1 Overview of the design

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

The MVC design pattern is specified and implemented in the following way: the module *Board* stores the state of the game board and the status of the game. A view module *UserInterface* can display the state of the game board and game using text-based graphics. The controller *GameController* is responsible for handling input actions.

The Singleton Pattern is applied for *GameController* and *UserInterface* by use the getInstance() method to obtain the abstract object.

Likely Changes my design considers:

- Data structure used for storing the game board
- The visual representation of the game such as UI layout.
- Change in peripheral devices for taking user input.
- Change in board size to adjust the difficulty of the game.

Tile ADT Module

Template Module

Board

Uses

None

Syntax

Exported Types

None

Exported Constant

None

Exported Access Programs

Routine name	**In**	**Out**	**Exceptions**
Tile		Tile	
Tile	$\mathbb{N}$	Tile	
Tile	$\mathbb{N}$, $\mathbb{N}$, $\mathbb{N}$	Tile	
merge		$\mathbb{N}$	
getValue		$\mathbb{N}$	
setValue	$\mathbb{N}$		
isEmpty		$\mathbb{B}$	
toString		String	

Semantics

State Variables

value: $\mathbb{N}$
row: $\mathbb{N}$
col: $\mathbb{N}$

State Invariant

None

Assumptions

- The constructor Tile is called for each object instance before any other access routine is called for that object.

Access Routine Semantics

Tile():

- transition: $value := 0$
- output: $out := self$
- exception: none

Tile(v):

- transition: $value := v$
- output: $out := self$
- exception: none

Tile(v, r, c):

- transition: $value, row, col := v,\ r,\ c$
- output: $out := self$
- exception: none

merge():

- transition: $value := value + value$
- output: $out := value$
- exception: none

getValue():

- output: $out := value$

- exception: none

setValue(v):

- transition: $value := v$
- exception: none

isEmpty():

- output: $out := value = 0$
- exception: none

toString():

- output: $out :=$ (String) $value$
- exception: none

Board ADT Module

Template Module

Board

Uses

Tile

Syntax

Exported Types

None

Exported Constant

None

Exported Access Programs

Routine name	**In**	**Out**	**Exceptions**
Board	$\mathbb{N}$	Board	
getTiles		seq of (seq of Tile)	
getTileAt	$\mathbb{N}$, $\mathbb{N}$	Tile	
setTileAt	$\mathbb{N}$, $\mathbb{N}$, Tile		
getScore		$\mathbb{N}$	
getSize		$\mathbb{N}$	
isFull		$\mathbb{B}$	
isMovePossible		$\mathbb{B}$	
isGameOver		$\mathbb{B}$	
moveUp			
moveDown			
moveLeft			
moveRight			
restart			

Semantics

State Variables

tiles: sequence [Size, Size] of Tile *// Representation of the board*
emptyTiles: $\mathbb{N}$ *// Number of empty tiles in the board*
size: $\mathbb{N}$ *// Size of the board*
score: $\mathbb{N}$ *// Score of the game*
gameOver: $\mathbb{B}$ | $gameOver = false$ *// Variable indicating if the game is over or not*

State Invariant

None

Assumptions

- The constructor Board 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, to generate a random tile with value 2 or 4.
- Assume that the random tile generator is called after the board is initialized with empty tiles in the constructor.
- User does not provide values that generate out of bounds error

Access Routine Semantics

Board(s):

- transition:

$$\text{tiles} := \langle \begin{array}{c} \langle 0_0, ..., 0_s \rangle \\ \langle 0_1, ..., 0_s \rangle \\ \langle 0_2, ..., 0_s \rangle \\ \langle \vdots, ..., 0_s \rangle \\ \langle 0_s, ..., 0_s \rangle \end{array} \rangle$$

  $size, emptyTiles, score := s, s * s, 0$
  newRandomTile()
- output: $out := self$

- exception: none

getTiles():

