Tabassum Fabiha

tf2478

1.1 - Conflicting design requirements

1) Price
   a) The learning curve of Java would relate to the price point of a restaurant since the time and effort it takes to learn a language is the initial price you have to pay to be able to utilize it.
2) Customer-service
   a) Java's documentation and open community of people provide a great support system, similar to customer-service at any restaurant or establishment.
3) Ability to customize orders
   a) Java's ability to create abstract classes and methods allows for programmers to customize one general object to different specific objects, similar to a restaurant having a general idea for a food option but allowing customers to decide what options to include or remove.
4) Food options
   a) Java has multiple ready to go packages that you can implement in your own programs, which relates to restaurants having multiple food options to choose from.
5) Wait time
   a) The time it takes for Java, or any programming language to perform what it's intended to do relates to the time a customer must wait in a restaurant to receive their food.
6) Service's ability to clean up after the customer
   a) Java has a garbage collector that is responsible for clearing unused memory, which relates to how some restaurants offer to clean up after the customers.

There is no “best” restaurant, just as there is no “best” programming language. There are different pros and cons to each restaurant, such as one might be far more affordable, but another might provide higher quality food. Different situations and circumstances require the use of different restaurants. The same holds true for programming languages.

1.2 - Hello World again

```
/**
 * @author: me
*/
public class NameMaker {

    String name;

    NameMaker() {
        name = world;
    }

    NameMaker(String inName) {
        name = inName;
    }

    NameMaker(String inName, String inUNI) {
        name = inName + “ (” + inUNI + “)”
    }

    public String whatToSay() {
        return name;
    }
}
```

1.3 - arrays, ArrayLists, LinkedLists

a. A suitcase → **array list**

    i. You can add as many things to a suitcase as needed and can remove them without disrupting the order of everything else. An array list allows the option of removing items very quickly while also giving you the option of adding as many items as possible, since there's no set limit on the number of items in a suitcase.

b. A tie, scard, or belt rack → **array**
    i. The capacity for a rack is fixed, and the only data structure that fits this is the array with its fixed size at creation.
c. A roll of paper towels → **linked list**
    i. Each sheet of paper towel only knows the ones immediately attached to it, and decreases in size by one in constant time with the removal of each sheet, much like a linked list.
d. That part of a wallet that holds bills, which the user keeps sorted by denomination → **array list**
    i. A wallet doesn't have a set limit of dollar bills that's placed in it like in an array, and the bills placed in the wallet are not connected to each other in any way like in a linked list, so the only reasonable structure to use is the array list.
e. A waiting line for an event, but where people sneak their friends into the line wherever they happen to be in the line → **linked list**
    i. This scenario is most appropriately fit to a linked list. The people already in line are like the nodes of a linked list. Nodes in linked lists connect to the nodes immediately next to them so their links can be manipulated to add additional nodes into the middle of the linked list, simulating people allowing their friends into the line.
f. A silverware drawer, which has separate bins for forks, knives, etc → **array list nested inside an array**
    i. There are a set number of bins for utensils in the drawer, which is represented by the array because of its fixed size. Each bin however, can

be represented by an array list, because there is no set limit of the number of utensils in the bin and the utensils have no linked connection to each other to warrant a linked list. To restate, the drawer can be represented with an array in which each element, the bins, is an array list.

1.4 - Bad user requirements

1) Can you buy drinks with no pizzas?
2) Do you have to buy cheap and premium toppings together?
3) Can you choose to forgo toppings altogether?
4) Can you handle more than one customer at a time?
5) If you can handle multiple customers at a time, at what point do you take their orders?
6) If you only have an order of one pizza do you want for 2 or 3 more pizza orders to come in before you start baking?
7) How are the orders made? Does the dough need to be created or does it already exist?

