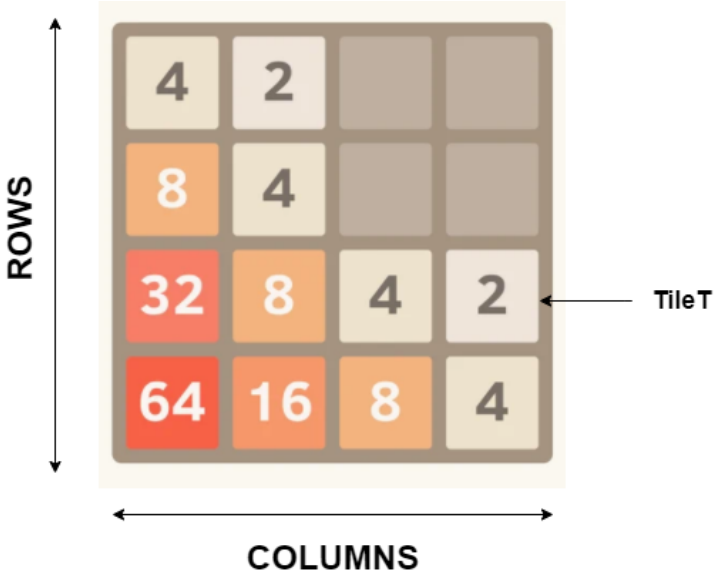
Assignment 4, Specification

SFWR ENG 2AA4, COMP SCI 2ME3

April 12, 2021

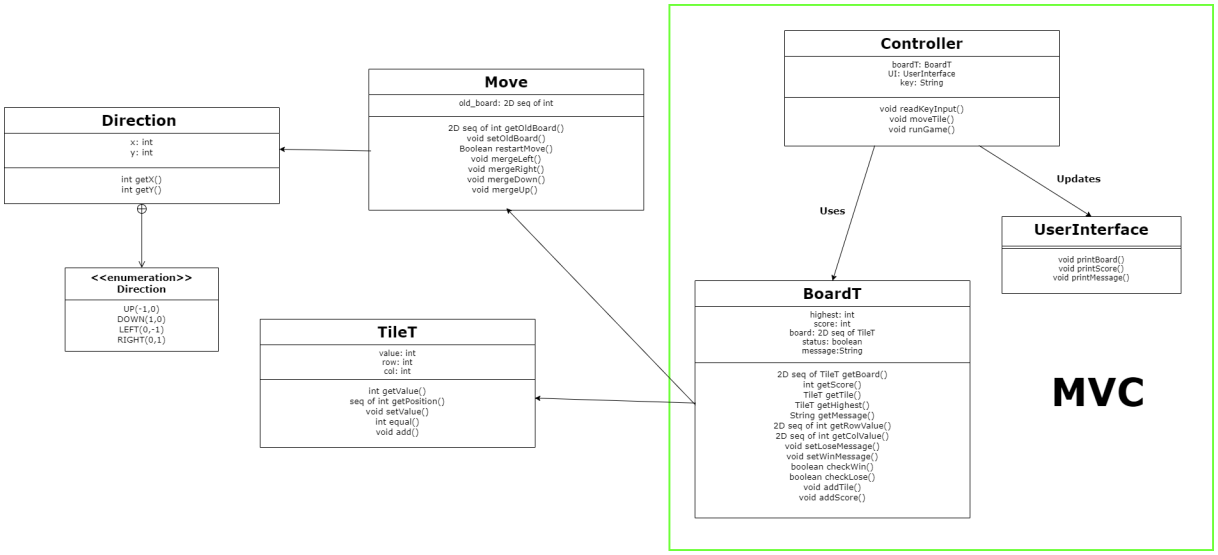
This Module Interface Specification (MIS) document contains modules, types and methods for implementing 2048. The game begins with a board that consist of two randomly generated tiles of values 2 or 4. The user can move the tiles up, down, left or right. When shifted, adjacent tiles with the same values will be combined together. Note that each tile, can only be combined once per move. Tiles with similar values will remain in their current position and will not be combined. If the adjacent value is zero, the tile will continue to move across the board, until it hits the edge of the board or a tile with a different value. The player loses when the all tiles cannot be moved anymore. On the other hand, the player wins when they have combined enough tiles to create a tile with a value of 2048. The game can be launched and play by typing `make expt` in terminal. Note. use "a", "w", "s", and "d" to control tiles.



