Release 2

Project Design Document

Fruit Cups

Kyle Mirley <kxm9453@rit.edu>

Lea Boyadjian <lb5255@rit.edu>

Regine Thimothee <rxt3820@rit.edu>

Ezana Kalekristos <eek9045@rit.edu>

Jordan Darling <jxd2395@rit.edu>

Project Summary

Wellness Manager is a program that will guide users to a healthier lifestyle. The program operates by allowing users to record the foods or recipes they eat on the daily basis. The user will provide the necessary information such as the name of the food, calories, grams of fat, grams of protein and grams of carbohydrates of the food. When it comes to recipes, the user will provide the ingredients. Wellness Manager can also log user's daily weight and provide an option of recording calorie limits. This can provide a visual diagram to help indicate if the user has met their calorie goal of the day. Along with the goal of leading a healthier lifestyle, there's also the option of recording exercises. When a user records an exercise, they log how many calories they burned in the process. Those calories are then deducted from the running calorie count of the day.

The daily logging featured within the wellness manager allows the user to keep track of what they eat on a daily basis as well as the amount of servings that they've consumed and review this information for any past date. If a user forgets to log their entries, they may return to a past date and enter their data. The same goes for exercises, users can return to prior dates and edit what exercises they performed on that day. Finally, users will be able to break their data down into easily understandable graphs. These graphs would highlight the distribution of nutrients across each day. All of the data provided to the wellness manager program will be collected into a csv file in order to maintain an easy export of information.

Design Overview

In our updated class diagram and sequence diagram below our group had made some design decisions to have eight main classes to be in charge of functionality:

Recipe: The recipe class is in charge of creating recipes through the food it receives.
Recipes: A collection of recipes.
Food: The food class is in charge of describing/creating instances of food that the user consumes
Foods: A collection of foods.
Log: The log class keeps track of when users consume food, their weight and calorie limits daily, and their exercises..
Logs: A collection of logs.
Exercise: The exercise class is in charge of describing and creating exercises.
Exercises: A collection of exercises.

Our group had also created a CommandParser class that would allow a user to input commands to create foods, recipes, exercises, and log information which would then be added to its corresponding collection class. Our group also incorporated the architectural pattern,

MVC with the command pattern as well as the composite pattern. We used the MVC pattern because we saw the pattern to this project and we wanted to follow the design principle “separation of concerns”. **The Model** in our project encapsulates the classes that contain the core functionality which would be classes such as Log, Food, Recipe. **The Controller** would be the command classes that directly access the model as well as the View. **The View** in our project would be the CommandParser class that is responsible for displaying information.

Project Assumptions:

Our group believes that we will be able to deliver a program that will allows user to log in data and track different nutritional information for each day
Some different aspects of the project that our group plans to accomplish are:

Resources:

- Our team will require the help of Dr. E. as well as ourselves to complete the project.

Delivery:

- End users will be able to input, log, and gain visual representation of their nutritional intake, as well as record how much exercise they’re getting.
- Project will be able to gain data from CSV inputs

Scope:

- Our team plans to deliver a Java program able to accomplish the tasks of a wellness manager

Methodology:

- Our team will follow a waterfall like approach in where the project will be broken down and complete in different sequential parts

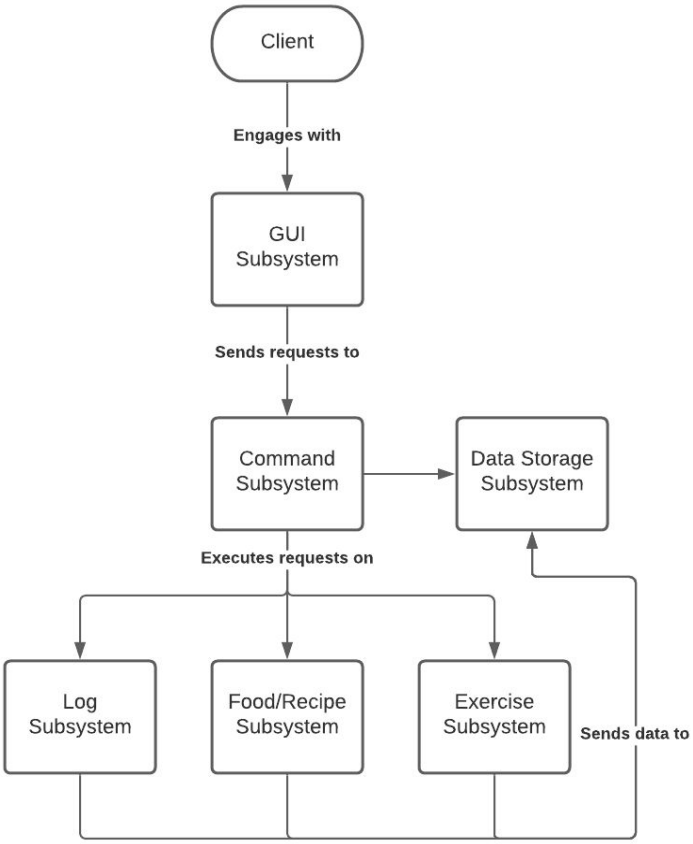
Technology:

- Our team plans to use :
    - Trello: for planning organization
    - Java/Intellij: for development
    - Lucidchart: for graphing
    - Zoom: for team meetings

Architecture and Design:

- Our team will utilize structural, design, and behavioral based pattern.

