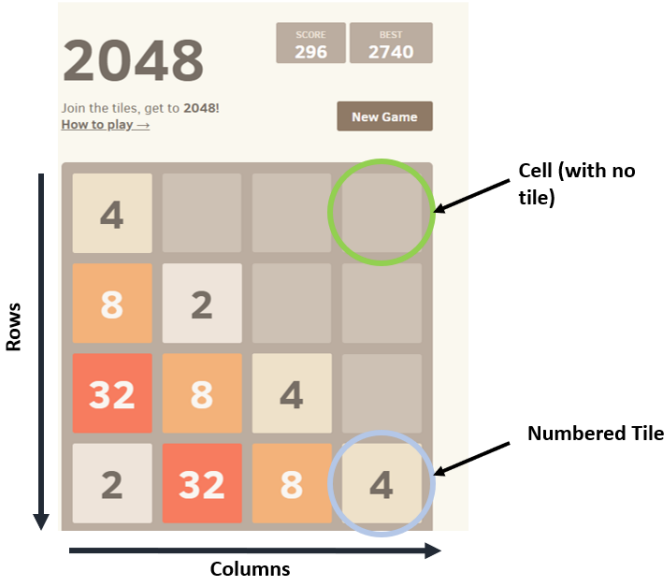
# Assignment 4 Design Specification

SFWRENG 2AA4

April 12, 2021

This Module Interface Specification (MIS) document will contain various modules, types and methods that are needed for the implementation of the game *2048*. Throughout this specification the word cell will be used to refer to space occupied by each tile in the grid. At the start of the game the user has a 4 x 4 grid with numbered tiles that will slide across the board as the user moves them with arrow keys. After every new click of the arrow, a new tile will appear in a random empty square in the board. This tile will have the value of 2 or 4. As the arrow keys move, the tiles will slide as far as is possible in the direction of the arrow key unless they are impeded by the boundary of the grid or by another tile. If two tiles of the same value collide during the sliding process, a new tile will form that is the sum of the previous tiles. Tiles can only merge in groups of two. The game is considered *finished* or *won* when the board is full and the player has no more moves or when a tile with the value *2048* is on the board, respectively.