The 2048 board image above is from
https://apps.apple.com/ca/app/2048-by-gabriele-cirulli/id868076805.

1 Overview of the design

This design applies Module View Specification (MVC) design pattern.The MVC components are *Controller* (controller module), *BoardT* (model module), and *UserInterface* (view module). *BoardT* uses Move *Move* to merge tiles in a given direction. While, *Direction* provides enumerated types for the tile movements and is used by *Move*.
A UML diagram is provided below to vvisualize the software architecture.



The MVC design pattern seprates the model, user interface, and the controls into three components. For this game, the MVC design pattern is implemented in the following way: the module *BoardT* stores the state and status of the board. The *Userinterface* is the view portion of MVC. It displays the state of the board using ASCII graphics. The controller uses the state of *BoardT* and is able to manipulate it. It also updates the UserInterface.

Likely Changes my design considers:

- Data structure, array, used to store the game board.
- The game halts, when 2048 has been reached.

Board Module

Template Module

BoardT

Uses

TileT

Syntax

Exported Constants

Size = 4 //Size of the board 4 x 4

Exported Types

BoardT = ?

Exported Access Programs

Routine name	In	Out	Exceptions
new BoardT		BoardT	
getBoard		seq of (seq [4] of TileT)	
getScore		$\mathbb{N}$	
getTile	$\mathbb{N}$, $\mathbb{N}$	TileT	
getHighest		TileT	
getMessage		String	
getRowValue	$\mathbb{N}$	seq [4] of $\mathbb{N}$	
getColValue	$\mathbb{N}$	seq [4] of $\mathbb{N}$	
setLoseMessage			
setWinMessage			
checkWin		$\mathbb{B}$	
checkLose		$\mathbb{B}$	
addTile			
addScore	$\mathbb{N}$		

Semantics

State Variables

$highest : \mathbb{N}$
$score : \mathbb{N}$
$board$: seq of (seq [4] of TileT)
$message$: String

State Invariant

None

Assumptions

- The constructor BoardT is called for each object instance and before any other access routine for that object.
- Assume a random function that generates a random value between 0 and 1.

Access Routine Semantics

new BoardT():

- transition:
  $$\text{board} := \left\langle \begin{array}{c} \langle \text{TileT}()_0, ..., \text{TileT}()_3 \rangle \\ \langle \text{TileT}()_0, ..., \text{TileT}()_3 \rangle \\ \langle \text{TileT}()_0, ..., \text{TileT}()_3 \rangle \\ \langle \text{TileT}()_0, ..., \text{TileT}()_3 \rangle \end{array} \right\rangle$$
  where the value of any two random tiles is 2 or 4 and the rest are 0,
  highest, score = 0, 0
- output: $out := self$
- exception: none

getBoard():

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

getScore():

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

getTile(row, col):

- output: $out := board[row][col]$
- exception: none

getHighest():

- output: $out := (\exists x : TileT | x \in board \wedge \forall (y : TileT | y \in board \wedge x >= y : x))$
- exception: none

getMessage():

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

getRowValue(row):

- output: $out := [i : \mathbb{N} | i \in [0..|board| - 1] : board[row][i]]$
- exception: none

getColValue(col):

- output: $out := [i : \mathbb{N} | i \in [0..|board| - 1] : board[i][col]]$
- exception: none

setLoseMessage():

- transition:
  $message$:= A message to represent the player lost
- exception: none

setWinMessage():

- transition:
  $message$:= A message to represent the player won
- exception: none

addTile():

- transition:
  $val = generateRandomTile \land pos = generateRandomPosition$
  $\implies board[pos[0][pos[1]].setValue(val)$
- exception: none

addScore(points):

- transition:
  $score := score + points$
- exception: none

checkWin():

- output := $\exists(t : TileT | t \in board : t.getValue() = 2048)$
- exception: none

checkLose():

