Assignment 4 Specification

Alan Scott

April 12, 2021

Design Introduction

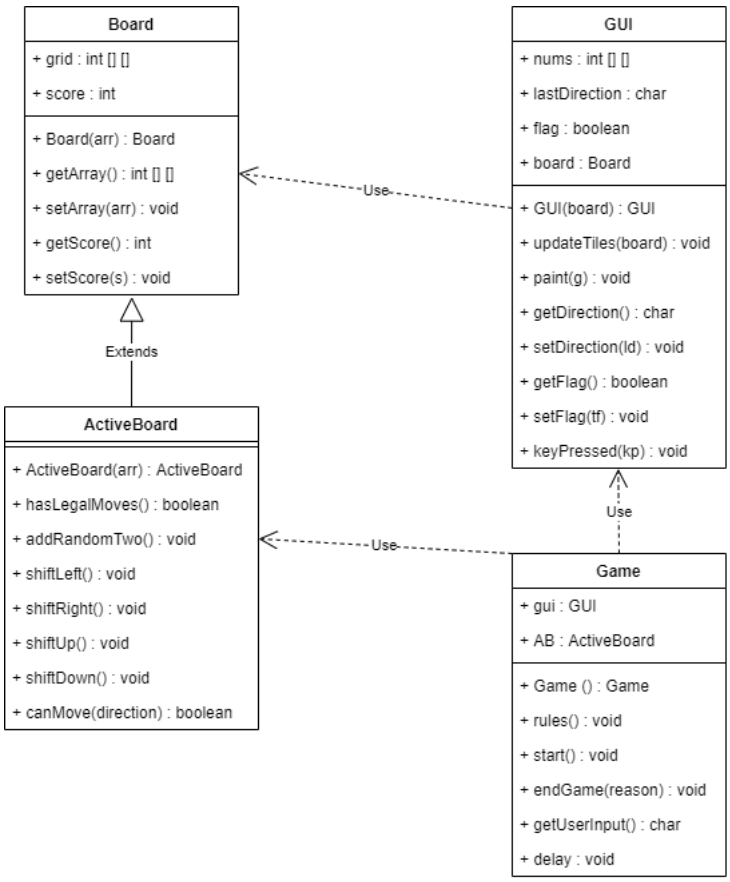
Likely Changes

This project was designed with the idea that the whole game of 2048 could be completely replaced with another game using a similar setup. For this reason, the 4x4 grid and the operations to carry out a game of 2048 are completely separate. The 4x4 board is defined as its own object, with only basic methods such as getters and setters for the grid data and score available. The setup for the game 2048 is a child class of Board, and it contains all the operations necessary for a 4x4 game of 2048. If you were however to create a different score based game based on a similar board, it could be re-purposed to be used for that setup.

Design Overview

The design for the 2048 game involves four separate modules. The first module is Board, which emulates a 4x4 grid of numbers, and keeps track of the score. The 4x4 grid data is stored using a two dimensional array, where the rows and column of the array correspond to the rows and columns of the 2048 board. The second module, ActiveBoard, is a Board object which contains all the necessary operations inherent to the 2048 game. It is a child class of Board, and thus inherits its data structure and state variables. The third module is the GUI module, which is a specialized JFrame (ie it inherits JFrame). This module covers the graphical display of the game state, namely the 4x4 grid and the current user score. It also handles user key inputs through the implementation of the KeyListener interface. The final module of the program is Game, which is the controller that operates the various classes. It takes in user input and computes the game states through operations performed on the ActiveBoard class. When Game reaches an end state (by lack of moves or user exit) it terminates the program.

