Assignment 4, Design Specification

COMPSCI 2ME3, Hyosik Moon

April 15, 2021

Before explaining the design specification of the program, let's look at how to play the game first. As shown in the graph, players have to move and add the blocks and make the score 2048. The player can move up, down, left, and right to merge the blocks. When two consecutive blocks that have a same value move a same direction, they are added. All blocks are moved at the same time, players should be cautious about which blocks they will add. When there are no places to move the game ends.

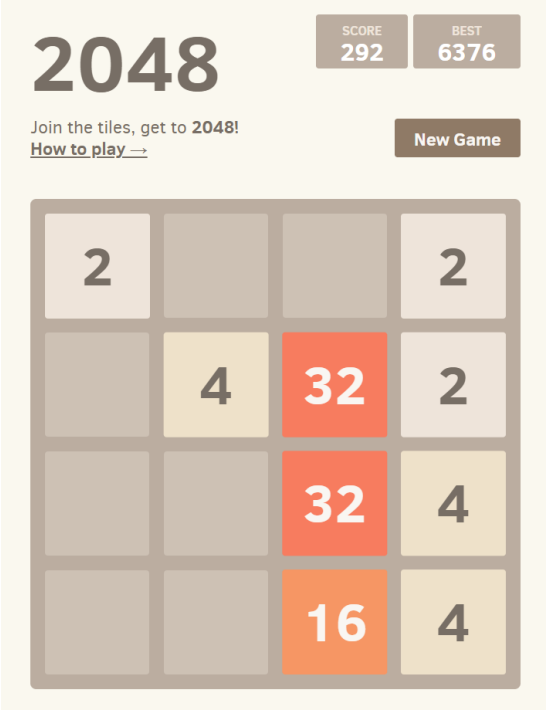

Figure 1: 2048 [3]

1 Overview of the design

This design applies Module View Specification (MVC) design pattern, Strategy design pattern and Singleton design pattern. The MVC components are *GameController* (controller module), *BoardT* (model module), and *UserInterface* (view module) [1]. For the Strategy design pattern, the *EndCondition* (interface) captures the abstraction and the derived classes: *EndByTime* and *EndByMoves* encapsulate the implementation details. Singleton pattern is specified and implemented for *GameController* and *UserInterface*.

An UML diagram is provided below for visualizing the structure of this software architecture [2].

