Design Document

Specification (Updated):

Between Phase 0 and Phase 1, we have added a function where users can store their basic as well as personal health data in the program. All they need to do is to keep track of their unique ID, and they will be able to log back in and use the functionalities without entering all of their information again.

Given a user and their corresponding personal health data, our application can perform five functionalities: generate a meal plan, generate a list of workout moves, predict if the user is at a high risk of a certain disease, analyze BMI, and calculate EER.

The user will interact with the application in the Console. When the program runs, the user will be asked if they are an existing user. If the user is new, they will enter basic information, such as their name, age, and gender. Then, the user will be prompted to enter personal health data such as their height and weight. If they are an existing user, they will be prompted to enter their user ID. If their ID is correct (i.e. exists in the program database), unlike new users, they will not be prompted to enter their information.

After the profile for the user has been created or the user has logged in, the program will prompt the user to choose one of the following: Analyze BMI, Analyze EER, Analyze Workout, Analyze Disease, or Create Meal Plan. Each functionality will require some additional user input such as food and exercise preferences, and they will return different String outputs to the user.

1. The “Analyze BMI” function calculates the BMI of the user using the user’s height and weight, then returns the user’s BMI and classify it as underweight, healthy, overweight, or obese. .
2. The “Analyze EER” function uses the user’s height, weight, gender, age, and activity levels, then calculates the Energy Requirement per day, and returns the value to the user.
3. The “Generate Workout” function prompts the user to enter the major and minor muscles they want to focus on, and the equipment they have. It returns a list of exercises recommended for the user, as well as a brief description for each of the exercises. If no exercises match the user’s preferences, the user will have to try again.
4. The “Analyze Disease” function prompts a list of common symptoms for the user to check the ones they have, and return a list of possible diseases that they might have based on the symptoms selected.
5. The “Create Meal Plan” function prompts the user to enter their food preferences. The user will also be asked to enter the number of foods they would like suggested. The output meal plan will be a combination of different foods based on the user’s food preferences and the number of foods requested, and will look something like: “1) Apple + Cooked Lettuce + Fried Chicken Breasts + some nuts. 2) “Tomato and egg + Grapefruit + Steak”.

Design Document

The program uses three different databases to accomplish the functionality outlined above, a Food and Nutrition Database, an Exercise Database, and a Diseases Database. The program contains APIs that access those dataset and create corresponding Objects such as Disease, Exercise, and Food.

- Exercise Dataset: https://airtable.com/shrKZ9lPpw7EvjZ3X/tblvscpkbagqlWKkH
- Disease Dataset: https://www.kaggle.com/itachi9604/disease-symptom-description-dataset
- Food Dataset: https://tools.myfooddata.com/nutrition-facts-database-spreadsheet.php

SOLID:

Single-Responsibility Principle:

Our program adheres to the Single Responsibility Principle as all our classes have a clear purpose and they each have only one task. We addressed an issue we faced with the SRP in Phase 0 by creating separate controller classes to send and retrieve information to and from the use cases, through the creation of the Presenter class.

Open-Closed Principle:

One example of how our program adheres to the Open-Closed Principle is in our MealPlanGenerator functionality. In order to filter the food according to the user's different preferences, we have an abstract superclass called FoodCriterion. So far, FoodCriterion has four children classes, FoodIsLowFat, FoodIsLowCarbs, FoodIsLowSugar, and FoodIsVegetarian. According to the user's input, this four children class helps filter down our food objects to only those that satisfy all the criteria. If in the future, we want to write new conditions to filter the food, such as FoodIsHighProtein, we can easily make this extension by writing a new children class of FoodCriterion. Therefore, we followed this Open-Closed Principle as we made this feature easy for extension without modifying its source code, the abstract FoodCriterion.

Liskov Substitution Principle:

Again with the FoodCriterion hierarchy, every subclass of FoodCriterion (FoodIsLowCarbs, FoodIsLowFat, FoodIsLowSugar, etc.) can substitute for FoodCriterion. without breaking the program. Therefore we have no violations to this principle.

Interface Segregation Principle:

We followed the Interface Segregation Principle as we didn't force the client to implement interfaces that our program doesn't use, and our program is not forcing the client to depend on methods that it doesn't use. The UserAnalyzer interface is a good example, because every analyzer has to analyze something, and the method "analyze" enforces that action. Since there are no methods in UserAnalyzer that we don't use, we do not violate the ISP.

