SFWRENG 2AA4 Assignment 4, MIS

Saruggan Thiruchelvan (thirus6)

May 16, 2021

This document is the Module Interface Specification (MIS) for implementing the classic game, *2048*. The game is started with an empty 4×4 board and two tiles with the value of 2 (or possibly 4) are added to random positions on the board. The player can move LEFT, RIGHT, UP, or DOWN causing the tiles to shift in the given direction. If two tiles with the same value are beside each other in the direction of a move, then those tiles merge resulting in a tile with the double their value. With each move, a random tile is added to the board and the score is accumulated with the value of all merged tiles during the move. The goal is to get the 2048 tile (or possibly beyond), but if the player cannot make a move in any direction, the game is over. A demo of the implementation can be launched using the command `make demo`. In the Demo.java file, the marking TA can comment/uncomment to select between starting a new game (default) or resuming/restoring a previous game.

Design Overview

The software architecture follows the Model View Controller (MVC) design pattern. The model modules represent the data structures and accessors/mutators of the data which includes *BoardT*, *ScoreT*, and *DirectionT*. The controller modules maintain and control the state of the game and the game logic which includes *Controller*, and *BoardManager*. The view modules work to display the state of the game using a graphical user interface (GUI) and translate player interactions which includes *View*, *ComponentUI*, *ScoreUI*, *TileUI*, *BoardUI*, and *MessageUI*.

I designed these software modules to be intuitive to use and support likely changes. For example, variants of the game should be supported such as: tiles with different values (non 2-multiples or even negative values), different board sizes, or timed gameplay to name a few. Additionally, a likely feature would be the ability to resume or restore the state of a previous game and continue with one's current score/ high score. With respect to the view modules, some likely additions include introduction of animations and window resizeability.

Design Critique

I tried to follow the MVC design pattern when designing my program to improve maintainability and design for change. The model modules (*BoardT, ScoreT, DirectionT*) are simply data structures that have services to access/mutate the state of the game. They purposely do not contain any game logic which is instead handled by the controller modules (*Controller, BoardManager*). The controller modules have services that manipulate the model modules to represent the current state of the game following a succession of actions by the user. Lastly, the view modules (*View*, *ComponentUI*, *ScoreUI*, *TileUI*, *BoardUI*, *MessageUI*) display the state of the model modules using a GUI and translate the user's interactions into method

invocations of the controller modules. Some aspects of the MVC pattern are broken for simplicity but for the most part, it allows the modules to be highly cohesive and have low coupling.

The *BoardT* module has a very consistent design as the method signatures (such as getTile(i, j), setTile(i, j, value), isTileEmpty(i, j)) follow the same ordering of parameters (i-index, then j-index, etc). Although the design is highly cohesive (state and routines are highly related to the board of the game), it is not very essential. I opted to make the design more general rather than essential to make the module more convenient to use both in the code and in the written specification; methods such as isTileEmpty(i, j) are not absolutely necessary when one can just check if getTile(i, j) = 0 but it makes understanding the code/specification more intuitive. Similar methods such as equals(other) and isFull() are useful for the user to perform checks, conduct tests and avoid exceptions.

The *ScoreT* module is designed to be highly opaque and provide the most essential routines for maintaining the score of the game. Instead of adding setScore and setHighScore methods, I added an updateScore(points) method which adds to the existing score and updates the high score if necessary as well. Similarly, a resetScore method is used to set the score back to zero but leave the high score untouched. This design greatly supports information hiding as it allows the user to just add points to the score without needed the value of the current score in addition to preventing the user from potentially corrupting the high score of the module with direct mutators. Similarly, the design is also essential as it includes accessors for the score and high score but omits (direct) mutators for the high score as that it unnecessary (and possibly dangerous) to change the high score in such a way. I reluctantly added an initializeScore(initialScore, initialHighScore) method to support restoring the state of a previous game but it is necessary to call it at the start of a new game as the score and high score state variable are already initialized to 0 by default.