1.5 - Good user requirements and CRC, UML

CRC

BankAccount	
- has money - can adjust it's balance	

Button	
- knows its name - knows the position of it's associated coil	

CreditCard	
- knows it's bankAccount - can pay	- BankAccount

VendingMachine	
- has a display - has a panel of buttons - holds packages - accepts Credit Card payment - gets price of snack - dispenses snack - displays error - shows messages	- SpiralCoil - Button - Display - CreditCard - MobilePayment

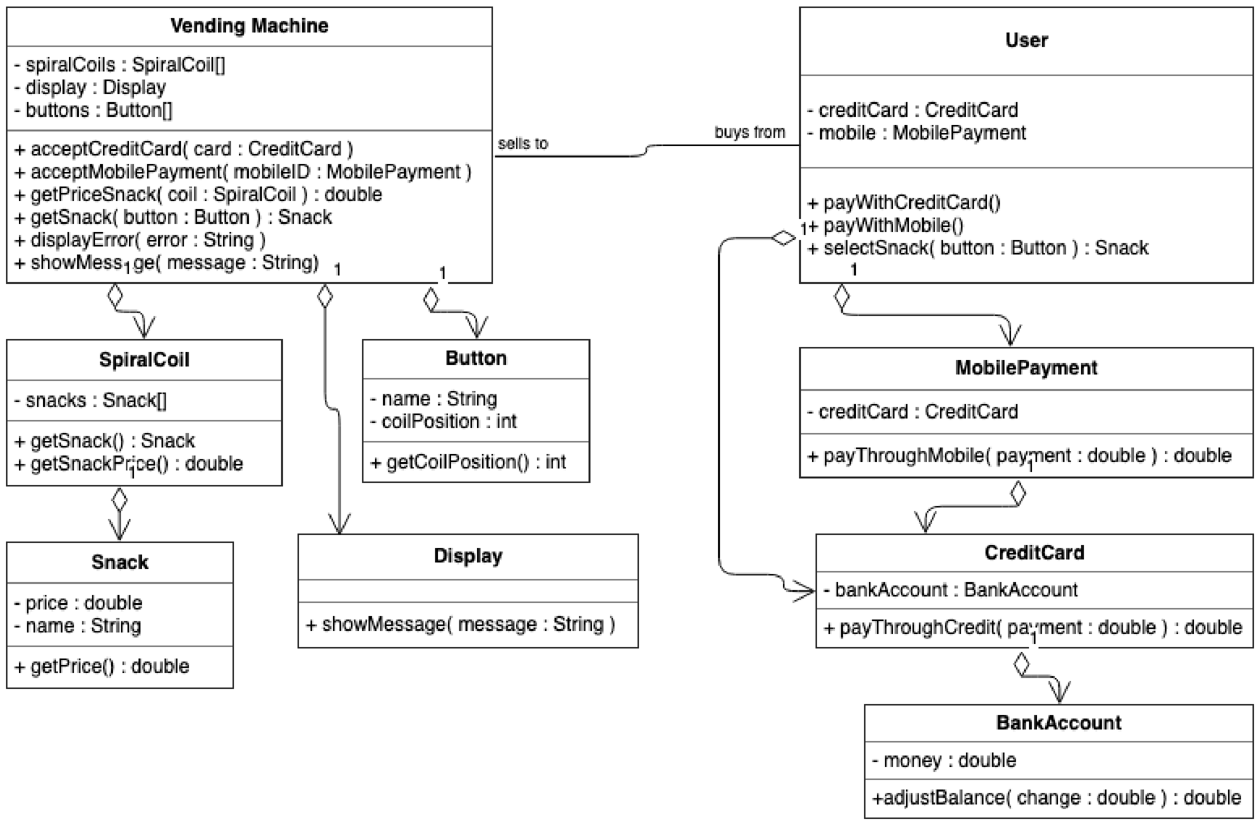
MobilePayment	
- knows it's credit card - can pay	- CreditCard

User	
- select snack - make credit card payment - make mobile payment	- Vending Machine - MobilePayment - CreditCard

SpiralCoil	
- gets snack price - gets snack	- Snack

Snack	
- knows name - knows price	

UML


Vending Machine
- spiralCoils : SpiralCoil[]
- display : Display
- buttons : Button[]
+ acceptCreditCard(card : CreditCard)
+ acceptMobilePayment(mobileID : MobilePayment)
+ getPriceSnack(coil : SpiralCoil) : double
+ getSnack(button : Button) : Snack
+ displayError(error : String)
+ showMessage(message : String)
sells to
buys from
User
- creditCard : CreditCard
- mobile : MobilePayment
+ payWithCreditCard()
+ payWithMobile()
+ selectSnack(button : Button) : Snack
1
1
1
1
1
SpiralCoil
- snacks : Snack[]
+ getSnack() : Snack
+ getSnackPrice() : double
1
Button
- name : String
- coilPosition : int
+ getCoilPosition() : int
MobilePayment
- creditCard : CreditCard
+ payThroughMobile(payment : double) : double
1
Snack
- price : double
- name : String
+ getPrice() : double
Display
+ showMessage(message : String)
CreditCard
- bankAccount : BankAccount
+ payThroughCredit(payment : double) : double
1
BankAccount
- money : double
+adjustBalance(change : double) : double

