Wellness / Release v1.0

Project Design Document

Beep Boop

Justin Newton <jmn6815@rit.edu>

Aaron Kelly <axk3897@rit.edu>

Aaron Parker <ajw2611[@rit.edu](mailto:jtoa@rit.edu)>

Edward Riley <emr9018@rit.edu>

Vincent Venutolo <vxv2326@rit.edu>

# Project Summary

The project Wellness is designed to monitor the health status of an individual to provide whether he or she is healthy or not based on what food they ate and when they did recently. As the project is primarily focused on tracking the user’s diet to promote or encourage the user to be healthier, this project contains the most basic functions a user needs to manage and to update information in order to produce a result.

Specifically, this program allows the user to add recipes and this program will hold a collection of basic foods. The user can add the name of the food, the number of calories, grams of fat, grams of carbohydrates, and grams of protein in the food.

The user will be able to add the daily log of their food consumption and store it in their data and be able to access it anytime. The user can also add other information such as weight and more. This can be used to determine if the user has accomplished their goal or not.

# Design Overview

The goal for the project design is to keep change or update the design until it is satisfied the requirements and met the expectations of what it should be. The change in design will be covered here.

The Wellness Project had been undergone the transformation from the beginning with a design sketch include a class and sequence diagram for UML design. As the process initiated for the first time, the team started to design using the existing project called WeatherStation as our source template to produce our results. The Weatherstation project contains a basic threaded program that focuses on using Observable and Observer to use the update method to notify all registered objects whenever the data like Kelvin and Temperature are changed. The first draft design was completed which contains the main class, several subclasses rely on it, and their respective data or manager classes. The main class is on Wellness and the other subclasses are User and Recipe classes.

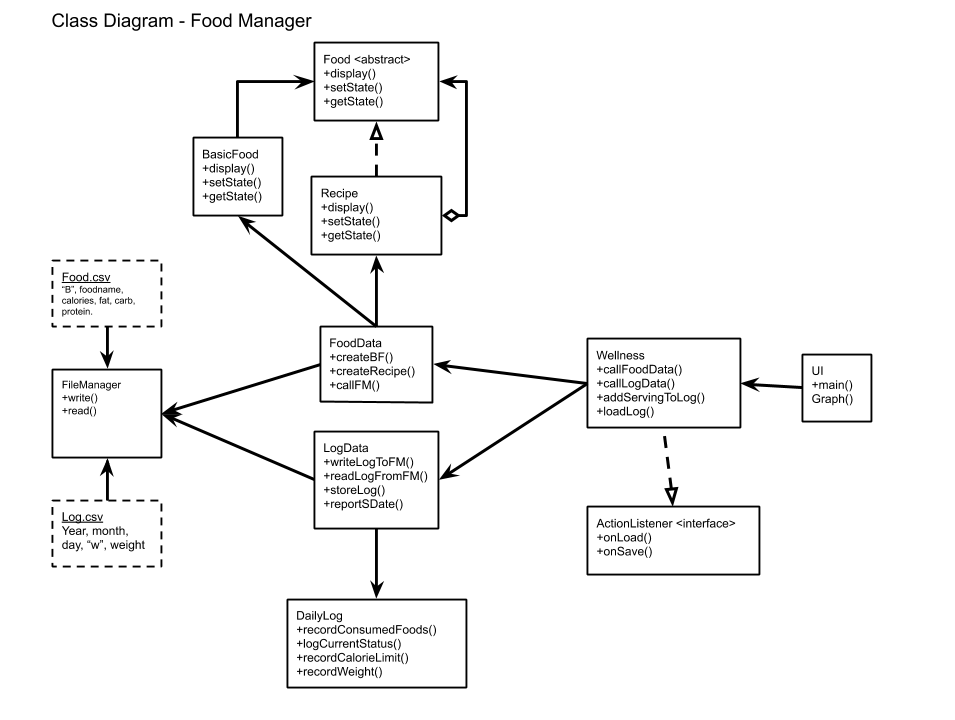
We had been in discussion over whether to meet requirements or considered what a client would be needed. Then the project in which classes are used to set up is changed to have a single class for a response to reading and writing files for both output and input of logs.csv and food.csv. Another class is User and Daily Log that they were configured to function separately so that the User will store only user data and their profile. Daily log class is set for monitoring nutrition status whenever the user eats food for each day. Three other classes which originally had a separate class in which one is responsible for collecting food called FoodCollection is now replaced with Recipe class and is treated as a composite for IFood and BasicFood classes. Thus, FoodCollection was removed and Recipe class is now a sub-main class for the Wellness project for storing and updating data of BasicFood and Recipe.

