PROJECT / RELEASE 1

Project Design Document

TEAM A Black Coffee
Adam Abramson <aea8656@rit.edu>
Cooper Gadd <ctg7866@rit.edu>
V.J. Goh <mcg4527@rit.edu>
Matt Grober <meg2746@rit.edu>
Andrew Tark <ayt1844@rit.edu>

Date	Changes
3/7	Initial Commit
3/19	Updated Subsystems -Matt
4/1	Final Implementation
4/21	Project summary and subsystems updated
4/23	Phase 2 Update

1 Project Summary - Matt

Created a program allowing users to manage a repository of their exercises, basic foods, and recipes, this data can be displayed through either the command line or a graphical user interface. Users can contribute by adding their own food items or exercises, specifying details such as name, calories, fat, carbohydrates, protein, and sodium. Recipes can be used as ingredients in other recipes.

Users log their daily food consumption, recording servings of each item. The program calculates and displays daily nutrient totals, featuring graphs depicting the distribution of fat, carbs, and protein. For instance, if a user logs only bread with 5g of carbs and 5g of protein, the graph shows 0% fat, 50% carbs, and 50% protein.

Logs may include additional details like weight or calorie targets. The program informs users about their calorie goal status, aiding in weight tracking. The system ensures data retention, encompassing daily exercise, food collections, recipes, daily consumption, calorie targets, and recorded weights.

2 Design Overview - Cooper

This project uses separation of concerns. models, views, controllers, and services all handle their own purpose. The models contain the object that will be used within the application. The views contain the different user interfaces that allow a user to interact with the application. The controllers contain the actions that control the application. The services contain the communication between the application and the csv files.

This project uses high cohesion since each file has its own specific task. For example, DataUtil is the only file that directly communicates with the data.

This project uses low coupling. For example, the services will use a utility to perform CRUD functions. The services don’t need to know how to save a record.

This project uses support for extendibility. For example, the view folder allows another type of interface for the user. If we would like to add another interface, like graphical user interface, we can without the program breaking.

This project uses MVC. Inside the source code, there’s folders for the models, views, and controllers.

This project uses the composite pattern. The user can log that they ate either a food or recipe which is a collection of food.

This project uses the facade pattern. The facade encapsulates the implementation and the interface.

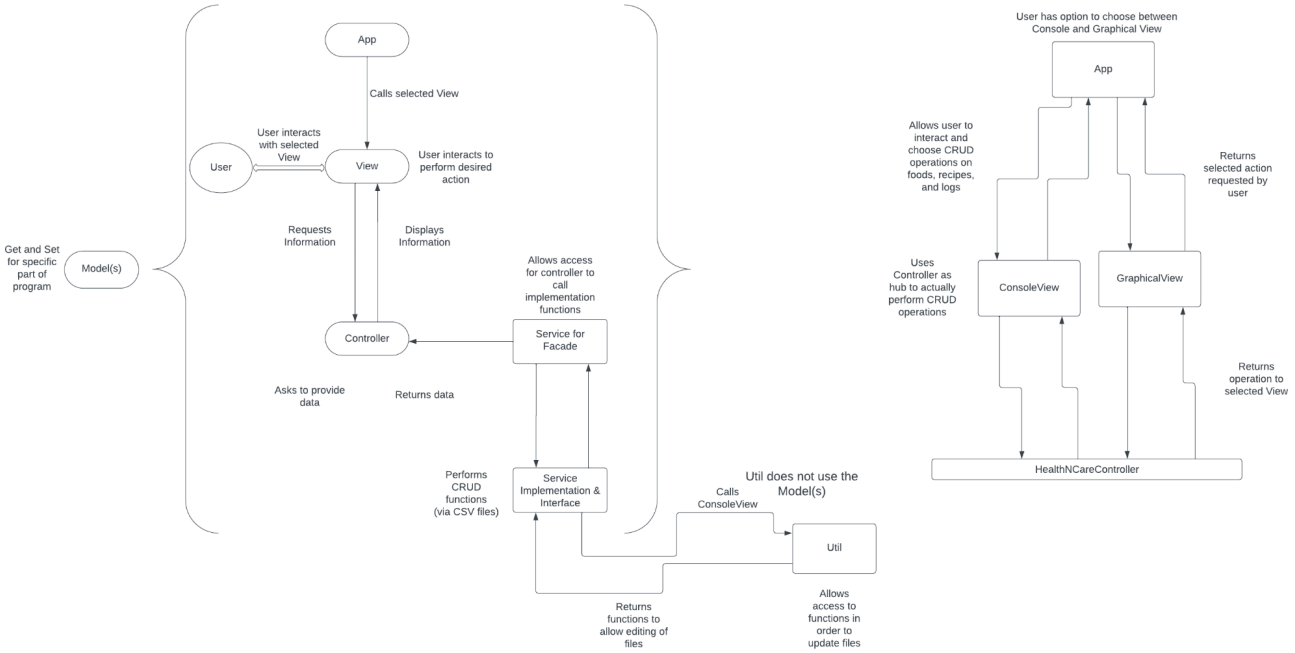
3 Subsystem Structure - Andrew

This section provides a graphical model of the subsystems that comprise the application as a whole.

