# Assignment 4, Specification

Mark Hutchison, COMP SCI 2ME3

April 10, 2021

The following specification is for the online game 2048. An online version of 2048 can be found at:

https://play2048.co/

The version we will be implementing will be slightly different, but will follow the majority of the practices found in that game.

The game 2048 is a square matrix, refereed to as a board, of tiles. These tiles can slide from one end of the board to the opposite end, and perform "merges" while sliding when appropriate. A tile can "merge" with an empty tile as many times as it likes during a slide, but may only "merge" once with a tile of the same value during a slide.

The game ends when a user is incapable of performing any further "merges" in any direction available on a directional pad containing "up", "down", "left", and "right". For future reference, the directional pad referenced above will be dubbed the "DPad" for convince.

In some variants of 2048, the game may end when the user gets a tile on the board with the value of 2048. We will not be implementing this behavior. The game should be able to go on as long as possible in my opinion. Once the game is over, the user must be allowed to play again if they so choose. Also, if they reached a tile with 2048 as its value, they should be congratulated in some way, shape, or form when the game ends.

I accounted for change in this design by making everything as modular as possible, with as many freedoms that I could allow without ruining the specification.

- Dividing the labour amongst the classes appropriately.
  - The TileT stores the individual data of each TileT, such as value and colour.
  - The BoardT handles the manipulation of said TileT's.
  - The GameController interacts with the board to play the game, and stores things like score and highest TileT.

  - The User then interacts with some sort of GUI to log their key inputs and play the game.

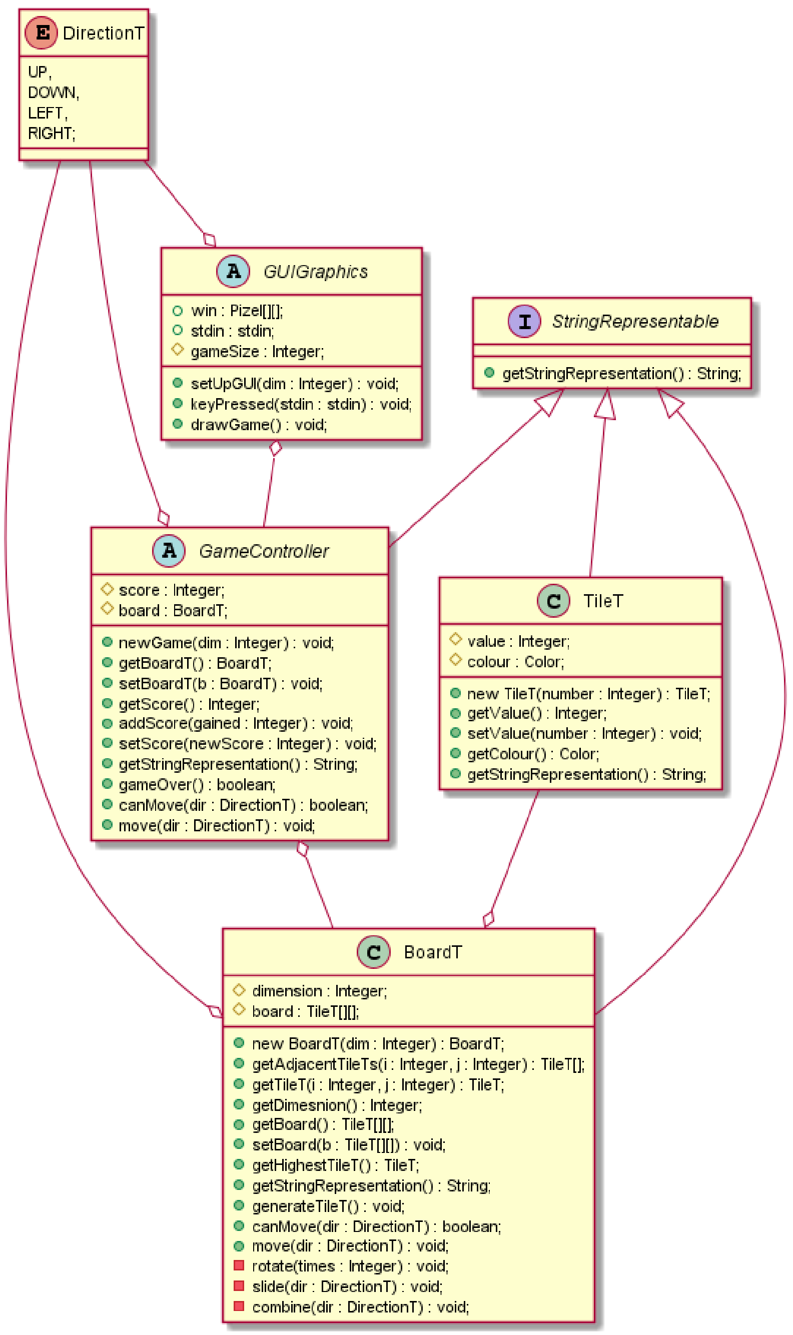
- Allowing the game dimension, while being greater than 3, to be changed to allow the game to have variations that are more fun.