As the first draft design is completed, it had been taken several changes over time. We applied some patterns to change the class diagram are composite and MVC patterns. The composite pattern was used to treat both a simple and complicated object as a simple object when reading for a client. In the project, it applied to several classes for collecting and storing basic foods and recipes: BasicFood, Recipe, and Food. BasicFood is considered a leaf node while Recipe can be either treated as a simple food or recipe contains another simple food. Food is an interface or abstract class, which is also a component, used to provide some data for both leaf and composite. For MVC, the class diagram was configured to set BasicFood and Recipes as a model, user interface as a view, and Wellness and other classes are treated as a controller.Subsystem Structure

# This section provides a general overview of all high-level class subsystems that which to comprise all together as the whole application. There are a total of about 3 high-level classes are wellness, foodData, Log, and user.

To start with the main class, Wellness is the center of the project and is considered the top of the highest level class among other classes. It is responsible for main functions such as exchange information between several of other high-level classes which are a user interface (UI), FoodData, and LogData. This class is composed of another function which to update all registered objects whenever one, many, or all of the data receive a change. Below is a graphical model of the Wellness class and its classes connecting to the Wellness:

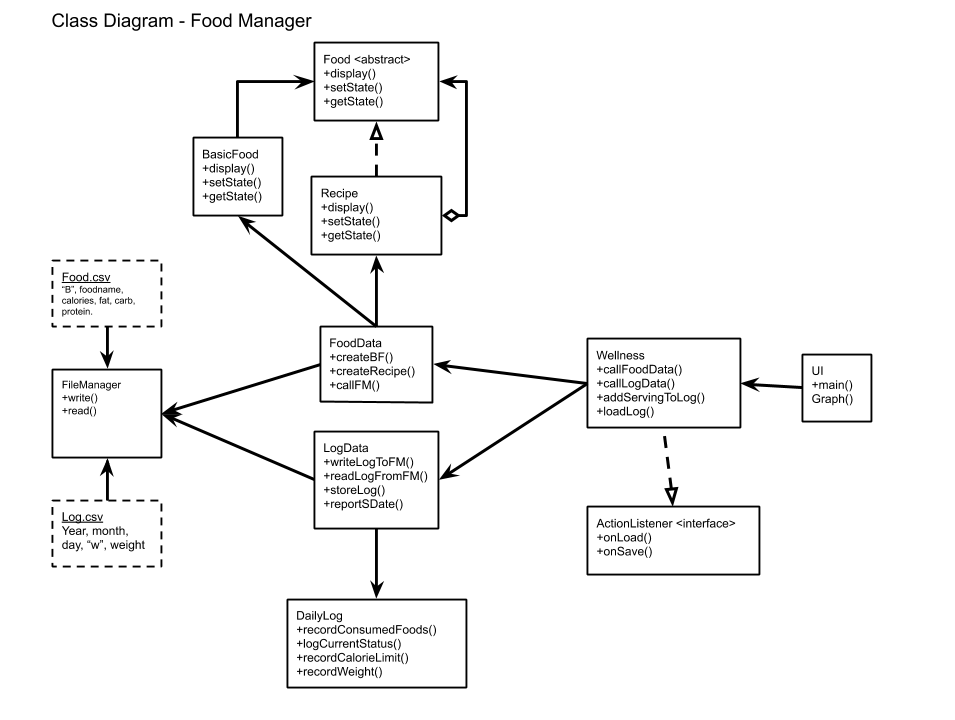
Wellness Figure



Start with the Wellness main class, it is responsible as a command center to exchange information and to update for the user interface. For the other classes except User Interface (UI), FoodData, and LogData, they are lower and also responsible for distributing needed information to and from any sources materials.

