Design Document

<u>Specification (Updated):</u>

<u>Changes between Phase 1 and Phase 2:</u>

Between Phase 1 and Phase 2, no new functionality of the program is implemented, the biggest change is that we implemented a GUI. The program is now much more user-friendly and visually appealing as the user does not have to interact with the program using the console anymore. Instead, using Java Swing, the user now can interact with a pop-up window, and interact with the program using easy-to-use checkboxes and buttons.

<u>Improvements based on the Feedbacks given in Phase 1:</u>

1. Modified Presenter class, so it now stores every single output that needs to be displayed onto the GUI screen. The GUI classes (Driver) calls Presenter (Controller) to retrieve the output, and it sets the String retrieved as text on its Jtextpane.
2. UserController does not directly interact with the entity User now, it now insteads interacts with the entity's interface (i.e. IUser)
3. Refactored Constant Class into different mores specific Constant Classes to avoid a bloater class
4. Package Names are all refactored into smaller cases
5. Wrote many new tests with meaningful names, added more documentations to explain what each of the cases of the tests are doing. Test Coverage improved.

<u>Functionalities of the program (Implemented all in Phase 1):</u>

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.

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.

Design Document

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.

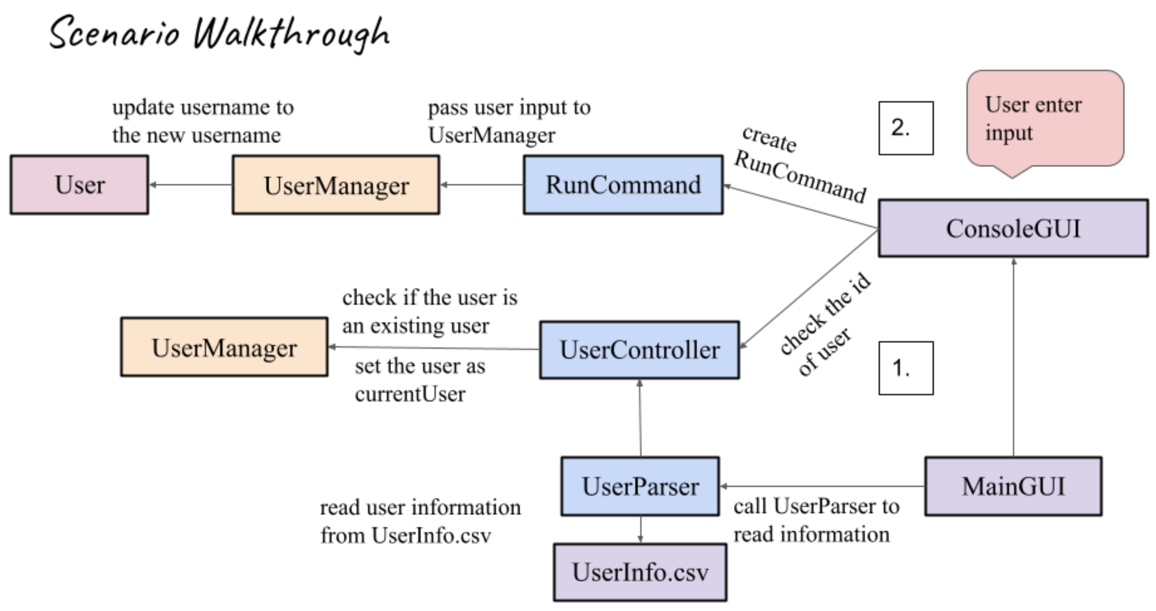
Dependency Inversion Principle:

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:


Scenario Walkthrough
update username to the new username
pass user input to UserManager
create RunCommand
2.
User enter input
User
UserManager
RunCommand
ConsoleGUI
check if the user is an existing user
set the user as currentUser
UserManager
UserController
check the id of user
1.
UserParser
MainGUI
read user information from UserInfo.csv
call UserParser to read information
UserInfo.csv