- Programming game mechanics to be modular within themselves, so changes can be done with only minor modifications. If something can be given a variable, it has been to allow the game to feel more customized to the user's preference without major modifications.

The View-Model-Controller pattern is also in itself a design for change, as it divides the 3 main concerns for further easy editing.

- View: `GUIGraphics`
- Controller: `GameController`
- Model: `DirectionT`, `TileT`, `BoardT`

The final file structure is detailed roughly by this UML diagram:



DirectionT
UP,
DOWN,
LEFT,
RIGHT;
GUIGraphics
win : Pizel[][];
stdin : stdin;
gameSize : Integer;
setUpGUI(dim : Integer) : void;
keyPressed(stdin : stdin) : void;
drawGame() : void;
StringRepresentable
getStringRepresentation() : String;
GameController
score : Integer;
board : BoardT;
newGame(dim : Integer) : void;
getBoardT() : BoardT;
setBoardT(b : BoardT) : void;
getScore() : Integer;
addScore(gained : Integer) : void;
setScore(newScore : Integer) : void;
getStringRepresentation() : String;
gameOver() : boolean;
canMove(dir : DirectionT) : boolean;
move(dir : DirectionT) : void;
TileT
value : Integer;
colour : Color;
new TileT(number : Integer) : TileT;
getValue() : Integer;
setValue(number : Integer) : void;
getColour() : Color;
getStringRepresentation() : String;
BoardT
dimension : Integer;
board : TileT[][];
new BoardT(dim : Integer) : BoardT;
getAdjacentTileTs(i : Integer, j : Integer) : TileT[];
getTileT(i : Integer, j : Integer) : TileT;
getDimesnion() : Integer;
getBoard() : TileT[][];
setBoard(b : TileT[][]) : void;
getHighestTileT() : TileT;
getStringRepresentation() : String;
generateTileT() : void;
canMove(dir : DirectionT) : boolean;
move(dir : DirectionT) : void;
rotate(times : Integer) : void;
slide(dir : DirectionT) : void;
combine(dir : DirectionT) : void;

# DirectionT Module

## Module

DirectionT

## Uses

None

## Syntax

### Exported Constants

None

### Exported Types

DirectionT = {
UP,
DOWN,
LEFT,
RIGHT
}

### Exported Access Programs

None

## Semantics

### State Variables

None

### State Invariant

None

### Assumptions

The only inputs available to the 2048 game are the 4 directions on a typical DPad.

**Considerations**

When implementing in Java, use enums to define this like in Assignment 3 and Tutorial 7.

## StringRepresentable Interface Module

### Interface Module

StringRepresentable

### Uses

None

### Syntax

### Exported Constants

None

### Exported Types

None

#### Exported Access Programs

| Routine name | In | Out | Exceptions |
|---|---|---|---|
| getStringRepresentation | | *String* | |

# TileT Module

## TileT Module inherits StringRepresentable

TileT

## Uses

StringRepresentable

## Syntax

### Exported Constants

None

### Exported Types

TileT = ?

### Exported Access Programs

| Routine name | In | Out | Exceptions |
|---|---|---|---|
| new TileT | $\mathbb{Z}$ | TileT | IllegalArgumentException |
| getValue | | $\mathbb{Z}$ | |
| setValue | $\mathbb{Z}$ | | IllegalArgumentException |
| getColor | | $Color$ | |

## Semantics

### State Variables

$value : \mathbb{Z}$
$c$: $Color$

### State Invariant

$value > 0$

### Assumptions

$String(v : \mathbb{Z})$ returns a string containing the integer value of v.

