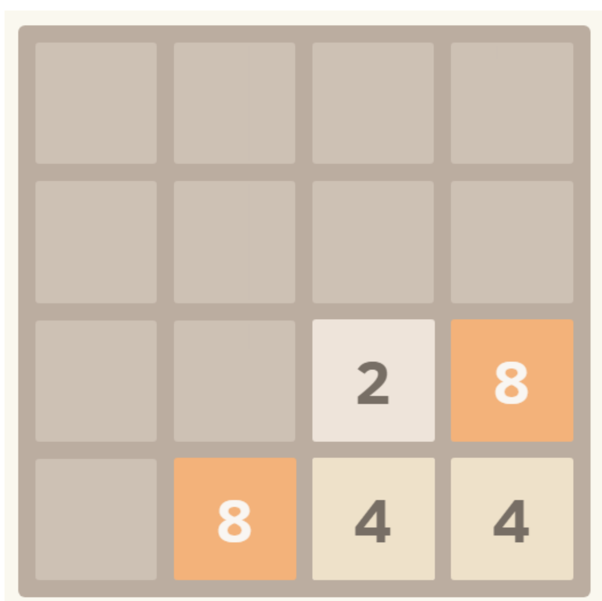
Assignment 4, Design Specification

SFWRENG 2AA4

April 11, 2021

The following Module Interface Specification outlines the necessary modules and their respective access routine programs, state variables, and local functions required for implementing the popular web/mobile game 2048. Players manipulate a 4x4 board that consists of randomly spawned tiles, and combine tiles with equal values into greater powers of two, until either the 2048 tile is reached or the player runs out of possible moves.

An example of the game being modelled in this design specification can be found on the website https://play2048.co/



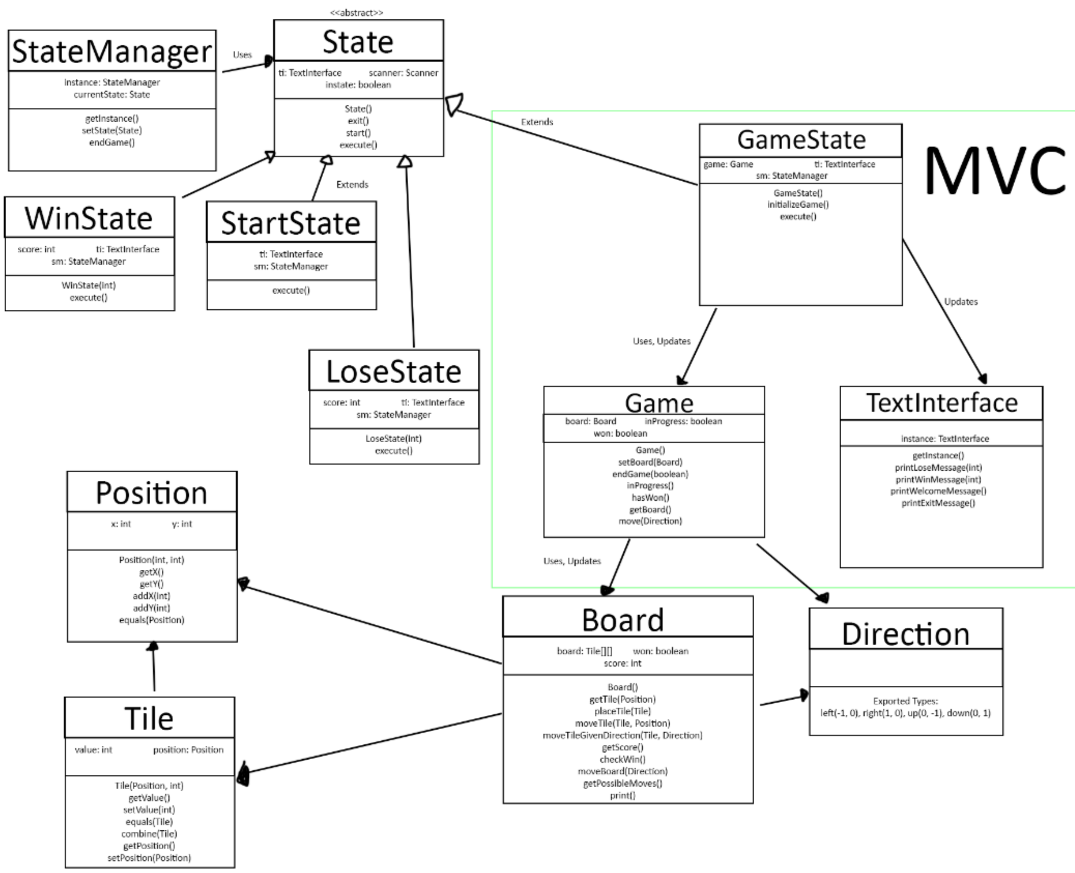
Changes my design considers or design changes in the future:

- Using Java UI libraries to display the board in a more user friendly way.
- Further generalize the board, make it more modular (i.e. allowing the ability to change dimension of the board during runtime).
- Take in user input via keyboard or mouse, rather than typing the actual direction the user wants the board to move.

1 Informal Design Overview

The design outlined in this specification adheres to the MVC design. The model, in this case, is the Game module itself. The Game module has direct access to the Board module and is managed by it. The module is directly updated and modified by the controller, and all aspects of the game is managed by this module. The view portion of the MVC design is the TextInterface module, which is accessible by the module's `getInstance()` function, and is updated by the controller and prints out any necessary messages to the user, including, but not limited to, the end messages and the end score. Finally, the controller is the GameState module, which inherits a State abstract module. Modules that inherit State repeatedly run its `execute` function until the State child exits the state, which serves as the basis for the controller's functionality. In terms of the GameState module, a child of the State abstract module, the GameState's `execute` functionality consists of taking in user input, as well as manipulating the Game module accordingly, depending on the user input. It checks repeatedly if the game is in progress, and if it is not, it exits the state and enters the `WinState` or `LoseState` accordingly.

To further visualize how the program runs and behaves, a UML diagram can be seen below:



As can be seen in the UML diagram, the MVC consists of the Game (model), TextInterface (view), and GameState (controller). The State system can be seen in the top left, with the WinState, StartState, and LoseState all being children of the State abstract module. The Game module manipulates the Board module, updating it depending on the user input that is provided by the controller GameState module.

Direction Module

Module

Direction

Uses

N/A

Syntax

Exported Constants

None

Exported Types

Direction = $\{$left, right, up, down$\}$

Exported Access Programs

None

Semantics

State Variables

None

State Invariant

None

Position ADT Module

Template Module

Position

Uses

None

Syntax

Exported Types

None

Exported Access Programs

Routine name	In	Out	Exceptions
Position	$\mathbb{N}, \mathbb{N}$	Position	
getX		$\mathbb{N}$	
getY		$\mathbb{N}$	
addX	$\mathbb{N}$		
addY	$\mathbb{N}$		
equals	Position	$\mathbb{B}$	

Semantics

State Variables

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

State Invariant

$x >= 0 \wedge x <= 3 \wedge y >= 0 \wedge y <= 3$

Assumptions

None

Access Routine Semantics

Position($xPos, yPos$):

- transition: x, y := $xPos$, $yPos$
- output: out := *self*

getX():

- transition: none
- output: out := x

getY():

- transition: none
- output: out := y

addX($xPos$):

- transition: x := x + $xPos$
- output: none

addY($yPos$):

- transition: y := y + $yPos$
- output: none

equals(p):

- transition: none
- output: out := (getX() = p.getX() $\wedge$ getY() = p.getY())

Local Functions

constrainXPos: void $\rightarrow$ void
constrainXPos() $\equiv ((x < 0) \Rightarrow x = 0) \wedge ((x > 3) \Rightarrow x = 3)$

constrainYPos: void $\rightarrow$ void
constrainYPos() $\equiv ((y < 0) \Rightarrow y = 0) \wedge ((y > 3) \Rightarrow y = 3)$

TextInterface Module

Module

UserInterface

Uses

None

Syntax

Exported Constants

None

Exported Types

None

Exported Access Programs

Routine name	In	Out	Exceptions
getInstance		TextInterface	
printLoseMessage	$\mathbb{N}$		
printWinMessage	$\mathbb{N}$		
printWelcomeMessage			
printExitMessage			

Semantics

Environment Variables

window: Section of the computer screen to display the text interface.

State Variables

instance: TextInterface

State Invariant

None

Assumptions

- Assume that each subroutine is called after the constructor has been called. The constructor can only be called once.