Design Document

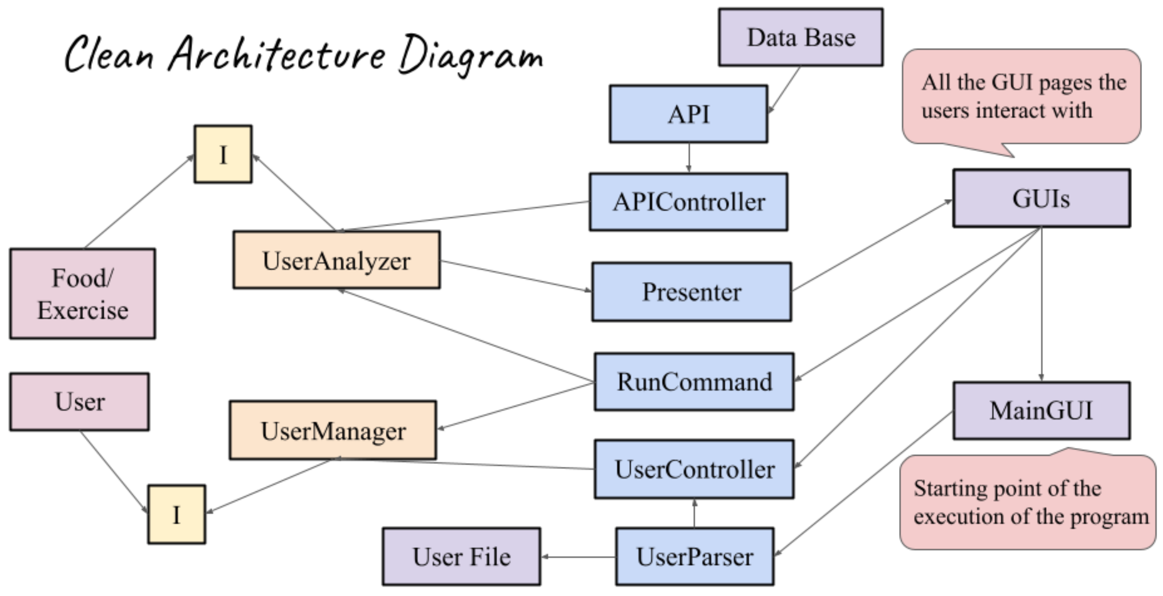
In this walkthrough, we will illustrate how our program runs when an existing user log-in and chooses to update their profile by changing their username.

The MainGUI 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 ConsoleGUI (Driver) is called, popping out a new window that allows the user to either click on a button that reads “Create New User”, or enter their ID to log-in as an existing user.

If the user identifies themselves as an existing user, ExistingUserConsole (helper class in Driver) is called and checks 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 a new window will display, showing 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 press the button “6. Edit Profile”, and change the first textfield to the new name, and press the “Update” Button. Once the “Update” button is activated, the helper function updateUser is called and collects the new username the user wants to change to from the textfield.

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 system will then prompt out a message, letting the user know the information has been updated.



Design Document

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:

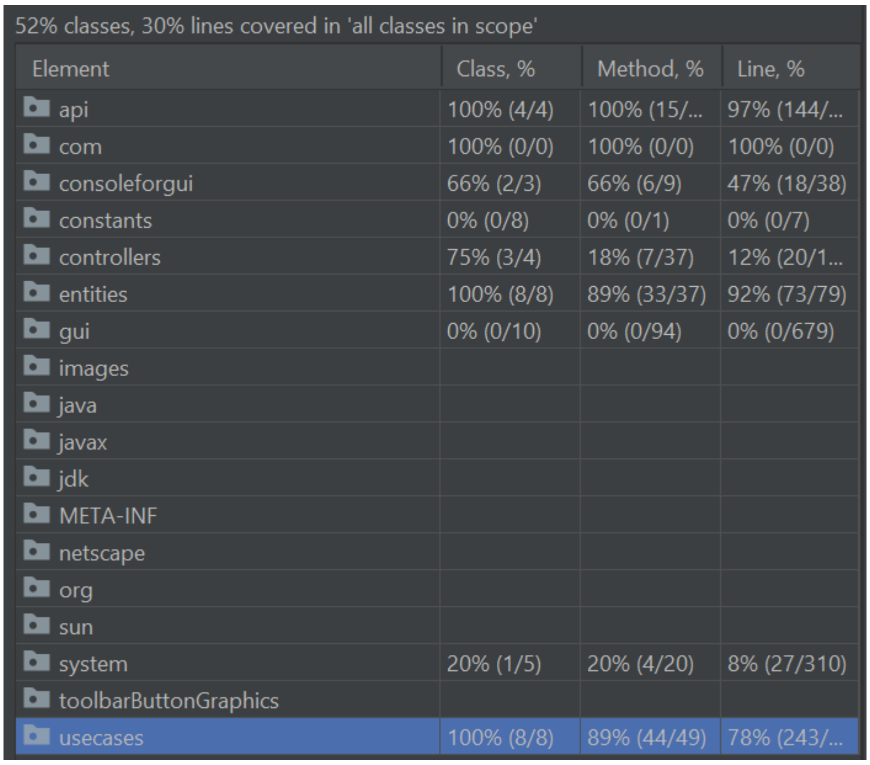
During Phase 1 and Phase 2, our group has collectively created over 36 Pull Requests, with almost all of them being reviewed and commented on by members of the group. Those that don't have a review have all been reviewed in-person when we worked together. We also thoroughly used the Issue feature of Github, as we posted 10 issues over the past phase, identifying the issue as either ‘bug’, ‘enhancement’, or ‘question’. All the issues have been properly addressed in our code and resolved. 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 improved effort of using various features of GitHub to facilitate development of our code during phase 2.

Code Style and Documentation:

- 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.
- Added documentation to all classes and most class variables.