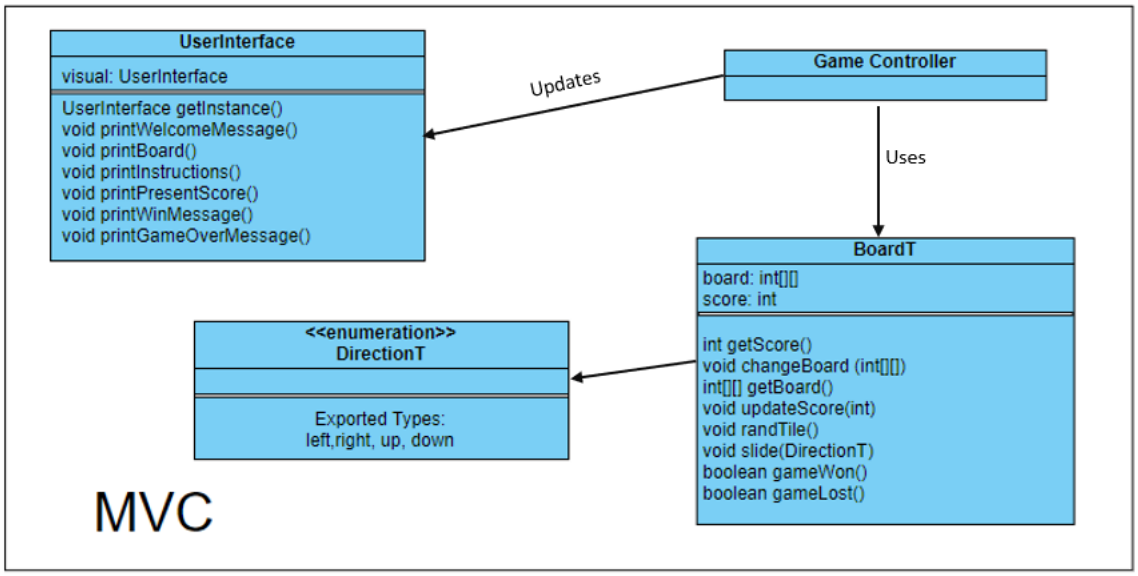
# Overview of the design

This design applies the Model-View-Controller (MVC) design pattern and Singleton design pattern. The MVC components are comprised of $BoardT$ which is the model module and $UserInterface$ which is the view module. The Singleton design pattern makes an appearance when getting the abstract object for the $UserInterface$ module.

The implementation of the MVC design pattern is as follows:

- $BoardT$ stores the state of the game board and all methods to make changes to the board throughout the game.
- $UserInterface$ displays the state of the game board using text-based graphics that are output to the screen.

Below is the UML diagram for visualizing this software architecture. The Game Controller is just for show. An actual game controller was not made as part of this MIS.




# Likely changes my design considers

- The design also considers the changing of displayed text as the user can go from win to loss to seeing the score and the board.

# DirectionT Module

## Module

DirectionT

## Uses

None

## Syntax

### Exported Constants

None

### Exported Types

DirectionT = {
right, *#Sliding to the right*
left, *#Sliding to the left*
up, *#Sliding upwards*
down, *#Sliding downwards*
} // Use enums to implement in Java

### Exported Access Programs

None

## Semantics

### State Variables

None

### State Invariant

None

### Assumptions

None

# BoardT ADT Module

## Template Module

BoardT

## Uses

Not sure yet

## Syntax

### Exported Types

None

### Exported Constants

None

### Exported Access Programs

| Routine name | In | Out | Exceptions |
|---|---|---|---|
| new BoardT | | BoardT | |
| getScore | | $\mathbb{N}$ | |
| changeBoard | seq [4, 4] of $\mathbb{N}$ | | |
| getBoard | | seq [4, 4] of $\mathbb{N}$ | |
| updateScore | $\mathbb{N}$ | | |
| randTile | | | |
| slide | DirectionT | | |
| gameWon | | $\mathbb{B}$ | |
| gameLost | | $\mathbb{B}$ | |

## Semantics

### State Variables

*board*: seq [4, 4] of $\mathbb{N}$
*score*: $\mathbb{N}$

### State Invariant

None

### Assumptions

- It is assumed that throughout the module the tile order (general order of tiles in rows and columns) remains the same. This interpretation will not change even if tiles are merged. It is assumed that the constructor is called before any other access routine and is only called once. It is also assumed that changeBoard() will only be used for testing purposes.

### Access Routine Semantics

new BoardT():

- transition: board, score := $\langle\langle 0,0,0,0\rangle, \langle 0,0,0,0\rangle, \langle 0,0,0,0\rangle, \langle 0,0,0,0\rangle\rangle$, 0
- output: *out* := *self*
- exception: None

getScore():

- output: *out* := *score*
- exception: None

changeBoard(*matrix*):

- transition: *board* := *b* # This will only be used for testing purposes.
- exception: None

getBoard():

- transition: *out* := *board*.
- exception: None

updateScore(*tile*):

- transition: *score* := *score* + *tile*

- exception: None

randTile():

- transition: $(\neg(value = random(2,4)) = 3) \wedge (i = random(0,3) \wedge j = random(0,3)) \wedge board[i][j] = 0) \Rightarrow board[i][j] := value$
  *# Check if the value of the random tile is not either 2 or 4 and that the random cell doesn't already have a tile. If these two conditions are met, place the random tile in the random cell.*

- exception: None

slide(*direction*):

- transition: $(direction = left \Rightarrow board := (slideLeft(), mergeLeft(), slideLeft())) \mid (direction = right \Rightarrow board := (slideRight(), mergeRight(), slideRight())) \mid (direction = up \Rightarrow board := (slideUp(), mergeUp(), slideUp())) \mid (direction = down \Rightarrow board := (slideDown(), mergeDown(), slideDown()))$