Program UML Diagram



Board Abstract Data Type

Module

Board

Uses

None

Syntax

Exported Constants

None

Exported Types

None

Exported Access Programs

Routine name	In	Out	Exceptions
new Board	sequence[4] of sequence[4] of $\mathbb{N}$	Board	
getArray		sequence[4] of sequence[4] of $\mathbb{N}$	
setArray	sequence[4] of sequence[4] of $\mathbb{N}$		
getScore		$\mathbb{N}$	
setScore	$\mathbb{N}$		
hasZero		$\mathbb{B}$	

Semantics

State Variables

$grid$: sequence[4] of sequence[4] of $\mathbb{N}$
$score$: $\mathbb{N}$

State Invariant

None

Assumptions

Assuming the input size of the 2D array will always be 4x4

Design Decision

The grid of the 2048 board was chosen to be represented as a 2D array of size 4x4, with the dimensions of the array corresponding to the dimensions of the 2048 board. The coordinates of the board are as follows:

(0,0)			(0,4)
(4,0)			(4,4)

Access Routine Semantics

new Board(*arr*):

- transition: *grid* := *arr*
- output: *out* := self
- exception: none

getArray():

- transition: none
- output: *out* := *grid*
- exception: none

setArray(*arr*):

- transition: *grid* := *arr*
- output: none
- exception: none

getScore():

- transition: none

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

setScore(s):

- transition: $grid := s$
- output: none
- exception: none

hasZero():

- transition: none
- output: $out := (0 \in grid)$
- exception: none

Board Operations Module

Module

ActiveBoard

Uses

Board

Syntax

Exported Constants

None

Exported Types

None

Exported Access Programs

Routine name	In	Out	Exceptions
new ActiveBoard	sequence[4] of sequence[4] $\mathbb{N}$	ActiveBoard	
hasLegalMoves		$\mathbb{B}$	
addRandomTwo			
shiftLeft			
shiftRight			
shiftUp			
shiftDown			
canMove	char	$\mathbb{B}$	

Semantics

State Variables

grid : sequence[4] of sequence[4] of $\mathbb{N}$
score : $\mathbb{N}$
Both of these state variables are inherited from Board

State Invariant

None

Assumptions

Assuming the array input to the constructor will be of size 4x4.

Access Routine Semantics

new ActiveBoard(arr)

- transition: score = 0, grid = arr
- output: $out := self$
- exception: none

hasLegalMoves()

- transition: none
- output: $out := (0 \in grid \implies True | \exists i, j | grid[i][j] = grid[i+1][j] \implies True | grid[i][j] = grid[i][j+1] \implies True | True \implies False)$
  # if a given 2D grid contains a zero or two adjacent identical values, the it is possible for a legal move to be made.
- exception: none

addRandomTwo()

- transition: $(\exists i, j | 0 \leq i \leq 3, 0 \leq j \leq 3 | grid[i][j] = 0) \wedge grid[i][j] = (randInt() < 0.9 \implies 2 | True \implies 4)$
  # choose a random element $grid[i][j]$ which equals zero and set it to 2, with a 10% chance of setting it to 4.
- output: none
- exception: none

shiftLeft()

- transition: $grid := (\exists n | n > j | grid[i][j] = grid[i][n] \implies grid[i][j] = grid[i][j] + grid[i][n]) \wedge grid[i][n] = 0) \wedge (\exists m | m > j | grid[i][j] = 0 \wedge grid[i][m] \neq 0 \implies grid[i][j], grid[i][m] = grid[i][m], grid[i][j])$
  # If two horizontally adjacent items (or items separated by a 0) are the same value, sum them in the leftmost element, and make the rightmost element blank. Then, shift all of the elements to the left.

- output: none

- exception: none

shiftRight()

- transition: $grid := (\exists n | n < j | grid[i][j] = grid[i][n] \implies grid[i][j] = grid[i][j] + grid[i][n]) \wedge grid[i][n] = 0) \wedge (\exists m | m < j | grid[i][j] = 0 \wedge grid[i][m] \neq 0 \implies grid[i][j], grid[i][m] = grid[i][m], grid[i][j])$
  # If two horizontally adjacent items (or items separated by a 0) are the same value, sum them in the rightmost element, and make the leftmost element blank. Then, shift all of the elements to the right.