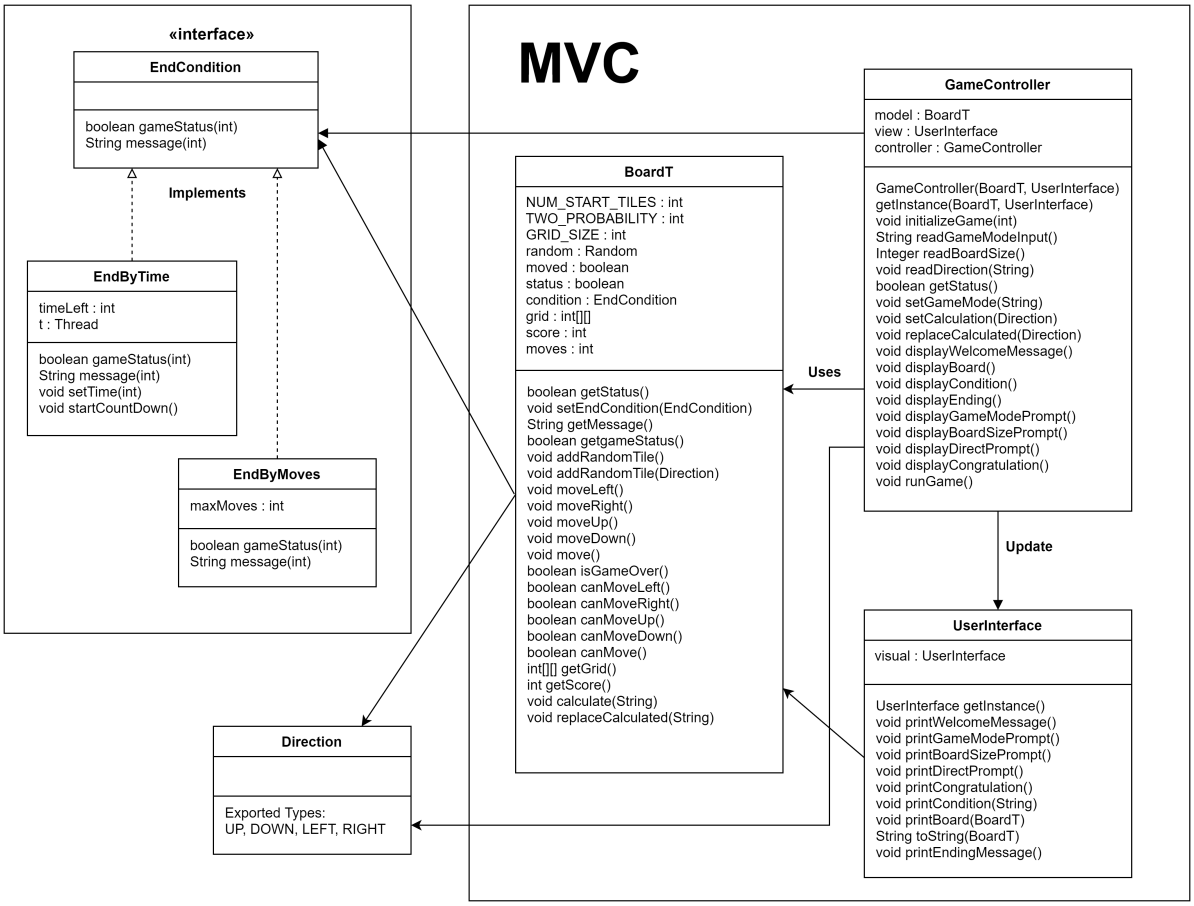

Figure 2: UML diagram of 2048

The MVC design pattern is specified and implemented in the following way: the module *BoardT* stores the state of the game board and the status of the game. A view module *UserInterface* can display the state of the game board and game using a text-based graphics. The controller *GameController* is responsible for controlling the flow of the game by handling input actions, and interacting with the model and view modules.

The implementation of the Strategy pattern enables the game ending condition to be changeable during runtime. The user can choose the game mode to end the game after a time limit *EndByTime* or after a specified amount of moves (*EndByMoves*).

For the Singleton design pattern, *GameController* and *UserInterface* use the `getInstance()` method to obtain the abstract object.

Likely Changes my design considers:

- Data structure used for storing the game board
- The visual representation of the game such as UI layout.
- Change in peripheral devices for taking user input.
- Change in game ending conditions to adjust the difficulty of the game.

Direction Module

Module

Direction

Uses

N/A

Syntax

Exported Constants

None

Exported Types

Direction = {UP, DOWN, LEFT, RIGHT}

Exported Access Programs

None

Semantics

State Variables

None

State Invariant

None

EndCondition Interface Module

Interface Module

EndCondition

Uses

None

Syntax

Exported Constants

None

Exported Types

None

Exported Access Programs

Routine name	In	Out	Exceptions
gameStatus	$\mathbb{N}$	$\mathbb{B}$	
message	$\mathbb{N}$	*String*	

EndByMoves Module

Template Module implements EndCondition Interface

EndByMoves

Uses

EndCondition

Syntax

Exported Constants

None

Exported Types

None

Exported Access Programs

Routine name	In	Out	Exceptions
EndByMoves		EndByMoves	

Semantics

State Variables

maxMoves: $\mathbb{N}$

State Invariant

None

Assumptions

None

Access Routine Semantics

new EndByMoves():

- transition: maxMoves := 2048
- output: *out* := self
- exception: none

gameStatus(moves):

- output: *out* := ¬ ((maxMoves) - moves <= 0)
- exception: none

message():

- output: *out* := A string containing information about the number of moves left
- exception: none

EndByTime Module

Template Module implements EndCondition Interface

EndByTime

Uses

EndCondition, Runnable

Syntax

Exported Constants

None

Exported Types

None

Exported Access Programs

Routine name	In	Out	Exceptions
EndByTime		EndByTime	
setTime	$\mathbb{N}$		
startCountDown			

Semantics

Environment Variables

t: Thread

State Variables

timeLeft: $\mathbb{N}$

State Invariant

None

Assumptions

None

Access Routine Semantics

new EndByTime():

- output: *out* := self
- transition: timeLeft, t := 600, new Thread(Run(this))
  // initiates the timer in thread t
- exception: none

gameStatus(moves):

- output: ¬ (timeLeft <= 0)
- exception: none

message(moves):

- output: *out* := A string containing information about the time left
- exception: none

setTime(time):

- transition: timeLeft := time
- output: none
- exception: none

startCountDown():

- transition: t.start() *// Starts the count down clock in the thread*
- output: none
- exception: none