- exception: None

gameWon():

- output: $out := (\exists i, j : \mathbb{N} |\ i, j \in [0...3] \wedge (board[i][j] \geq 2048))$

- exception: None

gameLost():

- output: $out := \neg(\exists i, j : \mathbb{N} |\ i, j \in [0...3] : areStillMoves(i, j))$

- exception: None

## Local Functions

fullBoard $:\rightarrow \mathbb{B}$
fullBoard $\equiv \langle(\forall i, j : \mathbb{N} | i, j \in board : \neg(board[i][j] == 0))\rangle$ *#Return whether or not the board is full of tiles.*

canMerge : $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{B}$
canMerge$(i, j) \equiv \langle i, j : \mathbb{N} | i, j \in board : (board[i][j] == board[i][j+1] | board[i][j] == board[i][j-1] | board[i][j] == board[i+1][j] | board[i][j] == board[i-1][j]) \rangle$

slideLeft : # *Has no input or output, runs a state change*
slideLeft $\equiv$ *board* $:= (\forall i, j : \mathbb{N} |\ i, j \in [0...3]) \Rightarrow$ Check that the current cell does not have a tile (value of 0) and travelling to the left of the board (column index decreases), check whether there are empty spaces for the current tile to slide to the left, the slide the tiles.

slideRight : # *Has no input or output, runs a state change*
slideRight $\equiv$ *board* $:= (\forall i, j : \mathbb{N} |\ i \in [0...3] \wedge j \in [3...0]) \Rightarrow$ Check that the current cell does not have a tile (value of 0) and travelling to the right of the board (column index increasing), check whether there are empty spaces for the current tile to slide to the right, then slide the tiles.

slideUp : # *Has no input or output, runs a state change*
slideUp $\equiv$ *board* $:= (\forall i, j : \mathbb{N} |\ i, j \in [0...3]) \Rightarrow$ Check that the current cell does not have a tile (value of 0) and travelling upwards on the board (row index decreases), check whether there are empty spaces for the current tile to slide upwards, then slide the tiles.

slideDown : # *Has no input or output, runs a state change*
slideDown $\equiv$ *board* $:= (\forall i, j : \mathbb{N} |\ i \in [0...3] \wedge j \in [3...0]) \Rightarrow$ Check that the current cell does not have a tile (value of 0) and travelling downwards on the board (row index increasing), check whether there are empty spaces for the current tile to slide to downwards, then slide the tiles.

mergeLeft : # *Has no input or output, runs a state change*
mergeLeft $\equiv ((\forall i, j : \mathbb{N} |\ i, j \in [0...3]) \wedge (board[i][j] = board[i][j+1])) \Rightarrow merge(i, j, i, j+1)$

mergeRight : # *Has no input or output, runs a state change*
mergeRight $\equiv ((\forall i, j : \mathbb{N} |\ i \in [0...3] \wedge j \in [3..0]) \wedge (board[i][j] = board[i][j-1])) \Rightarrow merge(i, j, i, j-1)$

mergeUp : # *Has no input or output, runs a state change*
mergeUp $\equiv ((\forall i, j : \mathbb{N} |\ i, j \in [0...3]) \wedge (board[i][j] = board[i+1][j])) \Rightarrow merge(i, j, i+1, j)$

mergeDown : # *Has no input or output, runs a state change*
mergeDown $\equiv ((\forall i, j : \mathbb{N} |\ i \in [0...3] \wedge j \in [3..0]) \wedge (board[i][j] = board[i-1][j])) \Rightarrow merge(i, j, i-1, j)$

merge : $\mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \rightarrow$
merge$(row, col, row1, col1) \equiv \langle (board[row][col] = board[row][col] + board[row][col]) \wedge board[row1][col1] = 0 \rangle$

random : $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$
random$(num1, num2) \equiv$ outputs a random number between num1 and num2 inclusive.