- output: $out := \langle i : \text{Tile} | i \in tiles : i \rangle$
- exception: none

getTileAt(r, c):

- output: $out := \text{tiles}[r][c]$
- exception: none

setTileAt(r, c,t):

- transition: $\text{tiles}[r][c] := t$
- exception: none

getScore():

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

getSize():

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

isFull():

- output: $out := emptyTiles = 0$
- exception: none

isMovePossible():

- output: $out := \langle \exists i, j : \mathbb{N} | i, j \in [0..size - 1] : tiles[i][j] = tiles[i][j + 1] \vee tiles[i][j] = tiles[i + 1][j] \rangle$
- exception: none

isGameOver():

- output: $out := \langle \exists i : \text{Tile} | i \in tiles : (isFull() \wedge \neg isMovePossible()) \vee i = 2048 \rangle$
- exception: none

moveUp():

- transition: tiles := board is updated by moving the existing tiles up and merging any appropriate tiles.
  tiles := mergeTiles()
- exception: none

moveDown():

- transition: tiles := board is updated by moving the existing tiles down and merging any appropriate tiles.
  tiles := mergeTiles()
- exception: none

moveLeft():

- transition: tiles := board is updated by moving the existing tiles left and merging any appropriate tiles.
  tiles := mergeTiles()
- exception: none

moveRight():

- transition: tiles := board is updated by moving the existing tiles right and merging any appropriate tiles.
  tiles := mergeTiles()
- exception: none

restart():

- transition: $tiles := Board()$
- output: $out := self$
- exception: none

Local Functions

mergeTiles: Seq of (Seq of Tile) $\rightarrow$ Seq of Tile
mergeTiles(tiles) $\equiv \langle i, j : \mathbb{N} | i, j \in [0..size-1] : tiles[i][j] = tiles[i][j+1] \rightarrow tiles[i][j+1] = 0 \wedge tiles[i][j] = 2 * tiles[i][j] \quad | \quad tiles[i][j] = tiles[i+1][j] \rightarrow tiles[i+1][j] = 0 \wedge tiles[i][j] = 2 * tiles[i][j] \rangle$
// Loop through the board and merge any appropriate tiles

UserInterface Module

UserInterface Module

Uses

None

Syntax

Exported Types

None

Exported Constants

None

Exported Access Programs

Routine name	In	Out	Exceptions
getInstance		UserInterface	
printBoard	Board		
printWelcomeMessage			
printInvalidInputMessage			
printBoardSizePrompt			
printRestartPrompt			
printQuitPrompt			
printMovePrompt			
printEndingMessage			

Semantics

Environment Variables

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

State Variables

visual: UserInterface

State Invariant

None

Assumptions

- The UserInterface 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 $\Rightarrow$ new UserInterface())
- output: *self*
- exception: None

printBoard(*board*):

- transition: window := Draws the game board onto the screen. Each cell of the board is accessed using the *getTileAt* method from *BoarT*. The board[x][y] is displayed in a way such that x is increasing from the left of the screen to the right, and y value is increasing from the top to the bottom of the screen. For example, board[0][0] is displayed at the top-left corner and board[3][3] is displayed at bottom-right corner.

printWelcomeMessage():

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

printInvalidInputMessage():

- transition: window := Displays a message when the user enters an invalid input.

printBoardSizePrompt():

- transition: window := Window appends a prompt message asking the user to select the desired board size

printRestartPrompt():

- transition: window := Window appends a prompt message asking the user if they would like to restart the game printQuitPrompt():

- transition: window := Window appends a prompt message asking the user if they would like to quit the game

printMovePrompt():

- transition: window := Appends a prompt message asking the user to enter the desired move

printEndingMessage():

- transition: Prints a ending message after the user quits the game.