BoardT ADT Module

Template Module

BoardT

Uses

EndCondition, Direction

Syntax

Exported Types

None

Exported Constant

None

Exported Access Programs

Routine name	In	Out	Exceptions
BoardT	$\mathbb{N}$, Random	BoardT	
getStatus		$\mathbb{B}$	
setEndCondition	EndCondition		
getMessage		String	
getgameStatus		String	
addRandomTile			
addRandomTile	Direction		
move	Direction	$\mathbb{B}$	
isGameOver		$\mathbb{B}$	
canMove	Direction	$\mathbb{B}$	
getGrid		seq of (seq of $\mathbb{N}$)	
getScore		$\mathbb{N}$	

Semantics

State Variables

NUM_START_TILES: $\mathbb{N}$
TWO_PROBABILITY: $\mathbb{N}$
GRID_SIZE: $\mathbb{N}$
random: Random
moved: $\mathbb{B}$
status: $\mathbb{B}$
condition: EndCondition
grid: sequence of (sequence of $\mathbb{N}$)
score: $\mathbb{N}$
moves: $\mathbb{N}$

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.
- Players can choose the board size from 1 to infinite.
- Players can choose the game mode which are “limited moves mode” and “limited time mode”.
- If there is a way for blocks to be added, the game continues, otherwise the game ends.
- If the board has more than 30% empty position, a new block is created at a random position.
- When the grid size is bigger than 6 and there are more than 30% empty places, then the number of created blocks increased by boardsize - 4.
- Block 2 is created with a possibility of 90%, and block 4 is created with a possiblity of 10%.

- When the game score reaches 2048, print "Congratulation" message and continue the game.

Design decision

Players can choose the size of the board and game mode which are "limited moves mode" and "limited time mode". When the board size is 1 there is no place to move, so it ends automatically. Players can choose any size of board but for the simplicity of implementation, the game mode's limit condition doesn't change. For example, it the "limited moves mode", the max_movement is 2048, and in the "limited time mode", the limited time is 600s.

Access Routine Semantics

BoardT(boardSize, random):

- transition:
  random := random
  GRID_SIZE := boardSize
  grid := $\langle t : \text{Seq of Integer} \mid t_i \wedge i \in [0..|boardSize| - 1] : |t| = |boardSize| \rangle$
  score, moves, status := 0, 0, true
- output: *out* := self
- exception: none

getStatus():

- output: *out* := status
- exception: none

setEndCondition(condition):

- transition: condition := condition
- output: none
- exception: none

getMessage():

- output: *out* := (condtion.message(moves))

- exception: none

getgameStatus():

- output: *out* := condition.gameStatus(moves)
- exception: none

addRandomTile():

- output: grid := $(\exists i, j, k, l \in \mathbb{N} \mid i, j, k, l < |boardSize| - 1 : grid[i][j] := 2 \wedge grid[k][j] := 2)$
- exception: none

addRandomTile(direction):

- transition: grid := canMove(direction) $\Rightarrow$ $(\exists i, j \in \mathbb{N} \mid i, j < |boardSize| - 1 : grid[i][j] := 2 \vee grid[i][j] := 4)$
- output: none
- exception: none

move(direction):

- output: *out* := ((direction = Direction.LEFT $\Rightarrow$ moveLeft()) || (direction = Direction.RIGHT $\Rightarrow$ moveRight()) || (direction = Direction.UP $\Rightarrow$ moveUp()) || (direction = Direction.DOWN $\Rightarrow$ moveDown()))
- exception: none

isGameOver():

- output: *out* := $(\neg$(canMoveLeft()) $\wedge$ $\neg$(canMoveRight()) $\wedge$ $\neg$(canMoveUp()) $\wedge$ $\neg$(canMoveDown()))
- exception: none

canMove(direction):

- output: *out* := ((direction = Direction.LEFT $\wedge$ canMoveLeft()) || (direction = Direction.RIGHT $\wedge$ canMoveRight()) || (direction = Direction.UP $\wedge$ canMoveUp()) ||(direction = Direction.DOWN $\wedge$ canMoveDown()))
- exception: none

getGrid():

- output: out := grid
- exception: none

getScore():

- output: out := score
- exception: none

Local Functions

moveLeft: void $\rightarrow$ void
moveLeft $\equiv$ $(\exists i, j \in \mathbb{N} \mid i, j < |boardSize| - 1 : grid[i][j-1] = 0 \wedge \neg(grid[i][j] = 0) \Rightarrow grid[i][j-1] := grid[i][j] \wedge grid[i][j] := 0) \wedge (\exists i, j \in \mathbb{N} \mid i, j < |boardSize| - 1 : (grid[i][j-1] = grid[i][j]) \Rightarrow grid[i][j-1] := grid[i][j-1] + grid[i][j] \wedge grid[i][j] := 0)$