**Access Routine Semantics**

new TileT($number : \mathbb{Z}$):

- transition: $value := number$
- output: $out := self$
- exception: $(number < 0) \Rightarrow IllegalArgumentException$

getTile():

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

setTile($number : \mathbb{Z}$):

- output: $out := number$
- exception: $(number < 0) \Rightarrow IllegalArgumentException$

getColor():

- output:

| Case | out |
|---|---|
| value = 0 | Color(128, 128, 128) |
| value = 2 | Color(238, 228, 218) |
| value = 4 | Color(237, 224, 200) |
| value = 8 | Color(242, 177, 121) |
| value = 16 | Color(245, 149, 99) |
| value = 32 | Color(246, 124, 95) |
| value = 64 | Color(246, 94, 59) |
| value = 128 | Color(237, 207, 114) |
| value = 256 | Color(237, 204, 97) |
| value = 512 | Color(237, 200, 80) |
| value = 1024 | Color(237, 197, 63) |
| True | Color(237, 194, 46) |

- exception: None

getStringRepresentation():

- output: $out := String(value)$
- exception: None

### 0.0.1 Considerations

Implementation of `Color` is facilitated by the java `Color` class.

Color is custom type, represented as $tuple(\mathbb{Z}, \mathbb{Z}, \mathbb{Z})$, where the integers correspond to $(r, g, b)$.

# BoardT Module

## BoardT Module inherits StringRepresentable

BoardT

## Uses

StringRepresentable, DirectionT, TileT

## Syntax

### Exported Constants

None

### Exported Types

BoardT = ?

### Exported Access Programs

| **Routine name** | **In** | **Out** | **Exceptions** |
|---|---|---|---|
| new BoardT | $\mathbb{Z}$ | BoardT | IllegalArgumentException |
| getDimension | | $\mathbb{Z}$ | |
| getBoard | | seq [N] of (seq [N] of TileT) | |
| setBoardT | seq [N] of (seq [N] of TileT) | | IllegalArgumentException |
| getTileT | $\mathbb{Z}$, $\mathbb{Z}$ | $TileT$ | IllegalArgumentException |
| getAdjacentTileTs | $\mathbb{Z}$, $\mathbb{Z}$ | $seq$ [4] $of\ TileT$ | IllegalArgumentException |
| getHighestTileT | | $TileT$ | |
| generateTileT | | | |
| canMove | $DirectionT$ | $\mathbb{B}$ | |
| move | $DirectionT$ | | |

## Semantics

### State Variables

*dimension*: $\mathbb{Z}$
*board* : seq [dimension] of (seq [dimension] of TileT)

**State Invariant**

$dimension > 0$

**Assumptions**

None

**Access Routine Semantics**

new BoardT($dim : \mathbb{Z}$):

- transition: $dimension, board := dim, \langle i : \mathbb{N} \mid 0 < i < dimension : \langle j : \mathbb{N} \mid 0 < j < dimension : \text{ new TileT}(0)\rangle\rangle$
- output: $out := self$
- exception: $(dim < 3) \Rightarrow IllegalArgumentException$

getBoard():

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

getDimension():

- output: $out := dimension$
- exception: None

setBoard($b$ : seq [N] of (seq [N] of TileT)):

- transition: $board, dimension := b, N$
- exception: $\neg(N < 3) \Rightarrow IllegalArgumentException$
- assumption: board is a square matrix of TileTs, and $N : \mathbb{N}$

getTileT($i : \mathbb{Z}, j : \mathbb{Z}$):

- output: $out := board$
- exception: $\neg(0 \leq i, j < dimension) \Rightarrow IllegalArgumentException$

getAdjacentTileTs($i : \mathbb{Z}, j : \mathbb{Z}$):

- output: $out := \langle$up, down, left, right$\rangle$ where:

$$\begin{aligned} up &:= \neg(0 \leq i-1 < dimension) \Rightarrow NIL \mid True \Rightarrow board[i-1][j] \\ down &:= \neg(0 \leq i+1 < dimension) \Rightarrow NIL \mid True \Rightarrow board[i+1][j] \\ left &:= \neg(0 \leq j-1 < dimension) \Rightarrow NIL \mid True \Rightarrow board[i][j-1] \\ right &:= \neg(0 \leq j+1 < dimension) \Rightarrow NIL \mid True \Rightarrow board[i][j+1] \end{aligned}$$

- exception: $\neg(0 < i, j < dimension) \Rightarrow IllegalArgumentException$

getHighestTileTs():

- output: $out := max(\langle i \in \mathbb{N}.(i < dimesnion) \Rightarrow maxValue(board[i])\rangle)$
  See local functions for implementation details of $maxValue()$
- exception: None
- assumptions: max is a defined function that simply returns the largest integer in a sequence of integers.

