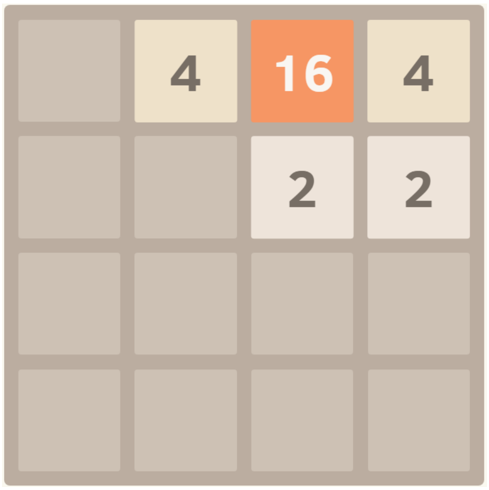
# Assignment 4, Design Specification

SFWRENG 2AA4

April 12, 2021

This Module Interface Specification (MIS) document contains modules, types and methods for implementing the game *2048*. A game commences by initializing a game board with two of the 16 tiles in the board populated by either a 2 or a 4. The user can choose whether they would like to move up by entering the number 1, move down by entering the number 2, move left by entering the number 3, or move right by entering the number 4. All of the tiles in the gameboard will shift according to the user's move. If two tiles that have the same value are moved in the appropriate direction, the two tiles will combine to form one tile, where its value will be the added value of the two equal tiles. The game can terminate in three ways. If the user wishes to exit the game, the number 0 can be entered. If there are no possible moves that the user can make to change the state of the gameboard, then the game terminates and the user has lost. If one of the tiles has a value of 2048, the game terminates and the user has won. The game can be launched and played by typing `make play` in terminal.

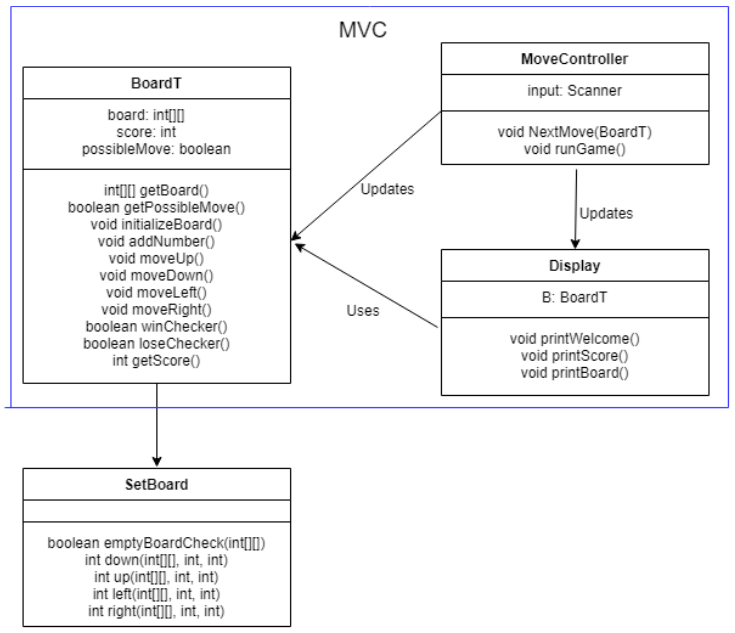


The above board is from https://play2048.co/

## 1 Overview of the design

The implemented design successfully applies the Module View Specification design pattern and applies the Module View Specification (MVC) design pattern. The MVC components are *MoveController* (controller module), *BoardT* (model module), and *Display* (view module). In addition to the MVC design pattern, a library module is used that encompasses static functions that are used by the model module for certain methods. A main function, *Play.java* allows for the game to be run with all three modules using one another as intended.

The UML diagram below provides a visualization of the structure of the designed software architecture.



