PROJECT / RELEASE

Project Design Document

TEAM A: NANCY

Kelly Appleton <kda9036@rit.edu>
Yun Yiu Cheng <yc6647@rit.edu>
Jordan Rabideau <jmr8990@rit.edu>
Tessa Zakroczemski <tez9600@rit.edu>

Project Summary

Health impacts every aspect of one's life; therefore, having good health is important. Nutrition is a crucial aspect of health and plays a major role in achieving and maintaining a healthy weight. Being over or underweight can negatively impact one's health. This connection between one's health, nutrition, and weight has created a need for people to be able to monitor their food intake, nutrition values, and weight. Wellness Manager is a software that meets the needs of those looking to monitor such aspects to improve their health and daily lives.

Users will be able to add and retrieve foods and recipes from the system. Nutrition facts about the foods including calories, grams of fat, grams of carbohydrates, and grams of protein will be included. Users will be able to log what they eat and calculate their daily nutrition amounts. The application will save user data so that they can set goals such as a target weight or their desired calorie intake and track their progress. Wellness Manager will make it convenient for users to log and track their daily consumption, calorie and nutrition targets, and recorded weights in order to make adjustments to achieve their health goals.

In the second version of Wellness Manager, users are able to record exercises to reduce their calorie totals. They will also be presented with an intuitive graphical user interface (GUI) in order to carry out any program functionality and view their progress, which will also be aided by a bar graph.

Design Overview

Our initial approach to designing the system was to use a Model-View-Controller (MVC) architectural pattern in order to separate concerns involving user interaction and system functionality. We want users to be able to interact with a view, which in our case will be a simple command line interface that will eventually become a GUI. A controller will listen for user input and inform the model of what functionality is needed. The model will store information about the user, foods, and recipes and will act accordingly when receiving instruction from the controller. While all three components - the model, view, and controller - are coupled together, we think it is important for our application to be separated into these components to implement high cohesion so that each can focus on handling its particular role and do so efficiently.

Our model will serve as the core of our application and consist of multiple design patterns. From the start of our design process, we found that the most logical pattern to employ is the Composite pattern. The main goal of the program is to be able to log foods and recipes a user consumes in order to monitor nutrition values associated with those items. A food can be thought of as a basic food and a recipe can be thought of as something that is composed of foods; therefore, we plan to use the Composite pattern to have a food interface serve as the component, a basic food class that is a leaf, and a recipe class that is a composite, since a recipe is an aggregation of one or more foods. This utilization of the Composite pattern will aid in the extensibility of our application in that users will be able to add basic foods and recipes easily, without changes to classes or other components of the program being needed.

After sketching our initial design and reviewing how different processes would work, we felt there was a need for either the Facade pattern or Mediator pattern because there are multiple complex processes that need to occur when carrying out various functionality. We decided to incorporate the Mediator pattern within the Model subsystem because there is a need for bidirectional communication between our mediator and the subsystems that comprise our model, such as the food and logging subsystems. In addition to improving communication between classes and subsystems, we also want our mediator to be able to add functionality by acting as a manager, and doing more work than that of a facade. The mediator will facilitate all nutritional aggregation functions and coordinate interactions between objects. This will decrease coupling within our application because classes will not be tightly coupled to each other and will not be coupled to multiple classes; instead, they will only know about the mediator and depend on it for their interactions.

We have also introduced the Observer pattern to our design. Since our mediator is serving as a manager, we decided to make it an Observable. Our command line interface (and eventual GUI) will be an Observer. This allows the Manager class to notify the view of any changes to the model so that the view can update itself automatically. By incorporating the Observer pattern within our design, we are also accounting for future extensibility of our application. The Observer pattern allows for one or more observers, so when our application progresses beyond a CLI, we will be able to accommodate users if they want to use the Wellness Manager on multiple platforms such as their smart watch, phone, tablet, laptop, etc.