The second high level subsystem structure is user which is responsible to display the information to an individual. It is considered one of the highest level structure due to its request to command over the Wellness and as well as any information will affect other functions to process in order to bring up. The graphical model is below:

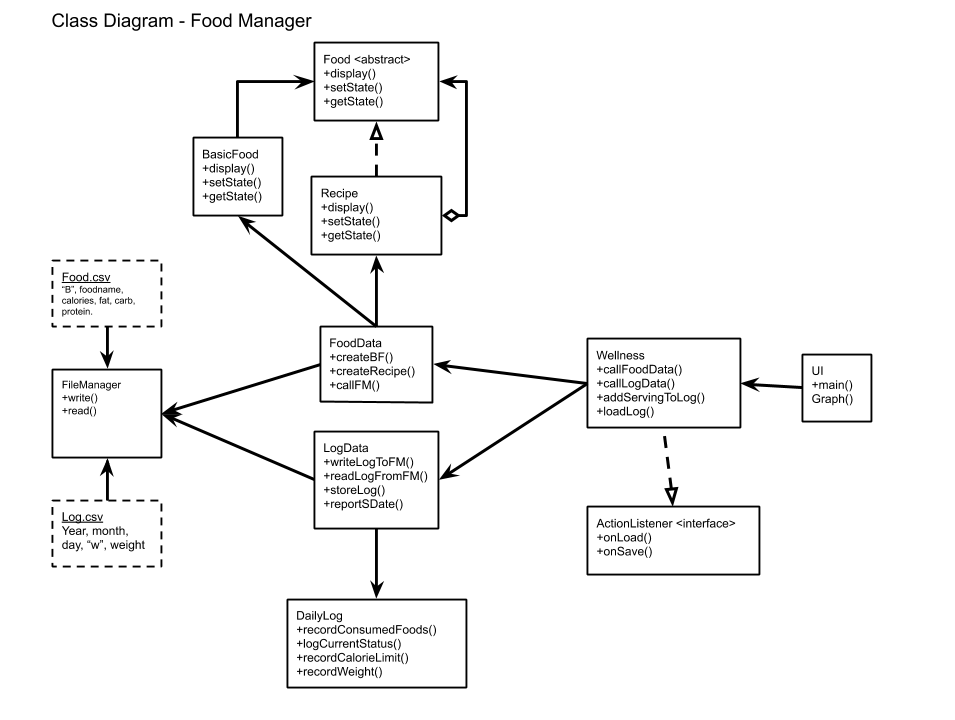
UI Figure



As the picture above shows, it provides a connection to the main station which is Wellness and contains major functions to provide what an individual needs. As the individual request for information to appear, it will command the main station to order any needed functions to process information to bring it up.

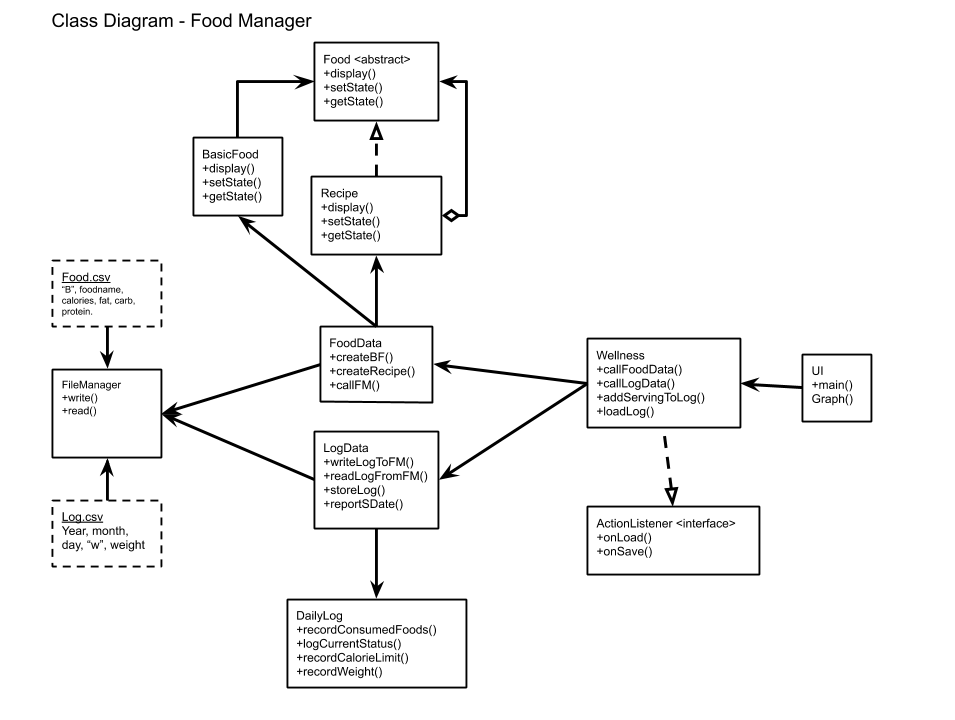
The third high level structure class is log class. It is not considered the top highest structure, but as a sub main class. Its level is the second level structure below the high level such as Wellness and User Interface and is used for the most common important functionalities needed of other low level classes such as DailyLog and FileManager. They provided data needed to establish data structures to the Log class to run. The graphical model for the LogData is below.