The model module *BoardT* stores the state of the game board as well as its status. The view module *Display* represents the game board visually through the terminal using text-based graphics. Finally, the controller *MoveController* is responsible for handling input actions. The MVC design pattern is specified and implemented in a way such that the display and controller use an instance of BoardT, and the controller effecitvely updates the contents of the display based on the user's input. The user is given the choice to execute their next move, or exit from the game based on the properties of the *MoveController*. Additionally, specific methods of the *BoardT* determine whether the game is in a winning or losing state, in which the game is terminated as well.

## Likely Changes my design considers:

- The visual representation of the game such as a graphical user interface layout.
- Change in game modes by increasing/decreasing the dimensions of the board, or increased difficulty through different winning conditions
- Change in scanner for taking user input such as using a KeyListener
- Data structure used for storing the board
- Change variable type for inputted board

# SetBoard Module

## Template Module

SetBoard

## Uses

None

## Syntax

### Exported Constants

None

### Exported Types

None

### Exported Access Programs

| **name** | **In** | **Out** | **Exceptions** |
|---|---|---|---|
| emptyBoardCheck | seq of (seq of $\mathbb{N}$) | $\mathbb{B}$ | |
| down | seq of (seq of $\mathbb{N}$), $\mathbb{N}$, $\mathbb{N}$ | $\mathbb{N}$ | |
| up | seq of (seq of $\mathbb{N}$), $\mathbb{N}$, $\mathbb{N}$ | $\mathbb{N}$ | |
| left | seq of (seq of $\mathbb{N}$), $\mathbb{N}$, $\mathbb{N}$ | $\mathbb{N}$ | |
| right | seq of (seq of $\mathbb{N}$), $\mathbb{N}$, $\mathbb{N}$ | $\mathbb{N}$ | |

## Semantics

### State Variables

None

### State Invariant

None

### Assumptions

- Complete access to the Random java utility library is available

## Access Routine Semantics

emptyBoardCheck(board):

- output: $out$ := (Arrays.deepEquals(emptyBoard, board) $\Rightarrow$ true | True $\Rightarrow$ false) where $emptyBoard \equiv$ [[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
- exception: none

down(board, row, col):

- output: $out$ := (row $<$ 3 $\Rightarrow$ board[row+1][col] | True $\Rightarrow$ 1)
- exception: none

up(board, row, col):

- output: $out$ := (row $>$ 0 $\Rightarrow$ board[row-1][col] | True $\Rightarrow$ 1)
- exception: none

left(board, row, col):

- output: $out$ := (col $>$ 0 $\Rightarrow$ board[row][col-1] | True $\Rightarrow$ 1)
- exception: none

right(board, row, col):

- output: $out$ := (col $<$ 3 $\Rightarrow$ board[row][col+1] | True $\Rightarrow$ 1)
- exception: none

## Local Functions

Arrays.deepEquals: seq of (seq of $\mathbb{N}$)$\times$ seq of (seq of $\mathbb{N}$) $\rightarrow \mathbb{B}$

Array.deepEquals$(a, b) \equiv (\forall x \cdot (\forall y \cdot a_{xy} = b_{xy}))$

# Board ADT Module

## Template Module

BoardT

## Uses

SetBoard

## Syntax

### Exported Types

None

### Exported Constant

SIZE = 4 // Size of the board which is assumed to always be 4 x 4

### Exported Access Programs

| **Routine name** | **In** | **Out** | **Exceptions** |
|---|---|---|---|
| BoardT | seq of (seq of $\mathbb{N}$) | BoardT | |
| getBoard | | seq of (seq of $\mathbb{N}$) | |
| getPossibleMove | | seq of (seq of $\mathbb{B}$) | |
| initializeBoard | | | |
| addNumber | | | |
| moveUp | | | |
| moveDown | | | |
| moveLeft | | | |
| moveRight | | | |
| winChecker | | $\mathbb{B}$ | |
| loseChecker | | $\mathbb{B}$ | |
| getScore | | $\mathbb{N}$ | |

## Semantics

### State Variables

board: sequence [Size, Size] of $\mathbb{N}$
score: $\mathbb{N}$
possibleMove: $\mathbb{B}$

