**Wellness Manager 2.0**

Project Design Document

TEAM DeepBlue

Tianpeng Kuang <tk5433@rit.edu>

Evan Hiltzik <eh8319@rit.edu>

Peter Schwarzkopf <pjs4644[@rit.edu](mailto:jtoa@rit.edu)>

Yongheng Mei <ym9276[@rit.edu](mailto:jtoa@rit.edu)>

Vincent Sze <vs7050[@rit.edu](mailto:jtoa@rit.edu)>

**Project Summary**

For this project, we are looking to make a program that is able to keep track of one’s wellness. We want to create a program that keeps track of vital health information from the user’s daily intake: calories, fat, carbohydrates, proteins. The program will provide a list of basic foods and their nutritional values, which can then be combined to create recipes within the system’s collection. The user will also be able to log foods and recipes into the collection; the program will display what the user ate during the day and will receive information about the foods’ nutrition and how it compares to their daily intake goal. Additionally, the user will be able to input their daily desired intake of calories and their current weight. There will be a section to keep track of daily exercising, how many calories burned from doing these exercises, and a comparison to the user’s goal versus what they burned. Keeping a log of daily nutrient intake and exercises will give the user a clearer picture of how their eating and exercise habits affect their wellness, as well as keeping the user responsible for their health.

# **Design Overview**

For our original design sketch, we went with a very simplified sketch. The food and recipe classes both went to a collective food database. The UI accessed both this database as well as the log which got its information from the food database. This system made it so that there was high coupling between the food collection, food, and recipe classes.

For our design sketch v1.0, we greatly improved upon the mistakes of our original design. We included much more detail into the workings of the Wellness Manager. The high coupling problem was fixed between the food and recipe classes by making the food and recipe classes controlled by commandController. The recipe class uses food for its data. An example of dependency inversion in our design is the foodComponent interface.The food class uses the food Component interface to manage items. Our separation of concerns is demonstrated by the food, recipe, and DailyLog classes writing to their respective CSV files. We also have the Wellness Manager class to initiate the view and the UI instead of putting that burden on a different class.

For our design sketch v2.0, we added a visual GUI and an exercise class to the diagram. The GUI contains four classes: NutritionCanvas, InteractionPanel, ButtonPanel, and HealthPanel, will initialize the view and display the immediate results of state changes. The exercise class will take in user input from the controller and write the specified exercise into a CSV file. Adding these classes will still follow the MVC pattern as the GUI, consisting of the four classes, fits into the view and the Exercise class fits into the model.

Our design sketch v2.0 clearly demonstrates good design principles as well as clear visuals of the inner workings of the Wellness Manager. The use of various classes to do their own tasks as well as the use of the MVC architectural structure improves the functionality and readability of the WellnessManager. The final document improves upon the errors of the original design in almost every aspect including S.O.C., high cohesion, low coupling, and dependency inversion.