LogData Figure



As the picture shows, it has two classes to issue command over the other two. With the methods such as writeLogToFM() and storeLog(), they are indicated an authority to control and to command to issue for information from the two classes. DailyLog and FileManager are to provide information for the Log to run.

FoodData Figure



The fourth high level structure is FoodData (above). It is considered the same level as the log class for the different functionality to issue on two classes, Recipe, and FileManager. As it's shown above, it has three functions to issue are createBF, createRecipe, and callFM in which the two classes are required to provide when such commands are issued. As a result, the FoodData class have an authority over the two classes and is also a sub main class for the higher up it is required to provide information to give.

# 

# Subsystems

|  |  |
| --- | --- |
| **Class** Basic Food | |
| **Responsibilities** | basic food that defines the object that is implemented by Food abstract class and will be used as a single basic food.  Name, Calories, Fat, Carb, Protein, Weight. |
| **Collaborators**  **(uses)** | Food - the interface that is implemented by Basic Food and considers a leaf. |

|  |  |
| --- | --- |
| **Class** Food (abstract) | |
| **Responsibilities** | Provide a generic abstract to basic food & recipe.  Includes the name, calories, carbs, protein, weight. |

|  |  |
| --- | --- |
| **Class** Recipe | |
| **Responsibilities** | A Composite that can add basic food & recipe forming a composite pattern. |
| **Collaborators (inheritance)** | Food - the interface that is implemented by Recipe and considers a composite. |

|  |  |
| --- | --- |
| **Class** Wellness | |
| **Responsibilities** | A controller class that is implemented by ActionListener will act by trigger event and send specific data to one of two data classes that will store the information. Forming an MVC and Observer Pattern. |
| **Collaborators (inheritance)** | An ActionListener implemented by Wellness. |

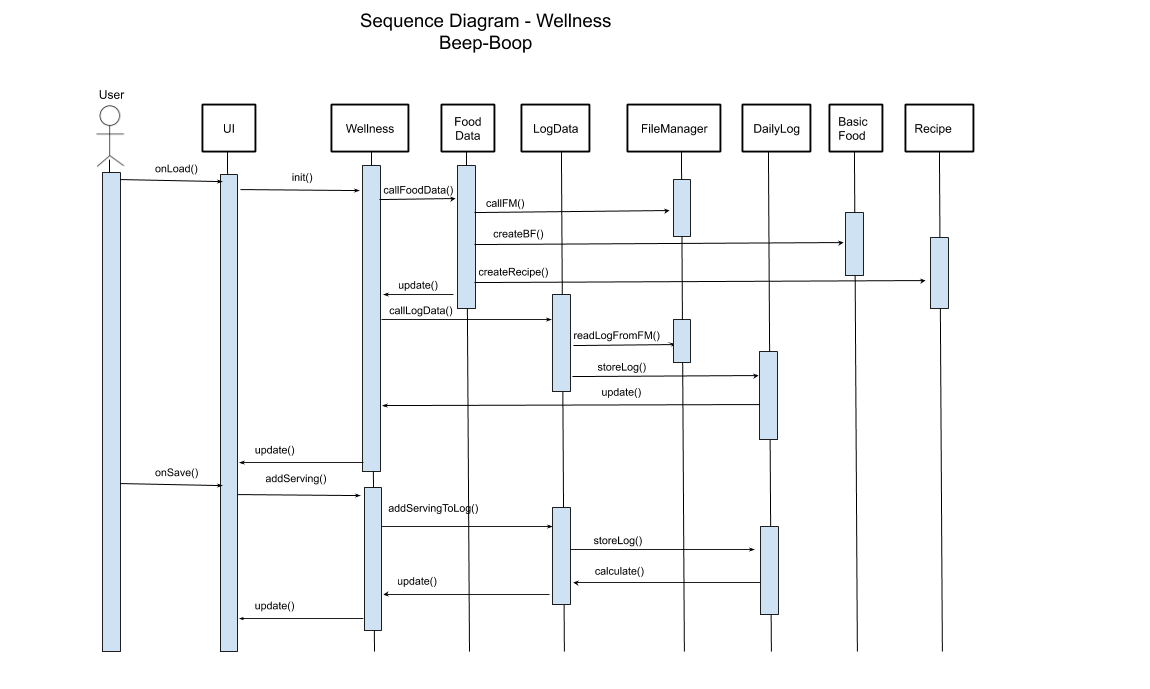
|  |  |
| --- | --- |
| **Class** Daily Log | |
| **Responsibilities** | A model that represents temporary data storage before it data being written to file in File Manager. |