Subsystem Structure



This subsystem shows the general flow of information through our system. The client will be interacting directly with the GUI subsystem, which will show all of our graphical components, and display the most up to date information. In this subsystem, the client is presented with places to initiate commands. Any request / command that is initiated will then go to the command subsystem.

The command subsystem is in charge of interpreting what the client is trying to accomplish, and instantiating the proper command objects to execute that command. This portion of our system allows us to completely abstract our view from logic, and encapsulates our requests as objects. The command subsystem is in charge of instantiating those command objects, and providing the appropriate data for those objects to execute their respective commands. The command subsystem is also responsible for calling upon the data storage subsystem on initialization to get the initial CSV data.

The command subsystem - once a command object has been instantiated and ‘execute’ has been called - will then talk to one of three different subsystems depending on what command needs to be executed. All three of these subsystems live within the model, and will perform the request initiated by the command subsystem. When a data object is created, updated, or deleted, it will automatically contact the Data Storage subsystem and update the records of our data objects.

This design is an improvement of our previous design because it removes some of the responsibility from our Command subsystem, increasing our cohesion without having any effect on our coupling. With this design, we maintain a healthy balance of

cohesion and coupling, much like our users will be able to maintain a healthy lifestyle through the use of our app.

Subsystems

This section provides descriptions of the classes in the subsystem and their relationships.

Subsystem CSV Writer/Parser

Class CSVParser	
Responsibilities	Support access to a CSV Read to a CSV files Read Food items and Recipes from the file
Collaborators	Food- basic food class Recipe - collection of foods

Class CSVWriter	
Responsibilities	Support access to a CSV Write to a CSV files Create Food items and Recipes to the file
Collaborators	Food- basic food class Recipe - collection of foods

Class CommandParser	
Responsibilities	Provide commands to use both CSVParser and CSVWriter

Collaborators	CSVParser CSVWriter

Interface Command	
Responsibilities	Provides a generic interface to all subsystems to use different commands

Subsystem Logging

Class Log	
Responsibilities	Represents a user's log of food that is taken in a day. Create a log of a user's daily intake Get an existing log Delete a log Edit an existing log
Collaborators	N/A

Class LogCommand	
Responsibilities	Execute a command to the Log Class to Create, Get, Edit, Delete a log
Collaborators (uses)	Log

Interface Command	
Responsibilities	Provides a generic interface to all subsystems to use different commands

Subsystem Recipe

Class Recipe	
Responsibilities	Represents a recipe in the system Create a recipe of existing food Get an existing recipe Delete a recipe Edit a recipe
Collaborators	Food - basic food class

Class Food	
Responsibilities	Represents a food in the system Create a food in the system Delete a food Edit a food Get existing food.
Collaborators	N/A

Interface Command

Responsibilities	Provides a generic interface to all subsystems to use different commands

Interface Composite	
Responsibilities	Interface to design food and recipes into a tree like pattern Includes saving to a file

Class RecipeCommand	
Responsibilities	Interact with the recipe class to perform certain actions.
Collaborators	Recipe

Class FoodCommand	
Responsibilities	Interact with the recipe class to perform certain actions.
Collaborators	Food

Subsystem Exercise

Class Exercise

Responsibilities	Represent Exercise in the system Create and exercise Edit and exercise Log different exercises
Collaborators	N/A

Class ExerciseCommand	
Responsibilities	Execute a command to the Exercise Class to Create, Get, Edit, Delete an Exercise
Collaborators (uses)	Exercise

Interface Command	
Responsibilities	Provides a generic interface to all subsystems to use different commands

Subsystem Command

Class FoodCommand	
Responsibilities	Interact with the recipe class to perform certain actions.