Dependency Inversion Principle:

Design Document

We are making sure that the classes between layers are all depending on abstractions. For example, when we are trying to analyze certain things about the user, we are depending on the interface UserAnalyzer instead of directly calling a specific analyzer to perform the task. Furthermore, the Use Cases are no longer directly manipulating the entities. Instead, they manipulate them through an interface. For example, the interfaces IUser, IFood, and IExercise are created as interfaces for the entities, User, Food, and Exercise.

There is still one violation of the Dependency Inversion Principle in our code, as currently our UserController is directly collaborating with the User Entity. This is a problem we will address and fix in phase 2.

Clean Architecture:

Using Scenario Walkthrough to demonstrate how our program follows the Clean Architecture:

In this walkthrough, we will illustrate how our program runs when an existing user log-in and chooses to update their profile by changing their username.

The Main class first runs, calling the UserParser (Data reader) to read the information stored in the userInfo.csv as existing user objects, and storing those user objects in the UserManager as existingUsers. Then HelperConsole (Driver) is called, prompting the user to choose whether they are an existing user or new user.

Once the user identifies themselves as an existing user, ExistingUserConsole (Driver) is called, and asks the user to enter their unique id. The ExistingUserConsole will check if the id is valid by calling UserController.checkUserExist(id) (Controller). The UserController calls UserManager (Use case) and sees if this id belongs to a user in the existingUsers. Once the id is identified to be valid, a RunCommand (Controller) is created and sets this logged-in user as the currentUser in UserManager (Use case).

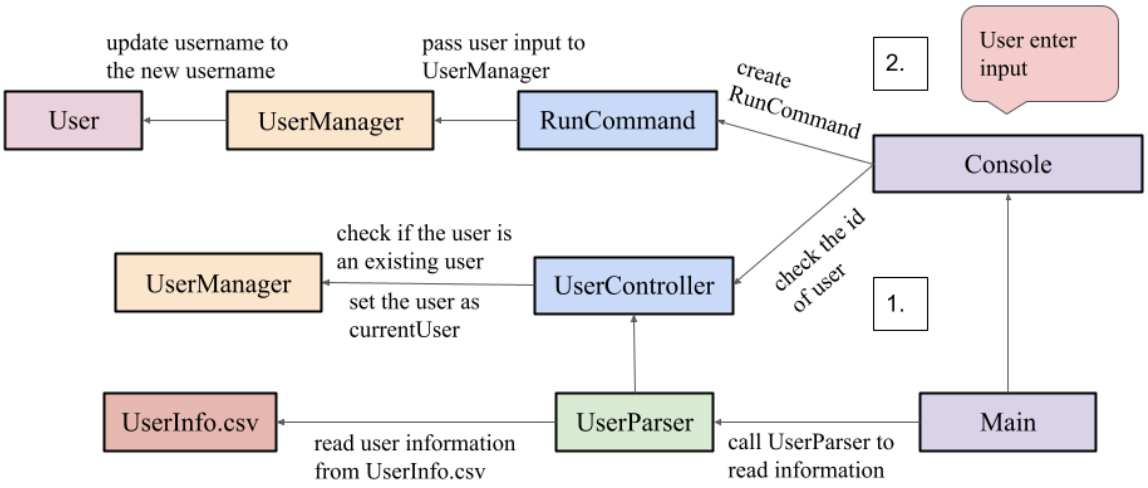
Then the console displays the existing user menu that allows the user to choose what they would like to do with the program. For this scenario walkthrough, we want the user to change their username, thus they will enter the command ‘6’ to change their profile, and then enter command ‘1’ to change their username specifically.

The helper function updateUser in the console is called and collects the new username the user wants to change to. The method RunCommand.ExecuteCommandUpdateInfo (Controller) will be called to execute this command, where it passes the new username along with the command ‘1’ to UserManager (Use Case). UserManager.changeUserInfo will set the currentUser’s username to the new username. The RunCommand.ExecuteCommand-UpdateInfo will then return a message to the console, letting the user know the information has been updated.