I separated the controller modules into two parts: the *Controller* which has services for the user to make moves and check the state the state of the game and the *BoardManager* library which the *Controller* uses for complex multi-stage operations on *BoardT* objects. The *BoardManager* library contains useful methods such as merge(board, direction) and align(board, direction) which are partial operations of the larger move(direction) operation in the *Controller* module. This design prevents the user from using the *Controller* module to leave the game board in an invalid state (such as a merge without aligning) while allowing the complex services in *BoardManager* to be developed separately and exported to potentially be used by other modules. This design is highly maintainable and supports design for change as the implementation details can be maintained (or possibly changed for variations of the game) without fear of potentially corrupting one another. Additionally, the *Controller* is designed to highly usable as it includes methods such as isGameOver() and canMove(direction) to allow the user to again perform checks, conduct tests and avoid exceptions. It is designed to be highly flexible as a new game can be initialized by invoking newGame() or previous game can be restored by invoking resumeGame(board, intialScore, initalHighScore). Currently, there is no means of restoring the state of a game from a file or a database, but the resumeGame method allows one to start from an existing board configuration, so in the future, that feature can seamlessly be integrated. I chose to add the initialScore and initialHighScore to the method signature which implicitly initializes the ScoreT module so that fewer lines of code need to be written to resume a game.

The *View* module is absolutely not minimal. It contains just one single method display() which calls many services to initialize and format the GUI components and finally display the window. This was done intentionally as I wanted the module to be able to display the game in as few

steps as possible so I wrapped all that functionality into a single method invocation. However, the GUI itself supports modularity as complex GUI components are wrapped in their own highly cohesive modules such as ScoreUI for displaying the score/high score and BoardUI for displaying the grid of TileUI components. This is achieved due in part to the ComponentUI interface which forces "UI components" to implement its necessary methods. UI components are GUI components that need to be updated regularly to reflect the state of the game and are designated by the "UI" suffix in their names and implementing the ComponentUI interface. Currently, the interface has just one method signature update() which is called following each key press or mouse click but in the future if I want to introduce more complex additions to my GUI design (such as animations), the infrastructure exists to do so and as a result, supports design for change.

DirectionT Module (Enumerated Type)

Module

DirectionT

Uses

None

Syntax

Exported Constants

None

Exported Types

DirectionT = {
LEFT,
RIGHT,
UP,
DOWN,
}

Exported Access Programs

None

Semantics

State Variables

None

State Invariant

None

Assumptions

None

BoardT Module (Abstract Data Type)

Template Module

BoardT

Uses

DirectionT

Syntax

Exported Constants

SIZE = 4

Exported Types

BoardT = ?

Exported Access Programs

Routine name	In	Out	Exceptions
new BoardT		BoardT	
getTile	$\mathbb{Z}$, $\mathbb{Z}$	$\mathbb{Z}$	IndexOutOfBoundsException
setTile	$\mathbb{Z}$, $\mathbb{Z}$, $\mathbb{Z}$		IndexOutOfBoundsException
isTileEmpty	$\mathbb{Z}$, $\mathbb{Z}$	$\mathbb{B}$	IndexOutOfBoundsException
isFull		$\mathbb{B}$	
equals	BoardT	$\mathbb{B}$	
copy		BoardT	

Semantics

State Variables

$tiles$: seq [SIZE] of seq [SIZE] of $\mathbb{Z}$

State Invariant

None

Assumptions

None

Notes

The index i specifies the horizontal position of the tile from left to right. The index j specifies the vertical position of the tile from top to bottom. The top-left position of the board is (0, 0) while the bottom-right position of the board is (SIZE-1, SIZE-1).

Access Routine Semantics

new BoardT():

- transition: $tiles := < \begin{matrix} < 0,0,0,0 >, \\ < 0,0,0,0 >, \\ < 0,0,0,0 >, \\ < 0,0,0,0 > \end{matrix} >$
- output: out := self
- exception: none

getTile(i, j):

- output: out := $tiles[i][j]$
- exception: exc := $(i < 0) \vee (i \geq \text{SIZE}) \vee (j < 0) \vee (j \geq \text{SIZE}) \Rightarrow$ IndexOutOfBoundsException

setTile(i, j, $value$):

- transition: $tiles[i][j] := value$
- exception: exc := $(i < 0) \vee (i \geq \text{SIZE}) \vee (j < 0) \vee (j \geq \text{SIZE}) \Rightarrow$ IndexOutOfBoundsException

isTileEmpty(i, j):

- output: out := $tiles[i][j] = 0$
- exception: exc := $(i < 0) \vee (i \geq \text{SIZE}) \vee (j < 0) \vee (j \geq \text{SIZE}) \Rightarrow$ IndexOutOfBoundsException

isFull():