|  |  |
| --- | --- |
| **Class** Food Data | |
| **Responsibilities** | A class that calls and instantiates Hashmap that accepts Food abstract as a data type that can store BasicFoods & Recipes.  Call file manager to write data to file when the user is done. |

|  |  |
| --- | --- |
| **Class** Log Data | |
| **Responsibilities** | A class that calls and instantiate ArrayList that accepts DailyLog as a data type that stores the data.  Call file manager to write data to file when the user is done.  Also, call the file manager to load data from the file. |

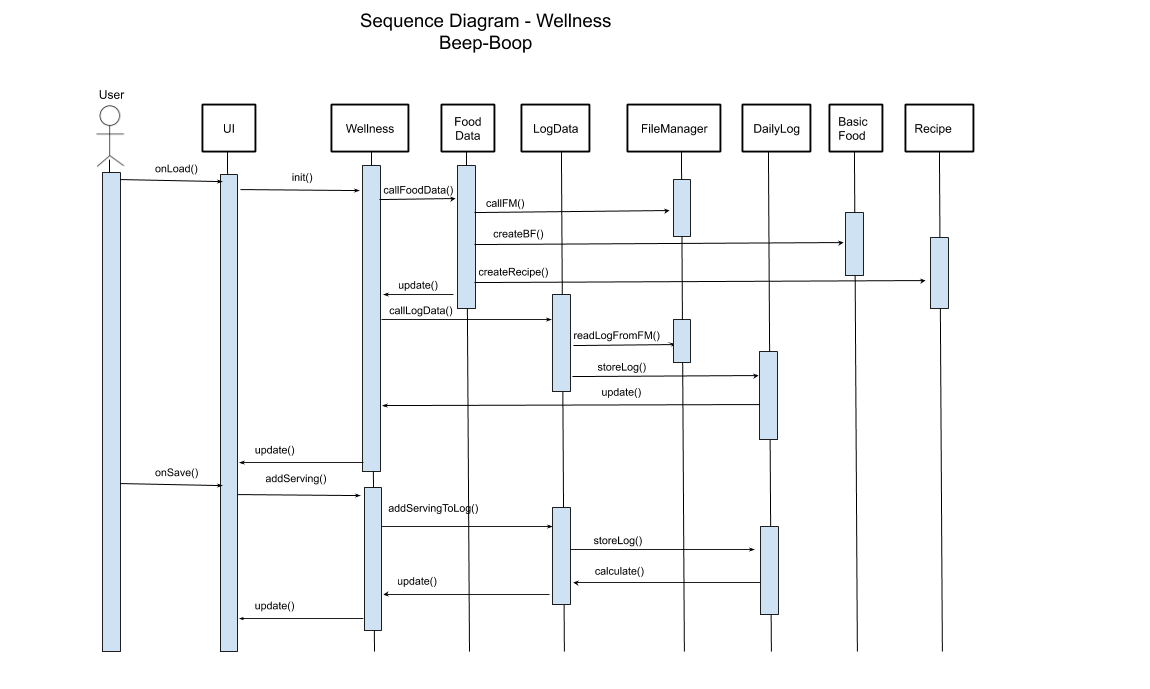
|  |  |
| --- | --- |
| **Class** File Manager | |
| **Responsibilities** | Write to file  Read from file |