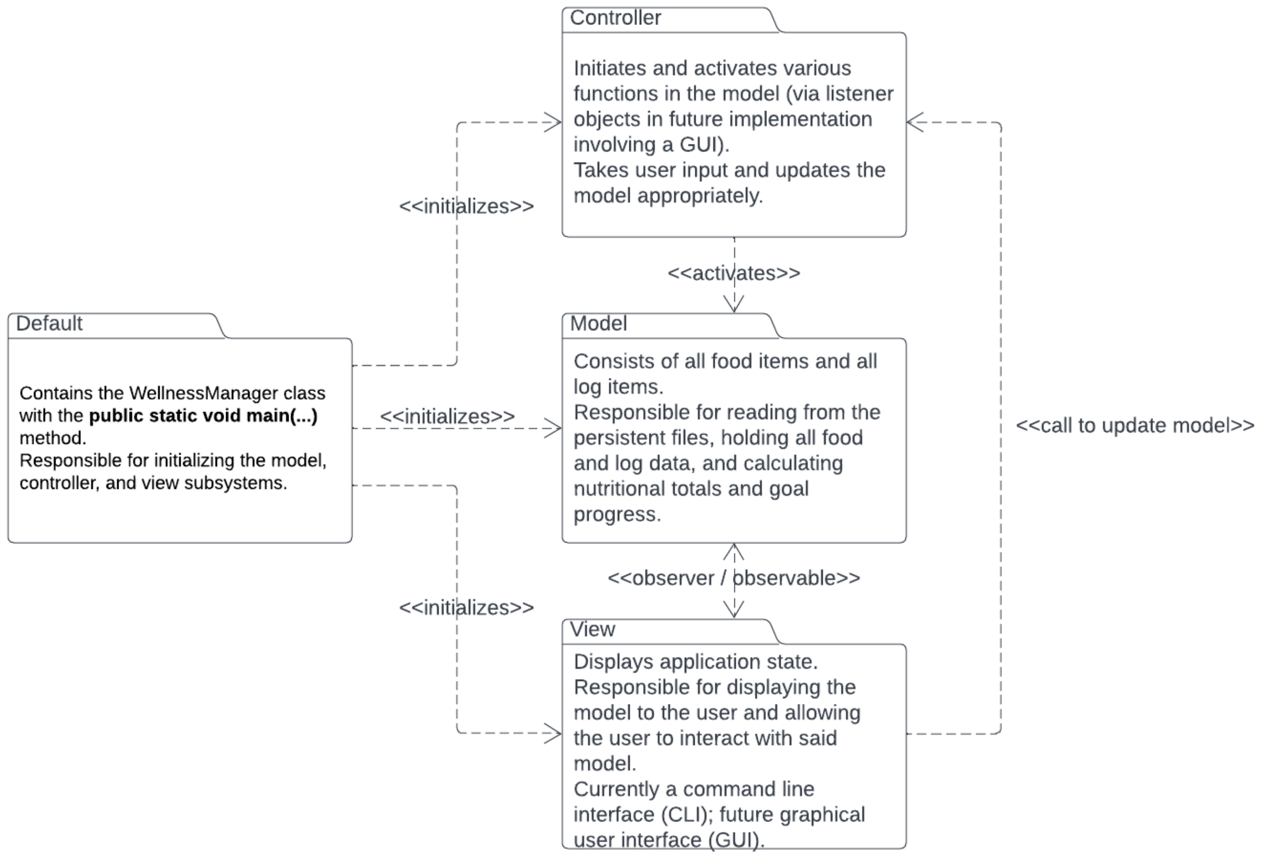
Phase 2

Phase 2 of our project involves adding the ability to log exercises and deduct calories from daily totals based on the calories burned from exercising. It also involves the creation of a graphical user interface (GUI) in addition to or in place of a command line interface (CLI). Since our original design attempted to be highly cohesive with low coupling, we will not need to change much to incorporate these new requirements.

We are able to simply extend the functionality of our application by adding the ability to log exercises. Exercises can be singular or done as a part of an exercise routine. While we do not know a desired format for logging an exercise routine, we feel that to accomplish our new requirements surrounding exercises and prepare to also be able to handle future needs of handling exercise routines, we are again looking to incorporate the Composite pattern in our design. Just like with foods and recipes, we will apply the Composite pattern to exercises and exercise routines, where exercises are basic components and exercise routines are composites of those components.

While some of our team members have experience with JavaFX, we feel that we would like to challenge ourselves to learn something new and implement our GUI using Swing. We will be able to use components like buttons, text fields, and option panes to create a user friendly interface that enacts the same functionality as our CLI.