# UserInterface Module

## UserInterface Module

## Uses

None

## Syntax

### Exported Types

None

### Exported Constants

None

### Exported Access Programs

| Routine name | In | Out | Exceptions |
|---|---|---|---|
| getInstance | | UserInterface | |
| printBoard | int[][] | | |
| printInstructions | | | |
| printWelcomeMessage | | | |
| printPresentScore | BoardT | | |
| printWinMessage | | | |
| printGameOverMessage | | | |

## Semantics

## Environment Variables

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

### State Variables

*visual*: UserInterface

### State Invariant

None

**Assumptions**

- The constructor is called for each object instance before any other access routine is called for that object and it is only called once.

**Access Routine Semantics**

getInstance():

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

printBoard(*board*):

- transition: window := Draws the game board onto the screen. The board is accessed using the *getBoard* method from *BoardT*. The board[x][y] is displayed so that x increases from the left of the screen to the right, and y increases 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.

printInstructions():

- transition: window := Displays the game instructions to the screen.

printWelcomeMessage():

- transition: window := Displays a message that welcomes the user.

printPresentScore(*board*):

- transition: window := Displays a message showing the user's score using board.getScore().

printWinMessage():

- transition: window := Displays a message to congratulate the user when they get the tile 2048.

printGameOverMessage():

- transition: window := Displays a message to tell the user the game is over when the 4 x 4 board is full of tiles and no further moves can be made.

# Critique of Design

- The $UserInterface$ module is specified as an abstract object as we only need a singular instance of the view to be used alongside a controller during the runtime. As such, there will be no possibility for changing state.

- The $BoardT$ module is specified as an ADT because it seems a better choice to create a new instance of BoardT when restarting or after losing a game.

- The majority of my methods are essential, they all serve a purpose and are divided so as to serve a single purpose at a time. By combining them, the modules as a whole are cohesive and complete.

- The separation of sliding and merging into methods for each direction is not considered essential. The reason I did this is for maintainability and verifiability. By separation of concerns, creating more methods allowed for me to more easily verify if the code was reliable and correct. In part the MVC design pattern also contributes to the maintainability by keeping related methods together in modules.
  Note: If there was something specifically wrong with merging or sliding tiles, it would be easy to find where the error occured.

- In terms of minimality, the specification ensures that different services are not offered by the same access program. I made sure that I did so even at the risk of essentiality at times. This is shown by the fact that every single move on the board, whether it be sliding, merging or updating score, has its own access program. This is especially prominent in the UserInterface module where all the printing methods are separated by content.

- Following off the previous point, I also ensured consistency by naming the methods that worked together similarly, as shown below:

  - slideUp(), slideLeft(), slideDown(), slideRight()
  - mergeUp(), mergeLeft(), mergeDown(), mergeRight()
  - printBoard(), printScore()...

- The MIS could have been more general in the sense that the board could have been able to work with different board size. Unfortunately, that implementation would have changed a lot of the methods I had made previously, so I had decided not to include that. However, the methods made for shifting in different directions was made general, in the sense that, had I chosen to allow for varying sizes in the board, they would be able to update easily with that change. In particular, these

methods rely on board.length which is subject to change depending on what length was specified by the code.

- As for information hiding/opacity, though the methods for making the tiles slide are public, the access programs that actually allow for the sliding and merging motions are all private e.g. (slideUp(), mergeLeft(), etc...). This enforces information hiding because the implementation information that is not necessary for the user is not made easily accessible and remains hidden in the background.

- This MIS design achieves high cohesion low coupling through the application of the MVC design pattern. The related functionalities are kept together in the model, and view parts of the design respectively. This enables high cohesion. As for low coupling, the controller would with both model and view but for the most part the implementations are independent of one another. Therefore, a change to any individual module does not severely damage any of the others.

## Answers to Questions

1. Please see the folder for UMLQ1.jpeg, the photo is too large to be implemented here. Any attempts to resize ruined the quality of the image.