

# HEALTH-N-CARE APP / RELEASE 2

## Project Design Document

### TEAM 3 Charlie

Zaher Asad <[zsa4432@rit.edu](mailto:zsa4432@rit.edu)>

Ethan Gapay <[etg5588@rit.edu](mailto:etg5588@rit.edu)>


Allan Flores <[arf7094@rit.edu](mailto:arf7094@rit.edu)>

Alfred Franco Salgado <[af8857@rit.edu](mailto:af8857@rit.edu)>

Alex Tedesco <[act2076@rit.edu](mailto:act2076@rit.edu)>

Nolan J Wira <[njw1389@rit.edu](mailto:njw1389@rit.edu)>

*Google Doc link to document with version history:*

 Design Documentation.docx

## 1 Project Summary

The HealthNCare App is developed for a Wellness Consortium to assist users in monitoring and improving their daily lives. The app allows users to track their food intake, weight, daily calorie goals, exercises, and calories expended during workouts. Users can add essential foods or recipes, log their daily food consumption, and view nutritional summaries with the calories depleted during exercise automatically subtracted from their net calories for the day. The app, which follows the MVC and Composite patterns, provides the user with a simple interface.

Key Features:

- The ability to add/define basic foods with nutrient information.
- The ability to add/define recipes composed of basic foods or other recipes.
- The ability to log a specific food intake with a specific quantity and date attached.
- The ability to view your total daily nutrition intake, along with basic recommendations.
- The program also provides a way to record exercises.
- The calories expended on exercise are deducted from the calories consumed in order to compute net calories per day.
- The user is able to add to or edit the collection of exercises provided.

## 2 Design Overview

The development of an application designed to keep track of daily food intake & exercises and calories depleted must carefully consider the scalability and ease of use available to the user. The goal is to create a program that can be personalized to a person's general diet and remain usable on a daily basis. This stresses the importance of modular code and a reasonably structured back-end. Naturally, our ideas and implementations towards achieving this goal will change over time and this section is tasked with recording those changes and their rationales.

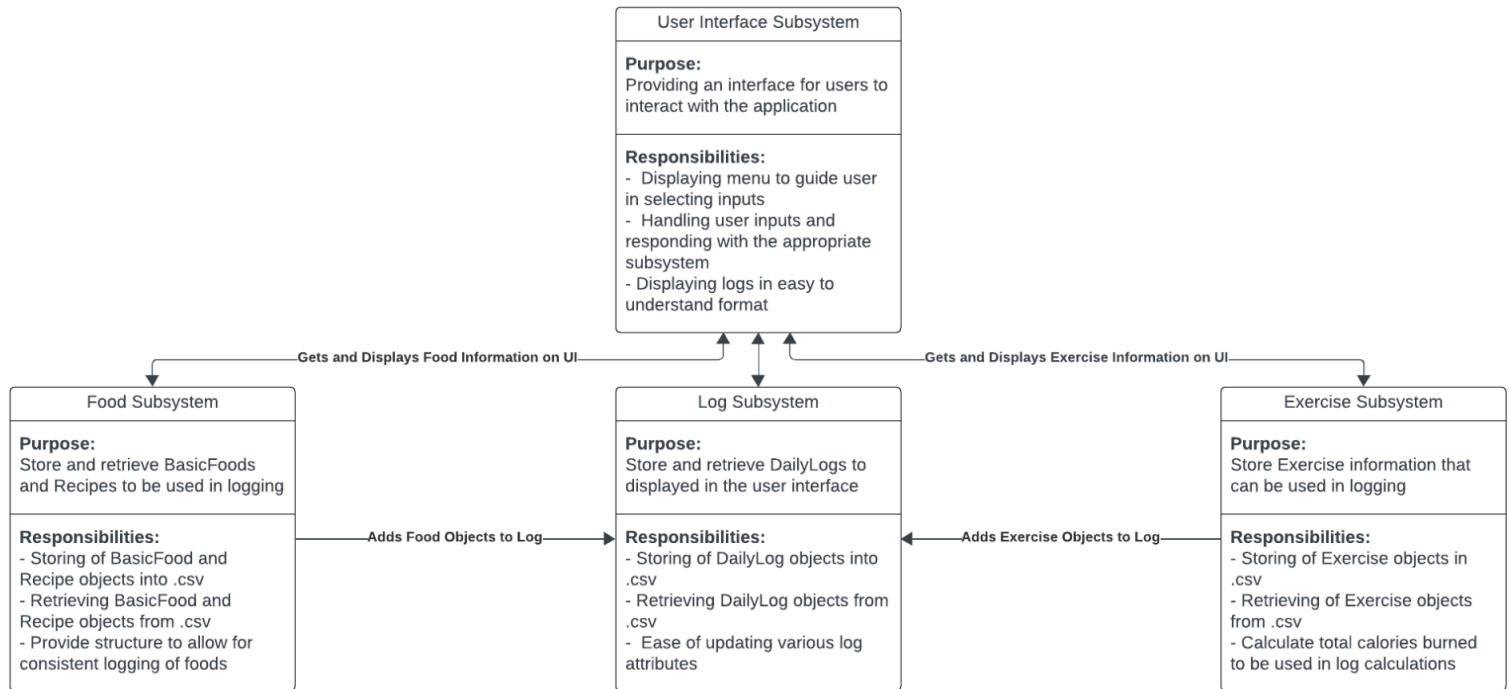
Working on the design sketch was the first instance of actual critical ideation into designing the implementation of our health app. With some guidance from the slides and descriptions of the app's inner workings, we extrapolated our first design. At the top level, we intended to implement a Model-View-Controller (MVC) architecture, which should function well for our needs as it separates the data from the behavior put upon it.