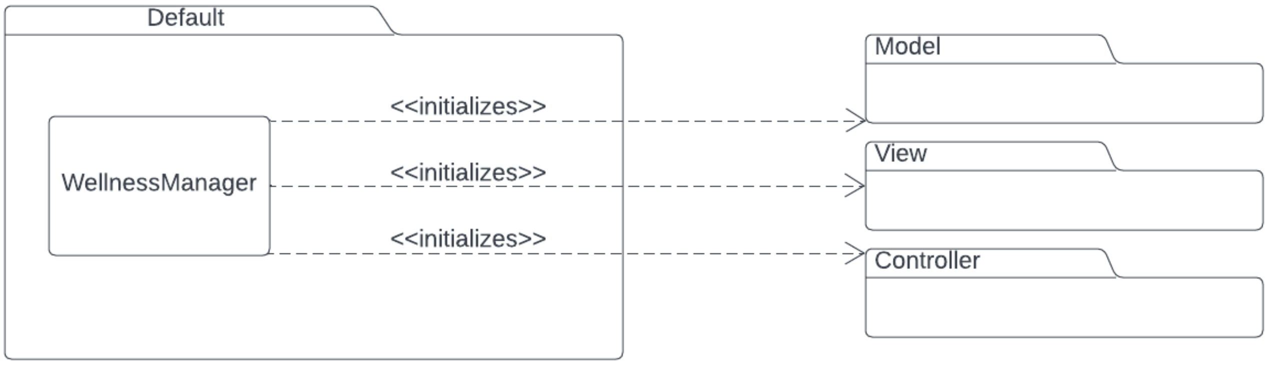
Subsystem Structure


Controller
Initiates and activates various functions in the model (via listener objects in future implementation involving a GUI).
Takes user input and updates the model appropriately.
<<initializes>>
<<activates>>
Default
Contains the WellnessManager class with the **public static void main(...)** method.
Responsible for initializing the model, controller, and view subsystems.
<<initializes>>
Model
Consists of all food items and all log items.
Responsible for reading from the persistent files, holding all food and log data, and calculating nutritional totals and goal progress.
<<call to update model>>
<<observer / observable>>
<<initializes>>
View
Displays application state.
Responsible for displaying the model to the user and allowing the user to interact with said model.
Currently a command line interface (CLI); future graphical user interface (GUI).

Subsystems

Default Subsystem

Class WellnessManager	
Responsibilities	Creates the mediator, controller, and CLI manager. Loads the data. Runs CLI manager.
Collaborators (uses)	model.WellnessController - the primary model class model.WellnessMediator - handling of functions of model class model.ManagerCLI - for user interaction with the program



Subsystem name: Model

Class Log	
Responsibilities	Saves user information. Timestamps the entries for the day. Provides access to foods and recipes consumed. Provides access to suggested caloric intake. Allows users to enter a single weight.
Collaborators (uses)	model.LogFood - contains food type and quantity

Class FoodReader	
Responsibilities	Read file. Saves file. Writes to food file. Retrieves ingredients.

Collaborators (uses)	model.BasicFood - holds name, calories, fat, carbs, and protein of a single food

Class LogReader	
Responsibilities	Read file. Saves file. Writes to log file. Retrieves ingredients.
Collaborators (uses)	model.Log - holds name, calories, fat, carbs, and protein of a single food

Class ExerciseReader	
Responsibilities	Read file. Saves file. Writes to exercise file.
Collaborators (uses)	model.Exercise - holds name and default calories burnt

Class Exercise	
Responsibilities	Hold name Hold Default calories burnt Calculate actual calories burnt.
Collaborators (uses)	

Class BasicExercise	
Responsibilities	Hold name Hold Default calories burnt Calculate actual calories burnt.
Collaborators (Implements)	Model.Exercise

Class ExerciseRoutine	
Responsibilities	Hold name

	Hold Default calories burnt Calculate actual calories burnt.
Collaborators (Implements)	Model.Exercise

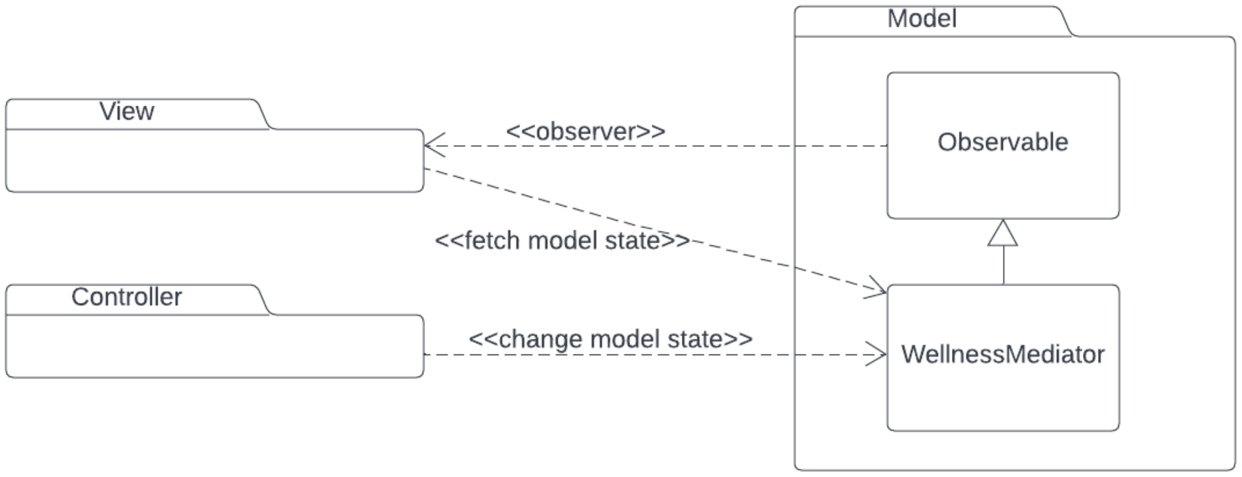
Class WellnessState	
Responsibilities	Hold list of available foods Hold list of available exercises Hold list of all logs Calculate daily progress
Collaborators (Implements)	Model.Exercise Model.Food Model.Log