Access Routine Semantics

getInstance():

- transition: instance := (instance = null $\Rightarrow$ new TextInterface())
- output: *self*
- exception: None

printLoseMessage(*score*):

- transition: window := Print a message to the screen when the user loses the game, which displays the total score the user achieved.

printWinMessage(*score*):

- transition: window := Print a message to the screen when the user wins the game, which displays the total score the user achieved.

printWelcomeMessage():

- transition: window := Prints a welcome message when the user first launches the game.

printExitMessage():

- transition: window := Prints an exit message when the user exits the game.

Local Function:

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

Tile ADT Module

Template Module

Tile

Uses

None

Syntax

Exported Types

None

Exported Access Programs

Routine name	In	Out	Exceptions
Tile	$Position, \mathbb{N}$	Tile	
getValue		$\mathbb{N}$	
setValue	$\mathbb{N}$		
equals	Tile	$\mathbb{B}$	
getPosition		Position	
setPosition	Position		

Semantics

State Variables

value: $\mathbb{N}$
position: Position

State Invariant

None

Assumptions

None

Access Routine Semantics

Tile(p, val):

- transition: value, position := val, p
- output: out := $self$
- exception: None

getValue():

- transition: none
- output: out := value

setValue(v):

- transition: value := v
- output: none

equals($tile$):

- transition: none
- output: out := value $\equiv$ tile.getValue()

getPosition():

- transition: none
- output: out := position

setPosition(newPos):

- transition: position := $newPos$
- output: none

State Abstract Module

Abstract Module

State

Uses

TextInterface, Scanner

Syntax

Exported Constants

None

Exported Types

None

Exported Access Programs

Routine name	In	Out	Exceptions
State		State	
exit			
start			
execute			

Semantics

Environment Variables

None

State Variables

ti: TextInterface
scanner: Scanner
instate: $\mathbb{B}$

State Invariant

None

Assumptions

None

Access Routine Semantics

new State():

- transition: ti, scanner, instate := TextInterface.getInstance(), new Scanner(System.in), true
- output: *self*
- exception: None

exit():

- transition: instate := false

start():

- transition: operational method that continuously runs the 'execute' abstract method until *instate* state variable equals false.

execute():
This is an abstract function that is inherited by State children. As mentioned in the start() method semantics, this function will continuously update as long as the state is active.

Local Function:

None

StateManager Module

Template Module

StateManager

Uses

State, TextInterface

Syntax

Exported Types

None

Exported Constant

None

Exported Access Programs

Routine name	In	Out	Exceptions
getInstance		StateManager	
setState	State		
endGame			

Semantics

State Variables

currentState: State
instance: StateManager
view: Area of window that displays text.

State Invariant

None

Access Routine Semantics

getInstance():

- transition: instance := (instance = null) $\Rightarrow$ new StateManager()
- output: *out* := instance
- exception: none

setState(state):

- transition: currentState := state$|$True $\Rightarrow$ currentState.start()
- output: none
- exception: none

endGame():

- transition: currentState, view := null, TextInterface.getInstance().printExitMessage()
- output: none
- exception: none

Board ADT Module

Template Module

Board, Direction, Position

Uses

Tile

Syntax

Exported Types

None

Exported Constant

None

Exported Access Programs

Routine name	**In**	**Out**	**Exceptions**
Board		Board	
getTile	Position	Tile	
placeTile	Tile		
moveTile	Tile, Position		
moveTileGivenDirection	Tile, Direction		
moveBoard	Direction		
getScore		$\mathbb{N}$	
checkWin		$\mathbb{B}$	
getPossibleMoves		$\mathbb{N}$	
print			

Semantics

Environment Variables

window: Section of the computer screen to display the game board.

State Variables

board: sequence [4, 4] of Tile
won: $\mathbb{B}$
score: $\mathbb{N}$

State Invariant

None

Assumptions

- Assume there is a random function that randomly generates a piece onto the board.

Design decision

Coordinates of the board is represented as a 2D sequence. Each index of this sequence contains a Tile object.

In the above illustration, cell at (0,0) is stored in board[0][7].

- The reasoning for this choice of data structure is how easy it is to get a tile at a specific point in the board. And since 2048 is essentially a two dimensional grid, it was obvious to use a two dimensional array to represent the game board.

