how to exit the game. Also, the window shows a message to ask the player which movement he/she wants to play and whether he/she wants to exit the game.
- output: none
- exception: none

print_game_ending():

- transition: window := If the player chooses to exit the game, just print an ending message.
- output: none
- exception: none

print_invalid_command():

- transition: window := Remind the user that the input command is invalid and shows a message to tell the user to enter a new command.
- output: none

- exception: none

print_game_fail():

- transition: window := Print a message when the game fails, which means no more moves can be made.
- output: none
- exception: none

print_wrong_size_input():

- transition: window := Print a message if the user enters an invalid board size. For example, 0 or negative numbers. Also, window shows a message to tell the user to enter a new number again.
- output: none
- exception: none

## GameControl Module

*#This part of specification is operational*

## GameControl Module

## Uses

BoardT, DisplayGame

## Syntax

### Exported Types

None

### Exported Constants

None

### Exported Access Programs

| **Routine name** | **In** | **Out** | **Exceptions** |
|---|---|---|---|
| initialize_game | | BoardT | |
| run_game | BoardT | | |
| main | | | |

## Semantics

## Environment Variables

keyboard: Scanner(System.in) *// reading inputs from players via keyboards*

### State Variables

None

### State Invariant

None

**Access Routine Semantics**

initialize_game():

- transition: operational method for initializing the game.
  1. Print the welcome message.
  2. Asking the player to enter the board size he/she wants to use and then receive the board size from the keyboard.
  3. Create an object of BoardT with the size received from step 2.
  4. Put 2 numbers(2 and 2 or 2 and 4) into two randomly chosen positions on the board.
  5. Display the board onto the screen.
  6. Return the board created from step 3.
- output: out := The BoardT object created from step 3 above.
- exception: None

run_game(board):

- transition: operational method for running the game.
  After printing the initialized game board onto the screen, the game will display the instruction and ask the user to make a move or exit the game. If the player chooses to make a move, then the game will display the game board after that move. If the player chooses to exit the game, the game will end by displaying the ending message. The game will continue as above until no more moves can be made.
- output: None
- exception: None

main(): *#This is the main function. The driver of the whole game*

- transition: operational method for running the game.
  1. BoardT model = initialize_game(); *//Initialize the game*
  2. run_game(model); *// run game*
- output: None
- exception: None

## 4 Critique of Design

- Consistency
  The consistency of this design is met. This includes naming conventions and ordering of parameters and exception handling.

- Essentiality
  There are two methods from `BoardT` class are not essential, which are `set_board` and `equals`. These two methods are not essential for implementing the game, however, they are useful for testing the model, therefore, they can be deleted after testing.

- Generality
  For a standard 2048 game, the game board is $4 \times 4$. However, this design has improved the generality so that the player can set up a $n \times n$ board. Also, because of `put_one_random_number` method from `BoardT` class, we can put different random values after each move, not only 2 or 4. However, the generality can still be improved, the next improvement can be different shapes of the board, not only square board.

- Minimality
  The Minimality should be improved for `move` method of class `BoardT`. The reason is that `move` provides two independent services. First, it changes the state of the game board according to the MoveDirection given. At the same time, it returns a boolean to tell if the board has changed before and after the move operation. The correct way of designing should be make a copy of the current board and define another method to compare the board after the move and copy of the board before the move.

- Cohesion
  The cohesion of this design is achieved. The reason is that the design follows the structure of MVC. Each module except `MoveDirection` just implements one of the units of MVC. As a consequence, the components within one module are highly related.

- Information Hiding
  All the state variables are private variables. However, since we need `set_board` method for testing, the programmer can still change the state of game board from outside. Therefore, we need to come up with a better way for testing when there are random numbers. Also, it is a good practice to just remove this method after testing.

The following are some other points about this design:

1. The view module are designed to be an abstract object just like a service module, the reason is that different game can use the same view module, therefore, there is not need to create an ADT.

2. About the testing, all the exceptions are tested and different move operations are tested. However, for each move, it is better to include more testing cases.