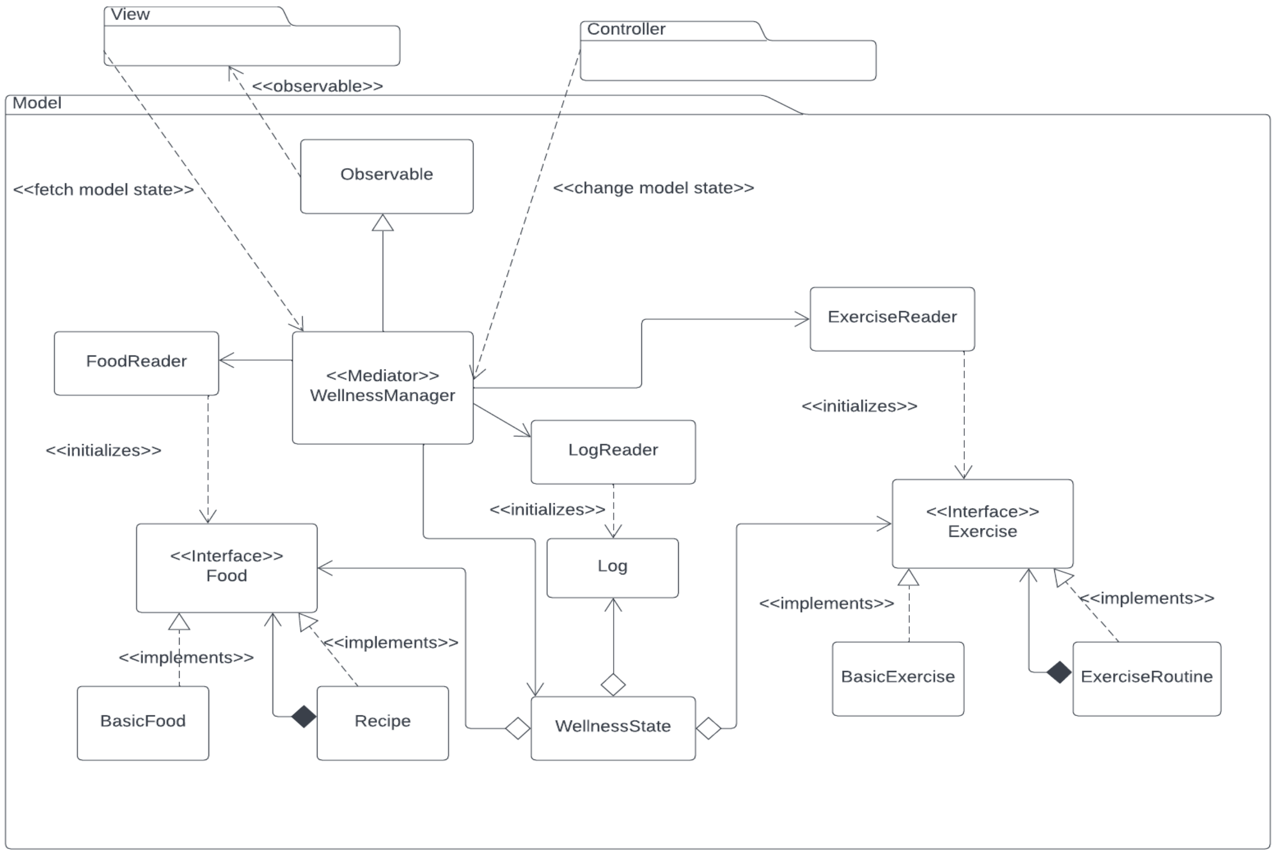
Class WellnessMediator	
Responsibilities	Store list of available foods. Store list of logs Add food to list of foods Update logs Add Logs
Collaborators (uses)	model.FoodReader - Reads food file and creates food objects model.LogReader - Reads log file and creates log objects

Class Recipe	
Responsibilities	Calculate total nutritional values Store a list of ingredients that makes up the recipe
Collaborators (implements)	model.Food - Holds nutritional information about foods