Access Routine Semantics

Board():

- transition:
  board := new Tile[4][4]
  generateRandomPiece(2)
- output: *out* := *self*
- exception: None

getTile(p):

- transition: none
- output: *out* := board[p.getX()][p.getY()]

- exception: None

placeTile(t):

- transition: board := $\{$tile : board$|tile \in board:$ (tile.getX() = t.getX() $\wedge$ tile.getY() = t.getY()) $\Rightarrow$ (tile = t)$\}$
- output: none
- exception: None

moveTile(tile, p):

- transition: board := $\{$t : board$|t \in board:$ (tile.getX() = t.getX() $\wedge$ tile.getY() = t.getY()) $\Rightarrow$ (t = null $\wedge$ placeTile(new Tile(p, tile.getValue())))$\}$
- output: none
- exception: none

moveTileGivenDirection(t, d):

- transition:
  board := ((getTile(newPosition) $\neq$ null $\wedge$ tile $\neq$ t $\wedge$ t.equals(tile)) $\Rightarrow$ t.combine(tile) $\wedge$ score += t.getValue() $\wedge$ (t.getValue() = 2048 $\Rightarrow$ won := true))$|$True $\Rightarrow$ (moveTile(t, newPosition) $\wedge$ t.setPosition(newPosition))

  where $newPosition \equiv$ calculateFinalTilePosition(t, d) and $tile \equiv$ getTile(newPosition)

moveBoard(d):

- transition: $\{$iterative : board$|iterative \in board$:$\{$t : iterative$|$ $t \in iterative$: ((t $\neq$ null $\wedge$ canTileMove(t, d)) $\Rightarrow$ moveTileGivenDirection(t, d))$\}\}$

getScore():

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

checkWin():

- transition: none

- output: out := won
- exception: none

getPossibleMoves():

- transition: none
- output: $+(t : \text{Tile}|t \in board : (t \neq null) \Rightarrow$ getMovesForTile(t)))
- exception: none

print():

- transition: window := Prints a display of the board with its corresponding tiles and current score of the game.

Local Functions

getMovesForTile: Tile $\rightarrow \mathbb{N}$
getMovesForTile(t) $\equiv$ ((getTile(left) = null $\vee$ getTile(left).equals(t))$\Rightarrow$ 1|True$\Rightarrow$0) + ((getTile(right) = null $\vee$ getTile(right).equals(t))$\Rightarrow$ 1|True$\Rightarrow$0) + ((getTile(up) = null $\vee$ getTile(up).equals(t))$\Rightarrow$1|True$\Rightarrow$0) + ((getTile(down) = null $\vee$ getTile(down).equals(t))$\Rightarrow$ 1|True$\Rightarrow$0)
where :
left $\equiv$ new Position(t.getPosition().getX()-1, t.getPosition().getY())
right $\equiv$ new Position(t.getPosition().getX()+1, t.getPosition().getY())
up $\equiv$ new Position(t.getPosition().getX(), t.getPosition().getY()-1)
down $\equiv$ new Position(t.getPosition().getX(), t.getPosition().getY()+1)

getTilesInColumn: $\mathbb{N} \rightarrow$ seq of Tile
getTilesInColumn(column): (column $< 0 \vee$ column $> 3) \Rightarrow$ IndexOutOfBoundsException|True $\Rightarrow$ board[column]

getTilesInRow: $\mathbb{N} \rightarrow$ seq of Tile
getTilesInRow(row): ((row $< 0 \vee$ row $> 3) \Rightarrow$ IndexOutOfBoundsException)|True $\Rightarrow$ [board[0][row], board[1][row], board[2][row], board[3][row]]

calculateFinalPosition: Tile $\times$ Direction $\rightarrow$ Position
calculateFinalPosition(t, d) $\equiv$ new Position(x, y)
where:
x $\equiv ((d.getX() = 0) \Rightarrow t.getPosition().getX())|\text{True} \Rightarrow t.getX() + d.getX()$

y $\equiv ((d.getY() = 0) \Rightarrow t.getPosition().getY())|$True $\Rightarrow t.getY() + d.getY()$
x and y coordinates update until a tile is reached or the edge of the board is reached.

canTileMove: $Tile \times Direction \rightarrow \mathbb{N}$
canTileMove(t, d) $\equiv \neg$(calculateFinalPosition(t, d).equals(t.getPosition()))

getEmptyCoordinates: $void \rightarrow$ seq of seq of int
getEmptyCoordinates() $\equiv \cup(t$:Tile$|t \in board :$ (t $\neq$ null) $\Rightarrow$ {t.getPosition().getX(), t.getPosition().getY()}|True $\Rightarrow$)

generateRandomPiece: $\mathbb{N} \rightarrow void$
generateRandomPiece(numPieces): $((numPieces < 0) \Rightarrow$ ArithmeticException|True $\Rightarrow$ (placeTile(new Tile(new Position(random(getEmptyCoordinates())), 2))))