While contemplating our design, we identified the 4 main areas our system needed to be divided into: the UI, Food management (adding/tracking foods), Exercises

management (Calories Depleted during Exercises), and data management (saving and retrieving log data). We decided to divide the last area into two separate subsystems: Log Subsystem and Exercise Subsystem, to keep them focused and cohesive. In our current design implementation, the Food Subsystem is considered separate from the controller that manages it, along with the Log Subsystem is separate from the log controller, and the Exercise Subsystem is separate from the exercise controller. We chose to visualize it as such since this helps us support our separation of concerns.

With the knowledge that the user needed to be able to add foods with nutrient information at a base level and be able to use those foods to create recipes, which can then be used in other recipes, we decided to use the composite pattern as it fulfilled our scalability requirements. With this structure, we should be able to compose foods from other foods easily and treat them similarly when tracking them. We intend to use observers to update the UI, to reduce coupling between systems.

### 3 Subsystem Structure



## 4 Subsystems

### 4.1 User Interface Subsystem

<b>Class</b> SwingUI	
<b>Responsibilities</b>	Display menu with prompts to guide users in selecting inputs Handle user inputs and responding with the food, log, or exercise subsystems Display the logs in an easy to understand format

### 4.2 Log Subsystem

<b>Class</b> LogController	
<b>Responsibilities</b>	Storing/Overriding of DailyLog objects into .csv Retrieving DailyLog objects from .csv Ease of updating various log attributes

<b>Class</b> DailyLog	
<b>Responsibilities</b>	Load and save food collection from/to CSV files. Calculate total calories and nutrient breakdown for each day. Determine if calorie intake exceeds set limit.

### 4.3 Food Subsystem

<b>Interface</b> Food	
<b>Responsibilities</b>	Outline methods to get basic food and recipe data. Ensure uniqueness and consistency of food and recipe data.

<b>Class</b> BasicFood	
<b>Responsibilities</b>	Represents a basic food item. Includes calories, fat, carbs, protein, sodium. Handles reading of the CSV of basic food.
<b>Collaborators (implements)</b>	Food

Class Recipe	
<b>Responsibilities</b>	Represents a recipe item. Can be composed of different BasicFood and Recipe. Handles reading of the CSV of recipes.
<b>Collaborators (implements)</b>	Food

Class FoodController	
<b>Responsibilities</b>	Controls functionality coming from both View and Model regarding add/removing Food data from CSV. Takes input from view and updates CSV. Gets information back from BasicFood/Recipe objects.
<b>Collaborators (Uses)</b>	BasicFood, Recipe, UserInterface

#### 4.4 Exercise Subsystem

Class Exercise	
<b>Responsibilities</b>	Represents an exercise. Handles reading of the CSV of exercises.
<b>Collaborators (used in)</b>	ExerciseController

Class ExerciseController	
<b>Responsibilities</b>	Controls functionality coming from both View and Model regarding adding/removing exercises. Takes input from view and updates CSV. Gets information back from an Exercise object.
<b>Collaborators (Uses)</b>	Exercise

## 5 Sequence Diagrams

Scenario: The scenario described is that the user adds to or edits the collection of exercises provided.