Class BasicFood

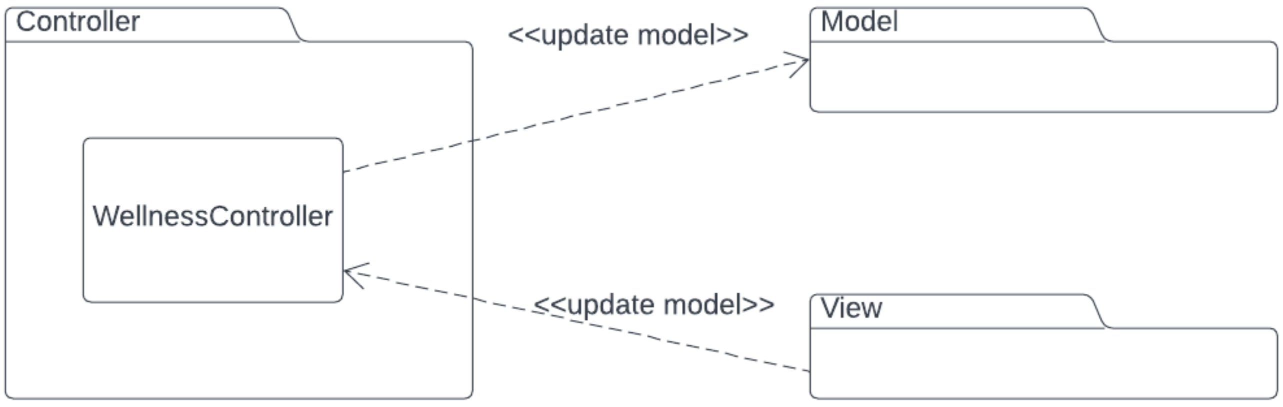
Responsibilities	Store nutritional values
Collaborators (implements)	model.Food - Holds nutritional information about foods





Subsystem name: Controller

Class WellnessController	
Responsibilities	Relay Log and Food updates from the view to the model
Collaborators (uses)	view.ManagerCLI: so the controller can obtain information entered by the user view.LogMenuCLI: so the controller can obtain information entered by the user model.WellnessMediator: so the controller can update the model when food is added