Game ADT Module

Template Module

Game

Uses

Board

Syntax

Exported Types

None

Exported Constant

None

Exported Access Programs

Routine name	**In**	**Out**	**Exceptions**
Game		Game	
setBoard	Board		
getBoard		Board	
endGame	$\mathbb{B}$		
inProgress		$\mathbb{B}$	
hasWon		$\mathbb{B}$	RuntimeException
move	Direction		

Semantics

State Variables

board: Board
inProgress: $\mathbb{B}$
won: $\mathbb{B}$

State Invariant

None

Access Routine Semantics

new Game():

- transition: inProgress, won := true, false
- output: $out := self$
- exception: none

setBoard(b):

- transition: board := b
- output: none
- exception: none

getBoard():

- transition: none
- output: board
- exception: none

endGame(w):

- transition: inProgress, won := false, w
- output: none
- exception: none

inProgress():

- transition: none
- output: inProgress
- exception: none

hasWon():

- transition: none
- output: won
- exception: $exc :=$ inProgress $\Rightarrow$ RuntimeException

move(d):

- transition: board, game := board.moveBoard(d), ((board.checkWin()) $\Rightarrow$ endGame(true)) $\wedge$ ((board.getPossibleMoves() $= 0$) $\Rightarrow$ endGame(false))
- output: none
- exception: none

StartState Module

Template Module inerhits State Abstract Class

StartState

Uses

State

Syntax

Exported Constants

None

Exported Types

None

Exported Access Programs

Routine name	In	Out	Exceptions
execute			

Semantics

State Variables

Any state variable inherited from State module
window: Section of the computer screen that displays text.

State Invariant

None

Assumptions

None

Access Routine Semantics

execute():

- transition: This operational method displays the welcome message and asks for user input for the game to start.
- output: None

Local Function

None

LoseState Module

Template Module inerhits State Abstract Class

LoseState

Uses

State, TextInterface

Syntax

Exported Constants

None

Exported Types

None

Exported Access Programs

Routine name	In	Out	Exceptions
LoseState	$\mathbb{N}$	LoseState	
execute			

Semantics

State Variables

Any state variable inherited from State module
window: Section of the computer screen that displays text. score: $\mathbb{N}$

State Invariant

None

Assumptions

None

Access Routine Semantics

new LoseState(s):

- transition: score := s

execute():

- transition: This operational method displays the 'lose' message from the TextInterface class. It waits for user input, starting a new game depending on the input that the user provides.
- output: None

Local Function

None

WinState Module

Template Module inerhits State Abstract Class

WinState

Uses

State, TextInterface

Syntax

Exported Constants

None

Exported Types

None

Exported Access Programs

Routine name	In	Out	Exceptions
WinState	$\mathbb{N}$	WinState	
execute			

Semantics

State Variables

Any state variable inherited from State module
window: Section of the computer screen that displays text.
score: $\mathbb{N}$

State Invariant

None

Assumptions

None

Access Routine Semantics

new WinState(s):

- transition: score := s

execute():

- transition: This operational method displays the 'win' message from the TextInterface class. It waits for user input, starting a new game depending on the input that the user provides.
- output: None

Local Function

None

GameState Module

Template Module inerhits State Abstract Class

GameState

Uses

State, WinState, LoseState, TextInterface, Direction

Syntax

Exported Constants

None

Exported Types

None

Exported Access Programs

Routine name	In	Out	Exceptions
GameState		GameState	
execute			

Semantics

State Variables

Any state variable inherited from State module
window: Section of the computer screen that displays text.
game: Game

State Invariant

None

Assumptions

None

Access Routine Semantics

new GameState():

- transition: game := new Game()
- game := game.setBoard(new Board())

execute():

- transition: This operational method utilizes the Game abstract data type and takes user input, updating the game board accordingly depending on the direction inputted. Furthermore, the method checks to see if the game is in progress, if not, changes state to either the WinState or LoseState accordingly.
- output: None
- exception: $exc :=$ direction invalid $\Rightarrow$ IllegalArgumentException

Local Function

None

Critique/Overview of my design

- The State abstract object and the StateManager module serves as a way for new states of the game to be added with ease. For example, different win states and lose states could easily be added depending on the score the user achieves (i.e. high scores) without cluttering the overall design of the controller. Menus can be added to further improve user experience. New states can be added by adding the `extends State` keyword to a new class.