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# **Subsystem Structure**

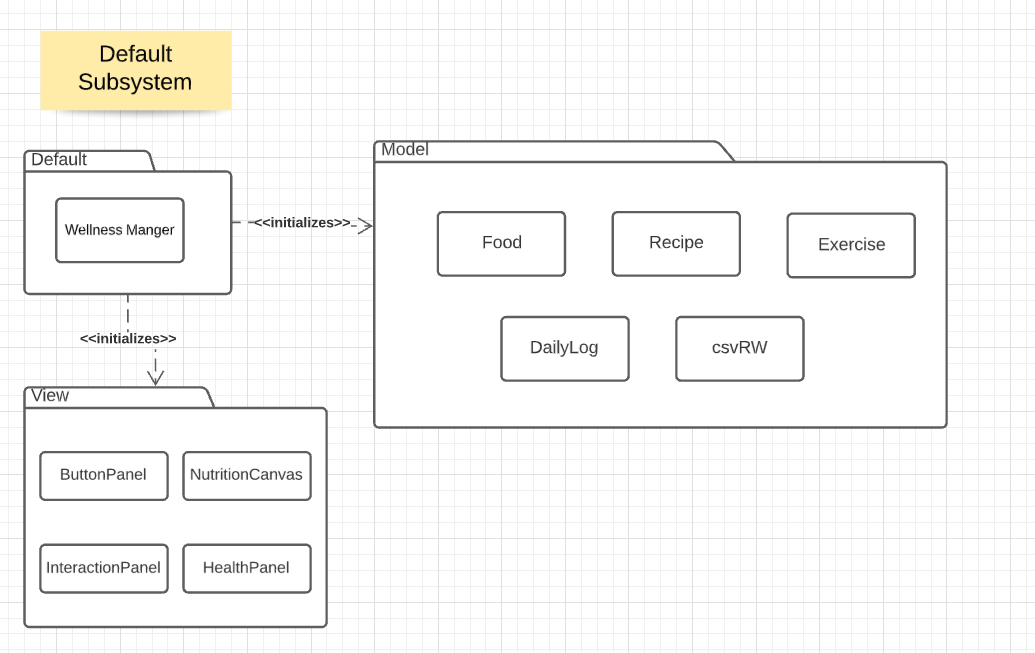
# 

# **Figure #1 : Subsystem Structure**

# **Subsystems**

## **Default Subsystem**

| **Class** Wellness Manager | |
| --- | --- |
| **Responsibilities** | Initialize the view  Initialize the controller |
| **Collaborators**  **(uses)** | **controller.GUIController** - creation of the controller  **view.GUI** - creation of GUI object |



# **Figure #2 : Default Subsystem**

**Model Subsystem**

| **Class** Food | |
| --- | --- |
| **Responsibilities** | Food objects are use to store food's nutrition information that user entered |
| **Collaborators**  **(uses)** | **NA** |

| **Class** Food | |
| --- | --- |
| **Responsibilities** | Food objects are use to store food's nutrition information that user entered |
| **Collaborators**  **(uses)** | **NA** |

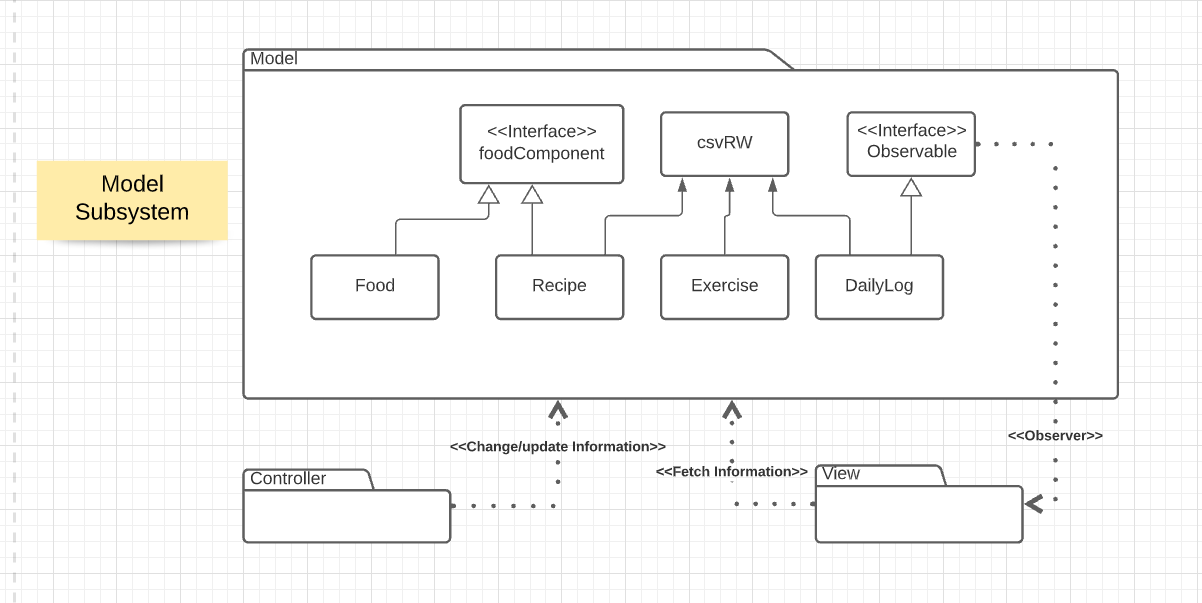
| **Class** csvRW | |
| --- | --- |
| **Responsibilities** | Reads from a csv file as well as writes to a csv file |
| **Collaborators**  **(implements)** | **NA** |
| **Collaborators**  **(uses)** | **java.io.\* -** use for BufferedReader, FileReader, BufferedWriter, FileWriter  **java.util.\* -** use for Arraylist and Arrays |