generateTileT():

- transition:

| | | $board[i][j] :=$ |
|---|---|---|
| $getTileT(i,j).getValue() = 0$ | $random(0,10) < 2$ | new TileT(4) |
| | True | new TileT(2) |
| $getTileT(i,j).getValue() \neq 0$ | $generateTileT()$ | |

  where:
  $i := random(0, dimension - 1)$
  $j := random(0, dimension - 1)$

- assumptions: The function $random(a : \mathbb{N}, b : \mathbb{N})$ described returns an integer in between $a$ and $b$ inclusive. Additionally, this function will only be called when there is a spot available for the TileT to be inserted.

canMove($dir : DirectionT$):

- out: $out := \exists(i, j : \mathbb{N} \mid 0 < i, j < dimension :$

$$\begin{aligned}
dir.equals(DirectionT.UP) &\Rightarrow (getAdjacentTileTs(i,j)[0].getValue() = 0 \\
&\vee getAdjacentTileTs(i,j)[0].getValue() = getTileT(i,j).getValue()) \mid \\
dir.equals(DirectionT.DOWN) &\Rightarrow (getAdjacentTileTs(i,j)[1].getValue() = 0 \\
&\vee getAdjacentTileTs(i,j)[1].getValue() = getTileT(i,j).getValue()) \mid \\
dir.equals(DirectionT.LEFT) &\Rightarrow (getAdjacentTileTs(i,j)[2].getValue() = 0 \\
&\vee getAdjacentTileTs(i,j)[2].getValue() = getTileT(i,j).getValue()) \mid \\
dir.equals(DirectionT.RIGHT) &\Rightarrow (getAdjacentTileTs(i,j)[3].getValue() = 0 \\
&\vee getAdjacentTileTs(i,j)[3].getValue() = getTileT(i,j).getValue()) \\
&)
\end{aligned}$$

- exception: None

move($dir$ : $DirectionT$):

- transition:

| | instructions |
|---|---|
| $dir.equals(DirectionT.UP)$ | $rotate(3)$<br>$slide()$<br>$combine()$<br>$slide()$<br>$rotate(1)$ |
| $dir.equals(DirectionT.DOWN)$ | $rotate(1)$<br>$slide()$<br>$combine()$<br>$slide()$<br>$rotate(3)$ |
| $dir.equals(DirectionT.LEFT)$ | $rotate(2)$<br>$slide()$<br>$combine()$<br>$slide()$<br>$rotate(2)$ |
| $dir.equals(DirectionT.RIGHT)$ | $slide()$<br>$combine()$<br>$slide()$ |

  See local functions for specifications regarding functions used.

- exception: None

getStringRepresentation():

- output: The string representation of a board is quite an involved process, and is hard to write mathematically. The returned value of this method is a string representation of the game board using design characters. The process of generating this string is broken into steps:

  1. Define $tileLength$ as $min(6,\ 2 + ||\text{gameController.getHighestTile().getStringRepresentation()}||)$
  2. Now we need to pick our Decoration Alphabet for the board. Your alphabet, at minimum, will include:
     - 4 corner characters: denoted as $TL$, $TR$, $BL$, $BR$
     - A horizontal line character: denoted as $-$
     - A vertical line character: denoted as $l$
     - At minimum 1 "join" character: denoted as $+$
     - A Blank Space character: denoted as $BS$
     - A newline character: denoted as LF.
  3. The final tools we will need are two operators, concatenate ($||$) and repeat ($*$)
  4. We first set $out$ := "$TL\ ||((-*tileLength)\ ||+)\ *dimension - 1\ ||((-*tileLength))\ ||TR\ ||LF$"
  5. Now we need to preform two iterations of code, one that occurs for every $arr:\ seq\ [dimension]\ of\ TileT \in board$, and one for every $t:\ TileT \in arr$.
     (a) For every $arr$, we concatenate a "$l$" character to the string.
     (b) Then for every $t:\ TileT \in arr$, if $t.getValue() > 0$,
     $out := outs\ ||$"$BS*(tileLength - \lfloor \frac{tileLength}{2} \rfloor)\ ||t.getStringRepresentation()\ ||BS*(tileLength - \lceil \frac{tileLength}{2} \rceil)\ ||l$",
     Else,
     $out := ||$"$BS*tileLength\ ||l$"
     (c) If $arr$ is not the last sequence found in $board$, $out := out\ ||$"$LF\ ||+\ ||((-*tileLength)\ ||+)\ *dimension - 1\ ||(-*tileLength)\ ||+\ ||LF$" Else, just $out := out\ ||$"LF"
  6. Finally, we conclude our string:
     $out := out\ ||$"$BL\ ||(-*tileLength\ ||+)\ *dimension - 1\ ||(-*tileLength)\ ||BR\ ||LF$"

  The method described above should render a complete ASCII board representation of the data in $board$.