moveRight: void $\rightarrow$ void
moveRight $\equiv$ $(\exists i, j \in \mathbb{N} \mid i, j < |boardSize| - 1 : (\neg(grid[i][j-1] = 0) \wedge grid[i][j] = 0 \Rightarrow grid[i][j] := grid[i][j-1] \wedge grid[i][j-1] := 0) \wedge (\exists i, j \in \mathbb{N} \mid i, j < |boardSize| - 1 : (grid[i][j-1] = grid[i][j]) \Rightarrow grid[i][j] := grid[i][j-1] + grid[i][j] \wedge grid[i][j-1] := 0)$

moveUp: void $\rightarrow$ void
moveUp $\equiv$ $(\exists i, j \in \mathbb{N} \mid i, j < |boardSize| - 1 : (grid[i-1][j] = 0 \wedge \neg(grid[i][j] = 0) \Rightarrow grid[i-1][j] := grid[i][j] \wedge grid[i][j] := 0) \wedge (\exists i, j \in \mathbb{N} \mid i, j < |boardSize| - 1 : (grid[i-1][j] = grid[i][j]) \Rightarrow grid[i-1][j] := grid[i-1][j] + grid[i][j] \wedge grid[i][j] := 0)$

moveDown: void $\rightarrow$ void
moveDown $\equiv$ $(\exists i, j \in \mathbb{N} \mid i, j < |boardSize| - 1 : (\neg(grid[i-1][j] = 0) \wedge grid[i][j] = 0 \Rightarrow grid[i][j] := grid[i-1][j] \wedge grid[i-1][j] := 0) \wedge (\exists i, j \in \mathbb{N} \mid i, j < |boardSize| - 1 : (grid[i-1][j] = grid[i][j]) \Rightarrow grid[i][j] := grid[i-1][j] + grid[i][j] \wedge grid[i-1][j] := 0)$

canMoveLeft: void $\rightarrow \mathbb{B}$
canMoveLeft $\equiv$ $(\exists i, j \in \mathbb{N} \mid i, j < |boardSize| - 1 : (grid[i][j-1] = 0 \wedge \neg(grid[i][j] = 0)) \vee (\neg(grid[i][j-1] = 0) \wedge grid[i-1][j] = grid[i][j]))$

canMoveRight: void $\rightarrow \mathbb{B}$
canMoveRight $\equiv$ $(\exists i, j \in \mathbb{N} \mid i, j < |boardSize| - 1 : (\neg(grid[i][j-1] = 0) \wedge (grid[i][j] = 0)) \vee (\neg(grid[i][j-1] = 0) \wedge grid[i][j-1] = grid[i][j]))$

canMoveUp: void $\to \mathbb{B}$
canMoveUp $\equiv (\exists i, j \in \mathbb{N} \mid i, j < |boardSize| - 1 : (grid[i-1][j] = 0 \wedge \neg(grid[i][j] = 0)) \vee (\neg(grid[i-1][j] = 0) \wedge grid[i-1][j] = grid[i][j]))$

canMoveDown: void $\to \mathbb{B}$
canMoveDown $\equiv (\exists i, j \in \mathbb{N} \mid i, j < |boardSize| - 1 : (\neg(grid[i-1][j] = 0) \wedge grid[i][j] = 0) \vee (\neg(grid[i-1][j] = 0) \wedge grid[i-1][j] = grid[i][j]))$

UserInterface Module (Abstract Object)

Module

UserInterface

Uses

BoardT

Syntax

Exported Types

None

Exported Constants

None

Exported Access Programs

Routine name	In	Out	Exceptions
getInstance		UserInterface	
printWelcomeMessage			
printGameModePrompt			
printBoardSizePrompt			
printDirectPrompt	String		
printCongratulation	String		
printCondition	String		
printBoard	String		
printEndingMessage			

Semantics

Environment Variables

None

State Variables

visual: UserInterface

State Invariant

None

Assumptions

- The UserInterface constructor is called for each object instance before any other access routine is called for that object. The constructor can only be called once.

Access Routine Semantics

getInstance():

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

printWelcomeMessage():

- output: none
- exception: none

printGameModePrompt():

- output: none
- exception: none

printBoardSizePrompt():

- output: none
- exception: none

printDirectPrompt():

- output: none
- exception: none

printCongratulation():

- output: none

- exception: none

printCondition(message):

- output: none //Print the message
- exception: none

printBoardT(boardT):

- output: none //Print toString(boardT)
- exception: none

printEndingMessage(boardT):

- output: none
- exception: none

Local Function:

UserInterface: void → UserInterface
UserInterface() ≡ new UserInterface()

toString: BoardT → String toString(boardT) ≡ Print grid information

GameController Module (Abstract Object)