| **Class** Recipe | |
| --- | --- |
| **Responsibilities** | Recipe objects are use to create new recipe according to user selection from GUIController and writes the information to a csv file |
| **Collaborators**  **(uses)** | **model.csvRW** - write user input into a csv  **java.util.\*** - uses hashmap to store collection |

| **Class** dailyLog | |
| --- | --- |
| **Responsibilities** | DailyLog objects are used to store log attribute information  Pass information to log database class to store information in csv |
| **Collaborators**  **(uses)** | **model.csvRW** - uses input to write data to a csv file  **java.util.\* -** use for Arraylist |

| **Class** foodComponent (interface) | |
| --- | --- |
| **Responsibilities** | Interfaces that provide methods that are implemented by the composite object. |

| **Class** Exercise | |
| --- | --- |
| **Responsibilities** | Detects burned calories; creates and names exercises |
| **Collaborators**  **(uses)** | **model.csvRW -** uses input to write data to csv file |



# **Figure #3 : Model Subsystem**

## 

## 

## 

## 

## **View Subsystem**

| **Class** NutritionCanvas | |
| --- | --- |
| **Responsibilities** | Contains the Bar Chart canvas showing nutrition |
| **Collaborators**  **(uses)** | **java.awt.\* -** user for drawing awt shapes and canvas |

## 

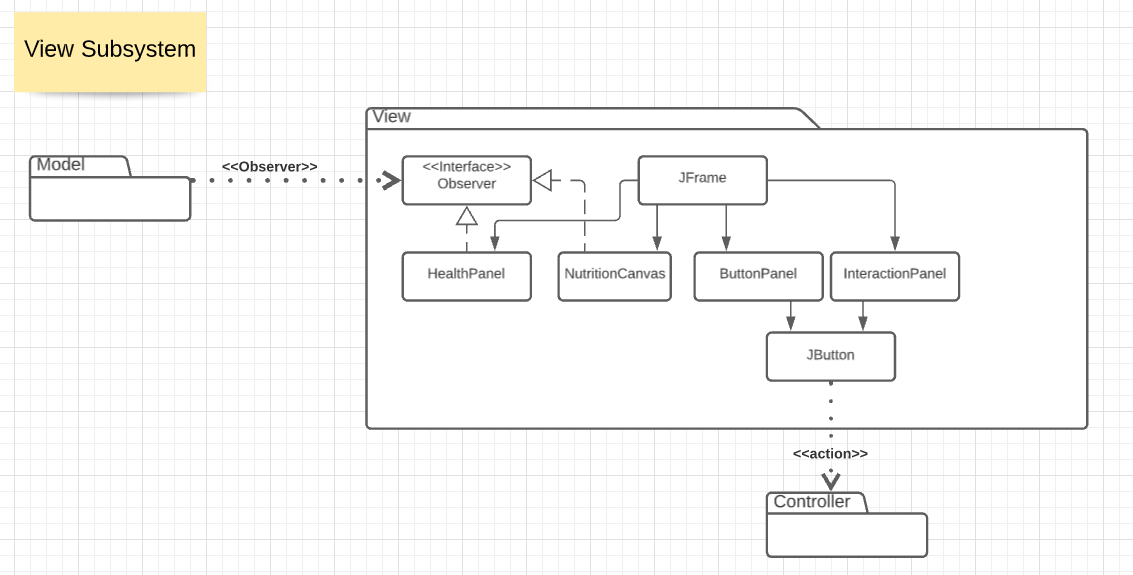
| **Class** InteractionPanel | |
| --- | --- |
| **Responsibilities** | Contains JTextArea to display messages  Contain JTextField to allow user to input data |
| **Collaborators**  **(uses)** | **java.awt.\* -** use for awt layout  **java.swing.\* -** use for jpanel, jtextarea, jtextfield, jbutton, ...etc components. |