- output := $\forall(i, j : \mathbb{N} \mid i, j \in [0..|board| - 1] : board[i][j].getValue() \neq board[i][j + 1] \land board[i + 1][j].getValue() \neq board[i + 1][j] \land board[i][j].getValue() \neq 0 \land board[i][j].getValue() \neq 2048)$
- exception: none

Local Functions

generateRandomTile: $\mathbb{N}$
generateRandomTile() $\equiv Math.random() < 0.9 \implies 2|True \implies 4$

generateRandomPosition: seq of $\mathbb{N}$
generateRandomPosition() $\equiv \exists(i, j : \mathbb{N} \mid board[i][j] \neq 0 : [i, j])$;

TileT Module

Template Module

TileT

Uses

None

Syntax

Exported Constants

None

Exported Types

TileT = ?

Exported Access Programs

Routine name	In	Out	Exceptions
new TileT		TileT	
getValue		$\mathbb{N}$	
getPosition		seq of $\mathbb{N}$	
setValue	$\mathbb{N}$		
equal	TileT	$\mathbb{N}$	
add			

Semantics

State Variables

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

State Invariant

None

Assumptions

None

Access Routine Semantics

new TileT(val, row, col):

- transition: $value, row, col := val, row, col$
- output: $out :=$ self
- exception: none

getValue():

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

getPosition():

- output: $out := [row, col]$
- exception: none

setValue(val):

- transition: $value := val$

equal(tile):

- output := $out := value = tile.getValue()$
- exception: none

add():

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

UserInterface Module

Module

UserInterface Module

Uses

BoardT

Syntax

Exported Constants

None

Exported Types

None

Exported Access Programs

Routine name	In	Out	Exceptions
printBoard	BoardT		
printScore	BoardT		
printMessage	BoardT		

Semantics

Environment Variables

window: A computer screen to display the game and the messages.

State Variables

None

State Invariant

None

Assumptions

- UserInterface can only be called at any moment after the initialization of BoardT.

Access Routine Semantics

printBoard(board):

- transition: window := Draws the game board onto the computer window with the associated tile value. The board dimension is a 4x4 grid. Each grid cell corresponds with an index in the `board` according to the column and row on the grid. The board[x][y].getValue() is found in row x and column y. For instance, board[3][3].getValue() is the tile value displayed in the bottom right corner of the grid.

printScore(board):

- transition: Displays the current score on the computer window.

printMessage(board):

- transition: Displays the winning or losing message on the computer window.

Local Functions

Move Module (Abstract Object)

Module

Move

Uses

TileT, BoardT, Direction

Syntax

Exported Constants

None

Exported Types

None

Exported Access Programs

Routine name	In	Out	Exceptions
getOldBoard		seq of (seq [4] of $\mathbb{N}$)	
setOldBoard	seq of (seq [4] of TileT)		
restartMove		$\mathbb{B}$	
mergeLeft	BoardT		
mergeRight	BoardT		
mergeDown	BoardT		
mergeUp	BoardT		

Semantics

State Variables

old_board : seq of (seq [4] of $\mathbb{N}$)

State Invariant

None

Assumptions

None

Access Routine Semantics

getOldBoard():

- output: $out := old\_board$
- exception: none

setOldBoard(board):

- transition: $\forall(i, j : \mathbb{N} | i, j \in [0..|board| - 1] : old\_board[i][j] = board[i][j])$
- exception: none

canMove(boardT):

- output: $out := \exists(i, j : \mathbb{N} \mid i, j \in [0..|boardT.getBoard()|-1] : boardT.getBoard()[i][j] \neq old\_board[i][j])$
- exception: none

mergeLeft(boardT):

- transition: $\forall(i : \mathbb{N} \mid i \in [0..|boardT.getBoard()|-1] : slideZero(i, Direction.LEFT, boardT) \wedge \forall(j : \mathbb{N} \mid j \in [1..|boardT.getBoard()|-1] : combine(i, j, Direction.LEFT, boardT)) \wedge slideZero(i, Direction.LEFT, boardT))$
- exception: none

mergeRight(boardT):