- output: out := $\neg \exists (i, j : \mathbb{Z} | 0 \leq i < \text{SIZE} \wedge 0 \leq j < \text{SIZE} : \text{self.isTileEmpty}(i, j))$
- exception: none

equals($other$):

- output: out := $(\forall i, j : \mathbb{Z} | i \in [0..board.\text{SIZE}-1] \wedge j \in [0..board.\text{SIZE}-1] : board.\text{getTile}(i, j) = other.\text{getTile}(i, j))$
- exception: none

copy():

- output: out := $newBoard$: BoardT such that self.equals($newBoard$)
- exception: none

ScoreT Module (Abstract Object)

Module

ScoreT

Uses

None

Syntax

Exported Constants

None

Exported Types

None

Exported Access Programs

Routine name	In	Out	Exceptions
initialize	$\mathbb{Z}, \mathbb{Z}$		IllegalArgumentException
updateScore	$\mathbb{Z}$		IllegalArgumentException
getScore		$\mathbb{Z}$	
getHighScore		$\mathbb{Z}$	
resetScore			

Semantics

State Variables

$score : \mathbb{Z} \,|\, score = 0$
$highScore : \mathbb{Z} \,|\, highScore = 0$

State Invariant

$score \geq 0 \wedge highScore \geq 0 \wedge highScore \geq score$

Assumptions

None

Access Routine Semantics

initialize($initialScore, initialHighScore$):

- transition: $score, highScore := initialScore, initialHighScore$
- exception: exc := $initialScore < 0 \lor initialHighScore < 0 \lor initialHighScore < initialScore \Rightarrow$ IllegalArgumentException

updateScore($points$):

- transition: $score, highScore := score + points, (score + points > highScore \Rightarrow score + points) | (\text{True} \Rightarrow highScore)$
- exception: exc := $points < 0 \Rightarrow$ IllegalArgumentException

getScore():

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

getHighScore():

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

resetScore():

- transition: $score := 0$
- exception: none

BoardManager Module (Library)

Module

BoardManager

Uses

BoardT, DirectionT

Syntax

Exported Constants

None

Exported Types

None

Exported Access Programs

Routine name	In	Out	Exceptions
merge	BoardT, DirectionT	BoardT	
scoreFromMerge	BoardT, DirectionT	$\mathbb{Z}$	
align	BoardT, DirectionT	BoardT	
addRandomTile	BoardT	BoardT	IllegalStateException

Semantics

State Variables

None

State Invariant

None

Assumptions

None

Access Routine Semantics

merge(*board*, *direction*):

- output: out := $newBoard$: BoardT such that $\forall i, j : \mathbb{Z} | i \in [0..board.\text{SIZE} - 1] \wedge j \in [0..board.\text{SIZE} - 1]$:

		Out
direction = LEFT	SMH($board$, $[0..board.\text{SIZE} - 1], i, j$)	$newBoard.\text{getTile}(i, j) = board.\text{getTile}(i, j) + board.\text{getTile}(i + 1, j)$
	$\neg(\text{SMH}(board, [0..board.\text{SIZE} - 1], i, j)) \wedge \text{SMH}(board, [0..board.\text{SIZE} - 1], i - 1, j)$	$newBoard.\text{isEmpty}(i, j)$
	$\neg(\text{SMH}(board, [0..board.\text{SIZE} - 1], i, j)) \wedge \neg(\text{SMH}(board, [0..board.\text{SIZE}-1], i-1, j))$	$newBoard.\text{getTile}(i, j) = board.\text{getTile}(i, j)$
direction = RIGHT	SMH($board$, $[board.\text{SIZE} - 1..0], i, j$)	$newBoard.\text{getTile}(board.\text{SIZE} - 1 - i, j) = board.\text{getTile}(board.\text{SIZE} - 1 - i, j) + board.\text{getTile}(board.\text{SIZE} - 1 - i - 1, j)$
	$\neg(\text{SMH}(board, [board.\text{SIZE} - 1..0], i, j)) \wedge \text{SMH}(board, [board.\text{SIZE} - 1..0], i - 1, j)$	$newBoard.\text{isEmpty}(board.\text{SIZE} - 1 - i, j)$
	$\neg(\text{SMH}(board, [board.\text{SIZE} - 1..0], i, j)) \wedge \neg(\text{SMH}(board, [board.\text{SIZE}-1..0], i-1, j))$	$newBoard.\text{getTile}(board.\text{SIZE} - 1 - i, j) = board.\text{getTile}(board.\text{SIZE} - 1 - i, j)$
direction = UP	SMV($board$, $[0..board.\text{SIZE} - 1], i, j$)	$newBoard.\text{getTile}(i, j) = board.\text{getTile}(i, j) + board.\text{getTile}(i, j + 1)$
	$\neg(\text{SMV}(board, [0..board.\text{SIZE} - 1], i, j)) \wedge \text{SMV}(board, [0..board.\text{SIZE} - 1], i, j - 1)$	$newBoard.\text{isEmpty}(i, j)$
	$\neg(\text{SMV}(board, [0..board.\text{SIZE} - 1], i, j)) \wedge \neg(\text{SMV}(board, [0..board.\text{SIZE}-1], i, j-1))$	$newBoard.\text{getTile}(i, j) = board.\text{getTile}(i, j)$
direction = DOWN	SMV($board$, $[board.\text{SIZE} - 1..0], i, j$)	$newBoard.\text{getTile}(i, board.\text{SIZE} - 1 - j) = board.\text{getTile}(i, board.\text{SIZE} - 1 - j - 1)$
	$\neg(\text{SMV}(board, [board.\text{SIZE} - 1..0], i, j)) \wedge \text{SMV}(board, [board.\text{SIZE} - 1..0], i, j - 1)$	$newBoard.\text{isEmpty}(i, board.\text{SIZE} - 1 - j)$
	$\neg(\text{SMV}(board, [board.\text{SIZE} - 1..0], i, j)) \wedge \neg(\text{SMV}(board, [board.\text{SIZE}-1..0], i, j-1))$	$newBoard.\text{getTile}(i, board.\text{SIZE} - 1 - j) = board.\text{getTile}(i, board.\text{SIZE} - 1 - j)$

- exception: none

scoreFromMerge(*board*, *direction*):

- output: out :=

	Out
direction = LEFT	$(+i, j : \mathbb{Z} \mid i \in [0..board.\text{SIZE} - 1] \wedge j \in [0..board.\text{SIZE} - 1] \wedge \text{SMH}(board, [0..board.\text{SIZE} - 1], i, j) : board.\text{getTile}(i, j) + board.\text{getTile}(i + 1, j))$
direction = RIGHT	$(+i, j : \mathbb{Z} \mid i \in [0..board.\text{SIZE} - 1] \wedge j \in [0..board.\text{SIZE} - 1] \wedge \text{SMH}(board, [board.\text{SIZE} - 1..0], i, j) : board.\text{getTile}(board.\text{SIZE} - 1 - i, j) + board.\text{getTile}(board.\text{SIZE} - 1 - i - 1, j))$
direction = UP	$(+i, j : \mathbb{Z} \mid i \in [0..board.\text{SIZE} - 1] \wedge j \in [0..board.\text{SIZE} - 1] \wedge \text{SMV}(board, [0..board.\text{SIZE} - 1], i, j) : board.\text{getTile}(i, j) + board.\text{getTile}(i, j + 1))$
direction = DOWN	$(+i, j : \mathbb{Z} \mid i \in [0..board.\text{SIZE} - 1] \wedge j \in [0..board.\text{SIZE} - 1] \wedge \text{SMV}(board, [board.\text{SIZE} - 1..0], i, j) : board.\text{getTile}(i, board.\text{SIZE} - 1 - j) + board.\text{getTile}(i, board.\text{SIZE} - 1 - j - 1))$

- exception: none

align(*board*, *direction*):

- output: out := $newBoard$: BoardT such that $\forall i, j : \mathbb{Z} | i \in [0..board.\text{SIZE} - 1] \wedge j \in [0..board.\text{SIZE} - 1]$:

	Out
direction = LEFT	$(\exists row_{\text{NE}} : \text{set of } \mathbb{Z} | row_{\text{NE}} = \text{nonEmptyRowIndices}(board, j) : (i < |row_{\text{NE}}| \Rightarrow newBoard.\text{getTile}(i, j) = board.\text{getTile}(row_{\text{NE}}[i], j)) | (i \geq |row_{\text{NE}}| \Rightarrow newBoard.\text{isTileEmpty}(i, j)))$
direction = RIGHT	$(\exists row_{\text{NE}} : \text{set of } \mathbb{Z} | row_{\text{NE}} = \text{nonEmptyRowIndices}(board, j) : (i < board.\text{SIZE} - |row_{\text{NE}}| \Rightarrow newBoard.\text{isTileEmpty}(i, j)) | (i \geq board.\text{SIZE} - |row_{\text{NE}}| \Rightarrow newBoard.\text{getTile}(i, j) = board.\text{getTile}(row_{\text{NE}}[i - (board.\text{SIZE} - |row_{\text{NE}}|)], j)))$
direction = UP	$(\exists col_{\text{NE}} : \text{set of } \mathbb{Z} | col_{\text{NE}} = \text{nonEmptyColumnIndices}(board, i) : (j < |col_{\text{NE}}| \Rightarrow newBoard.\text{getTile}(i, j) = board.\text{getTile}(i, col_{\text{NE}}[j])) | (j \geq |col_{\text{NE}}| \Rightarrow newBoard.\text{isTileEmpty}(i, j)))$
direction = DOWN	$(\exists col_{\text{NE}} : \text{set of } \mathbb{Z} | col_{\text{NE}} = \text{nonEmptyColumnIndices}(board, i) : (j < board.\text{SIZE} - |col_{\text{NE}}| \Rightarrow newBoard.\text{isTileEmpty}(i, j)) | (j \geq board.\text{SIZE} - |col_{\text{NE}}| \Rightarrow newBoard.\text{getTile}(i, j) = board.\text{getTile}(i, col_{\text{NE}}[j - (board.\text{SIZE} - |col_{\text{NE}}|)])))$

- exception: none

addRandomTile(*board*):

- output: out := $newBoard$: BoardT such that $(\exists i, j : \mathbb{Z} | i \in [0..board.\text{SIZE} - 1] \wedge j \in [0..board.\text{SIZE}-1] : i = \text{randomIndex}(board) \wedge j = \text{randomIndex}(board) \wedge board.\text{isTileEmpty}(i, j) \wedge (\forall x, y : \mathbb{Z} | x \in [0..board.\text{SIZE} - 1] \wedge y \in [0..board.\text{SIZE} - 1] : (x \neq i) \wedge (y \neq j) \wedge newBoard.\text{getTile}(x, y) = board.\text{getTile}(x, y) \wedge newBoard.\text{getTile}(i, j) = \text{randomTile}(board)))$
- exception: exc := $board.\text{isFull}() \Rightarrow$ IllegalStateException

Local Functions

randomIndex: BoardT $\rightarrow \mathbb{Z}$
$\text{randomIndex}(board) \equiv \lfloor random() \times board.\text{SIZE} \rfloor$

randomTile: BoardT $\rightarrow \mathbb{Z}$
$\text{randomTile}(board) \equiv (random() < 0.9 \Rightarrow 2) | (\text{True} \Rightarrow 4)$

nonEmptyRowIndices: BoardT $\times \mathbb{Z} \rightarrow$ set of $\mathbb{Z}$
$\text{nonEmptyRowIndices}(board, j)$
$\equiv \cup(i : \mathbb{Z} | i \in [0..board.\text{SIZE} - 1] \wedge \neg board.\text{isTileEmpty}(i, j) : \{i\})$

nonEmptyColumnIndices: BoardT $\times \mathbb{Z} \rightarrow$ set of $\mathbb{Z}$
$\text{nonEmptyColumnIndices}(board, i)$
$\equiv \cup(j : \mathbb{Z} | j \in [0..board.\text{SIZE} - 1] \wedge \neg board.\text{isTileEmpty}(i, j) : \{j\})$

SMH: BoardT × seq of $\mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \to \mathbb{B}$ # shouldMergeHorizontal
SMH($board, x_s, i, j$) $\equiv$

			Out
$board$.getTile($x_s[i], j$) = $board$.getTile($x_s[i+1], j$) $\wedge$ $\neg(i < 0 \vee i \geq board$.SIZE)	$i = 0$	True	True
	$i > 0$	$board$.getTile($x_s[i], j$) $\neq$ $board$.getTile($x_s[i-1], j$) $\vee$ SMH($board, x_s, i-2, j$)	True
		$\neg$($board$.getTile($x_s[i], j$) $\neq$ $board$.getTile($x_s[i-1], j$) $\vee$ SMH($board, x_s, i-2, j$))	False
$board$.getTile($x_s[i], j$) $\neq$ $board$.getTile($x_s[i+1], j$) $\vee$ $i < 0 \vee i \geq board$.SIZE	True	True	False