Components:

- UserInterface
- ExerciseController
- Exercise

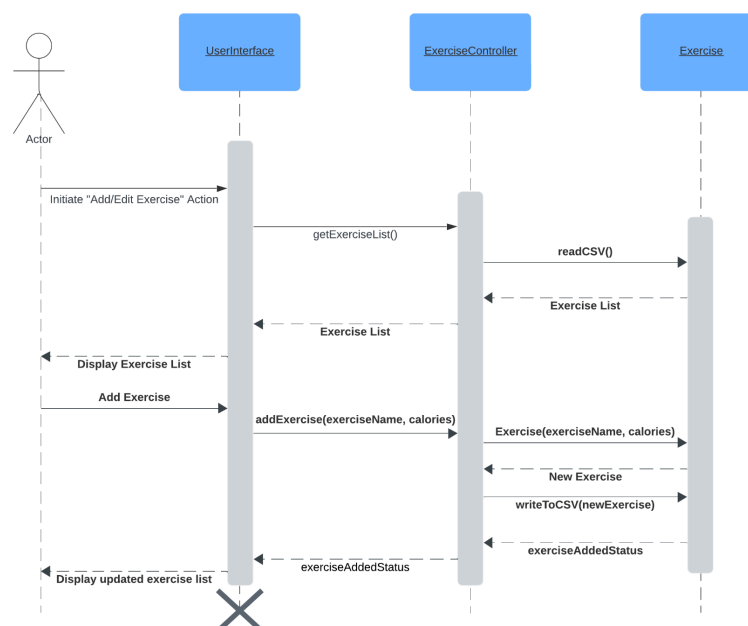
Features: The key features illustrated in the sequence diagram are:

1. The Actor initiates the "Add/Edit Exercise" action through the UserInterface.
2. The UserInterface calls `getExerciseList()` on the ExerciseController to retrieve the current exercise list.
3. The ExerciseController reads the exercise data from a CSV file using `readCSV()`.
4. The exercise list is displayed to the Actor via the UserInterface.
5. The Actor can choose to add a new exercise.
6. The UserInterface calls `addExercise(exerciseName, calories)` on the ExerciseController to add the new exercise.
7. The ExerciseController creates a new Exercise object with the provided details.
8. The new exercise is written to the CSV file using `writeToCSV(newExercise)`.
9. The exercise added status is returned to the UserInterface.
10. The updated exercise list is displayed to the Actor.

Operation: The overall operation being depicted is the following:

1. Retrieve the current list of exercises from a data source (CSV file).
2. Display the exercise list to the user.
3. Allow the user to add a new exercise or edit an existing one.
4. Update the data source with the new or modified exercise.
5. Display the updated exercise list to the user.

In summary, this sequence diagram illustrates the steps involved in allowing a user to manage a collection of exercises. The user can view the current list, add new exercises, and the changes are persisted to a data source (CSV file) and reflected in the updated exercise list displayed to the user.



Scenario: The scenario described is subtracting calories depleted from calories consumed in order to compute net calories per day.

Components:

- UserInterface
- LogController
- DailyLog
- ExerciseController
- Exercise

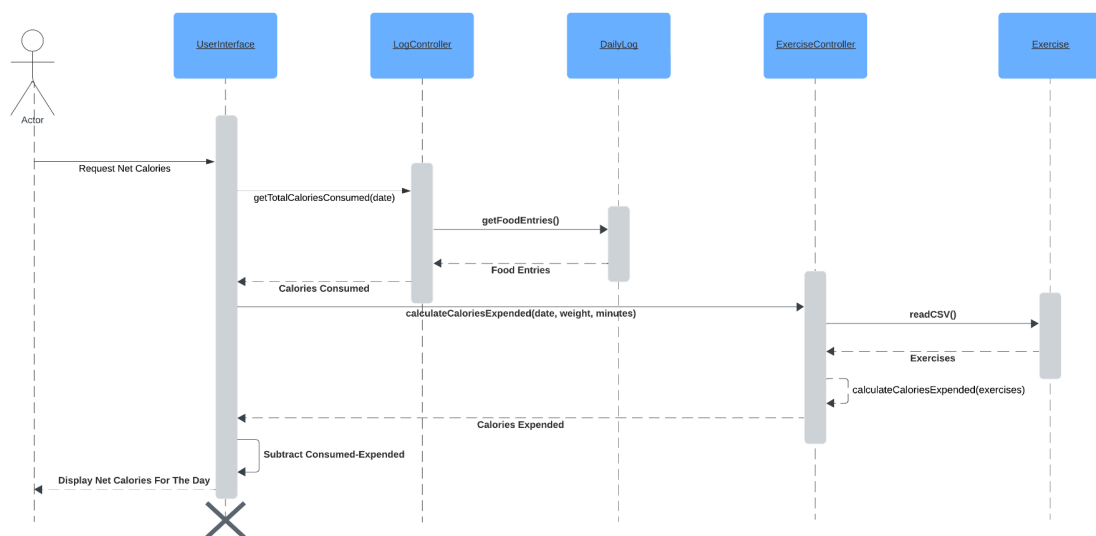
Features: The key features illustrated in the sequence diagram are:

1. The UserInterface calls `getTotalCaloriesConsumed(date)` on the LogController to retrieve the total calories consumed.
2. The LogController calls `getFoodEntries()` on the DailyLog to get the food entries and calculate the calories consumed.
3. The UserInterface calls `calculateCaloriesExpended(date, weight, minutes)` on the ExerciseController to get the calories expended.
4. The ExerciseController reads exercise data from a CSV file using `readCSV()` and calculates the calories expended based on the exercises.
5. The calories expended is returned to the LogController.
6. The LogController subtracts the calories expended from the calories consumed to compute the net calories.
7. The net calories for the day are displayed to the User via the UserInterface.

Operation: The overall operation being depicted is the following:

1. Retrieve the total calories consumed for the day from food entries.
2. Retrieve the calories expended for the day from exercise data.
3. Subtract the calories expended from the calories consumed.
4. Display the net calories for the day to the user.

In summary, this sequence diagram illustrates the steps involved in calculating the net calorie intake for a day by retrieving food and exercise data, computing calories consumed and expended, and then subtracting the expended calories from the consumed calories to get the net value.





Scenario: The scenario described is recording exercises performed by the user.

Components:

- `UserInterface`
- `ExerciseController`
- `Exercise`
- `LogController`
- `ExerciseEntry`
- `DailyLog`

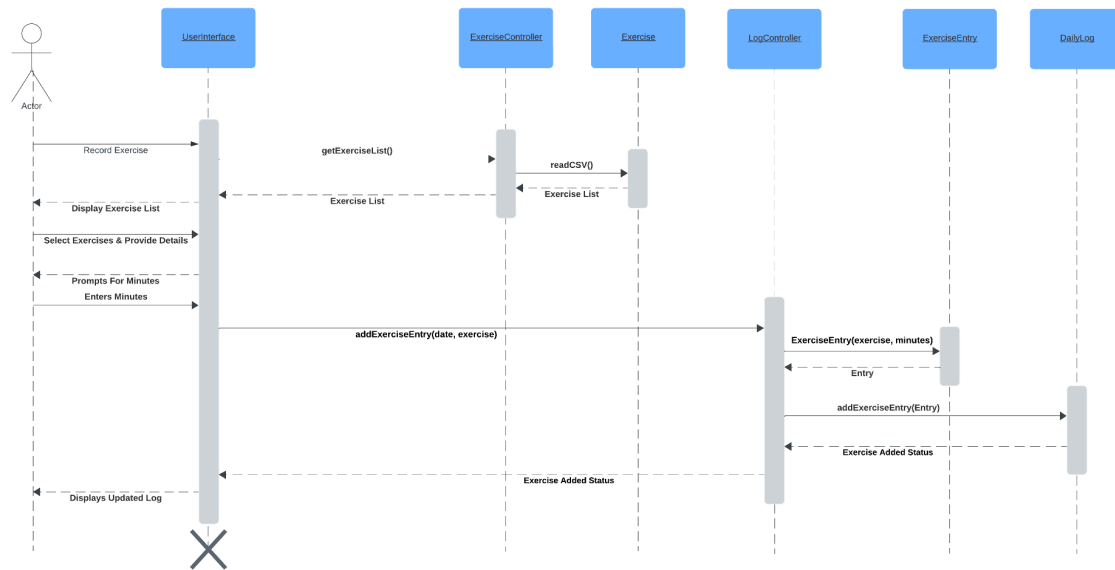
Features: The key features illustrated in the sequence diagram are:

1. The Actor initiates the "Record Exercise" action through the `UserInterface`.
2. The `UserInterface` calls `getExerciseList()` on the `ExerciseController` to retrieve the list of available exercises.
3. The `ExerciseController` reads the exercise data from a CSV file using `readCSV()`.
4. The exercise list is displayed to the Actor via the `UserInterface`.
5. The Actor selects an exercise and provides the duration (minutes).
6. For the selected exercise, the `UserInterface` calls `addExerciseEntry(date, exercise)` on the `LogController`.
7. The `LogController` creates a new `ExerciseEntry` object with the exercise and duration details.
8. The `ExerciseEntry` is added to the `DailyLog` by calling `addExerciseEntry(Entry)`.
9. The exercise added status is returned to the `LogController` and then to the `UserInterface`.
10. The updated daily log, including the new exercise entries, is displayed to the Actor.