- exception: None

**Local Routines**

maxValue($row : \ seq \ [dimension] \ of \ TileT$):

- output: $out := max(\langle i \in \mathbb{N} \ . \ (i < dimesnion) \Rightarrow row[i].getValue()\rangle)$
- exception: None
- assumptions: max is a defined function that simply returns the largest integer in a sequence of integers.

rotate($n : \ \mathbb{Z}$):

- transition:
```
For every k: $\mathbb{N}$ where $0 \leq k < n$:
    For every i: $\mathbb{N}$ where $0 \leq i < dimension$:
        For every j: $\mathbb{N}$ where $i \leq j < dimension$:
            $board[i][j], board[j][i] := board[j][i], board[i][j]$
    For every i: $\mathbb{N}$ where $0 \leq i < dimension$:
        For every j: $\mathbb{N}$ where $i \leq j < \lfloor \frac{dimension}{2} \rfloor$:
            $board[i][j], board[i][dimension{-}1{-}j] := board[i][dimension{-}1{-}j], board[i][j]$
```
- exception: None

slide():

- transition:
```
For every k: $\mathbb{N}$ where $0 \leq i < dimension$:
    For every i: $\mathbb{N}$ where $dimension - 1 > i \geq 0$:
        For every j: $\mathbb{N}$ where $i \leq j < dimension$:
            $(board[k][j].getValue() \neq 0 \wedge board[k][j{+}1].getValue() = 0) \Rightarrow board[k][j], board[k][j+1] := board[k][j+1], board[k][j]$
```
- exception: None

combine():

- transition:
```
For every k: $\mathbb{N}$ where $0 \leq i < dimension$:
    For every i: $\mathbb{N}$ where $dimension - 1 \geq i > 0$:
        $(board[k][i].getValue() \neq 0 \wedge board[k][i].getValue() = board[k][i{-}1].getValue()) \Rightarrow$
        {
            $GameController.addScore(board[k][i-1].getValue() * 2);$
            $board[k][i] := newTileT(board[k][i-1].getValue() * 2);$
            $board[k][i-1] := newTileT();$
        }
```

- exception: None

# GameController Module (Abstract Object)

## GameController Module inherits StringRepresentable

GameController

## Uses

StringRepresentable, DirectionT, TileT, BoardT

## Syntax

### Exported Constants

None

### Exported Types

GameController = ?

### Exported Access Programs

| Routine name | In | Out | Exceptions |
|---|---|---|---|
| newGame | $\mathbb{Z}$ | | |
| getBoard | | $BoardT$ | IllegalArgumentException |
| setBoard | $BoardT$ | | |
| getScore | | $\mathbb{Z}$ | |
| addScore | $\mathbb{Z}$ | | |
| setScore | $\mathbb{Z}$ | | |
| gameOver | | $\mathbb{B}$ | |
| canMove | $DirectionT$ | $\mathbb{B}$ | |
| move | | | |

## Semantics

### State Variables

$score : \mathbb{Z}$
$board$: $BoardT$

**State Invariant**

$score \geq 0$

**Assumptions**

None

**Access Routine Semantics**

newGame($dim : \mathbb{Z}$):

- transition: $board, score := newBoardT(dim), 0$
- exception: $dim >= 3 \Rightarrow IllegalArgumentException$

getBoardT():

- out: $out := board$
- exception: None

setBoardT($b : BoardT$):

- transition: $board := b$
- exception: None

getScore():

- out: $out := score$
- exception: None

addScore($num : \mathbb{N}$):

- transition: $score := score + num$
- exception: None

setScore($num : \mathbb{N}$):

- transition: $score := num$
- exception: None

canMove($dir : DirectionT$):

- out: $out := board.canMove(dir)$
- exception: None

move($dir : DirectionT$):

- transition: $board := board.move(dir); board.generateTileT()$
- exception: None

gameOver():