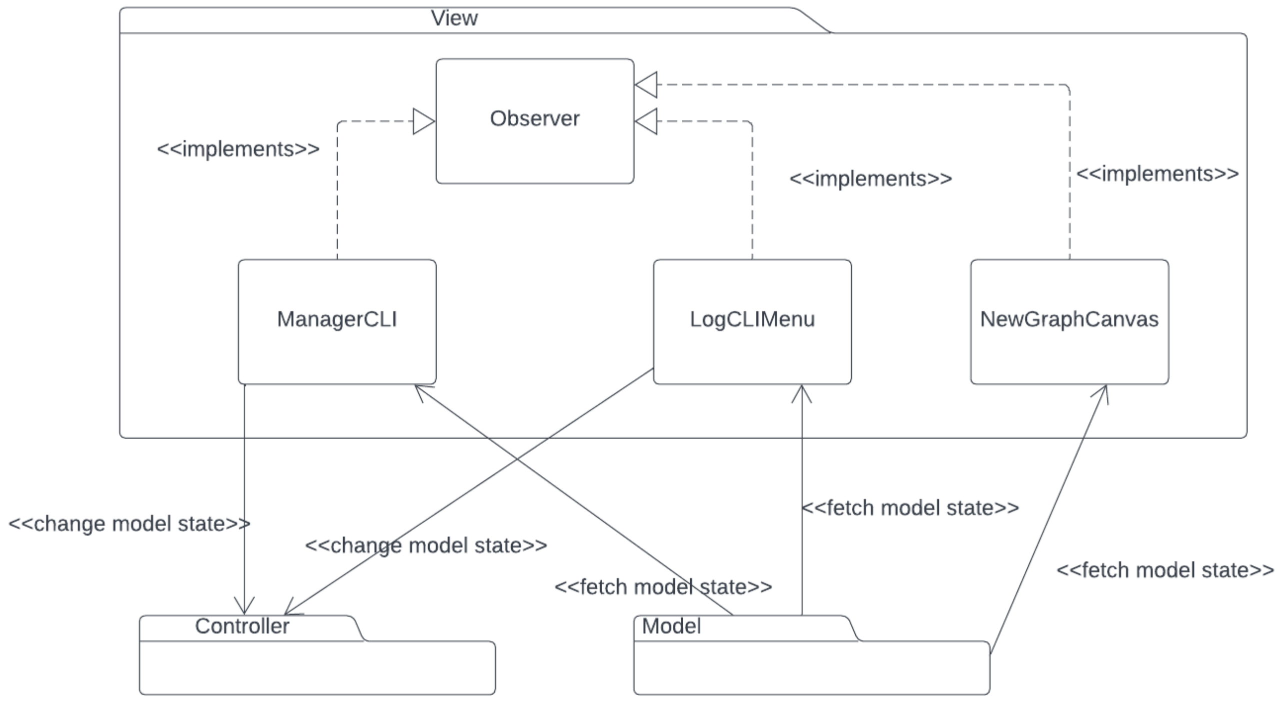


Subsystem name: View

Class ManagerCLI	
Responsibilities	Listen for the view to add a new food Listen for the view to add a new recipe Listen for the view to delete a food/recipe Listen for the view to list foods/recipes available Listen for the view to save and exit
Collaborators (uses)	model.WellnessController - call correct function model.WellnessMediator - handles function model.WellnessState - holds available food, logs, and exercises
Collaborators (implements)	Observer - allows the menu to update the state object when changes are made ActionListener - allows GUI buttons to call methods

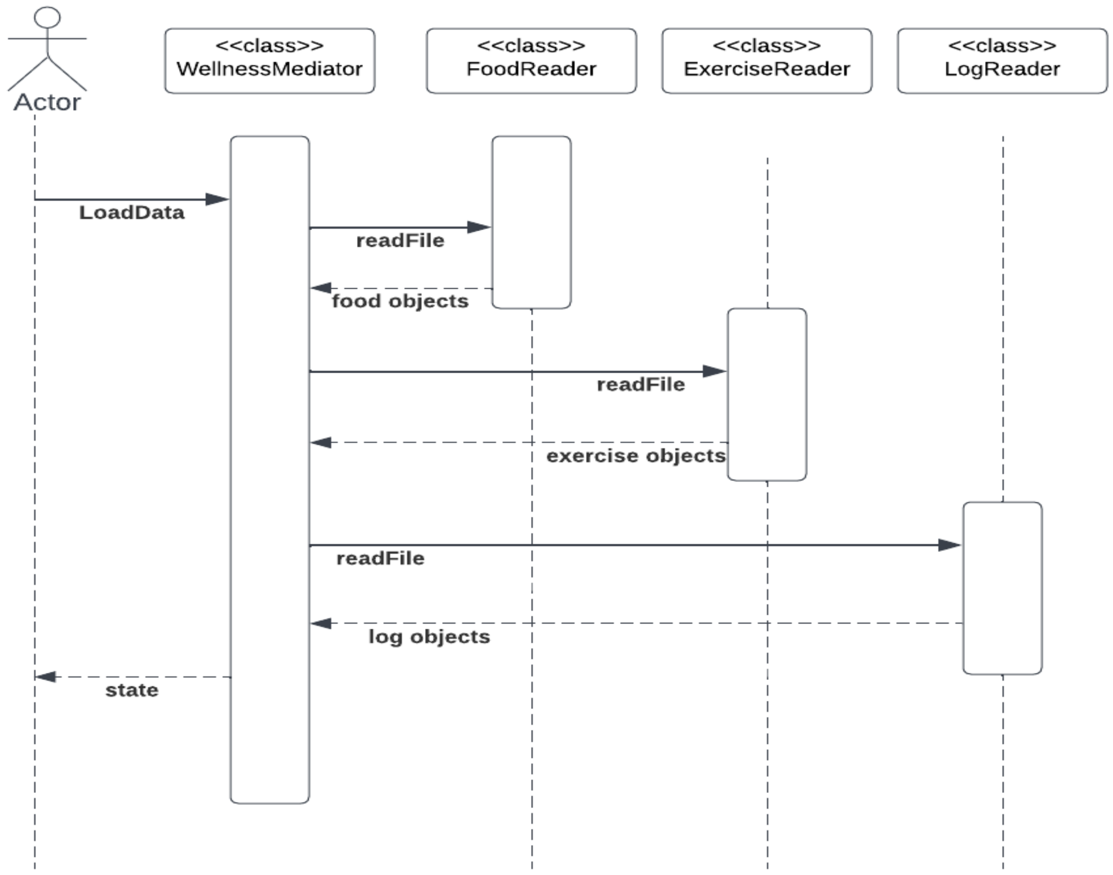
Class LogCLIMenu	
Responsibilities	Listen for the view to set weight. Listen for the view to set a calorie limit. Listen for the view to add food to the log. Listen for the view to view the logs.
Collaborators (uses)	model.WellnessController - call correct function model.WellnessMediator - handles function model.WellnessState - holds available food, logs, and exercises model.Food model.Log
Collaborators (implements)	Observer - allows the menu to update the state object when changes are made ActionListener - allows GUI buttons to call methods