Module

GameController

Uses

BoardT, UserInterface

Syntax

Exported Types

None

Exported Constants

None

Exported Access Programs

Routine name	In	Out	Exceptions
getInstance	BoardT, UserInterface	GameController	
initializeGame	$\mathbb{N}$		
readGameModeInput	String		IllegalArgumentException
readBoardSize	$\mathbb{N}$		IllegalArgumentException
readDirection	String	Direction	IllegalArgumentException
getStatus		$\mathbb{B}$	
getGameMode	String		
setCalculation	Direciton		
replaceCalculated	Direction		
displayWelcomeMessage			
displayBoard			
displayCondition			
displayEnding			
displayGameModePrompt			
displayBoardSizePrompt			
displayDirectPrompt			
displayCongratulation			
runGame			

Semantics

Environment Variables

keyboard: Scanner(System.in) *// reading inputs from keyboard*

State Variables

model: BoardT
view: UserInterface
controller: GameController

State Invariant

None

Assumptions

- The GameController constructor is called for each object instance before any other access routine is called for that object. The constructor can only be called once.
- Assume that model and view instances are already initialized before calling GameController constructor

Access Routine Semantics

getInstance(model, view):

- transition: controller := (controller = null ⇒ new GameController (model, view))
- output: *out* := self
- exception: none

initializeGame(boardSize):

- transition: model := new BoardT(boardSize, new Random())
- output: none
- exception: none

readGameModeInput():

- output: *out* := gameMode = input from an user by the keyboard
- exception: $exc := ((gamemode \neq m) \wedge (gamemode \neq t) \wedge (gamemode \neq e)) \Rightarrow$ IllegalArgumentException
  // "m" limited moves mode, "t" limited time mode, "e" to exit the game

readBoardSize():

- output: *out* := boardsize = input from an user by the keyboard
- exception: $exc := (boardsize \leq 0) \Rightarrow$ IllegalArgumentException

readDirection(input):

- output: *out* := direction = (input from an user by the keyboard) ∧ ((input=Direction.UP *Rightarrow* input := Direction.UP) ∨ (input=w *Rightarrow* input := Direction.UP) ∨ (input=s *Rightarrow* input := Direction.DOWN) ∨ (input=a *Rightarrow* input := Direction.LEFT) ∨ (input=d *Rightarrow* input := Direction.RIGHT))

- exception: $exc := ((direction \neq w) \wedge (direction \neq s) \wedge (direction \neq a) \wedge (direction \neq d)) \Rightarrow$ IllegalArgumentException

getStatus():

- output: *out* := (model.getStatus())
- exception: none

setGameMode(input):

- transition: model := (input = m $\Rightarrow$ model.setEndCondition(new EndByMoves()) $\vee$ input = t $\Rightarrow$ (condition := new EndByTime() $\wedge$ model.setEndCondition(condition) $\wedge$ condition.setCountDown())
- output: none
- exception: $exc := ((input \neq m) \wedge (input \neq t)) \Rightarrow$ IllegalArgumentException

setCalculation(direction):

- transition: model := model.move(direction)
- output: none
- exception: none

replaceCalculated(direction):

- transition: model := model.addRandomTile(direction)
- output: none
- exception: none

displayWelcomeMessage():

- transition: view := view.printWelcomeMessage()
- output: none
- exception: none

displayBoard():

- transition: view := view.printBoard(model)

- output: none
- exception: none

displayCondition():

- transition: view := view.printCondition(model.getMessage())
- output: none
- exception: none

displayEnding():

- transition: view := view.printEndingMessage()
- output: none
- exception: none

displayGameModePrompt():

- transition: view := view.printGameModePrompt()
- output: none
- exception: none

displayBoardSizePrompt():

- transition: view := view.printBoardSizePrompt()
- output: none
- exception: none

displayDirectPrompt():

- transition: view := view.printDirectPrompt()
- output: none
- exception: none

displayCongratulation():

- transition: view := view.printCongratulation()

- output: none
- exception: none

runGame():

- transition: operational method for running the game. The game will start with a welcome message, next asking the user to select a game mode, then display the board and let the user to play the game. Eventually, when the game ends, display the ending message.
- output: None

Local Function:

GameController: BoardT $\times$ UserInterface $\rightarrow$ GameController
GameController($model, view$) $\equiv$ new GameController($model, view$)

Critique of Design

- First of all, I referred to the previous year's assignment4 (two-dots), written by "Bill Song". Because "two-dots" and "2048" have a similiar software architecture, utlizing the proven design patterens can be a safe and efficient strategy. I took the advantageous design patterns from it such as Model View Specification, Strategy design pattern and Singleton design pattern.