1.6 - CRC for RPSLK

Talker	
- knows how to read from the console	
- knows what the sim is	- Sim
- knows how many rounds will be in the game	
- knows the current round	
- knows how many options can be played	
- can say welcome to users	
- can tell users the results for the round/game	
- can get the user's play	

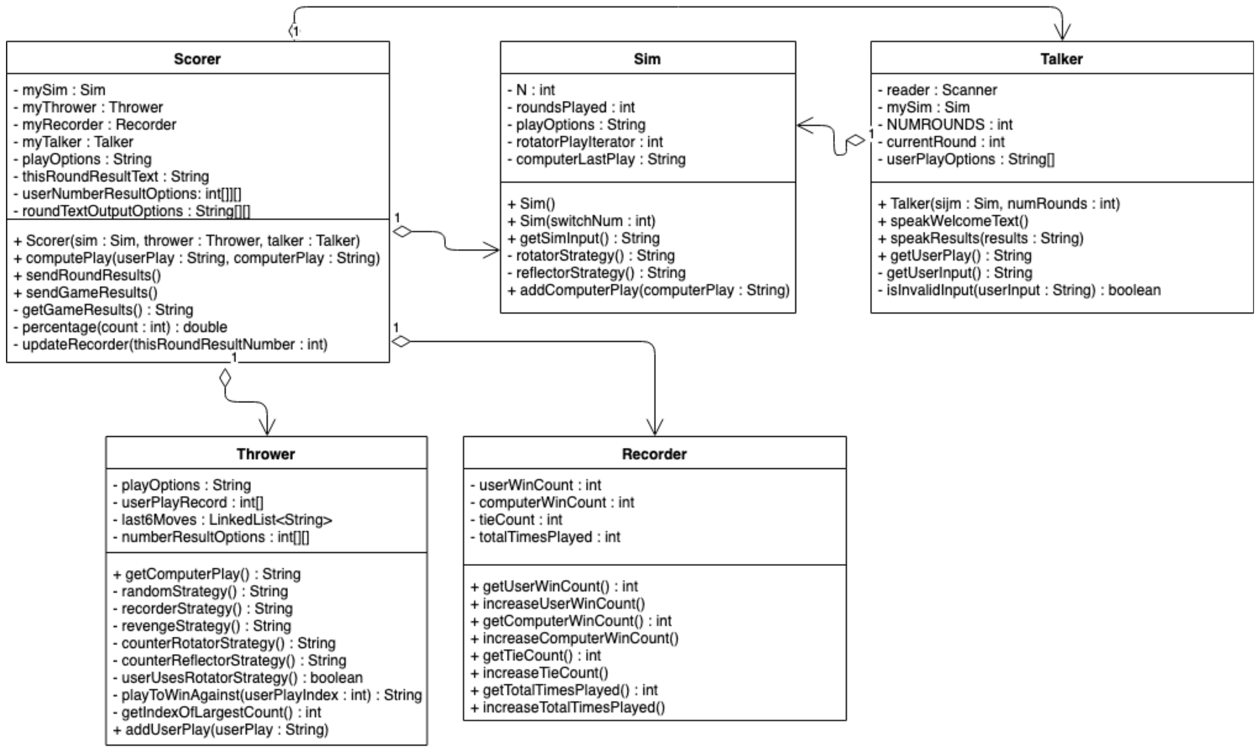
Sim	
- knows the number of times to play one strategy before switching to the other strategy	
- knows the number of rounds played so far	
- knows how many options can be played	
- knows the last move the computer plated	
- knows what move it last played when using the rotator strategy	
- can play user's move using the rotator and reflector strategies	
- can add the move the computer last played into it's database	

Scorer	
- knows the sim	
- knows the thrower	- Sim
- knows the recorder	- Thrower
- knows how many options can be played	- Recorder
- knows who won this round	- Talker
- knows who wins under what conditions	
- knows what to say when someone wins, loses, or ties	
- can compute who one or lost from the plays made	
- can tell talker to print the round/game results	
- can update the Recorder with this round's events	
- can update the Sim and Thrower about who played what last round	