- out: $out := True$
```
For every i: $\mathbb{N}$ where $0 \leq i < board.getDimension()$:
    For every j: $\mathbb{N}$ where $0 \leq i < board.getDimension()$:
        $(board.getTileT(i, j).getValue() = 0) \Rightarrow\ out := False$
        For every $tile : TileT \in board.getAdjacentTileTs(i, j)$:
            $(board.getTileT(i, j).getValue() = tile.getValue()) \Rightarrow\ out := False$
```
- description: This function will return the final result of *out* after preforming the operations described. *out* is by default $True$, and becomes $False$ when a condition violates the gameOver condition.
- exception: None

getStringRepresentation():

- output: $out :=$ "2048 ||LF ||$board.getStringRepresentation()$ ||LF ||Score: ||$score$ ||LF ||Highest Tile: ||LF ||$board.getHighestTileT().getValue().getStringRepresentation()$"

  $out := (board.getHighestTileT().getValue() = 2048) \Rightarrow out$ ||"Congratulations! ||LF ||Press 'Enter' to Restart or Keep Playing! ||LF" |

  $True \Rightarrow out$ ||"Press 'Enter' to Start! ||LF ||Press 'wasd' or Arrow Keys to slide! ||LF"
- exception: None

## GUIGraphics Module

### Module

GUIGraphics

### Uses

None

### Syntax

#### Exported Constants

None

#### Exported Access Programs

| **Routine name** | **In** | **Out** | **Exceptions** |
|---|---|---|---|
| *setUpGUI* | $\mathbb{N}$ | | IllegalArgumentException |
| *keyPressed* | *stdin* | | |
| *drawGame* | | | |

### Semantics

#### Environment Variables

win: two dimensional sequence of coloured pixels
stdin: the standard input used by the user, like a keyboard

#### State Variables

$gameSize : \mathbb{N}$

#### State Invariant

$gameSize >= 3$

**Assumptions**

stdin will, at minimum, contain one set of inputs that clearly and explicitly translates to the standard DPad directions.

The GUI will be designed for a 4x4 game board, but should be able to render larger even if not athletically pleasing.

**Access Routine Semantics**

setUpGUI($dim : \mathbb{N}$):

- transition: $GameController.newGame(dim); gameSize := dim$

  After the game controller is set up, preform the necessary steps to set up a game window on $win$, and populate it with the appropriate information. For the information you should render, refer to $GameController.getStringRepresentation()$.

  After this setup, call $drawGame$.

- exception: $dim < 3 \Rightarrow IllegalArgumentException$

keyPressed():

- transition: Monitor $stdin$ for the user pressing any valid DPad direction or the starting key of your choice. Upon pressing the starting key, preform the following:

  $$GameController.newGame(gameSize)$$
  $$GameController.getBoardT().generateTileT()$$
  $$GameController.getBoardT().generateTileT()$$

  Upon pressing any DPad key, find the $dir : DirectionT$ that matches it, and preform the following $GameController.canMove(dir) \Rightarrow GameController.move(dir)$

- exception: None

drawGamer():

- transition:

  Simply render the game state to $win$, referring to $GameController.getStringRepresentation()$ for items that need to be rendered.

  Every transition of the GameController's board state should re-render the game board using this routine.

- exception: None

### 0.0.2 Considerations

Implementation of `GUIGraphics.java` is facilitated by the java `swing` library.

## Critique of my Design

The critique of a design comes down to 4 properties:

- Consistent: I did everything in my power to remain consistent in this specification. I used Camel Case whenever possible, I kept variable names similar, and it is all done in my writing style (good or bad as that may be). I did everything in my power to maintain a level of consistency and opacity when it came to writing this specification.
- Essential: Essential means that no two routines in the same module do the same thing, and I can say with confidence that this is true. However, I wanted to point out a place of worry: The gameController does contain routines that do nothing but call the routine of the game BoardT and propagates the value. This is still technically essential, and is the point of the design patter, but it is still a place that I think needs to be discussed when talking about essential design.
- Minimal: I kept my Model as minimal as possible, however I sacrificed this quality when finishing my `BoardT` and coding my `GameController`. For example, `GameController.move()` both moves the board, but also generates a `TileT` at the same time. It makes sense to combine these into one, but it isn't minimal.
- Generality: This is very literal sadly, and not very general. Looking at the specification, there is only 1 use case for every type and it is extremely explicit in the use case. I didn't even try to make this general, as that is time I couldn't spend due to the amount of work.