- transition: $\forall(i : \mathbb{N} \mid i \in [0..|boardT.getBoard()|-1] : slideZero(i, Direction.RIGHT, boardT) \wedge \forall(j : \mathbb{N} \mid j \in [0..|boardT.getBoard()|-2] : combine(i, j, Direction.RIGHT, boardT)) \wedge slideZero(i, Direction.RIGHT, boardT))$
- exception: none

mergeDOWN(boardT):

- transition: $\forall(i : \mathbb{N} \mid i \in [0..|boardT.getBoard()|-1] : slideZero(j, Direction.DOWN, boardT) \wedge \forall(j : \mathbb{N} \mid j \in [0..|boardT.getBoard()|-2] : combine(j, i, Direction.DOWN, boardT)) \wedge slideZero(j, Direction.DOWN, boardT))$

- exception: none

mergeUP(boardT):

- transition: $\forall(i : \mathbb{N} \mid i \in [0..|boardT.getBoard()|-1] : slideZero(j, Direction.UP, boardT) \wedge \forall(j : \mathbb{N} \mid j \in [1..|boardT.getBoard()| - 1] : combine(j, i, Direction.UP, boardT)) \wedge slideZero(j, Direction.UP, boardT))$
- exception: none

Local Functions

slideZero: $\mathbb{N}, Direction, BoardT$
slideZero(pos, dir, boardT) $\equiv \forall(i, j : \mathbb{N} \mid i, j \in [0..|board| - 1] \wedge adj(i, j, dir).getValue() = 0 \implies adj(i, j, dir).setValue(board[i][j].getValue())$ //Assumes that rows and column are within the bounds of board.

combine: $\mathbb{N}, \mathbb{N}, Direction, BoardT$
combine(row col, dir, boardT) $\equiv board[row][col] \neq 0 \wedge board[row][col].getValue() = adj(row, col, dir).getValue() \implies adj(row, col, dir).add() \wedge board[row][col].setValue(0))$

adj: $\mathbb{N}, \mathbb{N}, Direction \to \mathbb{N}$
adj(row, col, dir) $\equiv board[row + dir.getX()][col + dir.getY()]$ // Assumes that rows and column are within the bounds of board.

Direction Module

Module

Direction

Uses

None

Syntax

Exported Constants

None

Exported Types

Direction = {
UP(-1,0),
DOWN(1,0),
LEFT(0,-1),
RIGHT(0,1)
}

Exported Access Programs

Routine name	In	Out	Exceptions
Direction	$\mathbb{N}$, $\mathbb{N}$		
getX		$\mathbb{N}$	
getY		$\mathbb{N}$	

Semantics

State Variables

$x : \mathbb{N}$
$y : \mathbb{N}$

State Invariant

None

Assumptions

None

Access Routine Semantics

Direction(xVector, yVector):

- transition: $x, y := xVector, yVector$
- exception: none

getX():

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

getY():

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

Considerations

When implementing in Java, use enums as shown in
https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html.

Controller Module

Template Module

Controller

Uses

BoardT, UserInterface, Move

Syntax

Exported Constants

None

Exported Types

Controller = ?

Exported Access Programs

Routine name	In	Out	Exceptions
Controller	BoardT, UserInterface	Controller	
readKeyInput			
moveTile	BoardT		
runGame	BoardT		

Semantics

Environment Variables

$keyboard : Scanner(System.in)$ // reads inputs from the keyboard

State Variables

$boardT : BoardT$
$UI : UserInterface$
$key : String$

State Invariant

None

Assumptions

None

Access Routine Semantics

Controller(model, view):

- transition: $boardT, UI := model, view$
- output: $out :=$ self
- exception: none

readKeyInput():

- transition: $key :=$ String of either a, w, d, s entered by the user.
- exception: none

