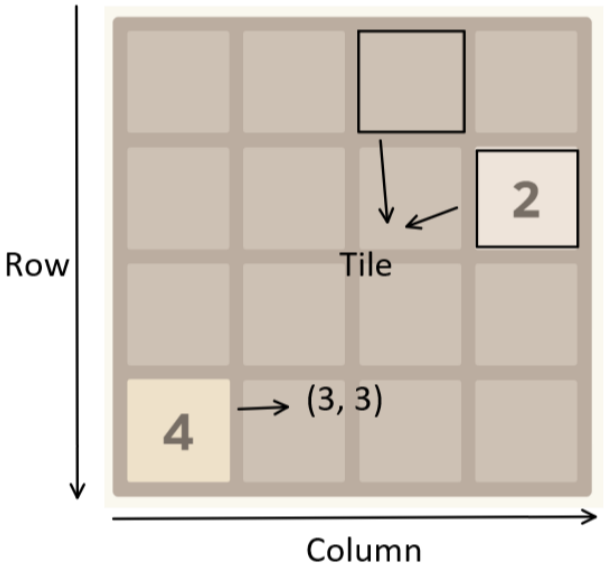
# COMPSCI 2ME3, Assignment 4, Design Specification

Jie Zhang - zhanj265

September 10, 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 will be given a 4x4 matrix with two 2's or two 4's or one 2 and one 4. The user can move all tiles in this matrix to up, down, left or right in one round, and tiles with the same number will merge into one when they touch. After each round, if there is still empty space in matrix, there is 90 percent possibility that a random empty space will be filled with 2, and 10 percent possibility that a random empty space will be filled with 4. Throughout this specification document, each tile will be referred as to a space in matrix and a cell(x, y) refers to the position of a tile, where x represents the index of row and y represents the index of column. In addition, row number increases when moving from top to bottom and column number increases when moving from left to right. The game can be launched and play by typing `make demo` in terminal.

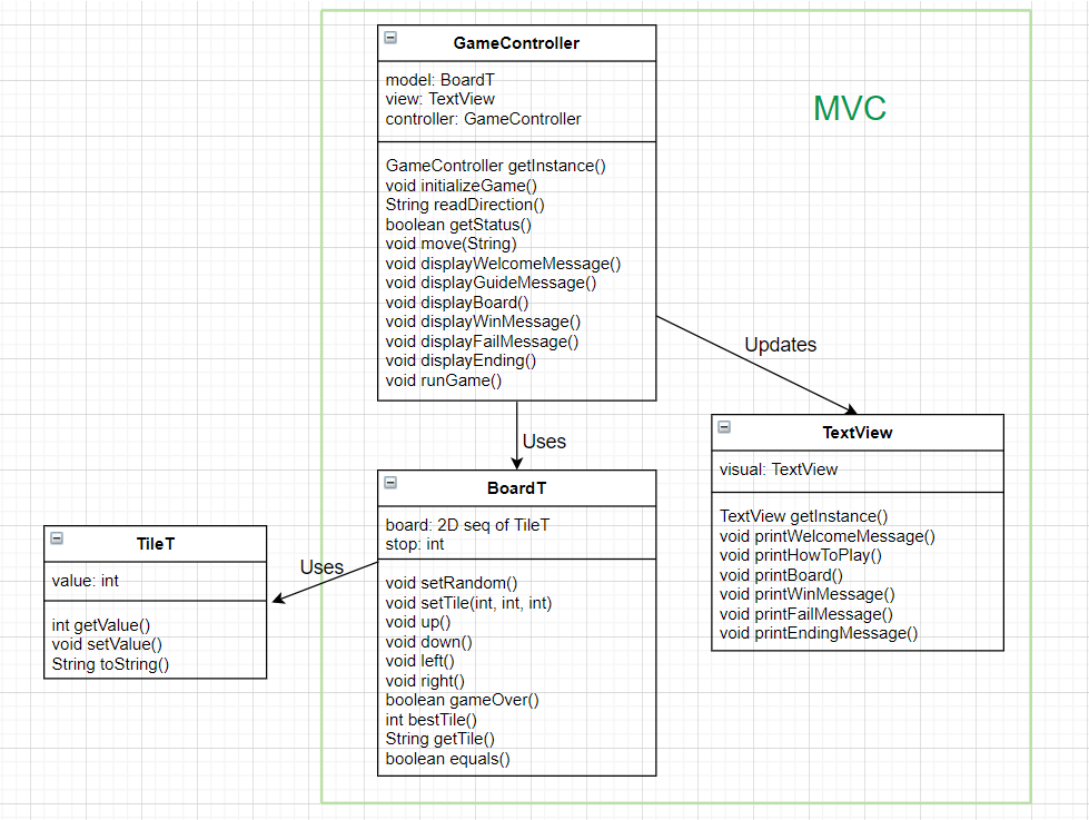


The above board visualization is from https://play2048.co/

# 1 Overview of the design

This design applies Module View Specification (MVC) design pattern and Singleton design pattern. The MVC components are *GameController* (controller module), *BoardT* (model module), and *TextView* (view module). Singleton pattern is specified and implemented for *GameController* and *TextView*