## 

| **Class** ButtonPanel | |
| --- | --- |
| **Responsibilities** | Contains JButtons for different types of command |
| **Collaborators**  **(uses)** | **java.awt.\* -** use for jpanel layout  **java.swing.\* -** primary use for jbutton, and jpanel |

## 

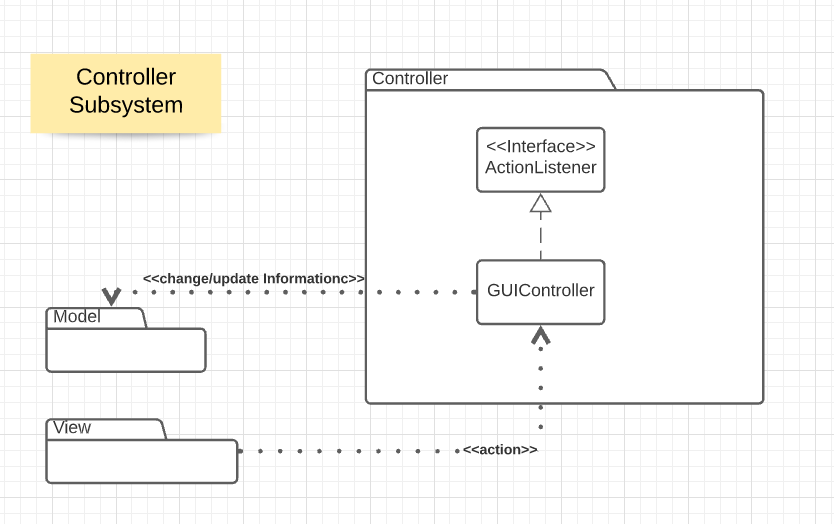
| **Class** HealthPanel | |
| --- | --- |
| **Responsibilities** | Contains user information such as calorie, weight, and nutrition consumed.  Contains JLabel for displaying information |
| **Collaborators**  **(uses)** | **java.awt.\* -** use for jpanel layout  **java.swing.\* -** use for jlabel, jpanel, ...etc components.  **java.util.\* -** use for observer and observable |



# **Figure #4 : View Subsystem**

## **Controller Subsystem**

| **Class** GUIController | |
| --- | --- |
| **Responsibilities** | GUIController controls all of the input obtained from the user and writes it to csv files |
| **Collaborators**  **(uses)** | **model.csvRW** - uses input to write data to a csv file  **model.food** - creates food object  **model.dailyLog** - creates dailyLog object  **model.recipe** - creates recipe object  **view.HealthPanel** - gets user input |