# Sequence Diagrams



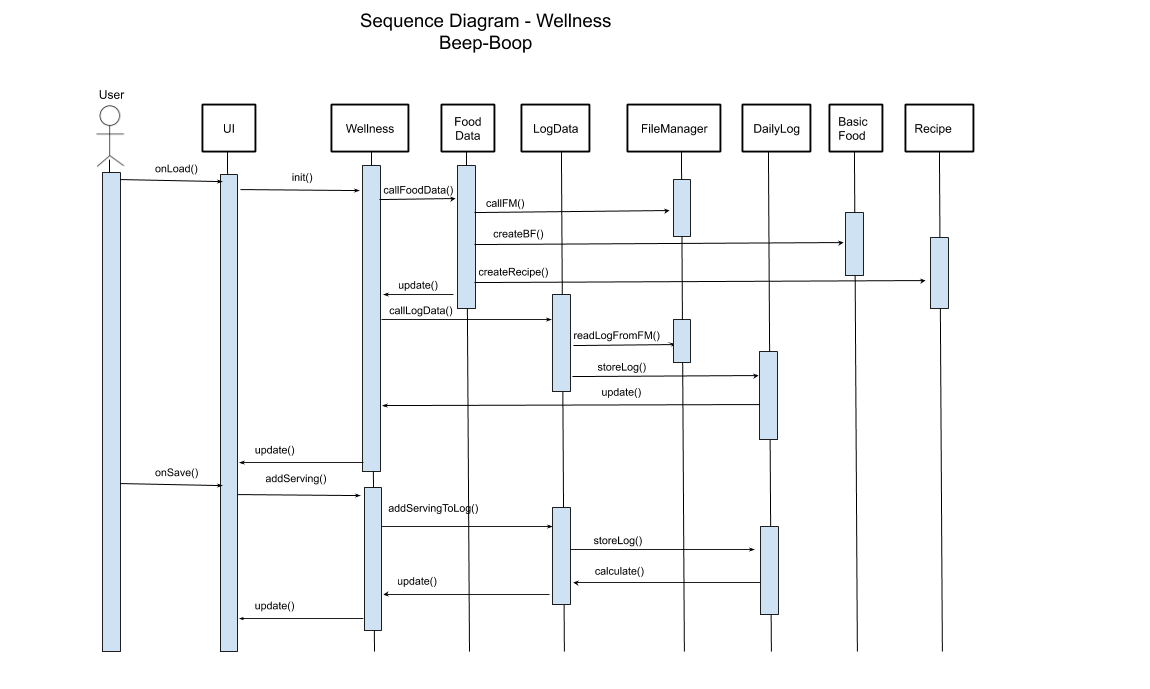
When the program executes, the user interface starts up first by calling init() to the wellness class. The wellness class will then call the Food Data then FileManager. The purpose of this is to retrieve the food information and assign it to the food data. The process is the same for the log information. It will get all of the cumulative logs and assign it to the log data. Once you have both models, you can now create basic food and recipes objects based on the food and recipe models. Next, the user will add a serving to the daily log. The UI will call the addServing() in Wellness then call addServingToLog in LogData class then add it to the daily log. Once a serving has been added to the daily log, the daily log will return the total number of calories. This number will be updated every time a new serving has been added just in case the user wants to view the current total of calories consumed the current date.

**Sequence Diagram Part #1**



This part of the sequence represents the software reading the food file and storing them into the array of objects of either food or recipe model when the program was opened. Then the software will call the LogData to proceed reading the log file from FileManager then store it into the DailyLog. Once all of the FoodData and LogData were set up ready then process to display all the data into UI for the user to view.

**Sequence Diagram Part #2**



This part of the sequence represents the user adding new servings of selected basic food and recipes to the LogData class then proceed to be recorded into DailyLog object. After that, the DailyLog will return the calculation of total calories consumed on the current date to be displayed on the UI. It will check if it passed the log’s date then it will be passed to the FileManager to add log to the file before process to create a new log for the new day.

# 

# Pattern Usage

## **Composite Model-View-Controller**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Composite Pattern** | |  | **Model-View-Controller Pattern** | |
| **Composite** | Recipe |  | **Model** | User  BasicFood  Recipe |
| **Leaf** | BasicFood |  | **View** | UI |
| **Component** | Food <abstract> |  | **Controller** | Wellness |

**Observer Mediator**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Observer Pattern** | |  | **Mediator Pattern** | |
| **Observer** | ActionListener <interface> |  | **Mediator** | FileManager |
| **Observable** | Wellness |  | **Objects** | Food.csv  Log.csv  Food data  Log data |