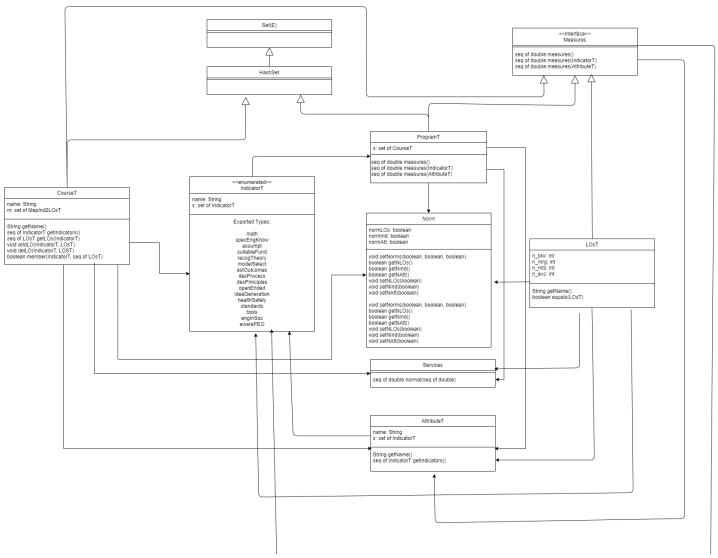
moveTile():

- transition: $key = a \implies Move.mergeLeft(boardT) \mid key = w \implies Move.mergeUp(boardT) \mid key = d \implies Move.mergeRight(boardT) \mid key = s \implies Move.mergeDown(boardT)$
- exception: none

runGame():

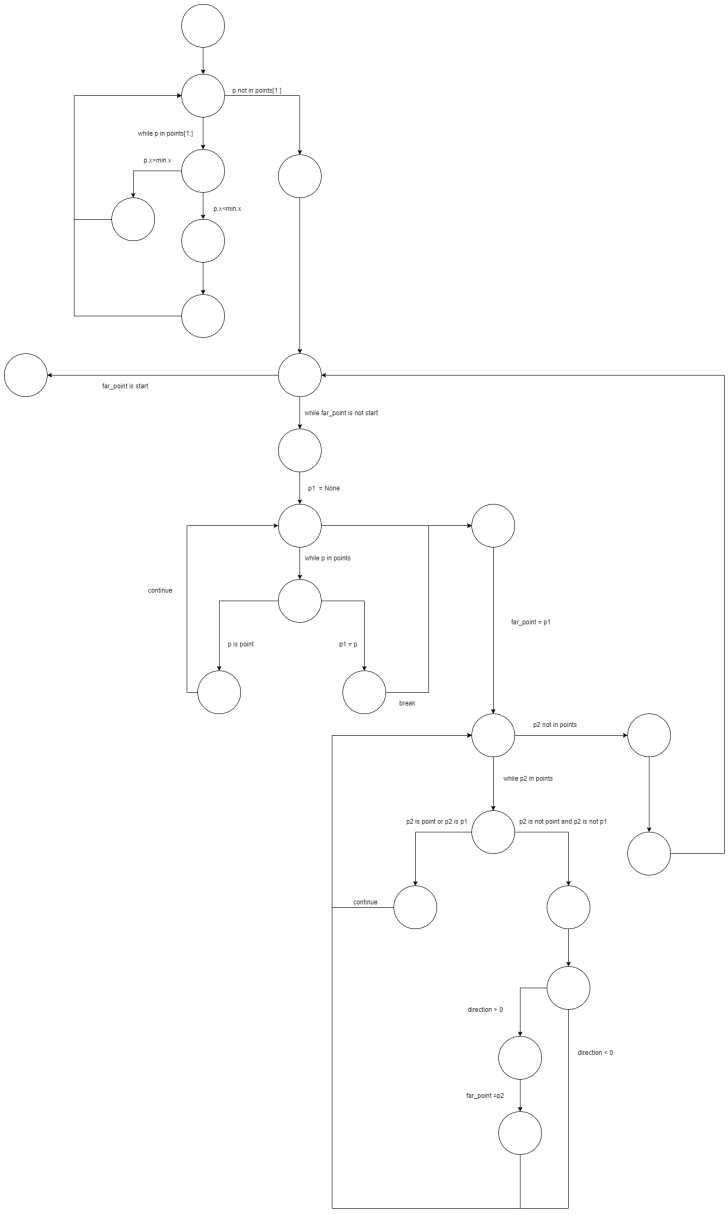
- transition: operational method for running the game. The game begins by displaying the board consisting of two tiles with values 2 or 4 and the rest, 0. This board is stored and copied by `Move.setOldBoard(boardT.getBoard())`. The user is then asked for the next move (up, down, left, right) and moves the tile when appropriate. A new tile is added when the tiles are able to move. The score is printed on the window. Eventually, the game will end, so the board is check for win or lose states using `boardT.checkWin()` and `boardT.checkLose`.