Collaborators	Food

Class LogCommand	
Responsibilities	Execute a command to the Log Class to Create, Get, Edit, Delete a log
Collaborators (uses)	Log

Class RecipeCommand	
Responsibilities	Interact with the recipe class to perform certain actions.
Collaborators	Recipe

Class Recipe	
Responsibilities	Represents a recipe in the system Create a recipe of existing food Get an existing recipe Delete a recipe Edit a recipe
Collaborators	Food - basic food class

Class Food	
Responsibilities	Represents a food in the system Create a food in the system Delete a food Edit a food Get existing food.
Collaborators	N/A

Interface Command	
Responsibilities	Provides a generic interface to all subsystems to use different commands

Class CommandParser	
Responsibilities	Provide commands to use both CSVParser and CSVWriter
Collaborators	CSVParser CSVWriter

Interface Composite

Responsibilities	Interface to design food and recipes into a tree like pattern Includes saving to a file

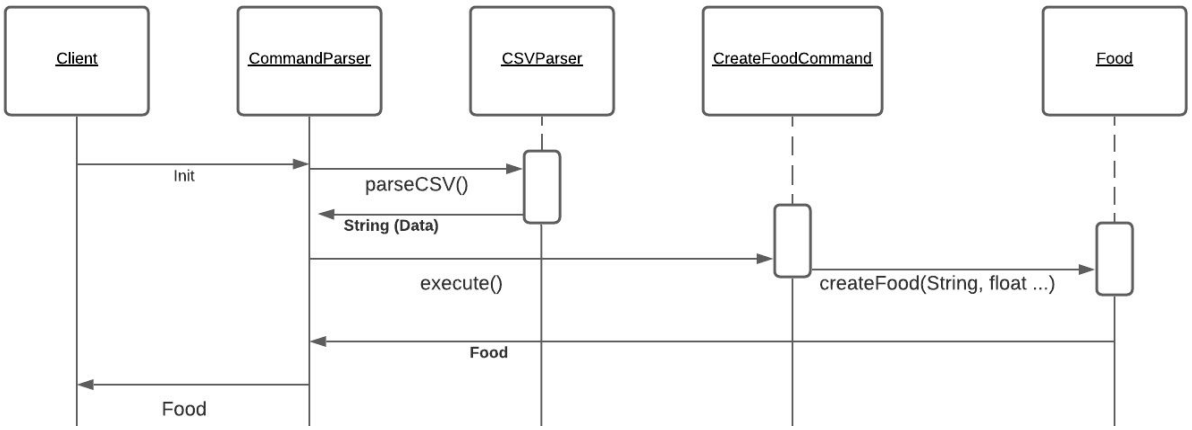
Class LogCommand	
Responsibilities	Execute a command to the Log Class to Create, Get, Edit, Delete a log
Collaborators (uses)	Log

Subsystem Graphing Subsystem

Class CommandParser	
Responsibilities	Provide commands to use both CSVParser and CSVWriter
Collaborators	CSVParser CSVWriter

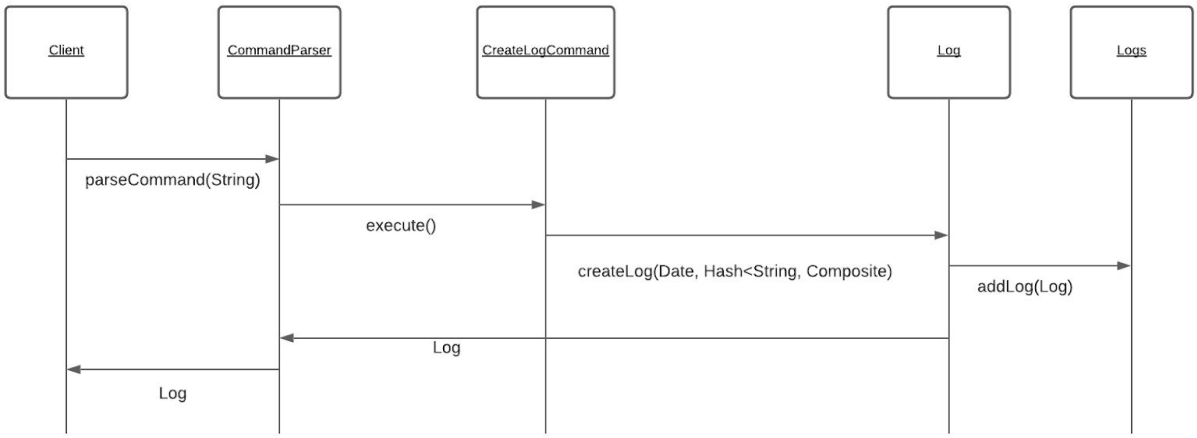
Sequence Diagrams

Instantiating new objects:



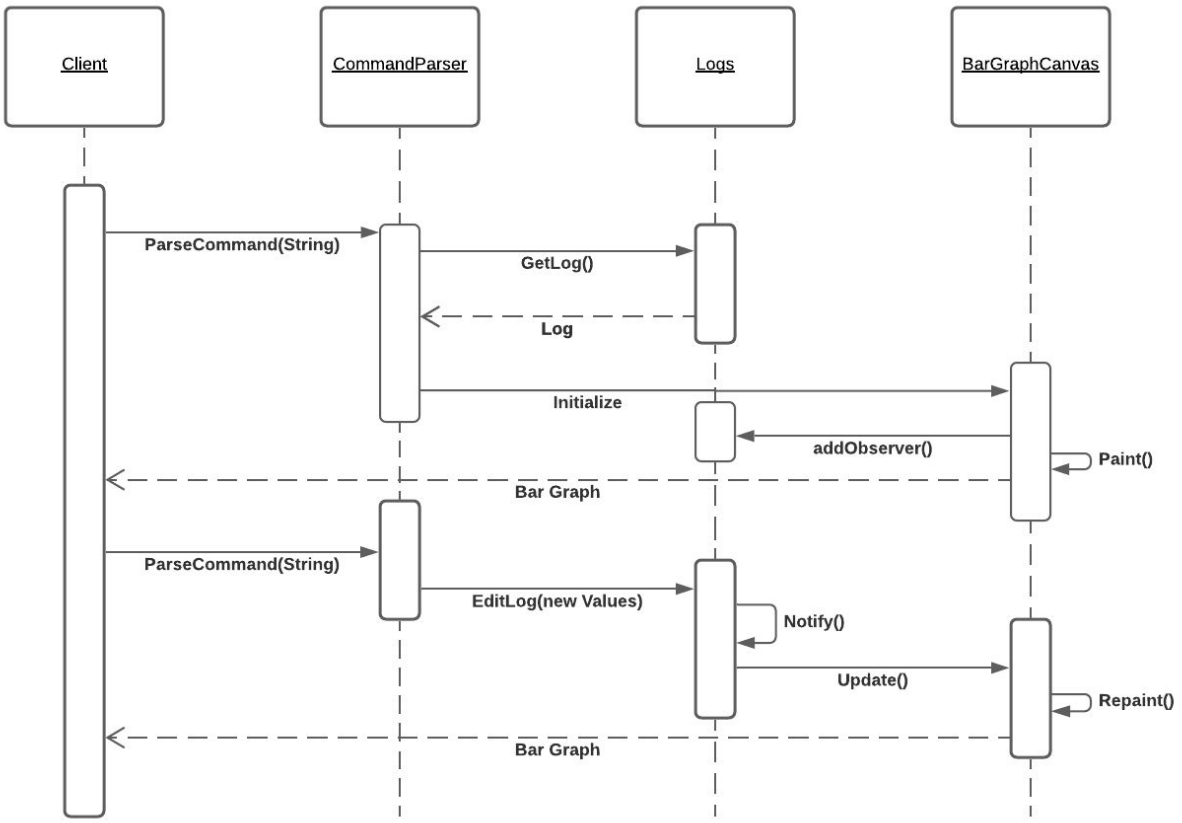
This sequence diagram above shows the process of creating a Food object, and looks exactly the same if we were creating a Recipe or Exercise. The Command Pattern is used in order to create these objects. The players are the Command Parser as the invoker, the Food,Recipe, and Exercise as the receivers, and the different CreateFoodCommand,CreateRecipeCommand,CreateExerciseCommand, as the concrete command classes. These concrete classes return the appropriate classes once they are invoked

Creating logs:



This sequence diagram shows the process of creating a Log object. We are utilizing the command pattern here in order to create a Log. The client initiates the interaction by making a request to the command parser to create a log. The command parser then instantiates the CreateLogCommand object once it has gathered the proper parameters. The CreateLogCommand object then initiates its ‘execute’ command, and calls on the Log object to create a new Log. That Log object is then added to the encompassing list of Logs, and the successful operation of the command is verified by returning the newly created Log object back to the Command Parser, and ultimately back to the Client.

Creating and updating Bar Graph



In this sequence diagram, we are seeing the client tell the command parser that they want to see the bar graph of their log for the day. The command parser uses the same tactics of reaching out to the logs as it did in previous examples, but for brevity, it has been cut out. The command parser looks for the log posted on a specific date, and then initializes the BarGraphCanvas, which then adds that log as its observable, and notifies the log that it will be observing it. The BarGraphCanvas then calculates the fat, protein and carbs of the food items in the log, and then calls paint() to draw the graph. When the user eventually updates the log, the Log object gets edited, and notify is called. The

log then notifies the BarGraphCanvas by calling update and sending over its new data. The BarGraphCanvas then repaints itself to show the most current information.

Patterns:

MVC Pattern	
Model	Food, Log, Recipe, Exercise
View	UI
Controller	Command Objects

Composite Pattern	
Component	Composite Interface
Leaves	Food
Composite	Recipe

Command Pattern	
Request	Foods, Recipe, Logs, Exercise
Invoker object	Command Parser
Command	FoodCommand,Recipe Command,Log Command,Exercise Command, Command

ObserverPattern	
Concrete Observer	BarGraphCanvas
Subject	Logs