Class NewGraphCanvas	
Responsibilities	Display nutritional information for current log
Collaborators (uses)	model.WellnessMediator - handles changes to the state model.WellnessState - holds available food, logs, and exercises model.Log
Collaborators (implements)	Observer - allows the menu to update the state object when changes are made
Collaborators (extends)	Canvas - allows the program to draw on the screen

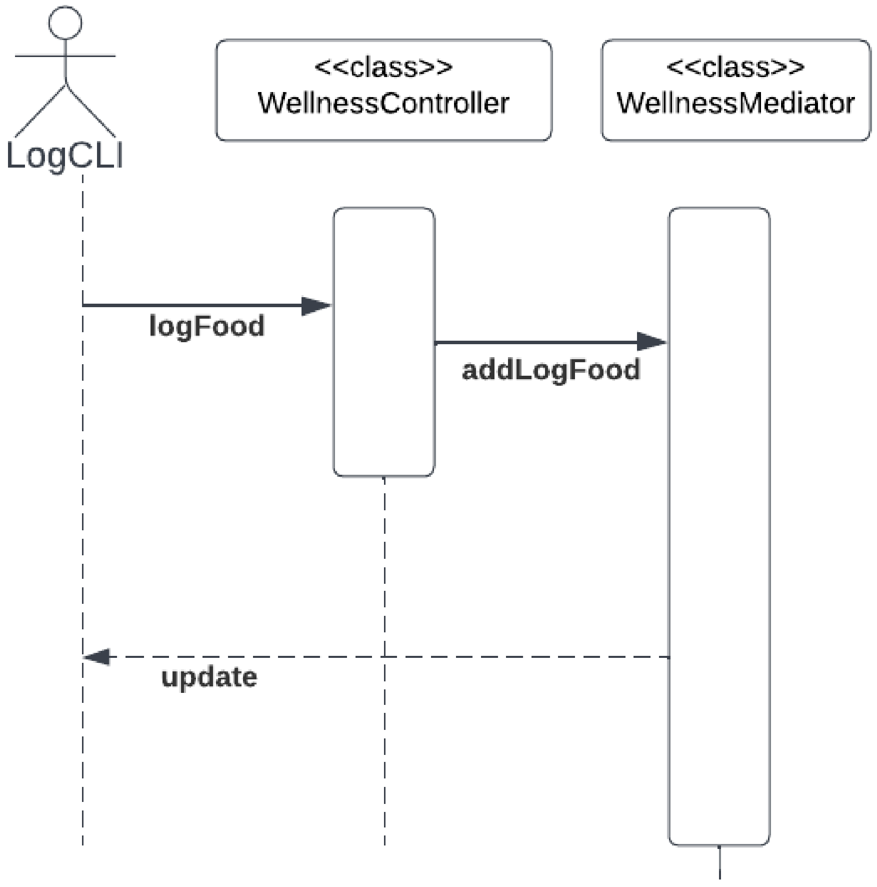

View
Observer
<<implements>>
<<implements>>
<<implements>>
ManagerCLI
LogCLIMenu
NewGraphCanvas
<<change model state>>
<<change model state>>
<<fetch model state>>
<<fetch model state>>
<<fetch model state>>
Controller
Model