Answers to Questions:



1. Draw a UML diagram for the modules in A3

2. Draw a control flow graph for the convex hull algorithm.



Based on `compute_hull()` in
https://startupnextdoor.com/computing-convex-hull-in-python/.

Critique of Specification

Consistency is shown in the specification through the order of parameters and the consistant naming convention among each module. The ordering of each parameter is the same

for each method. For instance, the parameter `row` would always go before `col`. This is almost a standard when we use these two naming conventions together. In addition, the names of each parameter is consistant through each module. Anything named boardT would mean a `BoardT` object. Similarly, a state variable with the name `board` is the two dimentional array of tiles that is found within a `BoardT` object. Overall, the specification is consistant.

Essentiality omits any unecessary features and ensures that the service is only offered once. The specification is essential as each method does not have a similar method that provides the same service. For instance, within `Move`, `mergeLeft` cannot be achieved through `mergeRight` or any other methods withn `Move`. In `BoardT`, one can argue that getting the value of each tile within a row can be achieved by getting the board through `getBoard` and then looping through each tile within the row. However, this would be a painful and tedious process to repeat each time. Thus, having a method that returns an integer of values in a specific row, would be deemed more convenient. Overall, the access programs within the specification are essential.

Generality means open endness and anticipating changes. Due to the separation of concerns (i.e. having multiple logical modules instead of one), implementing changes to the specification and code would be easier. For instance, having the controls separate from the board would allow one to reuse the board module not just in ASCII graphics, but also with GUI implementation. In addition, using a data structure such as an array would allow the programmer to easily make changes in the future as all the data is stored in a single space. However, there are other ways to make a specification more general, but was not achieved here in this specification. For instance, instead of putting 4 when looping through the board, the design would be more general by replacing that with a variable such as *size*. *Size* can modified easily if one wants to change the size of the board. In addition, a generic module is possible to implement for certain types such as integers, but is awkward if implemented with a string, for instance.

Minimality means that each access program only does one thing. I have tried to design a specification that is minimal. For instance, `checkWin` within `BoardT` only checks if there is a tile with the value of 2048. After checking, it just returns a boolean and does not, for instance, sets the message to the winning message through `setWinMessage`. Almost all of the access programs are minimal. One that I could not make minimal was the local function `combine` in `Move`. `combine` merges tiles of the same value together. Within `combine`, it is more convenient if the scores are updated right when the tiles are merged. In a sense, it made the implementation of the specification easier, however, minimality was lost in the process.

Cohesion is an internal property of the module. The components of the modules are grouped together through logical reasoning and not by chance. The specification consists of modules that concern the board, the interface, and the control. Through MVC

design pattern, it provides separation of concerns, allowing the modules to be separated appropriately and focus on one related goal per module. Furthermore, there is high cohesion among the components of each module as the methods within the module are related to each other and logically belong in their respective module. For instance, all methods within `TileT` focuses on setting or getting a tile within the board. On the other hand, `BoardT` methods focus on changing and updating the state of the board game. For instance, checking for win or lose make sense to be in `BoardT` and not `TileT` as winning and losing concerns the state of the board and not just the tiles within the board. Thus, the high cohesion is shown through the strong relationship of the components within the module and the separation of concerns of each module.

Information hiding is characterized by the secrets that are hidden from the clients. Through the MVC design pattern, information hiding is already seen as there is a specific module for the user interface. The user interface is what the client would see and interact with. Thus, the backend logic behind how the game works is hidden and encapsulated through `BoardT`. In addition, small design details such as private variables, prevents unanticipated changes to be made within the code and ensures that it cannot be directly accessed through other modules. Therefore, information hiding is seen in the specification through MVC design pattern, separation of concerns and encapsulation.