- `Game` and `Board` have no cyclic 'use' relations. This results in low coupling between the two modules, as the Game module uses the Board module, but the Board module does not use the Game module.

- Over the course of the design, I opted to go for a design that starts at the smaller details first, then work my way up to achieve generality. For example, I designed the Position and Direction modules first, which I then used in the Tile module. Then, in the Board module I used the Tile module, and then I used the Board module in the Game module. This bottom up design approach allowed for smaller details like the position of a tile to be fine tuned.

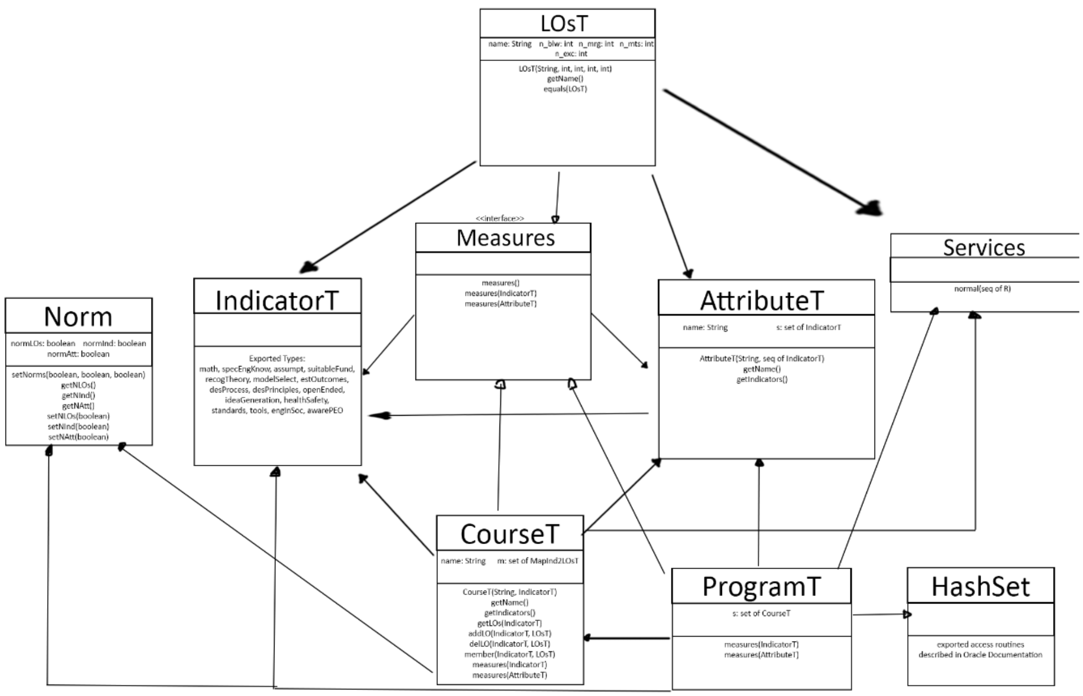
- `Game` and `Board` are both abstract data types, since I wanted to restrict a client's ability to add new features to either abstract data types. If I made both abstract objects, anyone could add new features to them, which is not the goal of the design specification.

- The Board abstract data type seemed to be a bit too overwhelming, and had a lot of local functions. I could have created a BoardManager that would be responsible for all the calculations required for managing Board moves and tile calculations. I also could have added some more elementary functions in the Tile module that could make calculations in the Board a lot easier. I also could have moved the print function to the TextInterface module, as it would make more sense and have most of the text functions all in one place.

- Many functions in Board could have been privated. Functions such as "moveTile" or "moveTileGivenDirection" could be privated, since external modules should not even use this function directly as all external board interaction should be limited to moveBoard.

Answers to Questions:

1. Draw a UML diagram for the modules in A3.



2. Draw a control flow graph for the convex hull algorithm. The graph should follow the approach used by the Ghezzi et al. textbook. In particular, the code statements should be edges of the graph, not nodes. Code for the convex hull algorithm can be found at: https://startupnextdoor.com/computing-convex-hull-in-python/. To match the diagrams available from Ghezzi, replace the for loop in the code with a while loop.