An 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 *BoardT* stores the state of the game board. A view module *TextView* can display the state of the game board using a text-based graphics. The controller *GameController* is responsible for handling input actions and updating the view received by user.

For *GameController* and *TextView*, use the getInstance() method to obtain the abstract object.

## Likely Changes my design considers:

- Change in the form of view. This specification uses TileT class to store value at each cell in this board. If we want to change the view from text-based to GUI, TileT object can store color based on its value. This class makes future GUI development more convenient.
- Change in the size of the game board.

# Tile ADT Module

## Template Module

TileT

## Uses

None

## Syntax

### Exported Types

TileT = ?

### Exported Constant

None

### Exported Access Programs

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

## Semantics

### State Variables

value: $\mathbb{N}$

### State Invariant

None

### Assumptions

- The value of TileT is always 0 or the power of 2 (except 1).

**Access Routine Semantics**

TileT():

- transition: value := 0
- output: *out* := *self*
- exception: None

TileT(num):

- transition: value := num
- output: *out* := *self*
- exception: None

getValue():

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

setValue(num):

- transition: value := num
- output: None
- exception: None

toString():

- transition: none
- output: *out* := value.toString()
- exception: None

# 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 | |
| BoardT | seq of (seq of TileT) | BoardT | |
| setRandom | | | |
| setTile | $\mathbb{N}$, $\mathbb{N}$, $\mathbb{N}$ | | IndexOutOfBoundsException |
| up | | | |
| down | | | |
| left | | | |
| right | | | |
| gameOver | | $\mathbb{B}$ | |
| bestTile | | $\mathbb{N}$ | |
| getTile | $\mathbb{N}$, $\mathbb{N}$ | String | IndexOutOfBoundsException |
| equals | BoardT | $\mathbb{B}$ | |

## Semantics

### State Variables

board: sequence [size, size] of TileT
stop: $\mathbb{N}$ *// help to store which tile cannot merge when moving them.*

### State Invariant

None

### 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 beteern 0 and 1 (not included 1).

### Design decision

The coordinates of the board is stored in a 2D sequence. In later specification, board[x][y] and board(x,y) both mean the TileT at row x and column y of the board. Row number increases when moving from top to bottom and column number increases when moving from left to right, and they both start from 0. Since this game is text-based, all empty spaces in board are represented by tiles with value of 0.

### Access Routine Semantics

BoardT():

- transition:
  $$\text{board} := \langle \begin{matrix} \langle 0,0,0,0 \rangle \\ \langle 0,0,0,0 \rangle \\ \langle 0,0,0,0 \rangle \\ \langle 0,0,0,0 \rangle \end{matrix} \rangle$$
  stop := 0
- output: *out* := *self*
- exception: None

BoardT($b$):

- transition: board, stop := $b$, 0
- output: *out* := *self*
- exception: None

setRandom():

- transition: board := board(randomFrom($x, y : \mathbb{N} | 0 <= x < size \wedge 0 <= y < size \wedge board[x][y].getValue() = 0 : (x, y)$)) := randomTile()

  *// Set a random empty space to be 2 (90 percent possibility) or 4 (10 percent possibility).*
- output: None
- exception: None

setTile(x, y, t)

- transition: board := board[x][y].setValue(t)
- output: None
- exception: ($\neg$ validateCell($x, y$) $\Rightarrow$ IndexOutOfBoundsException)

up()

- transition: board := Move all tiles with positive values in this game board upward until no tiles can move. From top to bottom, when 2 vertically adjacent tiles have the same value, they can merge and the upper one's value become their sum and the lower one's value become 0, which means the cell is empty. A merged tile cannot merge with other tiles in this round again. After this, all tiles with positive values move upward again. In this process, the state variable stop can be used to store which tile cannot merge anymore.
- output: None

down()

- transition: board := Move all tiles with positive values in this game board downward until no tiles can move. From bottom to top, when 2 vertically adjacent tiles have the same value, they can merge and the lower one's value become their sum and the upper one's value become 0, which means the cell is empty. A merged tile cannot merge with other tiles in this round again. After this, all tiles with positive values move downward again. In this process, the state variable stop can be used to store which tile cannot merge anymore.