Subsystems are groups of closely related classes. For example, an email program might have subsystems for contact list management, message composition, and message delivery, among many others. In large applications, a subsystem might even have sub-subsystems, though this should not be necessary in this course.

A design goal is to have much lower coupling among classes of *different* subsystems than among the classes *within* a subsystem. Put another way, each subsystem should be highly cohesive in purpose, and the subsystems as a group should exhibit separation of concerns, just as the set of classes *within* a given subsystem should each be highly cohesive, while the set as a whole exhibits separation of concerns.

Draw the subsystems as simple boxes with lines (possibly terminated in arrows) showing relationships between them. Include in each subsystem box a brief narrative of the subsystem's purpose and/or responsibilities. Label important relationship lines with an indication of what the relationship represents.



4 Subsystems - Matt

Class healthNCareController	
Responsibilities	View recipes, logs, and food. Displays data gathered from other classes.

Class BasicFood	
Responsibilities	Stores methods to use collect foods from the foods CSV.

Class CalorieLimitLog	
Responsibilities	Stores methods to use collect calorie logs from the log CSV.

Class FoodLog	
Responsibilities	Stores methods to use collect food information from the log CSV.

Class Ingredients	
Responsibilities	Stores methods to use basic foods as ingredients for recipes.

Class NutritionFacts	
Responsibilities	Stores methods to create nutrition facts for foods.

Class Exercise	
Responsibilities	Stores methods to use collect exercises from exercise.csv.

Class ExerciseLog	
Responsibilities	Stores methods to use collect exercise information from log.csv.

Class Recipe	
Responsibilities	Stores methods to create and get new Recipes.

Class WeightLog	
Responsibilities	Stores methods to interact with the weight logs in the log CSV.

Class DailyLogServiceImpl	
Responsibilities	Methods that update the calorie log, food log, and weight log CSV files.
Collaborators (uses)	CalorieLimitLog, FoodLog, WeightLog, DailyLogServiceInterface, DataUtil.

Class DailyLogServiceInterface	
Responsibilities	Interface for DailyLogService, outlines the classes used in it.

Class ExerciseServiceImpl	
Responsibilities	Methods that update the exercise.csv file
Collaborators (uses)	Exercise, DataUtil.

Class ExerciseServiceInterface	
Responsibilities	Interface for ExerciseService, outlines the classes used in it.

Class ExerciseService	
Responsibilities	Functionality for exercises (get and create).

Class FoodServiceImpl	
Responsibilities	Methods that update the food, ingredients, nutrition facts, and recipes CSV files
Collaborators (uses)	BasicFood, Ingredients, NutritionFacts, Recipe, FoodServiceInterface, DataUtil.

Class FoodServiceInterface	
Responsibilities	Interface for FoodService, outlines the classes used in it.

Class FoodService	
Responsibilities	Functionality for basic foods (get and create).

Class DataUtil	
Responsibilities	Gets data from the CSV files.

Class Console View	
Responsibilities	Outputs selectable options to interact with the CSV files.

Class Graphical View	
Responsibilities	Creates a GUI from which users can view and change information within the CSV files