Local Function:

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

GameController Module

GameController Module

Uses

Board, UserInterface

Syntax

Exported Types

None

Exported Constants

None

Exported Access Programs

Routine name	**In**	**Out**	**Exceptions**
getInstance	Board, UserInterface	GameController	
initializeGame			
readInput		String	
displayWelcomeMessage			
displayInvalidInputMessage			
displayBoard			
displayEndingMessage			
displayBoardSizePrompt			
displayRestartPrompt			
displayQuitPrompt			
displayMovePrompt			
runGame			

Semantics

Environment Variables

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

State Variables

model: Board
view: UserInterface
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))
- output: *self*
- exception: None

initializeGame():

- transition: model := $newBoard()$
- output: None
- exception: None

readInput():

- output: *out* := *input // a String, entered from the keyboard by the User*

displayWelcomeMessage():

- transition: view := view.printWelcomeMessage()

displayInvalidInputMessage():

- transition: view := view.printInvalidInputMessage()

displayBoard():

- transition: view := view.printBoard()

displayEndingMessage():

- transition: view := view.printEndingMessage()

displayBoardSizePrompt():

- transition: view := view.printBoardSizePrompt()

displayRestartPrompt():

- transition: view := view.printRestartPrompt()

displayQuitPrompt():

- transition: view := view.printQuitPrompt()

displayMovePrompt():

- transition: view := view.printMovePrompt()

runGame():

- transition: operational method for running the game. The game will start with a welcome message, asking the user if they would like to start the game, next asking the user to select a board size, then display the board and let the user to play the game. Eventually, when the game ends, prompt a message thanking the player for playing the game and informing the player how to start a new game.
- output: None

Local Function:

GameController: Board $\times$ UserInterface $\rightarrow$ GameController
GameController($model, view$) $\equiv$ new GameController($model, view$)

Critique of Design

- The design is consistent because all the method names and variables are consistent. For example, getters start with 'get', setters start with 'set', condition methods start with 'is', the view methods start with 'print', and the control methods that use the interface start with 'display'.
- The design is not general as it is specifying a game with certain rules. Therefore, those rules cannot be changed and made general such as having tiles with different types of numbers. However, the separation of the tiles from the board in the `Tile` module allows for some generality and design for change as the way each tile is represented can be changed in the future without affecting the board.
- The design is not essential because the model can be implemented with only the `Board` module without the `Tile`. However, I chose to make the design with the `Tile` module as it provides some generality as well as allows for design for change and information hiding.
- I choose to specify the `Board` module as well as the `Tile` module as an ADT, because It is more convenient to create a new instance of the board whenever a user wants to restart the game, as well as creating new instances of each tile whenever the board is updated.
- The controller and view modules are specified as a single abstract object which follows the Singleton design pattern because only one instance is required to control the action within those modules during runtime. Therefore, any unexpected state changes can be eliminated.
- The multiple constructors in `Tile` improve the flexibilty of the module. Tiles can be added automatically with an empty value or with a specified value, as well as with a defined position, which helps when generating a new random tile.
- Inside the `Board` constructor, there is the possibility of creating an entirely empty board with no random tiles. This is helpful from a testing perspective in order to control the testing environment.
- The design is minimal as every access method only has one transition.
- The test cases are designed with the goal to validate the correctness of and any possible reveal errors during program execution, with every access routine having at least one test case.

- I did not build any test cases for testing the controller module since the implementation of the controller's access methods uses methods from the model. The test cases for the model are in *TestBoard.java* as well as in *TestTile.java*

- Applying the MVC design pattern makes my design maintainable and the risk of unexpected errors when making changes. This is done by decomposing into three components based on the separation of concerns where the model component encapsulates the internal data and status of the game, the view displays the state of the game, and the controller handles the input actions to execute related actions to respond to events.

- My design achieves high cohesion and low coupling by applying the MVC design pattern. The design has high cohesion since it groups related functionalities within each module. The design also has low coupling because it was created with design for change in mind, and thus, the modules (model, view, controller) are mostly independent of each other. So a change in one of the modules does not heavily impact the others.