- MVC consists of three components, which are "controller", "model", and "view". Each component has a role such as *controller* handles input actions, *view* displays the data from the model component, and *model* encapsulates system's data as well as operations on the data. As MVC components have their own roles, we can separate concerns increasing the maintainability and decreaseing risks about changes. MVC also keeps high cohesion since it groups related functionalities within each module. The deisgn is also low coupling because the modules (model, view, controller) are mostly independent of each other. So, a change in one of the modules does not heavily impact the other. However in oder to maintain high cohesion, I implemented the `printBoard` method in UserInterface module (*view*), which increases the coupling because I had to use BoardT module (*model*).

- Strategy design pattern provides convenience and flexibility in designing for change. For example, this program applies strategy design pattern making the interface of *EndCondition* with two options such as *EndByTime* and *EndByMoves*. This makes it easier to switch between different algorithms during runtime through polymorphism. In addition, it increases maintainability and readability in a way that the concerns are separated into classes instead of using conditional statements to switch strategy in runtime.

- Singleton design pattern ensures that a class has only one instance and provides a global point of access to that instance. Because it has only one instance, we can decrease the potential conflicts in the software which incerases the reliability. For instance, as one GameController instance handles all inputs and communicates with the one UserInterface instance, we can significantly increase the clarity, verifiability, and reduce the ambiguity of software.

- I made the game more general by allowing users can choose the board size. However, when the board size is 1, there is no place to move, so the game will end right away. I also set that when the grid size is bigger than 6 and there are more than 70% empty places, then the number of created blocks increased by boardsize - 4. And I increased the usability after reflecting a real feedback from a user(my brother), I changed the input keys from "U, D, L, R" to "w, s, a, d".

- When I checked the MIS, I found that the terms of board and grid were used for the same meaning. Thus we can say that this is not consistent.
- We can also see the information hiding in the interface module(*EndCondition*) and its implementations protecting other parts of the program from extensive modification. And as I mentioned in the MVC design pattern, it has high cohesion because each components of MVC are highly related to each others.
- We can also check it's not essential because I implemented `addRandomTile()` method two times in BoardT, with and without parameter for the simplicity of the implementation and readability of the code. But I was able to implement it in a same method by using a conditional sentence.
- As we can see in the *moveUp*, *moveDown*, *moveLeft*, *moveRight*, and *move* methods, each method has only one role. So, we can say it's minimal.
- Because the game states change randomly according to the input of the user, testing all methods automatically was impossible. I made *AllTests.java* file for testing *TestBoardT.java*, *TestEndByMoves.java*, and *TestEndByTime.java*. Except for some methods which are possible to test automatically, I verified and tested the software incrementally. Whenever there are small changes, I tested them. You can run the game with the Demo.java file and also "make expt" command on mills. The test cases are designed to validate the correctness of the program based on the requirement and reveal errors or unusual behavior during program execution, every access routine has at least one test case.

Answers to Questions:

Q1: UML diagram for the modules in A3.

- Abstrac Data Types : AttributeT, CourseT, LOsT, ProgramT(subclass), IndicatorT(enum), HashSet(generic)
- Module : Measures(Interface)
- Abstract Objects : Norm
- Libraries : Services