SMV: BoardT × seq of $\mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \to \mathbb{B}$ # shouldMergeVertical
SMV($board, y_s, i, j$) $\equiv$

			Out
$board$.getTile($i, y_s[j]$) = $board$.getTile($i, y_s[j+1]$) $\wedge$ $\neg(j < 0 \vee j \geq board$.SIZE)	$j = 0$	True	True
	$j > 0$	$board$.getTile($i, y_s[j]$) $\neq$ $board$.getTile($i, y_s[j-1]$) $\vee$ SMV($board, y_s, i, j-2$)	True
		$\neg$($board$.getTile($i, y_s[j]$) $\neq$ $board$.getTile($i, y_s[j-1]$) $\vee$ SMV($board, y_s, i, j-2$))	False
$board$.getTile($i, y_s[j]$) $\neq$ $board$.getTile($i, y_s[j+1]$) $\vee$ $j < 0 \vee j \geq board$.SIZE	True	True	False

Controller Module (Abstract Object)

Module

Controller

Uses

BoardT, BoardManager, ScoreT, DirectionT

Syntax

Exported Constants

None

Exported Types

None

Exported Access Programs

Routine name	In	Out	Exceptions
newGame			
resumeGame	BoardT, $\mathbb{Z}, \mathbb{Z}$		IllegalStateException
move	DirectionT		IllegalStateException
canMove	DirectionT	$\mathbb{B}$	
isGameOver		$\mathbb{B}$	
nextTileToGet		$\mathbb{Z}$	
getBoard		BoardT	

Semantics

State Variables

board : BoardT

State Invariant

None

Assumptions

Assume newGame or resumeGame is called before using any of the other services.

Access Routine Semantics

newGame():

- transition: *board* := BoardManager.addRandomTile(BoardManager.addRandomTile(new BoardT())) and ScoreT module following ScoreT.resetScore()
- exception: none

resumeGame($resumeBoard, initialScore, initialHighScore$):

- transition: *board* := $resumeBoard$
  and ScoreT module following ScoreT.initialize($initialScore, initialHighScore$)
- exception: exc := $initialScore < 0 \lor initialHighScore < 0 \lor initialHighScore < initialScore \Rightarrow$ IllegalArgumentException
- assumptions: Assume that $resumeBoard$ will not be an empty BoardT object (i.e. it will not be a new BoardT())

move(*direction*):

- transition: *board* := (self.canMove(*direction*) $\Rightarrow$ BoardManager.addRandomTile(BoardManager.align(BoardManager.merge(BoardManager.align(*board*, *direction*), *direction*), *direction*)))and ScoreT module following ScoreT.updateScore(BoardManager.scoreFromMerge(BoardManager.align(*board*, *direction*), *direction*))
- exception: exc := self.isGameOver() $\Rightarrow$ IllegalStateException

canMove(*direction*):

- output: out := $\neg$*board*.equals(BoardManager.merge(BoardManager.align(*board*, *direction*), *direction*))
- exception: none

isGameOver():

- output: out := $\neg$self.canMove(DirectionT.LEFT) $\land$ $\neg$self.canMove(DirectionT.RIGHT) $\land$ $\neg$self.canMove(DirectionT.UP) $\land$ $\neg$self.canMove(DirectionT.DOWN)
- exception: none

nextTileToGet(): # the next tile to be achieved in the game; first it is 2048, then 4096, 8192,...

- output: out := (maxTile(*board*) $< 2048 \rightarrow 2048$)|(True $\rightarrow$ maxTile(*board*) $\times$ 2)
- exception: none

getBoard():

- output: out := *board*.copy()
- exception: none

Local Functions

maxTile: BoardT $\rightarrow \mathbb{Z}$
maxTile(*board*) $\equiv$ max($\cup(i, j : \mathbb{Z} | i \in [0..board.\text{SIZE}-1] \land j \in [0..board.\text{SIZE}-1] : \{board.\text{getTile}(i, j)\})$)

max: set of $\mathbb{Z} \rightarrow \mathbb{Z}$
max(*numbers*) $\equiv$ the largest/maximum value in *numbers*

ComponentUI Module (Interface)

Interface Module

ComponentUI

Uses

None

Syntax

Exported Constants

None

Exported Types

None

Exported Access Programs

Routine name	In	Out	Exceptions
update			

Semantics

State Variables

None

State Invariant

None

Assumptions

None

ScoreUI Module (Abstract Data Type)

Template Module inherits ComponentUI

ScoreUI

Uses

ComponentUI, ScoreT

Syntax

Exported Constants

None

Exported Types

ScoreUI = ?