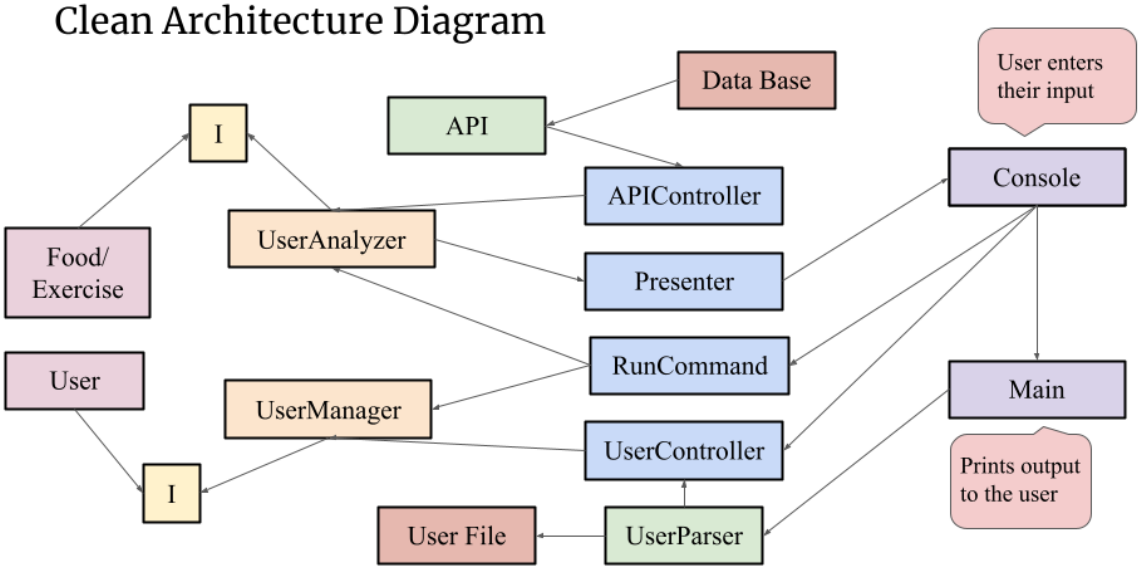
Design Document

The only problem that still persists is that there are 2 controllers (Interface Adapter layer) that need to use Entity Classes (Entity layer), and this is a violation to the Clean Architecture (skipping layers). We will be addressing this issue during phase 2.

Scenario Walkthrough



Clean Architecture Diagram



Design Patterns:

Factory Design Pattern:

As we know, the Factory Design Pattern is best used for cases where there are many subclasses that can be initialized all under a parent class. It creates a "Factory" that will initialize and return the appropriate subclass based on an inputted value. This removes the

need of having excessive amounts of “new” statements and provides a cleaner and smoother way to initialize new objects of the subclasses.

This design pattern was implemented for the subclasses of the UserAnalyzer, with the corresponding pull request being #62 on Github. UserAnalyzer has multiple subclasses: BMIAnalyzer, EERAnalyzer, ExerciseAnalyzer, DiseaseAnalyzer, and MealPlanGenerator. They each correspond to a functionality of our program, and take on the integers from 1-5 respectively. A “Factory” class, UserAnalyzerFactory, was created. This class takes in an integer called command and outputs the appropriate UserAnalyzer subclass based on it. For example, a command value of “1” would properly initialize a new BMIAnalyzer. UserAnalyzerFactory is an extremely important class used in RunCommand, which will initialize and then use these subclasses for other functionalities.

Template Method Design Pattern:

The Template pattern utilizes an abstract class that defines the “template” (a skeleton of operations) of an algorithm. Its subclasses can then override its methods to create different concrete implementations that conform to the template.

This pattern is implemented in our project (see pull request #44 on Github). FoodFilterCriterion is the abstract parent class with the method isSatisfiedBy(), and its subclasses FoodIsLowCarbs, FoodIsLowFat, FoodIsLowSugar, and FoodIsVegetarian override this method to define different concrete boolean algorithms used to filter Food objects. These boolean algorithms are used in the method getFoodByCriteria in the FoodManager class to filter a list of Food objects passed from APIController.

The Template Method Pattern conforms to the Open-Closed Principle. It allows our program to be open to the extension of new filter algorithms for Food objects while no modification of existing code in our program will be needed.