Recorder	
- knows how many times the user won	
- knows how many times the computer won	
- knows how many times there were ties	
- knows how many rounds were played in total	
- can give and increase the number of user wins	
- can give and increase the number of computer wins	
- can give and increase the number of ties	
- can give and increase the number of rounds played	

Thrower	
- knows what options can be played	
- knows the counts of the moves the user's made so far	
- knows the last 6 moves made in the game	
- knows who wins/loses under any condition	
- can come up with the computer's play using the random, recorder, and revenge strategies	
- can add the move the user just made to it's database	

1.7 - UML for RPSLK


Scorer
- mySim : Sim
- myThrower : Thrower
- myRecorder : Recorder
- myTalker : Talker
- playOptions : String
- thisRoundResultText : String
- userNumberResultOptions: int[][]
- roundTextOutputOptions : String[][]
+ Scorer(sim : Sim, thrower : Thrower, talker : Talker)
+ computePlay(userPlay : String, computerPlay : String)
+ sendRoundResults()
+ sendGameResults()
- getGameResults() : String
- percentage(count : int) : double
- updateRecorder(thisRoundResultNumber : int)
Sim
- N : int
- roundsPlayed : int
- playOptions : String
- rotatorPlayIterator : int
- computerLastPlay : String
+ Sim()
+ Sim(switchNum : int)
+ getSimInput() : String
- rotatorStrategy() : String
- reflectorStrategy() : String
+ addComputerPlay(computerPlay : String)
Talker
- reader : Scanner
- mySim : Sim
- NUMROUNDS : int
- currentRound : int
- userPlayOptions : String[]
+ Talker(sijm : Sim, numRounds : int)
+ speakWelcomeText()
+ speakResults(results : String)
+ getUserPlay() : String
- getUserInput() : String
- isInvalidInput(userInput : String) : boolean
Thrower
- playOptions : String
- userPlayRecord : int[]
- last6Moves : LinkedList<String>
- numberResultOptions : int[][]
+ getComputerPlay() : String
- randomStrategy() : String
- recorderStrategy() : String
- revengeStrategy() : String
- counterRotatorStrategy() : String
- counterReflectorStrategy() : String
- userUsesRotatorStrategy() : boolean
- playToWinAgainst(userPlayIndex : int) : String
- getIndexOfLargestCount() : int
+ addUserPlay(userPlay : String)
Recorder
- userWinCount : int
- computerWinCount : int
- tieCount : int
- totalTimesPlayed : int
+ getUserWinCount() : int
+ increaseUserWinCount()
+ getComputerWinCount() : int
+ increaseComputerWinCount()
+ getTieCount() : int
+ increaseTieCount()
+ getTotalTimesPlayed() : int
+ increaseTotalTimesPlayed()
1
1
1
1
1

1.8 - Clean Code for RPSLK

a. Three names I used that correlate to the suggestions in Chapter 2 are getUserPlay(), getComputerPlay(), and updateRecorder(). These names explicitly describe what the methods will do. They are pronounceable, searchable and contain no encodings. In addition, because all three are methods all the names are also verbs.
b. Three methods I used that correlate to the suggestions in Chapter 3 are getSimInput(), getComputerPlay() and revengeStrategy(). These methods only do a single thing and split all complex responsibilities down into other methods.
c. Two comments that I used that correlate to the suggestions in Chapter 4 are written below and can be found at the end of the Scorer class. These comments were the nonnegotiable kind to explain the aggregations I used to hold win/loss/tie conditions and what to say at the end of the round for any combination of moves.

```
/**
        * The 2d array is used to store the info on who wins or loses given
all possible combinations of user and computer plays. The rows going
down represent the user's play in the sequence 'rpslk'. The columns going
across represent the computer's play in the sequence 'rpslk'.
        -1      --> loss for the user
        0       --> tie
        1       --> win for the user

        To find the result of any particular combination:
                        userNumberResultOption[userPlay][computerPlay]
*/
and
/**
```

The 2d array is used to store the info on the text results for all possible combinations of user and computer plays. The rows going down represent the user's play in the sequence 'rpslk'. The columns going across represent the computer's play in the sequence 'rpslk'.

To find the text outputs of any particular combination:

```
roundTextOutputOptions[userPlay][computerPlay]
*/
```