- output: None

left()

- transition: board := Move all tiles with positive values in this game board to left until no tiles can move. From left to right, when 2 horizontally adjacent tiles have the same value, they can merge and the left one's value become their sum and the right one's value become 0, which means the cell is empty. A merged tile cannot merge with other tiles in this round again. After this, all tiles with positive values move to left again. In this process, the state variable stop can be used to store which tile cannot merge anymore.
- output: None

up()

- transition: board := Move all tiles with positive values in this game board to right until no tiles can move. From right to left, when 2 horizontally adjacent tiles have the same value, they can merge and the right one's value become their sum and the left one's value become 0, which means the cell is empty. A merged tile cannot merge with other tiles in this round again. After this, all tiles with positive values move to right again. In this process, the state variable stop can be used to store which tile cannot merge anymore.
- output: None

gameOver()

- output: $out := \forall(\text{x, y: } \mathbb{N} | 0 \leq x < size \wedge 0 \leq y < size \Rightarrow board[x][y] \neq 0) \wedge$
  $\forall(\text{x, y: } \mathbb{N} | 0 \leq x < size - 1 \wedge 0 \leq y < size \Rightarrow board[x][y] \neq board[x+1][y]) \wedge$
  $\forall(\text{x, y: } \mathbb{N} | 0 \leq x < size \wedge 0 \leq y < size - 1 \Rightarrow board[x][y] \neq board[x][y+1])$
- exception: None

bestTile()

- output: out := $\max(\{\text{x, y: } \mathbb{N} | 0 \leq x < size \wedge 0 \leq y < size : board[x][y].getValue()\})$
- exception: None

getTile(x, y)

- output: out := board[x][y].toString()

- exception: ($\neg$ validateCell$(x, y) \Rightarrow$ IndexOutOfBoundsException)

equals(b)

- output: out := $\forall$(x, y: $\mathbb{N} | 0 \leq x < size \wedge 0 \leq y < size \Rightarrow board[x][y].toString() = b.getTile(x, y)$)
- exception: None

### Local Functions

randomFrom: seq of T $\rightarrow$ T

randomFrom(Seq) $\equiv$ Seq[$floor(Seq.size * random())$]

randomTile: void $\rightarrow \mathbb{N}$

randomTile() $\equiv (floor(random() * 10) = 0 \Rightarrow 4 | True \Rightarrow 2)$

validateCell: $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{B}$

validateCell$(x, y) \equiv x <$ size $\wedge\ y <$ size $\wedge\ x >= 0 \wedge y >= 0$

# TextView Module

## Module

## Uses

None

## Syntax

### Exported Types

None

### Exported Constants

None

### Exported Access Programs

| Routine name | In | Out | Exceptions |
|---|---|---|---|
| getInstance | | TextView | |
| printWelcomeMessage | | | |
| printHowToPlay | | | |
| printBoard | BoardT | | |
| printWinMessage | | | |
| printFailMessage | | | |
| printEndingMessage | | | |

## Semantics

## Environment Variables

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

### State Variables

visual: TextView

### State Invariant

None

**Assumptions**

- The TextView constructor is called for each object instance before any other access routine is called for that object. The constructor can only be called once.

**Access Routine Semantics**

getInstance():

- transition: visual := (visual = null ⇒ new TextView())
- output: *self*
- exception: None

printWelcomeMessage():

- transition: window := Displays a welcome message when user first enters the game.

printHowToPlay():

- transition: window := Displays a message to teach user how to play this game.

printBoard(*board*):

- transition: window := Draws the game board onto the screen. Each cell of the board is accessed and printed using the $getTile$ method from $BoardT$. The board[x][y] is displayed in a way such that x is increasing from the top of the screen to the bottom, and y value is increasing from the left to the right of the screen. For example, board[0][0] is displayed at the top-left corner and board[size-1][size-1] is displayed at bottom-right corner.

printWinMessage():

- transition: window := Displays a win message when the user wins the game.

printFailMessage():

- transition: window := Displays a fail message when the user fails the game.

printEndingMessage():

- transition: Prints a ending message after the user exit the game (entered "e").