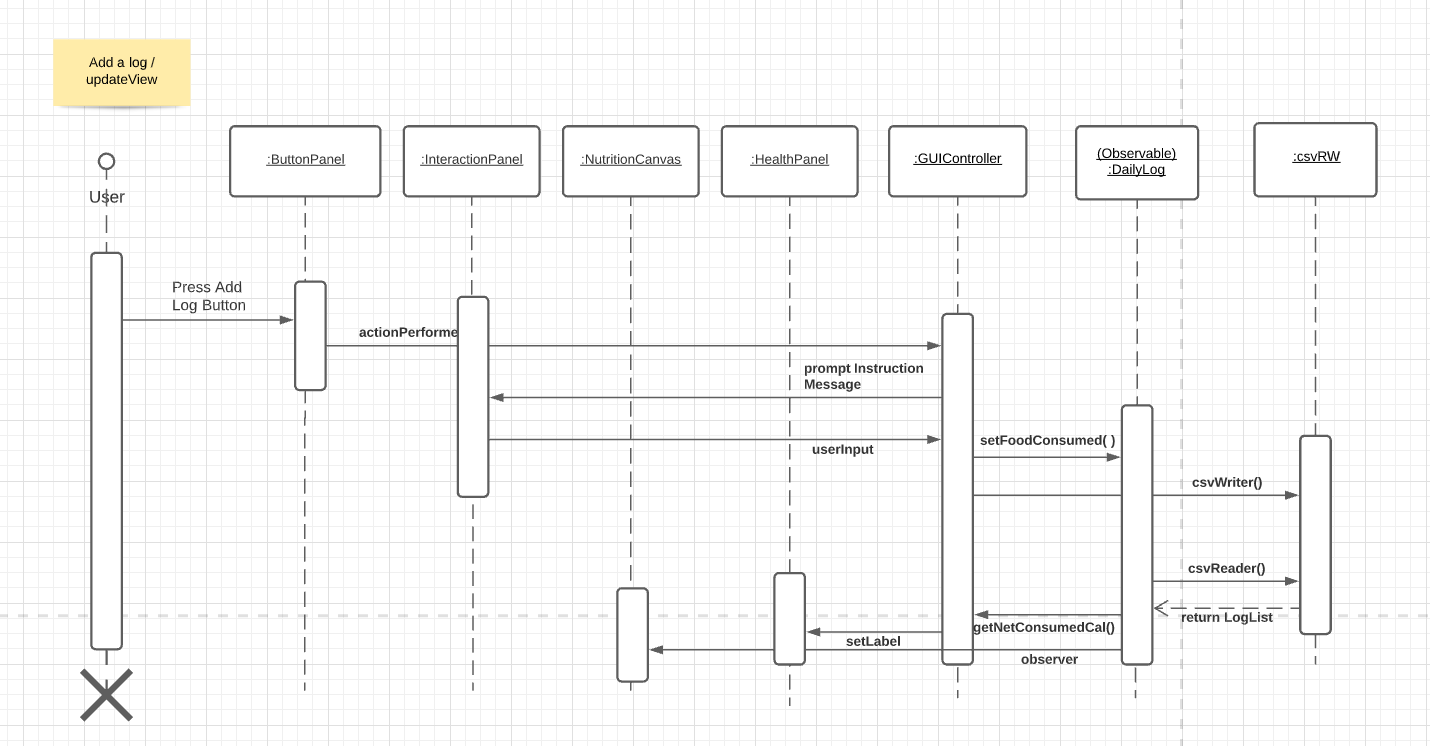


# **Figure #5 : Controller Subsystem**

# 

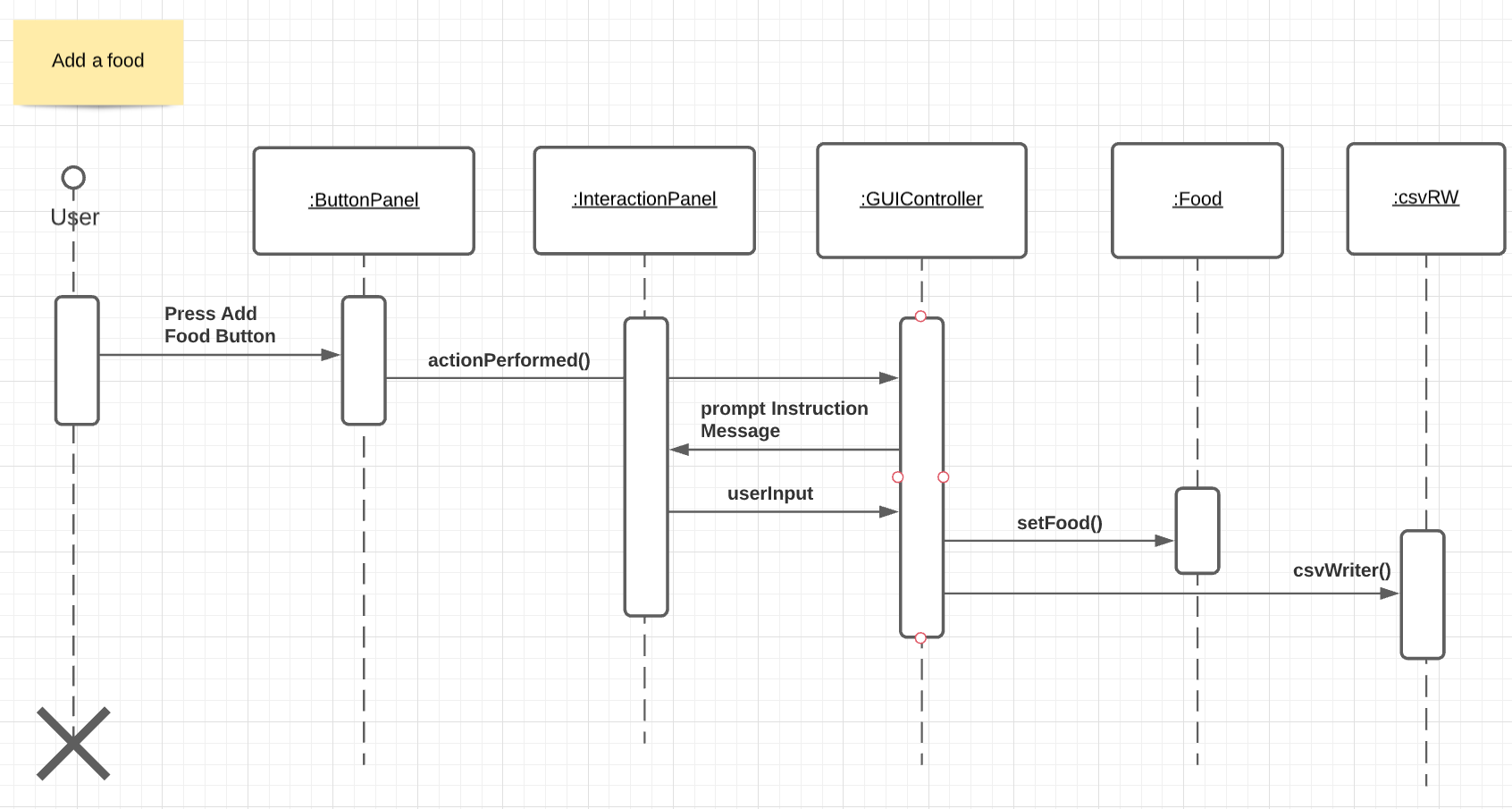
# **Sequence Diagrams**

## 



## **Sequence #1 : Add log entry and update view**

To begin, the client presses the Add Log button. A signal is detected by our GUIController, who then tells the InteractionPanel to prompt the user for further information (calories consumed, calories burnt, exercise names). After this information is entered and confirmed, GUIController will signal DailyLog to update our CSV with the