### 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.
- Complete access to the Random java utility library is available
- The addNumber method will always be called following a possible move

### Access Routine Semantics

BoardT($board$):

- transition: board, score, possibleMove = $board$, 0, $false$
- output: $out$ := $self$
- exception: None

getBoard():

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

getPossibleMove():

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

InitializeBoard():

- transition:

```
if(SetBoard.emptyBoardCheck(this.board)) {
        addNumber();
        addNumber();
}
```

addNumber():

- transition:

```
String numstring = "4222222222"
for(int i = 0; i < SIZE; i++) {
    for(int j = 0; j < SIZE; j++) {
        if(this.board[i][j] == 0) {
            count++;
        }
    }
}
String[] storeFreeBlocks = new String[count];
int localcounter = 0;
for(int i = 0; i < SIZE; i++) {
    for(int j = 0; j < SIZE; j++) {
        if(this.board[i][j] == 0) {
            storeFreeBlocks[localcounter] = "i, j";
            localcounter++;
        }
    }
}
int[] getRowCol = storeFreeBlocks[randomNum]
int a = getRowCol[0];
int b = getRowCol[1];
this.board[a][b] = Integer.parseInt(numstring);
```

moveUp():

- transition: All values within *this.board* that are not zero must be shifted upwards if there is a zero above it. If the number directly above the selected cell is equal to the selected cell itself, and is not equal to zero, the cell above becomes double this amount, and the cell below becomes zero. Once any necessary combines have occured, a second shit of the cells upwards must be done to clean up the board following the combine.

moveDown():

- transition: All values within *this.board* that are not zero must be shifted downwards if there is a zero below it. If the number directly below the selected cell is equal to the selected cell itself, and is not equal to zero, the cell below becomes double this amount, and the cell above becomes zero. Once any necessary combines have occured, a second shit of the cells downwards must be done to clean up the board following the combine.

moveLeft():

- transition: All values within *this.board* that are not zero must be shifted leftwards if there is a zero to the left of it. If the number directly to the left of the selected cell is equal to the selected cell itself, and is not equal to zero, the cell to the left becomes double this amount, and the cell to the right becomes zero. Once any necessary combines have occured, a second shit of the cells leftwards must be done to clean up the board following the combine.

moveRight():

- transition: All values within *this.board* that are not zero must be shifted rightwards if there is a zero to the right of it. If the number directly to the right of the selected cell is equal to the selected cell itself, and is not equal to zero, the cell to the right becomes double this amount, and the cell to the left becomes zero. Once any necessary combines have occured, a second shit of the cells rightwards must be done to clean up the board following the combine.

winChecker():

- transition: none
- output: $out := (\forall x \cdot (\forall y \cdot (board_{xy} = 2048) \Rightarrow \text{true} \mid \text{True} \Rightarrow \text{false}))$
- exception: None

loseChecker():

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

```
      for(int i = 0; i < SIZE; i++) {
          for(int j = 0; j < SIZE; j++) {
              if(this.board[i][j] == 0) {
                  return false;
              }
              if (i > 0 && this.board[i-1][j] == this.board[i][j]) {
                return false;
              }
              if (i < SIZE-1 && this.board[i+1][j] == this.board[i][j]) {
                return false;
              }

              if (j > 0 && this.board[i][j-1] == this.board[i][j]) {
                return false;
              }

              if (j < SIZE-1 && this.board[i][j+1] == this.board[i][j]) {
                return false;
              }
          }
      }
      return true;
  }
```

- exception: None

getScore():

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

# Display Module

## Display Module

## Uses

BoardT, Move

## Syntax

### Exported Types

None

### Exported Constants

BOARD = [[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]] // Empty Game Board

### Exported Access Programs

| Routine name | In | Out | Exceptions |
|---|---|---|---|
| printWelcome | | | |
| printScore | | | |
| printBoard | | | |

## Semantics

## Environment Variables

window: A portion of the computer screen, specifically the terminal, to display the game and messages.