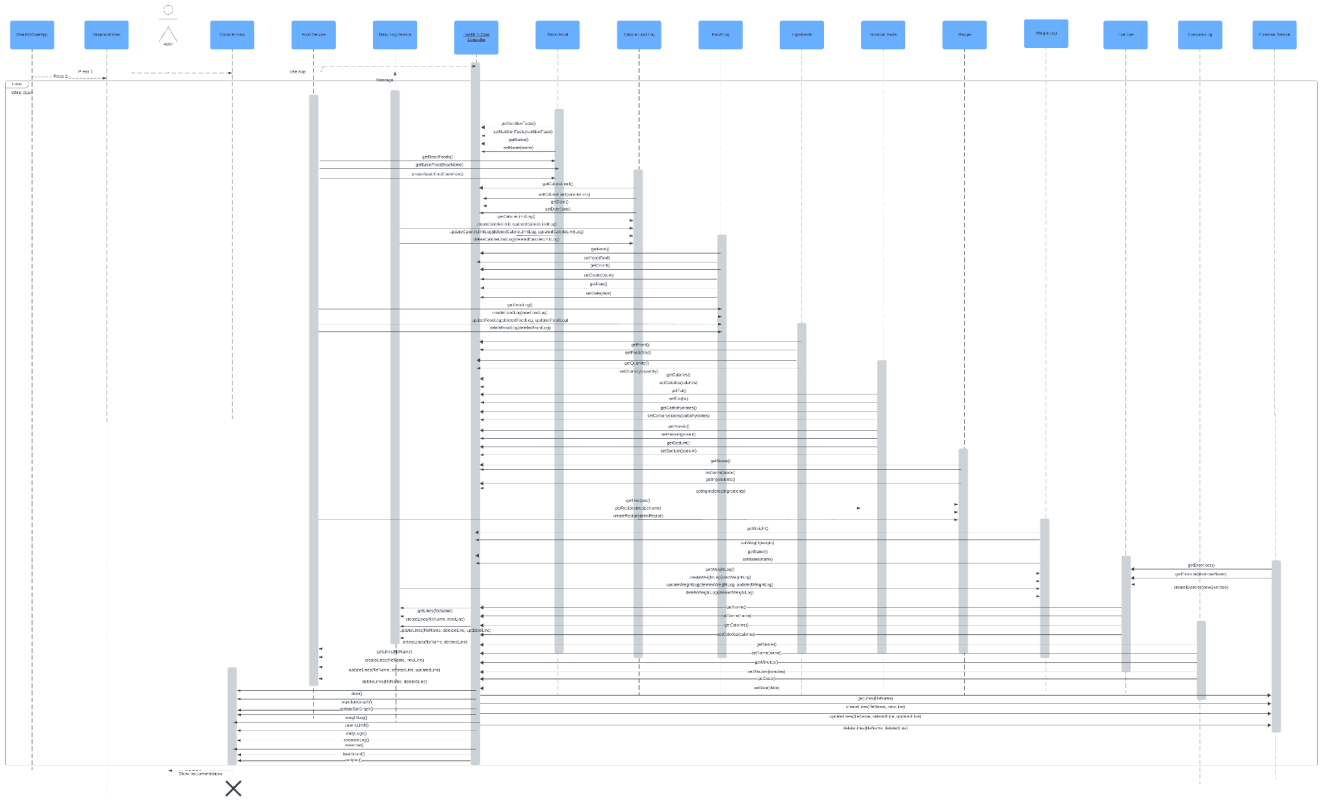
Class HealthNCareApp	
Responsibilities	Runs the Console View

5 Sequence Diagrams - Adam

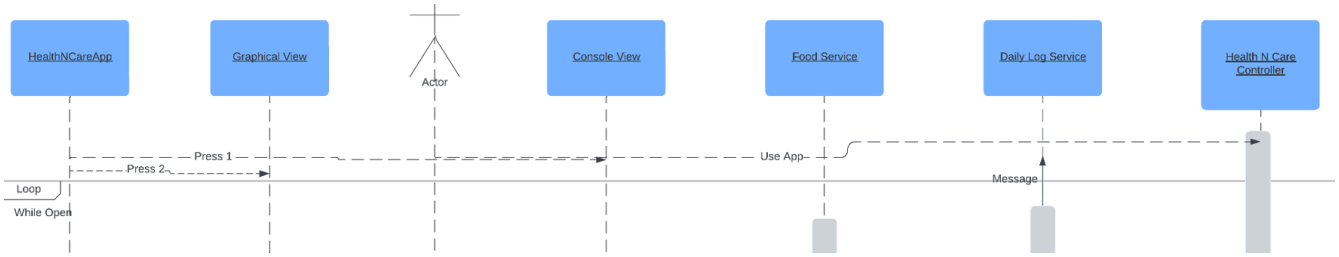
5.1 Description of labeled Sequence diagram #1 and (what feature / operation / scenario the diagram shows).