Exported Access Programs

Routine name	In	Out	Exceptions
new ScoreUI			

Semantics

Environment Variables

score: A label to display text on an application window.
highScore: A label to display text on an application window.

State Variables

None

State Invariant

None

Assumptions

None

Considerations

When implementing in Java, use a JPanel as a container for two JLabel instances representing *score* and *highScore* respectively.

Access Routine Semantics

new ScoreUI():

- transition: Initialize the visual appearance of the *score* and *highScore* labels (i.e. font, color, size, etc) to resemble the original 2048 game as close as possible. Set the text for the *score* label to read: "SCORE" and the value of ScoreT.getScore(). Set the text for the *highScore* label to read: "BEST" and the value of ScoreT.getHighScore().
- exception: none

update():

- transition: Set the text for the *score* label to read: "SCORE" and the value of ScoreT.getScore(). Set the text for the *highScore* label to read: "BEST" and the value of ScoreT.getHighScore().
- exception: none

TileUI Module (Abstract Data Type)

Template Module inherits ComponentUI

TileUI

Uses

ComponentUI

Syntax

Exported Constants

None

Exported Types

TileUI = ?

Exported Access Programs

Routine name	In	Out	Exceptions
new TileUI	$\mathbb{Z}$		
setValue	$\mathbb{Z}$		

Semantics

Environment Variables

tile : A square label to display text on an application window.

State Variables

$value : \mathbb{Z}$

State Invariant

None

Assumptions

Assume that the *value* is never greater than 4 digits (or monospace characters) in length.

Considerations

When implementing in Java, use a JLabel to represent *tile*.

Access Routine Semantics

new TileUI($initialValue$):

- transition: $value := initialValue$
  Initialize the visual appearance of the *tile* label (i.e. font, border, size, etc) to resemble the original 2048 game as close as possible. Set the text for the *tile* label to equal *value* (If $value = 0$, set the text for the *tile* label to an empty string). The color of the text for *tile* should be equal to the hexadecimal code of foreground(*value*). The background color for *tile* should be equal to the hexadecimal code of background(*value*).

- exception: none

setValue($newValue$):

- transition: $value := newValue$

update():

- transition: Set the text for the *tile* label to equal *value* (If $value = 0$, set the text for the *tile* label to an empty string). The color of the text for *tile* should be equal to the hexadecimal code of foreground(*value*). The background color for *tile* should be equal to the hexadecimal code of background(*value*).

- exception: none

Local Functions

foreground: $\mathbb{Z} \rightarrow$ Hexidecimal code
foreground($value$) $\equiv$

value	**Out**
0	0xe6e3e0
2	0x776e65
4	0x776e65
> 4	0xf9f6f2

background: $\mathbb{Z} \rightarrow$ Hexidecimal code
background($value$) $\equiv$

value	**Out**
0	0xe6e3e0
2	0xeee4da
4	0xeee1c9
8	0xf3b27a
16	0xf69664
32	0xf77c5f
64	0xf75f3b
128	0xedd073
256	0xedcc62
512	0xedc950
1024	0xedc53f
2048	0xedc22e
4096	0x5eda92

BoardUI Module (Abstract Data Type)

Template Module inherits ComponentUI

BoardUI

Uses

ComponentUI, TileUI, Controller, BoardT

Syntax

Exported Constants

None

Exported Types

BoardUI = ?

Exported Access Programs

Routine name	In	Out	Exceptions
new BoardUI			

Semantics

Environment Variables

tiles : A square grid of TileUI components to be displayed on an application window.

State Variables

board : BoardT

State Invariant

None

Assumptions

None

Considerations

When implementing in Java, use a JPanel with a GridLayout as a container for the TileUI instances in *tiles*.

Access Routine Semantics

new BoardUI():

- transition: *board* := Controller.getBoard()
  $(\forall i, j : \mathbb{Z} | i \in [0..board.\text{SIZE}-1] \wedge j \in [0..board.\text{SIZE}-1]$: add new TileUI($board.getTile(i, j)$) to *tiles*) and initialize the visual appearance of the *tiles* grid (i.e. colors, borders, etc) to resemble the original 2048 game as close as possible. Add the TileUI instances in a way such that *board*.getTile(0, 0) corresponds with the top-left of the grid and *board*.getTile(*board*.SIZE - 1, *board*.SIZE - 1) corresponds with the bottom-right of the grid.