### State Variables

B: BoardT

### State Invariant

None

**Assumptions**

- The game board used only consists of numbers that are powers of 2.
- The terminal window is large enough to display the dimensions of the game board, the score, and the prompt for the user's next move

**Access Routine Semantics**

printWelcome():

- transition: window := Displays a welcome message when user first enter the game along with instructions for possible next moves.

printScore():

- transition: window := Prints out the updated score after every turn is made

printBoard(*board*):

- transition: window := Draws the game board onto the screen. The board is first initialized through the *B.initializeBoard* method if the board is empty. Then the board is formatted using the *B.boardFormatter* method, so that the user sees a 4x4 grid populated by numbers that are an exponent of 2. The current state of the board is verified to see if the user has lost through the *B.loseChecker* method, or if the user has won through the *B.winChecker* method. Then, once the user has inputted their desired next move, the *Move.nextMove* method will alter the board to reflect the next move made by the user.

# MoveController Module

## MoveController Module

## Uses

BoardT, Display

## Syntax

### Exported Types

None

### Exported Constants

None

### Exported Access Programs

| Routine name | In | Out | Exceptions |
|---|---|---|---|
| nextMove | BoardT | | |
| nextMove | | | |

## Semantics

## Environment Variables

input: Scanner(System.in) *// Stores user input representing the desired move.*

### State Variables

None

### State Invariant

None

**Assumptions**

- The user only inputs a number of type $\mathbb{N}$ to reflect their desired turn
- The user does not wish to immediately play another game once the game baord is deemed to either be in a winning or losing state. The game must be re-run if the game is terminated.

**Access Routine Semantics**

nextMove($B$):

- transition: (choice = 0) $\Rightarrow$ System.exit(0); |
  (choice = 1) $\Rightarrow$ B.moveUp() $\wedge$ B.getPossibleMove $\Rightarrow$ B.addNumber() |
  (choice = 2) $\Rightarrow$ B.moveDown() $\wedge$ B.getPossibleMove $\Rightarrow$ B.addNumber() |
  (choice = 3) $\Rightarrow$ B.moveLeft() $\wedge$ B.getPossibleMove $\Rightarrow$ B.addNumber() |
  (choice = 4) $\Rightarrow$ B.moveRight() $\wedge$ B.getPossibleMove $\Rightarrow$ B.addNumber() |
  True $\Rightarrow$ System.out.println("Move not possible, try again!")
  where *choice* $\equiv$ input.nextInt()
- exception: none

runGame():

- transition: operational method for running the game. The game will start with a welcome message as well as explanations surrounding the rules. Every turn, the score and board are printed and updated based on the user's turn.
- output: None

**Local Function:**

nextInt: void $\rightarrow \mathbb{Q}$
nextInt() $\equiv$ The user input stored as type integer.

## Critique of Design

- The BoardT module, representing the model, is implemented as an abstract data type over an abstract object, because It is more convenient to create a new instance of the board after the user chooses to restart the game. Additionally, the state of the score can be saved and added to following every move through the state variable *this.score*.

- By defining the BoardT constructor with a single argument that takes in a 2D array of integers that represents the starting state of the gameboard, the model can be initialized with an empty game board, or any desired game board state for testing purposes.

- Essentiality was upheld as a crucial characteristic of the design, as a majority of the functions play a necessary role for creating and updating the game. The display uses print statements, instead of defining void methods that print out these statements to avoid implementing functions that have an output that doesn't change regardless of how the function is called. These print statements are not necessarily required for the implementation of the game, however they were kept in to improve the quality of the view module.

- Minimality was upheld by ensuring that all access routines, especially within the Model module, are not independent of one another. Any methods that may have went against the minimality requirement were included as a static method in *SetBoard.java*.

- Generality was sacrificed in order to fit the requirements of the game. The modules should perform exactly how the 2048 game performs, and for this reason there is no reason to keep generality in mind. However, the random function used to populate an empty cell with either a 2 or 4 when a move is made does allow for generality, as it is impossible to predict the state of the game board after a series of moves.