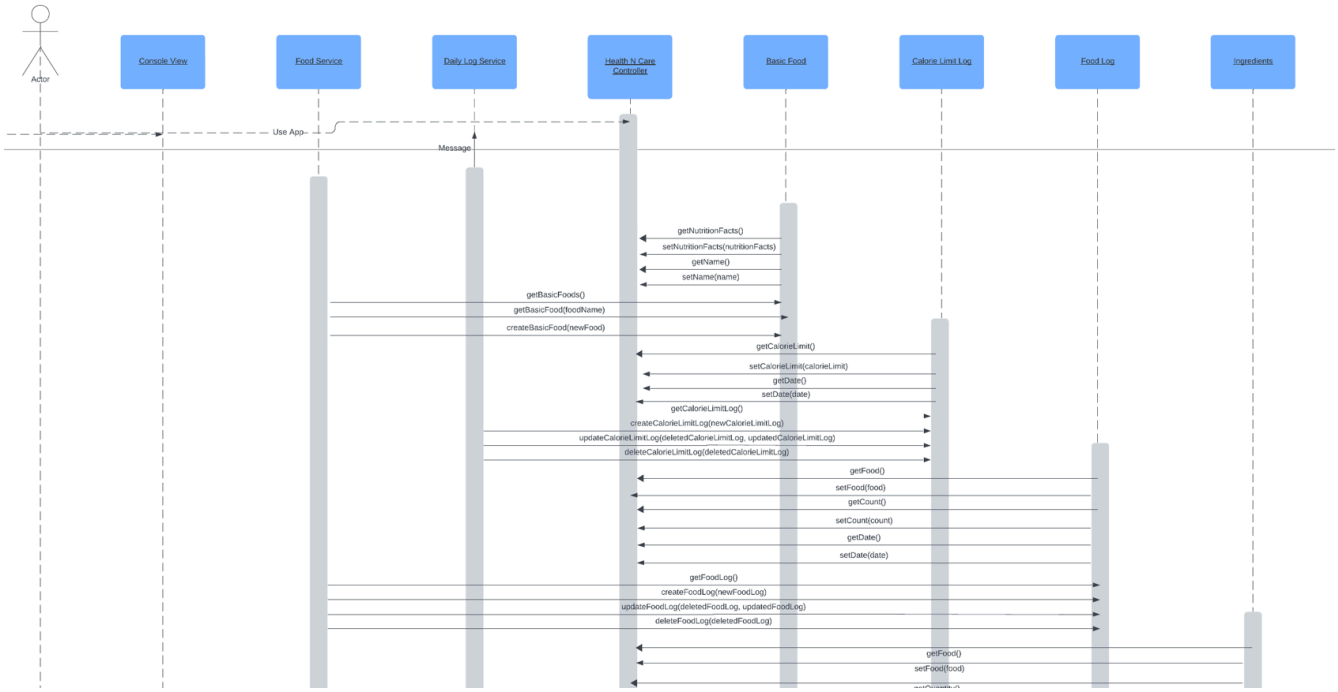
https://lucid.app/lucidchart/f416e9e5-b036-401b-852e-3e5e4074f49f/edit?viewport_loc=1367%2C1343%2C2905%2C1322%2C0_0&invitationId=inv_d44e0df3-ff82-4c2f-bc68-c6997c452016

(Full Diagram)



(Top Left Corner)





(Bottom Left Corner)

5.2 Description of labeled Sequence diagram #N (what feature / operation / scenario the diagram shows).

At the start of the program outside of the loop, the user first chooses whether they use the console or graphical views. Pressing 1 goes to the console and pressing 2 opens the GUI. The data for the code largely follows the MVC design pattern. Part of why the chart is so large is that many of the classes to the right are models for the different logs and

their uses can radically differ such as a food being part of a recipe or basic. The console view does not feature any methods other than itself, so it takes the data from the controller and presents it onto the screen. The graphical view however uses all of the different functions for the controllers. Every button used must interact with the controller's many classes in order for the app to function. The controller does this by getting the data from the model, and displaying it in the view. The services classes then store the new data onto the csv files of food, exercise, or the daily log. Food services manage the number of food and recipes being eaten by the user. The daily log service allows the user to record their calorie intake and display progress made on a bar graph. This includes exercises removing calories consumed. The exercise service allows the user to find an exercise to do or add a new exercise to the program.

6 Pattern Usage - V.J.

There will be a subsection for each pattern you use in your design (***including*** those that may be required in the project description or by your instructor).

6.1 Observer Pattern

Observer Pattern	
Observer(s)	View (App)
Observable(s)	User (Inputs)

6.2 Facade Pattern

Facade Pattern	
Client(s)	User (Inputs)
Facade	HealthNCare App
Implementation	MVC System

6.3 Composite Pattern

Composite Pattern	
Component	DailyLogServiceImpl
Composite	DailyLogService
Leaf	DailyLogServiceInterface

7 RATIONALE

Be sure to incorporate all major DESIGN and ARCHITECTURAL decisions and reasons for pursuing them. It helps to add entries with a time stamp (e.g., 9/27/2023 –We decided to..) >

Decided to use MVC and services 3/7/2024

Decided to use util 3/25/2024