Use of GitHub:

During Phase 0 and Phase 1, our group has collectively created over 50 Pull Requests, 50% of which have been reviewed on Github. Those that don't have a review have all been reviewed in-person when we worked together. We only have 1 issue post but it has been properly addressed. We also had several merge conflicts, where a pull request was not able to be merged automatically. This was fixed using the command line, and the pull request was successfully merged, with no issues in the code. Each member of the group also created their own branch, and made changes on their own branch before merging on to the main branch. This all demonstrates our group’s effort of using various features of GitHub to facilitate development of our code.

Code Style and Documentation:

Design Document

- Every Public method that we have is properly documented, and every class is also documented.
- We added comments to our code to make it easier to understand and read.
- As for the coding style, we've eliminated unnecessary imports and unused variables, and fixed as many style errors as we can.
- Packaged our files according to their layers in the Clean Architecture

Testing:

- We have written at least 1 test case for each of the functionality classes (BMIAnalyzer, EERAnalyzer, etc.).
- We also wrote a test case to test our UserParser, making sure the reading and writing of the file into userInfo.csv is working as desired. We created a separate csv file just for this test.
- One thing we can improve on is to write more complex test cases for phase 2, as the current test cases are all fairly simple, and are mostly testing the basic features of the program.

Refactoring:

Here are some of the things we've done that provided evidence that our team has refactored code in a meaningful way:

- We split out a Presenter class from RunCommand to satisfy SRP. As previously in phase 0, RunCommand was responsible for putting the input into and retrieving the output from the use cases.
- We created helper methods and a Constant class to clean up our code.
- We broke down Console into several small Classes to eliminate the bloater code smell. (Previously there were more than 500 lines in total in Console). Now Console is split up into HelperConsole, HelperUserInfo, NewUserConsole, and ExistingUserConsole.
- We deleted the API interface and the Disease class because they are not too helpful/necessary to the program.

Code Organization:

We decided to package our code by layers. This way we can see how our code abides by the Clean Architecture, and to see if there are any dependency violations or if our high-level classes are skipping layers (e.g. Interface Adapter layer trying to access an entity). This packaging makes it easy to find a specific file, and less effort is needed when we are navigating through the files. This is because after we created our CRC model, we became familiar with which file corresponds to which layer of our design.

Design Document

Functionality:

Click here to access a screen recording of how our program runs

Above is a video showing:

- Creating a new user profile and exploring all the different functionalities as the user
- The user logs out and logs back in with its personal id
- The user chooses to change it username after logging back in

Progress Report:

Open questions we are struggling with:

- We have a few style errors indicated by IntelliJ that we can't figure out how to address, including "Raw use of parameterized class" and "unchecked cast". Would it be fine to leave them as is or is there a way to fix them?

What has worked so far with our design:

- With one exception, the design patterns and packaging of our code has helped us generally adhere well to the SOLID principles, and the clean architecture layers.
- UserAnalyzerFactory implements the Factory pattern.
- Creating abstraction layers such as IUser, IFood, IExercise as well as UserAnalyzer allowed us to satisfy the Dependency Inversion Principle.

Member contributions:

All members contributed to the project in the time between Phase 0 and Phase 1. Since everyone just worked on whatever they found needed the most work, it's impossible to say what exactly everyone did, but here is a list of group members and some of their major contributions.

- David: Bulk code for APIs and Analyzers, fixed style errors, wrote interfaces for Entities, wrote first draft for Design Document.
- Jenny: DiseaseAnalyzer, Check for New user or existing User (console).
- Enid and Winnie: Worked on UserParser, allowed the Program to save state between runs. Connected the analyzers (use cases) with the console, gathered desired inputs from the user and sent them to the corresponding use cases. Created a log-in menu that allows users to go back to the main page without existing the program. Commented and documented most of the code.
- Paul: Wrote MealPlanGenerator, FoodManager and related classes (and unit tests for them). Wrote Template Method design pattern. Minor code edits/fixes in other classes.
- Naomi: packaging, Constants class, refactoring and code cleanup, updating CRC cards and design document
- Yifan: ExerciseAnalyzer, UserAnalyzerFactory/Factory design pattern, minor code cleanups in Console