- The controller and view modules could have been specified as a single abstract object since the have shared resources with one another. This would have allowed for only one instance to be required to control the action during runtime to avoid any unexpected state changes. However, because there were no additional state conditions required to define the controller or view, this was not implemented to increase the efficiency of the development process.

- The test cases are designed to validate the correctness of the program based on the requirements of the 2048 game. The test cases should reveal any errors or unusual behaviour that don't match the rules of the game. Every access routine in the Model module BoardT has at least one test case, and multiple instances of BoardT's are created that represent many different states in the game. One exception is made for the *getPossibleMove* method in BoardT because its purpose is to obtain whether or not a move is possible once it has actually been performed. Therefore its effectiveness is tested in the *moveUp*, *moveDown*, *moveLeft* and *moveRight* methods.

- No test cases were built for testing the controller module since the implementation of the controller's access methods uses methods from the model and view. The test cases for the model are in *TestBoardT.java* Due to the nature of the game, where a random number populates an empty cell following a possible move, the addNumber functionality was tested separately to ensure that a random number was in fact being inserted into the game baord only when a valid move was made. This prevented test cases where multiple turns were conducted right after one another, as it would be difficult to locate the random populated cell after each turn.

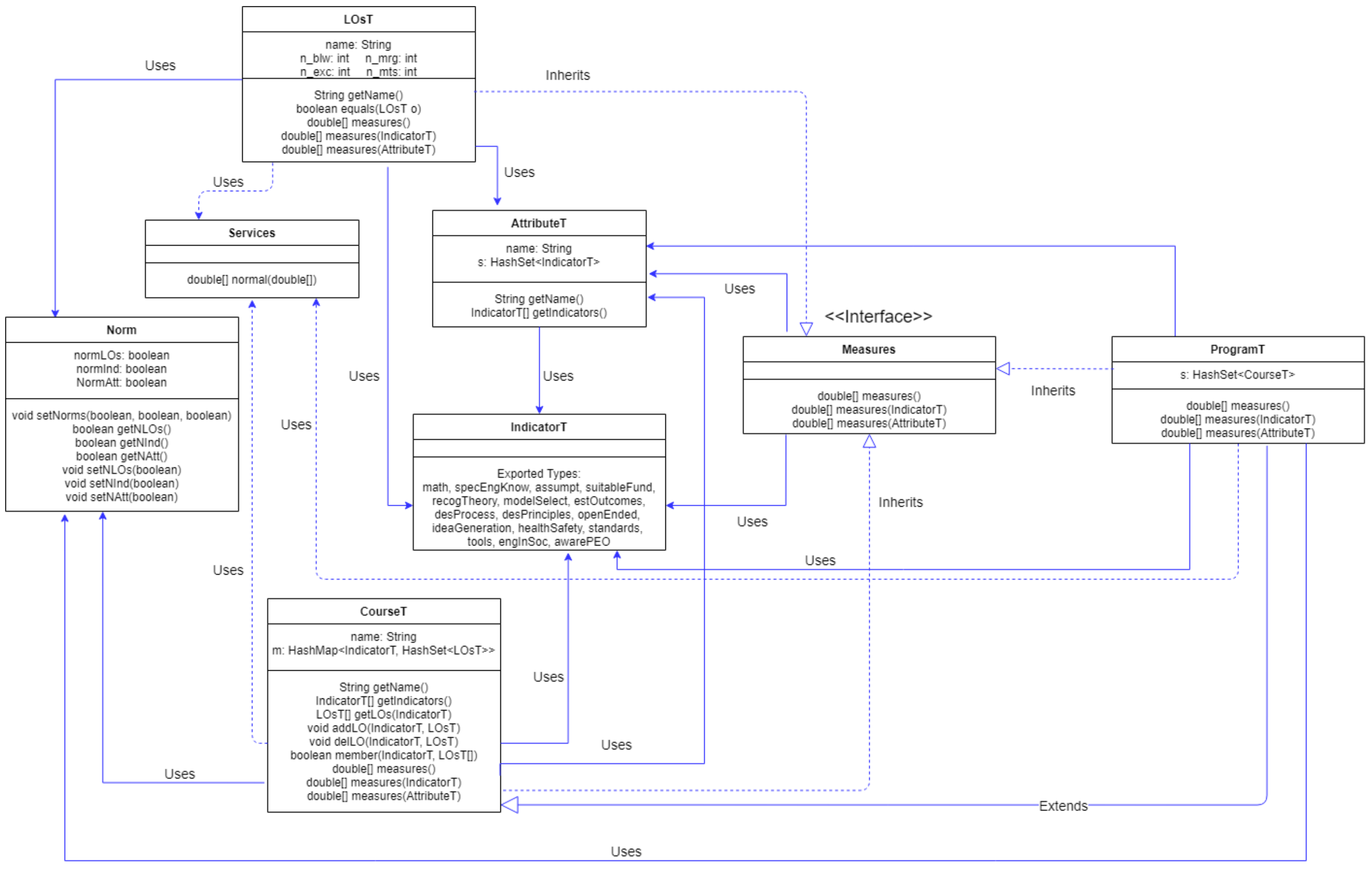
- The MVC design pattern improves maintainability as risks are reduced when making changes to either of the 3 modules. The modules are decomposed into three components based on the separation of concerns ideology, where the model will hold the data and status of the board, the display prints out the state of the board, and the controlled effectively handles the user's inputs that will execute the appropriate actions to reflect the user's input.