- output: none

- exception: none

shiftUp()

- transition: $grid := (\exists n | n > j | grid[i][j] = grid[n][j] \implies grid[i][j] = grid[i][j] + grid[n][j]) \wedge grid[n][j] = 0) \wedge (\exists m | m > j | grid[i][j] = 0 \wedge grid[m][j] \neq 0 \implies grid[i][j], grid[m][j] = grid[m][j], grid[i][j])$
  # If two vertically adjacent items (or items separated by a 0) are the same value, sum them in the uppermost element, and make the lower element blank. Then, shift all of the elements upward.

- output: none

- exception: none

shiftDown()

- transition: $grid := (\exists n | n < j | grid[i][j] = grid[n][j] \implies grid[i][j] = grid[i][j] + grid[n][j]) \wedge grid[n][j] = 0) \wedge (\exists m | m < j | grid[i][j] = 0 \wedge grid[m][j] \neq 0 \implies grid[i][j], grid[m][j] = grid[m][j], grid[i][j])$
  # If two vertically adjacent items (or items separated by a 0) are the same value, sum them in the lowermost element, and make the upper element blank. Then, shift all of the elements downward.

- output: none
- exception: none

canMove(direction)

- transition: none
- output: *out* :=
  $(direction = "D") \implies (\exists i, j | 0 \leq i, j \leq 3 | (grid[i][j] = grid[i-1][j] \wedge grid[i][j] \neq 0) \vee (grid[i][j] = 0 \wedge grid[i-1] \neq 0))$
  $(direction = "U") \implies (\exists i, j | 0 \leq i, j \leq 3 | (grid[i][j] = grid[i+1][j] \wedge grid[i][j] \neq 0) \vee (grid[i][j] = 0 \wedge grid[i+1] \neq 0))$
  $(direction = "R") \implies (\exists i, j | 0 \leq i, j \leq 3 | (grid[i][j] = grid[i][j-1] \wedge grid[i][j] \neq 0) \vee (grid[i][j] = 0 \wedge grid[i][j-1] \neq 0))$
  $(direction = "L") \implies (\exists i, j | 0 \leq i, j \leq 3 | (grid[i][j] = grid[i][j+1] \wedge grid[i][j] \neq 0) \vee (grid[i][j] = 0 \wedge grid[i][j+1] \neq 0))$
  # given a direction ['L', 'R', 'U', 'D'] corresponding to the directions [left,right,up,down], determine if there exists a zero that can be shifted into when shifting in that direction, or if values adjacent on that shift axis are the same, and can be merged.
- exception: none

Controller Module

Module

Game

Uses

Board, ActiveBoard, GUI, JFrame, KeyListener

Syntax

Exported Constants

None

Exported Types

None

Exported Access Programs

Routine name	In	Out	Exceptions
new Game		Game	

Semantics

State Variables

$gui : GUI$
$AB : ActiveBoard$

Environment Variables

console : Console for displaying text output

State Invariant

None

Assumptions

None

Access Routine Semantics

new Game():

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

rules()

- transition: none
- output : $out :=$ output to console strings representing the title and the rules of the game
- exception: none

start()

- transition: $AB := AB.hasLegalMoves() \implies (getUserInput() = "U" \implies AB.shiftLeft() | getUserInput() = "D" \implies AB.shiftDown() | getUserInput = "L" \implies AB.shiftLeft() | userInput() = "R" \implies AB.shiftRight())$ — True $\implies$ end game
- output: $output :=$ output the ActiveBoard to the GUI class based on user input.
- exception: none

endGame(reason)

- transition: none
- output: out := Output to console user score and the reason for the game ending.
- exception: none

getUserInput()

- transition: none
- output: $out := gui.getDirection()$
- exception: none

delay(millis)

- transition: none
- output: out := millisecond delay corresponding to given parameter
- exception: none

GUI Module

Module

GUI

Uses

JFrame, KeyListener

Syntax

Exported Constants

None

Exported Types

None

Exported Access Programs

Routine name	In	Out	Exceptions
new GUI		GUI	
updateTiles	Board		
paint	Graphics		
getDirection		char	
setDirection	char		
setFlag	$\mathbb{B}$		
getFlag		$\mathbb{B}$	
keyPressed	KeyEvent	char	