Testing:

- Use cases are thoroughly tested.
- Entity classes are thoroughly tested.
- APIs for databases are thoroughly tested.
- Relevant methods in the consoleforgui package are mainly tested.
- Relevant methods in the controllers package are mainly tested (note that the Presenter class due to its nature is not suited for testing).
- system package not tested in depth as the package is deemed deprecated.
- gui package not included in the testing scheme as the classes primarily deal with creation of gui elements.



52% classes, 30% lines covered in 'all classes in scope'

Element	Class, %	Method, %	Line, %
api	100% (4/4)	100% (15/...	97% (144/...
com	100% (0/0)	100% (0/0)	100% (0/0)
consoleforgui	66% (2/3)	66% (6/9)	47% (18/38)
constants	0% (0/8)	0% (0/1)	0% (0/7)
controllers	75% (3/4)	18% (7/37)	12% (20/1...
entities	100% (8/8)	89% (33/37)	92% (73/79)
gui	0% (0/10)	0% (0/94)	0% (0/679)
images			
java			
javax			
jdk			
META-INF			
netscape			
org			
sun			
system	20% (1/5)	20% (4/20)	8% (27/310)
toolbarButtonGraphics			
usecases	100% (8/8)	89% (44/49)	78% (243/...

(latest as of Dec 5, 2021)

Design Document

Refactoring:

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

Phase 2:

- Refactored the Constant class by breaking them down into many smaller classes
- Refactored all Package names into lower case so they are not confused with the classes name.
- Changed all the places that used the Food and User entity to use the interface IFood and IUser instead.
- Refactored the Console classes as many of the functions are not needed any more, as we are no longer using the terminal to take input from/output to the user. Deleted all Scanners and System.out.println from the classes.
- Refactored a method that was too long in UserParser (writeIntoFile).
-

Phase 1:

- 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.

Functionality:

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:

A brief summary of what each group member has been working on since phase 1:

1. Enid: The first to set up the GUI by writing MainGUI.java and ConsoleGUI.java to replace the previous Main.java file. Wrote the first page the user sees when they run the program, help connect the previous Console and analyzers with the current GUI, and also setted up the other GUI file for everyone else to work on.

   David: Created EERPromptGUI and BMIPromptGUI, refactored Constants class and broke it down into smaller groups. Documented all the GUI classes and related instance variables and code. Contributed on Accessibility Document.

   Yifan: Unified/modified the formatting and aesthetics for all of the GUIs.

   Winnie: Created ExercisePreferenceGUI, EditProfile GUI and NewUserLoginGUI. Helped modify the Presenter class so it follows the clean architecture. Connected the previous Console and analyzers with the current GUI.

   Paul: Wrote additional unit tests for methods not covered in phase 1. Refractored consoleforgui.

Each group member should include a link to a significant pull request (or two if you can't pick just one) that they made throughout the term. Include a sentence or two explaining why you think this demonstrates a significant contribution to the team.

- David:
  - https://github.com/CSC207-UofT/course-project-dj-wepny/pull/35
    - Created ExerciseAPI and manually wrangled the ExerciseMovesData Into the CSV we have now. ExerciseAPI was used for ExerciseAnalyzer and the API was barely changed.
  - https://github.com/CSC207-UofT/course-project-dj-wepny/pull/69
    - Created interfaces to make sure that we satisfy the Dependency Inversion Principle. Edited most of the codes in the Use Cases that originally takes in a concrete Entity object before
- Enid:
  - https://github.com/CSC207-UofT/course-project-dj-wepny/pull/87.
    - This pull request demonstrates contribution to the team as it sets up the basis of the GUI for everyone else to work on during phase 2. It also connects the GUI (driver) to the controller layer of the program.
  - https://github.com/CSC207-UofT/course-project-dj-wepny/pull/63/files.

    - Wrote documentations and comments for all the classes and code. Factored Console classes to avoid code smell, and refactored all the variable names into camelCase.
- Winnie
  - https://github.com/CSC207-UofT/course-project-dj-wepny/pull/95
    - Created GUI for Exercise Analyzer
  - https://github.com/CSC207-UofT/course-project-dj-wepny/pull/99
    - Linked write/update user information to the log out button on the user menu page so that user information is saved to the csv file when they log out.
    - Created GUI for Edit Profile.
- Naomi
  - https://github.com/CSC207-UofT/course-project-dj-wepny/pull/96
    - created GUI for MealPlanGenerator.
  - https://github.com/CSC207-UofT/course-project-dj-wepny/pull/60/files
    - packaged all the classes and created a Constants class
- Paul
  - https://github.com/CSC207-UofT/course-project-dj-wepny/pull/44
    - Created FoodManager class and related classes implementing the Template design pattern that conforms to the Open-Closed Principle.
  - https://github.com/CSC207-UofT/course-project-dj-wepny/pull/70
    - Wrote tests for FoodManager and MealPlanGenerator that revealed major bug in code, and subsequently fixed it.
- Yifan
  -