userInput provided. Finally, we update our display (NutritionCanvas) using the input our CSV was updated with.



**Sequence #2 : Add Basic Food**

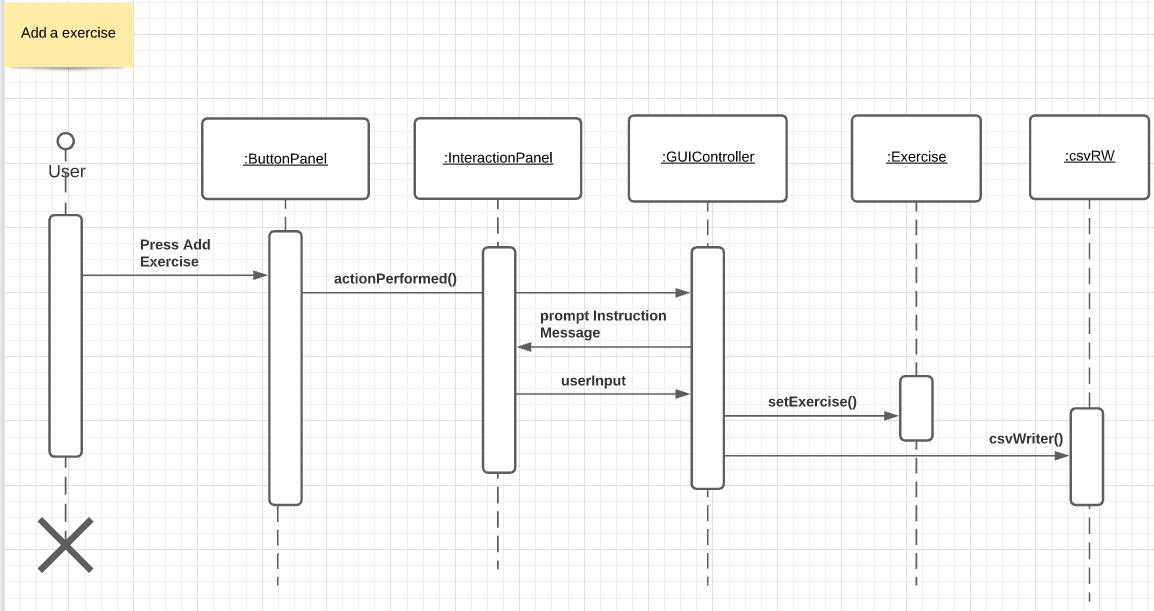
First, the user presses the Add Food button on our ButtonPanel. This, similar to the Add Log function, notifies our GUIController to ask our InteractionPanel to prompt for further information (food name, calories, macros, and servings). The userInput is then sent to our Food class, to ensure it's properly identified, as well as to our CSV for updating.