Semantics

Environment Variables

window : JFrame window displaying graphics to user

State Variables

board : Board
lastDirection : char
nums : sequence[4] of sequence[4] of $\mathbb{N}$
flag : $\mathbb{B}$

State Invariant

None

Assumptions

None

Access Routine Semantics

new GUI(board):

- transition: $this.board, this.nums := board, board.getArray()$
- output: *out* := self
- exception: none

updateTiles(board):

- transition: $nums := board.getArray()$
- output: none
- exception: none

paint(g):

- transition: window := Draws the game grid onscreen. A 4x4 grid is drawn onscreen, with numberical values occupying each slot. The 4x4 grid represents the data taken from the 4x4 2D array from the given Board object, with the numbers within the spots representing the given values corresponding to the elements within the 4x4 array.
- output: none
- exception: none

getDirection():

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

setDirection(ld):

- transition: $lastDirection := ld$
- output: none
- exception: none

setFlag(tf):

- transition: $flag := tf$
- output: none
- exception: none

getFlag():

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

keyPressed(kp):

- transition: $lastDirection := (kp \equiv KeyEvent.VK\_DOWN \implies 'D' | kp \equiv KeyEvent.VK\_UP \implies 'U' | kp \equiv KeyEvent.VK\_LEFT \implies 'L' | kp \equiv KeyEvent.VK\_RIGHT \implies 'R' | kp \equiv KeyEvent.VK\_ESCAPE \implies 'E')$

- output: none
- exception: none

Design Critique

Information Hiding

The concept of information hiding was applied throughout the modules. For example, and methods that did not need to be accessed outside of the class, such as `getUserInput()` or `endGame()` in Game were made private, whereas methods that needed to be accessed outside of the class were made public, such as `canMove()` for ActiveBoard. State variables were all made private, as any accessing of these variables would be done through their respective get methods. The exception to this rule was the state variables for Board, which were given a privacy level of protected, as they would need to be accessed by any child classes of Board.

Cohesion

On a general level, the modules in the 2048 program had a high degree of cohesion. Each module only contains methods that directly relate to the effective function of the program, with no unnecessary methods being included. For example, the ActiveBoard module, which emulates the behaviour of a 2048 board, only contains methods that are necessary for the correct function of the board, containing only methods to shift the board values, a method to add a random new value, and to determine if certain operations are valid. Otherwise, no other functions are added as no other computation is necessary for the correct operation of the program.

Generality

The whole setup of the 4x4 board was made as general as possible, as the Board module only contained enough functions to set and get state variables. Using this generalized module, the ActiveBoard module is designed to be a specialization on the Board module, with more specific restraints and operations applied to the Board.

Consistency

The specification strives to make the program as consistent as possible. On a superficial level, the naming conventions for both the methods and variables follow the same format, namely camel notation. On a more structural level, the structure of the classes and methods were designed consistently. For example, any operations on ActiveBoard were consistently performed using the built in methods (via blackbox methods) rather than extracting the data from the object, performing the computations, then updating the ActiveBoard afterwards. On that note, all methods were implemented with the concept of

blackboxing in mind, only taking in a parameter and performing the necessary transitions without any further interaction.

