October 15, 2021
Group: DJ WEPNY

Progress Report

Summary on Specification:

In the Specification, we wrote an explanation of what our program is doing and how the user will be interacting with the console. It gives a clear instruction of what the user should expect when they run the program, such as what they will need to enter, and what the program will do with the User's inputs. It also explains in detail what each functionality of our program does and outlines what input those functionalities need the user's to enter, as well as what they will output to the user. Our specification also indicates the databases we will be working with in this project, what we need them for, and the sources of these databases.

CRC model:

Our CRC model is displayed on PowerPoint so it gives a clear visual representation of all our classes. We have designed our CRC model such that it follows the clean architecture principles. Our CRC model contains four layers: Entities, Use cases, Controllers, and Drivers. The Entities represent the fundamental objects the program uses, which are Food, Disease, Exercise, and User as outlined in our CRC model. The use cases manipulate the entities, including CreateUser and UserAnalyzer. The controller, RunCommand and Console, connects the use cases with the driver, Main, and the driver takes in input from the user and outputs the result to the user. These Classes' instance attributes and methods are all clearly outlined in our CRC card. As well as their parent and child classes, if there are any.

Scenario Walkthrough:

In the scenario walkthrough, we provided a detailed instruction on how to run our code and what inputs the user needs to enter to ensure the walkthrough returns the desired output. It also step by step breaks down the different classes involved when the system calculates the user's BMI. First, the Main class will run, calling Console, which will ask for the user's information and pass it to RunCommand (a controller). RunCommand will call CreateUser (a use case) and create a User object (entity) with the information provided by the user. The BMIAnalyzer (a use case) will then use the User object to calculate the user's BMI. The final output the user sees will be printed on the console for the user to see.

Skeleton Program:

Our skeleton code closely follows our scenario walkthrough and successfully runs and outputs the desired result. We have set up all our entity classes, and coded the use cases: CreateUser, UserAnalyzer and the BMIAnalyzer for the skeleton code. We also coded the controller, RunCommand, to direct input to the use cases and return the results from the use

cases to the Console. Lastly we coded the Main Class as a Driver, and the Main class serves as the entry point of the execution of our program.

Reflection:

After writing the skeleton code for phase 0 and modifying our CRC cards according to any changes in the code, we noticed things that both worked well with our design and things that we might need to think about the structure more.

One thing that worked well is the structuring of our code. We adhered to the five principles of clean architecture, and our code shows a clear dependency hierarchy between the Entities, Use Cases, Interface Adapters, and Frameworks & Drivers. There are no signs of classes in lower layers trying to access a class in higher layers, or a class in higher layers (e.g Framework & Drivers) skipping layers (e.g trying to access Use Cases directly.)

However, there are still some issues that need to be handled. One of them is with our controller, RunCommand. Currently, it passes the user input from the Console to the Use Cases, and returns their output back to the console. We believe that the current implementation of RunCommand violates the Single Responsibility Principle because it has more than one responsibility.

Another issue would be how the classes are linked. According to the Dependency Inversion Principle, classes in the dependency hierarchy should depend on the abstractions instead of direct implementations. Instead of RunCommand directly calling BMIAnalyzer, it should call an Interface (Such as "IAnalyzer"), which the BMIAnalyzer will implement. In our current program, there are no abstraction layers, which is a violation to the DIP and needs to be handled.

Future plans:

First, we need to split up the responsibilities that RunCommand is currently taking on. One way to do that is to create a class called "Presenter", which will also be in the Interface Adapter layer. The Presenter's responsibility would be to retrieve the output from the Use Cases and return it to the Console. This resolves RunCommand's issue with the Single Responsibility Principle (SRP).

Next, we will create Layers of interfaces so that the high level classes won't directly depend on the lower level classes. This resolves our current issue with the DIP.

Finally, we are going to start implementing other functionalities (e.g Analyze Disease, Analyze Exercise, Create Meal plans, etc.) One of the biggest components we will need to work on is writing the data processing classes, as we have to process our current three datasets into

Entities. The most challenging functionality that we will be facing is the MealPlanGenerator, as it involves a large amount of data analyzing and we are working with a very large database (over 10000 entries.) We would also have to concrete an algorithm of forming reasonable combinations of foods objects into meals, taken into consideration of their nutrient level and calories.

Questions:

- Currently have no Interfaces in our program, would this be a problem?
- How are we going to implement the Presenter class? One of the possible methods that we came up with was to create an instance User variable inside the UserAnalyzer so that the use case stores the necessary information of the user, which can be retrieved by the Presenter. But the problem with this method is that we have to make sure that both RunCommand and Presenter have the same Use Case, or else the Presenter cannot get the data at all. If this is the case, then it would make the Presenter redundant because it's doing the same thing as RunCommand.
- Some of our use cases calculate new data and put them into the User object. This means that some of the instance attributes inside the User object (such as the BMI Hashmap) would depend on the use cases. Would this be a violation to the clean architecture? If so, how can we change this? Can we keep this implementation if it's well justified?

Member responsibilities

Everyone worked together and edited specifications, and CRC cards. With David, Winnie and Enid working more on the Progress report, and Jenny working on the Progress Report's powerpoint.

Classes that members wrote:

- David: Console, RunCommand, First Build of a successful scenario walkthrough run
- Jenny: API, Progress report presentation
- Winnie: User, Disease, TestBMIAnalyzer
- Enid: Exercise, Food, CreateUser, Console, Main, RunCommand
- Paul: RunCommand
- Naomi: BMIAnalyzer, README
- Yifan: UserAnalyzer, Main, minor contributions in written portions