- exception: none

update():

- transition: *board* := Controller.getBoard()
  Then, $(\forall i, j : \mathbb{Z} | i \in [0..board.\text{SIZE} - 1] \wedge j \in [0..board.\text{SIZE} - 1]$: set the value for the corresponding *tile*:TileUI in *tiles* using *tile*.setValue(*board*.getTile(i, j))).
  Next, $(\forall tile : \text{TileUI} | tile \in tiles : tile.\text{update}())$

- exception: none

MessageUI Module (Abstract Data Type)

Template Module inherits ComponentUI

MessageUI

Uses

ComponentUI, Controller

Syntax

Exported Constants

None

Exported Types

MessageUI = ?

Exported Access Programs

Routine name	In	Out	Exceptions
new MessageUI			

Semantics

Environment Variables

message: A label to display text on an application window.

State Variables

None

State Invariant

None

Assumptions

None

Considerations

When implementing in Java, use a JLabel to represent *message*.

Access Routine Semantics

new MessageUI():

- transition: Initialize the visual appearance of the *message* (i.e. font, color, size, etc) to resemble the original 2048 game as close as possible. Set the text for the *message* label to read: "Game Over!" if Controller.isGameOver() is true. Set the text for the *message* label to read: "Join the tiles, get to 2048!" if Controller.isGameOver() is false and Controller.nextTileToGet() < 2048. Set the text for the *message* label to read: "You won! Try to get to " and the value of Controller.nextTileToGet() if Controller.isGameOver() is false and Controller.nextTileToGet() > 2048.

- exception: none

update():

- transition: Set the text for the *message* label to read: "Game Over!" if Controller.isGameOver() is true. Set the text for the *message* label to read: "Join the tiles, get to 2048!" if Controller.isGameOver() is false and Controller.nextTileToGet() < 2048. Set the text for the *message* label to read: "You won! Try to get to " and the value of Controller.nextTileToGet() if Controller.isGameOver() is false and Controller.nextTileToGet() > 2048.

- exception: none

View Module (Abstract Object)

Module

View

Uses

Controller, ScoreUI, BoardUI, MessageUI

Syntax

Exported Constants

None

Exported Types

None

Exported Access Programs

Routine name	In	Out	Exceptions
display			

Semantics

Environment Variables

window : An application window to display the game on a portion of the computer screen.
title : A label to display text on an application window.
newGameButton : A button with text that can be displayed on an application window and clicked with the mouse.

State Variables

score : ScoreUI
board : BoardUI
message : MessageUI

State Invariant

None

Assumptions

None

Considerations

When implementing in Java, use a JFrame to represent *window*, use a JLabel to represent *title*, use a JButton to represent *newGameButton*.

Access Routine Semantics

display():

- transition: $score, board, message$:= new ScoreUI(), new BoardUI(), new MessageUI()

  Initialize the *window* with a width of 400 pixels and a height of 500 pixels. Next, initialize the *title* label with the text "2048" and format it (color, font, size etc.) to closely resemble the title label on the original 2048 game. Then, initialize the *newGameButton* with the text "New Game" and format it (color, size, shape) to closely resemble the New Game button on the original 2048 game.

  Add the *title* label to the top left area of *window*. Add *score* to the top right area of *window*. Add *message* to the window below the title and add *newGameButton* to the window below *score* such that there is ample space between them and enough space for the *board* to be added as large as possible. Lastly, add the *board* to the bottom of the *window* so that it is cantered and large as possible.

  When the window is displayed, it should listen for the user's key presses so that if Controller.isGameOver() is false, it should invoke Controller.move() according to which key is pressed. If the left arrow key is pressed, invoke Controller.move(Direction.LEFT), if the right arrow key is pressed, invoke Controller.move(Direction.RIGHT), if the up arrow key is pressed, invoke Controller.move(Direction.UP), and if the down arrow key is pressed, invoke Controller.move(Direction.DOWN). Consequently, *score*.update(), *board*.update(), *message*.update() are invoked.

  When the window is displayed, it should listen for the user's mouse clicks so that if the user clicks *newGameButton*, then Controller.newGame() is invoked. Consequently, *score*.update(), *board*.update(), *message*.update() are invoked.

  When the window is exited, the method should terminate.

- exception: none