Essentiality

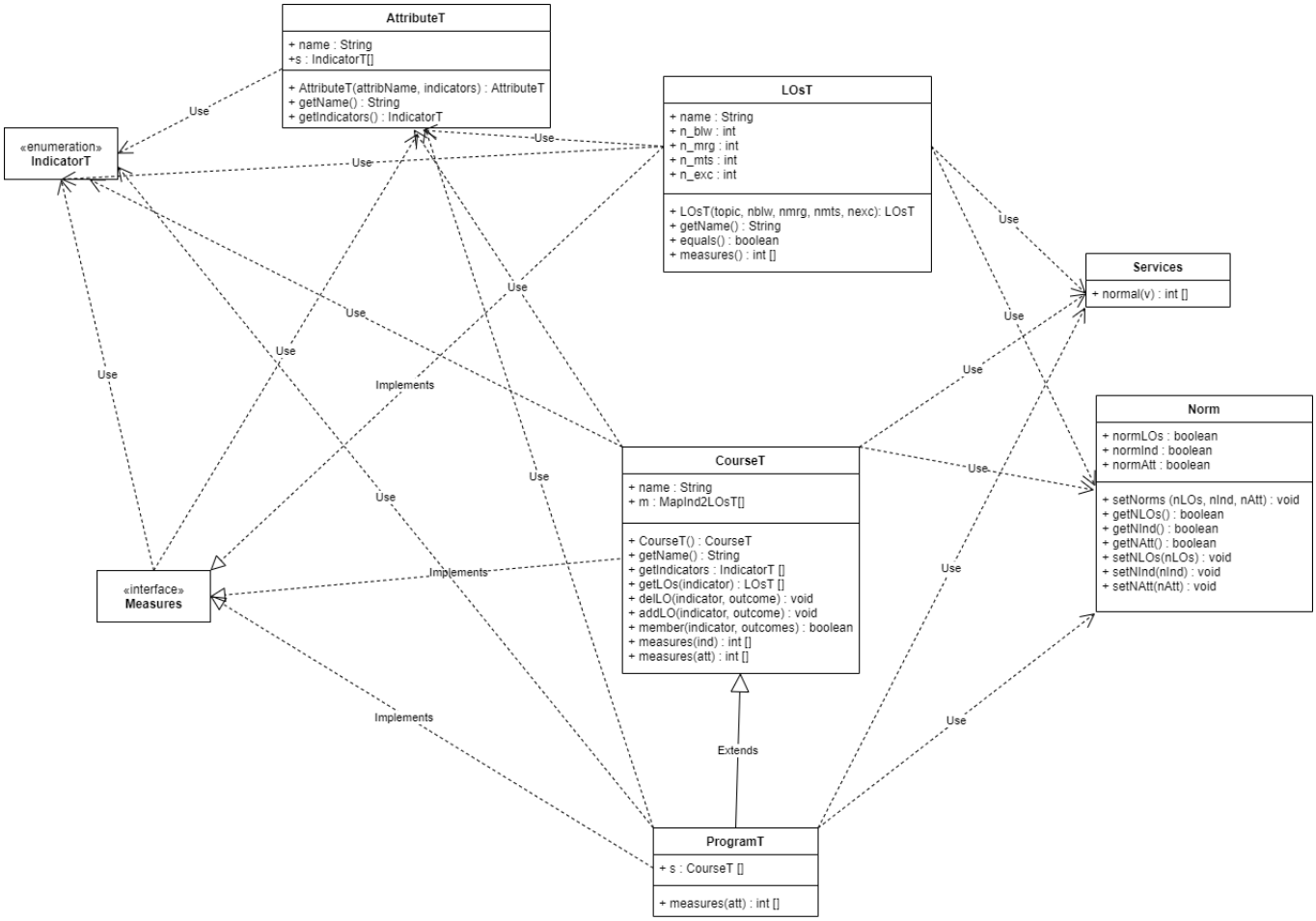
All of the modules within the program can be considered to be essential, as no module contains multiple methods performing the same task. The shift methods in ActiveBoard do perform the same general task, but since the execution of the task in each direction is different, the task was separated across four separate methods for the sake of program simplicity and understandablity (whereas the `canMove(direction)` method was kept as one method due to the similarity in the tasks in each direction and the simplicity of the tasks).

Minimality

The modules outlined in the MIS were kept as minimal as possible. With the exception of the controller module, all methods (not including the constructors) in the other three modules were minimal, as no method both mutated the state of the object or returned an output value. The exception for the controller class was made since the controller would have to handle user input and output as well as update the game state at the same time, requiring the method in control of the game flow to both cause a transition and an output.

Questions

A3 UML Diagram



Convex Hull Algorithm CFG