Operation: The overall operation being depicted is the following:

1. Retrieve the list of available exercises from a data source (CSV file).
2. Display the exercise list to the user.
3. Allow the user to select an exercise and provide the duration.
4. Create an entry for the selected exercise with the provided duration.
5. Add the exercise entry to the daily log.
6. Display the updated daily log, including the newly added exercise entries, to the user.

In summary, this sequence diagram illustrates the steps involved in allowing a user to record the exercises they have performed. The user can select from a list of available exercises, provide the duration for each, and the exercise entries are then added to the daily log and displayed to the user.



## 6 Pattern Usage

### 6.1 Pattern #4

Composite Pattern	
Client	SwingUI
Component	Food
Composite	Recipe
Leaf	BasicFood

## 7 RATIONALE

Design Decisions:

### Scenario 1,2,& 3 Sequence Diagrams:

The sequence diagrams demonstrate a Model-View-Controller (MVC) architecture pattern across different scenarios. In the first diagram, the controller allows the user to add or edit the collection of exercises provided, retrieving the current exercise list from a data source (CSV file), displaying it to the user, and persisting any new or modified exercises back to the data source. The second diagram shows the controller calculating the net calories for the day by retrieving food entries to compute calories consumed and exercise data to compute calories expended, then subtracting the expended calories from the consumed calories. The third diagram depicts the controller interacting with the user to get the list of available exercises and then recording the exercises performed by the user, including duration details.

The MVC model prevents the user from directly interacting with the data stored in CSV files, such as food databases or exercise lists, which enhances the application's security. The controller acts as an intermediary, handling user inputs and interactions, while the models (e.g., BasicFood, Recipe, DailyLog, Exercise) manage the underlying data. The views (e.g., UserInterface) display information to the user based on the model data processed by the controller.

### Pattern Usages:

- **MVC Pattern:** This pattern allows us to have three separate components that all play a part in the overall execution of the application. By separating these responsibilities, a more organized and maintainable system is built. The MVC pattern gives us the opportunity to scale our application effectively, taking changes into account and additions to functionality with minimal impact on other parts of the system.

### Subsystems:

- **User Interface:** Main component users will be interacting with. Users are able to control program functionality through the interface as well as getting information back from the system.
  - (4/12/2024) We intend to split up each of the subsystems into their own display views, Food and Logs
  - (4/16/2024) Adding Exercise to have its own display view as well
  - (4/20/2024) We decided to have all the functionality in one interface that the user can interact with.
- **Food:** System that handles all of the food related functionality in terms of reading and adding to the foods.csv file.
  - (4/11/2024) We decided to have Recipe, Basic Foods, and FoodController be the main working parts for this as they most closely work together.
  - (4/16/2024) We decided to move reading functionality to the food classes (Recipe and BasicFood) and have the controller (FoodController) write to the csv. We are practicing

- separation of concerns as the model deals with reading the data while the controller handles manipulating it.
- (4/20/2024) After implementing the functionality to the SwingUI, we realized that LogController needs to FoodController to function, so we edited the FoodController class to work better with LogController
  - **Log:** System to handle the logging aspect of the program. Users are able to add logs, add foods/exercises to those logs, add calorie limit for each log and weight for each log.
    - (4/12/2024) Due to a previous meeting, the log team convinces the whole team to move log.csv reading and writing to LogController because it made more logical sense to have LogController maintain and keep all of the logs together in one class rather than separate it for ease of updating the log.csv database
    - (4/16/2024) Added methods to LogController that would add Exercise functionality to it (ie adding an exercise to DailyLog, removing an exercise, calculating calories expended, etc). Also fixed certain methods so they work with SwingUI instead of CLI.
    - (4/18/2024) Implementation of Exercise subsystem led us to do all calorie calculations in LogController because it is where a user's daily logs and the calories are all going to be
  - **Exercise:** System to handle exercise portion of the program. Users will be able to add and remove exercises. Allows the program to calculate the total number of burned calories.
    - (4/16/2024) Along with the other exercise information, we also intend to store a DailyLog object since they relate to each other.
    - (4/18/2024) We decided that it does not need to store a DailyLog object as DailyLog would be storing the Exercise object.
    - (4/19/2024) We decided that Exercise will not have to store the total number of calories or time exercised as that will be determined at the ExerciseController.
    - (4/20/2024) Seeing the need for exercises to be logged, we took a similar approach with the Food subsystem in regards to interaction between the ExerciseController and LogController to ensure functionality between the two.