**Local Function:**

TextView: void → TextView
TextView() ≡ new TextView()

# GameController Module

## GameController Module

## Uses

BoardT, TextView

## Syntax

### Exported Types

None

### Exported Constants

None

### Exported Access Programs

| Routine name | In | Out | Exceptions |
|---|---|---|---|
| getInstance | BoardT, TextView | GameController | |
| initializeGame | | | |
| readDirection | | String | IllegalArgumentException |
| getStatus | | $\mathbb{B}$ | |
| move | String | | |
| displayWelcomeMessage | | | |
| displayGuideMessage | | | |
| displayBoard | | | |
| displayWinMessage | | | |
| displayFailMessage | | | |
| displayEnding | | | |
| runGame | | | |

## Semantics

## Environment Variables

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

### State Variables

model: BoardT
view: TextView
controller: GameController

### State Invariant

None

### Assumptions

- The GameController constructor is called for each object instance before any other access routine is called for that object. The constructor can only be called once.
- Assume that model and view instances are already initialized before calling GameController constructor

### Access Routine Semantics

getInstance($m$, $v$):

- transition: controller := (controller = null $\Rightarrow$ new GameController $(m, v)$) item output: *self*
- exception: None

initializeGame():

- transition: model := sum({x, y: $\mathbb{N} | 0 \leq x < size \wedge 0 \leq y < size \wedge board[x][y].getValue() = 0 : 1$}) = size*size - 2 $\wedge$ sum({x, y: $\mathbb{N} | 0 \leq x < size \wedge 0 \leq y < size \wedge (board[x][y].getValue() = 2 \vee board[x][y].getValue() = 4) : 1$}) = 2

  *// Initialize an board with only two values, 2 or 4 randomly, at random positions*
- output: None
- exception: None

readDirection()

- output: *input* : String, entered from the keyboard by the User

- exception: *exc* := (input ≠ “w” ∧ input ≠ “s” ∧ input ≠ “a” ∧ input ≠ “d” ∧ input ≠ “e” ⇒ IllegalArgumentException)

  *// “w”, “s”, “a”, “d” for direction, “e” to exit the game*

getStatus():

- transition: None
- output: *out* := model.gameOver()
- exception: None

move(dir)

- transition: (dir = “w” ⇒ model.up() | dir = “s” ⇒ model.down() | dir = “a” ⇒ model.left() | dir = “d” ⇒ model.right())
- output: None

displayWelcomeMessage():

- transition: view := view.printWelcomeMessage()

displayGuideMessage():

- transition: view := view.printHowToPlay()

displayBoard():

- transition: view := view.printBoard(model)

displayWinMessage():

- transition: view := view.printWinMessage()

displayFailMessage():

- transition: view := view.printFailMessage()

displayEnding():

- transition: view := view.printEndingMessage()

runGame():

- transition: operational method for running the game. The game will start with a welcome message, next teaching the user how to play this game, then display the board and let the user enter keyboard input to play the game. Eventually, when the game ends, prompt a win or fail message depending on how this game is ended. If the user does not want to play this game in the middle of the game, he/she can enter "e" to exit. Any unacceptable keyboard input will cause the game to end. When a keyboard input is acceptable (except "e), no matter whether it can cause change to the game board, a 2 or 4 will be placed in an empty space after this round. If there is exception, handle it in your way.

- output: None

**Local Function:**

GameController: BoardT $\times$ TextView $\rightarrow$ GameController
GameController($model, view$) $\equiv$ new GameController($model, view$)

# Critique of Design

- This specification is consistent on name conventions, ordering of parameters in argument lists, and exception handling.
- This specification is not essential since some methods are not necessary. For example, all display methods in GameController can be replaced by calling the corresponding methods in view. But they are kept, because the specification wants to highlight the functionality of GameController: connect model and view.
- This design is not general. The seperation of concerns (MVC) of the program allows user to easily predict how the modules can be used.
- This design is minimal since no method can have transition of state variables and output at the same time.
- MVC design pattern makes this program more maintainable. It decomposes this program into three components: model component stores the data and state of the game, view component displays the state of the game to the player, and the controller component handles the inputs given by the player and updates view. This design allows programmers to work in parallel. Moreover, development in one component does not require other components to change at the same time. Because of this, this design achieves high cohesion and low coupling.
- This design is good at information hiding. All state variables are private, but there are also methods that allow data to be accessed from other modules.
- The two constructors in BoardT improve the flexibilty of the module. The user can choose to play a regular game board by entering `make demo` or a board that is customized by him/herself. Moreover, the constructor that receives given board makes the testing for BoardT easilier, since we can predict how the board changes after each method and checks its correctness.
- The GameController and TextView modules are both designed to be an abstract object, because for the game, only one instance is required to control the action and diaplay corresponding message during runtime. Singleton design pattern is also shown here.