- The MVC design effectively achieves high cohesion and low coupling. High cohesion is enforced by grouping related functionalities within each module. All functionality that relates to the appearance of the board is maintained within the View module, and the same can be said for the Model and Controller modules. Low coupling is also achieved because each model is, for the most part, independent of one another. A change in one module doesn't necessarily impact one another directly, however because the controller is responsible for updating the module and view, there are implications for changes to this code.

- Static methods were used for the abstract objects for ease of utilizing the methods to aid in the model module. However, the singleton design pattern could be implemented to avoid warnings about variances in static accesses, to improve the development of the code.

## Answers to Questions:

Q1: Draw a UML diagram for the modules in A3.


LOsT
name: String
n_blw: int n_mrg: int
n_exc: int n_mts: int
String getName()
boolean equals(LOsT o)
double[] measures()
double[] measures(IndicatorT)
double[] measures(AttributeT)
Services
double[] normal(double[])
AttributeT
name: String
s: HashSet<IndicatorT>
String getName()
IndicatorT[] getIndicators()
<<Interface>>
Measures
double[] measures()
double[] measures(IndicatorT)
double[] measures(AttributeT)
ProgramT
s: HashSet<CourseT>
double[] measures()
double[] measures(IndicatorT)
double[] measures(AttributeT)
Norm
normLOs: boolean
normInd: boolean
NormAtt: boolean
void setNorms(boolean, boolean, boolean)
boolean getNLOs()
boolean getNInd()
boolean getNAtt()
void setNLOs(boolean)
void setNInd(boolean)
void setNAtt(boolean)
IndicatorT
Exported Types:
math, specEngKnow, assumpt, suitableFund, recogTheory, modelSelect, estOutcomes, desProcess, desPrinciples, openEnded, ideaGeneration, healthSafety, standards, tools, engInSoc, awarePEO
CourseT
name: String
m: HashMap<IndicatorT, HashSet<LOsT>>
String getName()
IndicatorT[] getIndicators()
LOsT[] getLOs(IndicatorT)
void addLO(IndicatorT, LOsT)
void delLO(IndicatorT, LOsT)
boolean member(IndicatorT, LOsT[])
double[] measures()
double[] measures(IndicatorT)
double[] measures(AttributeT)
Uses
Inherits
Extends

Q2: Draw a control flow graph for the convex hull algorithm.