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

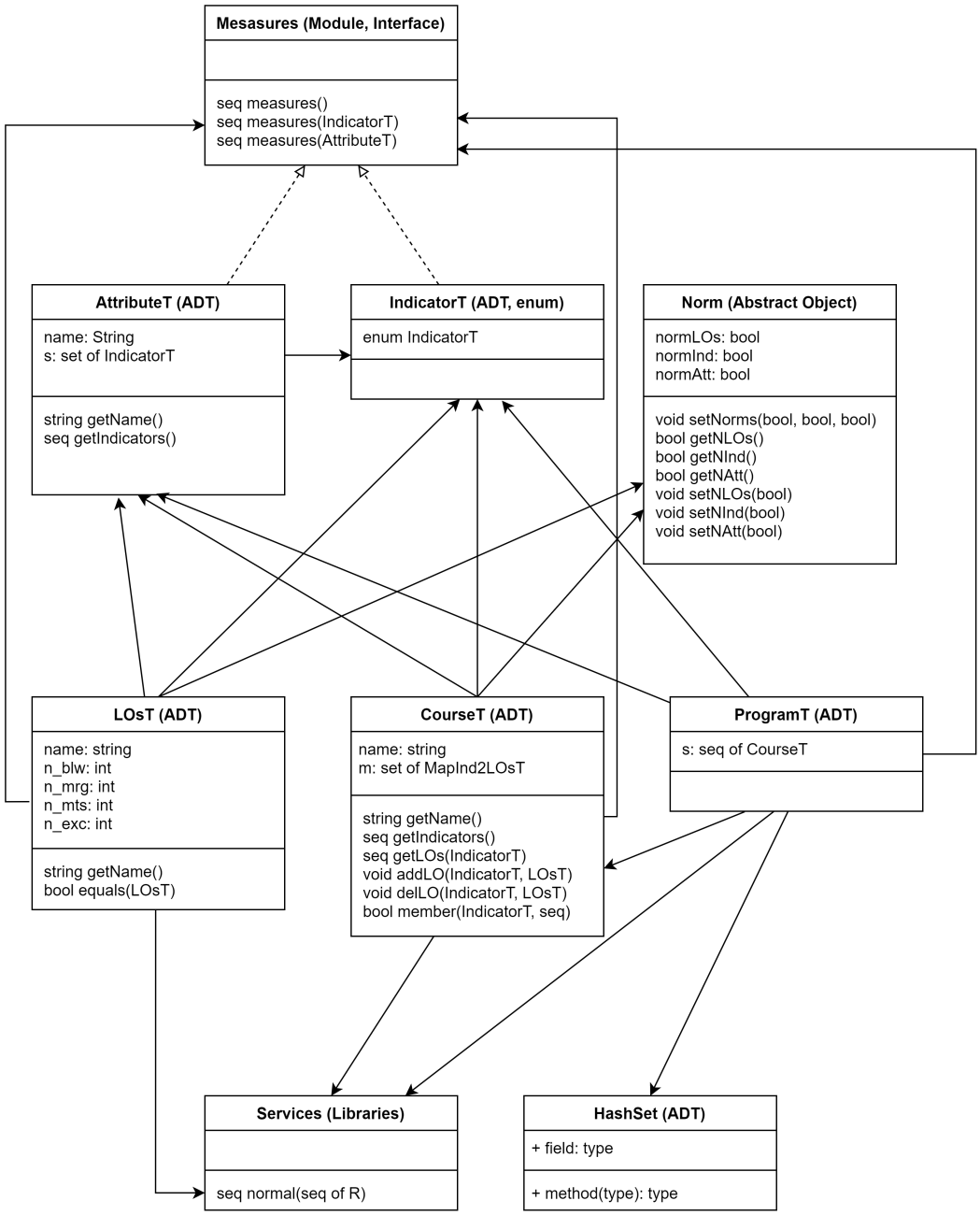

Figure 3: Question 1. UML diagram of A3

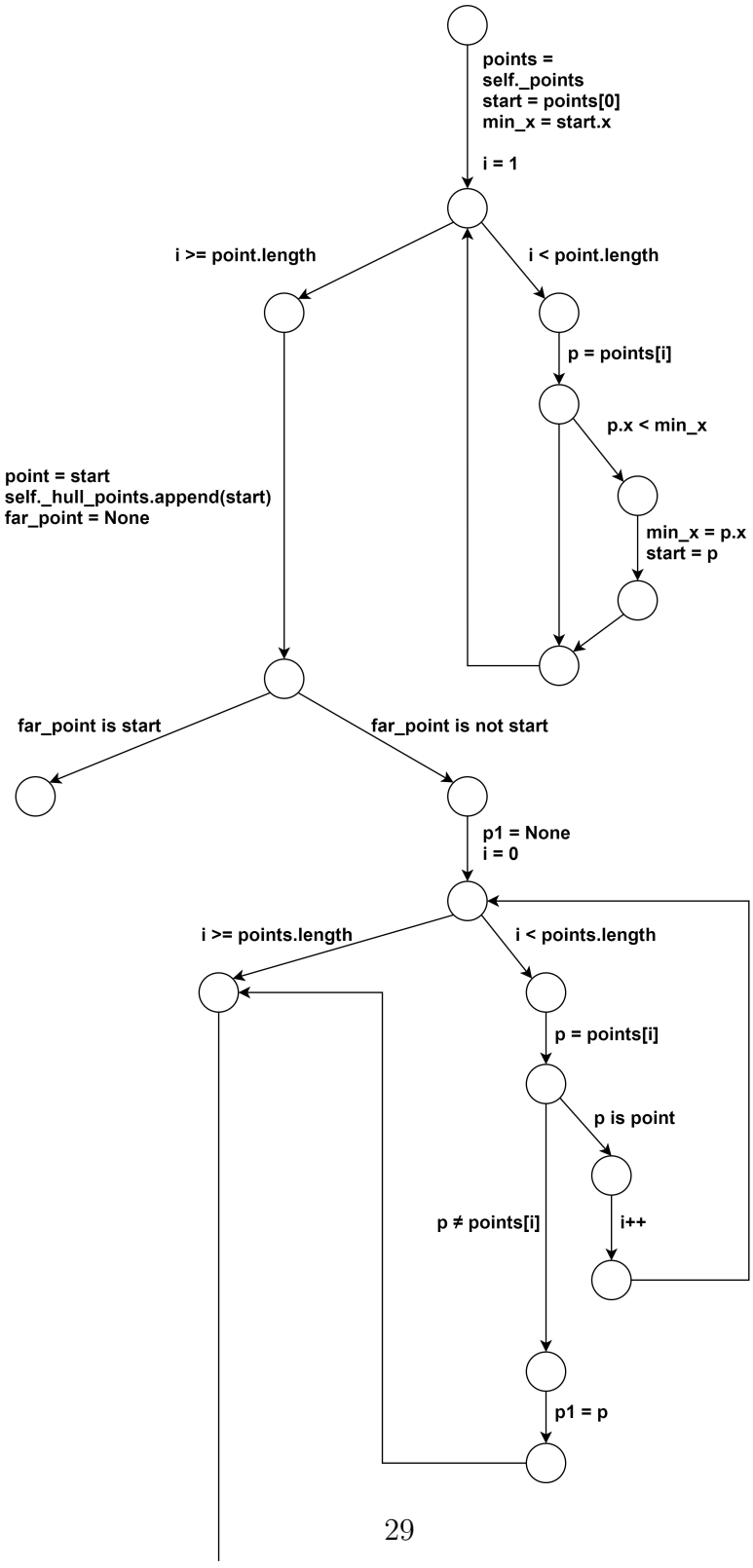

Figure 4: Question 2. Control flow graph for convex hull(1) - continue

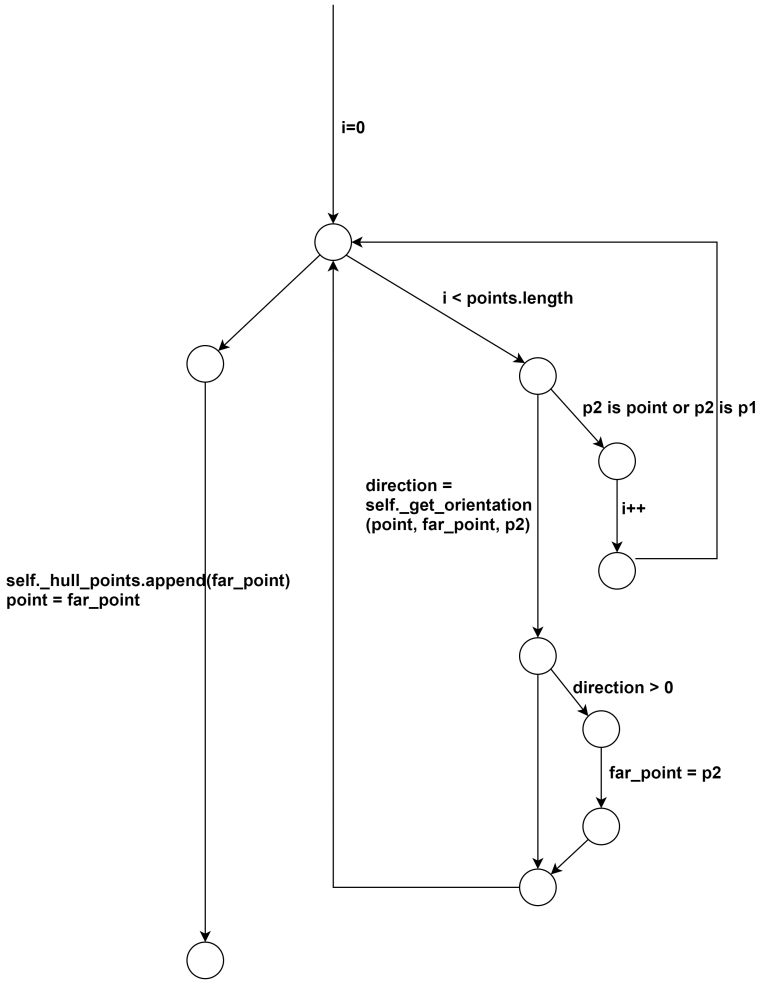

Figure 5: Question 2. Control flow graph for convex hull(2)

References

[1] 2020 Assignment 4, Design Specification (two-dots),
`Written by ‘‘Bill Song’’`

[2] UML diagram,
`https://app.diagrams.net/`

[3] 2048,
`https://play2048.co/`

[4] design strategy pattern,
`https://sourcemaking.com/design_patterns/strategy`