Sequence Diagrams

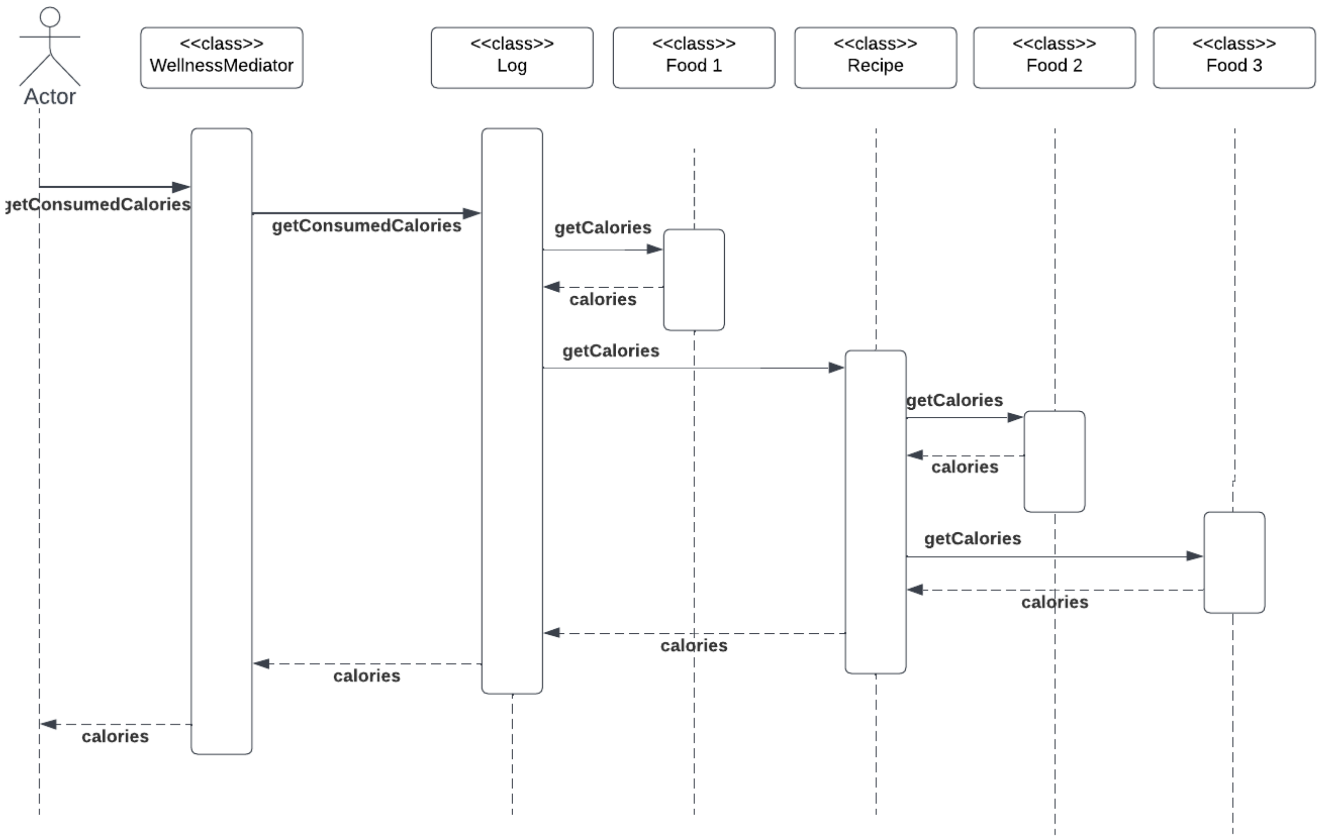
Sequence Diagram #1- Loading Objects into Memory



Sequence Diagram #2- Adding Serving of Food to Log Entry



Sequence Diagram #3- Compute Total Calories of a Log



Pattern Usage

Pattern #1: Composite

Composite Pattern	
Component	Food, Exercise
Leaf	Basic Food, Basic Exercise
Composite	Recipe, Exercise Routine*

Pattern #2: MVC (architectural pattern)

MVC	
Model	Manager Food Basic Food Recipe Food Reader Log Reader Exercise Basic Exercise Exercise Routine* Exercise Reader
View	ManagerCLI
Controller	Add Food Listener Add Exercise Listener Update Log Listener

Pattern #3: Mediator

Mediator Pattern	
Mediator	Manager
Mediated	Food Food Reader Log Reader Exercise Reader WellnessState

Pattern #4: Observer

Observer Pattern

Observable	Manager
Observer	ManagerCLI LogMenuCLI NewGraphCanvas

** not implemented yet*

RATIONALE

20221012 - We decided to incorporate the composite pattern by having Basic Food and Recipe classes both implement a common Food interface. This ensures that Recipes (as composites) can consist of both Basic Foods and other Recipes. We also decided to utilize the MVC architecture in order to cleanly separate the concerns of the different portions of the application.

20221020 - We decided to utilize the mediator pattern on the Model subsystem in order to simplify communication systems within the Model, as well as other subsystems in the application. We chose to use the mediator pattern rather than a facade because the mediator will facilitate all nutritional aggregation functions and store the result, as opposed to just passing information from one subsystem to another. We also decided to utilize the observer pattern by making the Manager class an Observable object, and the Manager CLI class an observer. This will allow the CLI (or any other future UI) to update itself when any changes are made to the model.

20221028 - We decided to add another class to the view in order to prevent the CLI from getting too bloated. We also believe that adding this specialized menu class will aid in the later implementation of actionlisteners.

20221114 - A GUI is required for phase 2 and our team has agreed to try to implement it using Java Swing. While some of the team has experience with JavaFX, our instructor/client seems to prefer Swing and we would like the challenge of learning something new. If we run into problems that cannot be resolved, we will consider revisiting the topic. We are also considering changing any class names containing “CLI” to “UI” for clarification since our program will have a GUI.

20221128 - We did not get around to changing any classes containing “CLI” to “UI” for clarity. We agreed that it is not worth our time or worth risking missing a change to rename classes at this point.