# 

**Sequence #3 : Set weight for the day**

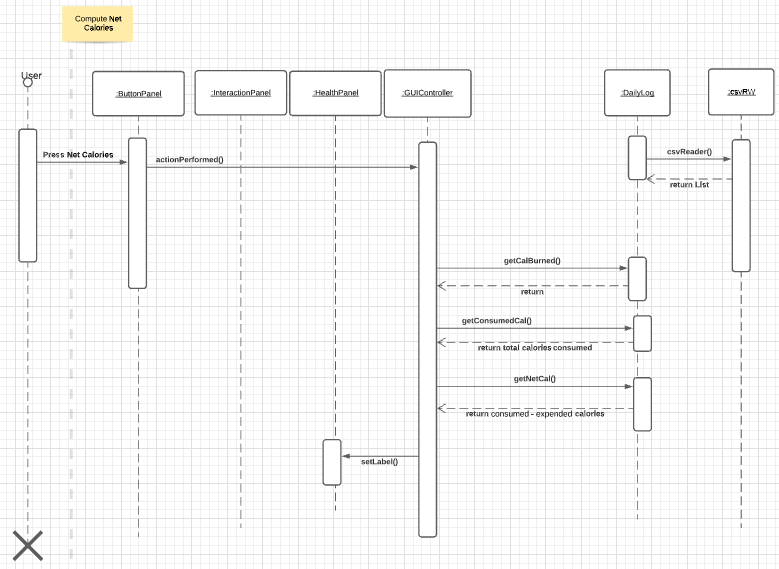
After pressing the Set Weight button, this will prompt the GUIController to tell the InteractionPanel to prompt the user with instructions on how to insert their weight. After the user inputs their weight, this input will then go to the GUIController. The GUIController will then call the setWeight() function and add the user’s weight to the DailyLog. After doing that, DailyLog will get that weight and then return that value to the GUIController, which then updates the label on the HealthPanel that is labelled weight to the correct weight for the day.

# 

****

**Sequence #4 : add Exercise Entry**

The user presses the Add Exercise button - our GUIController tells the story to prompt for more information (exercise name). The userInput is then sent to both the Exercise class, to ensure it's properly labelled, and to our csvRW to update our log info.

****

**Sequence #5 : Net Calories**

The user clicks the Net Calories button, the controller who calls getCalBurned from daily log. Then, getConsumedCal is retrieved from DailyLog, who is then asked to take both CalsBurned and ConsumedCals and getTotalCal from this information. After the net calories have been calculated, the information is sent back to our GUIController, who updates the graph.

# 

# 

# 

# 

# 

# 

# **Pattern Usage**

## **Pattern #1 Composite Pattern**

Description: Use a composite pattern to describe a group of objects that is treated the same way as a single instance of the same type of object. RecipeFood Database Log Database BasicFood Database is leaf, composite is FoodCollection. It provides flexibility of structure with manageable class or interface.

| **Composite Pattern** | |
| --- | --- |
| **Component(s)** | foodComponent |
| **Composite(s)** | recipe |
| **Leaf** | food |

## 

## **Pattern #2 Model-View-Controller Pattern**

Description: use to decouple user-interface data , and application logic. This pattern helps to achieve separation of concerns. recognize commands to the controllers, display the change to the view, manipulate BasicFood Database recipeFood Database Log Database as a model.

| **MVC Pattern** | |
| --- | --- |
| **Model(s)** | Food, Recipe, DailyLog, csvRW, foodComponent, Exercise |
| **View(s)** | NutritionCanvas, InteractionPanel, ButtonPanel, HealthPanel |
| **Controller(s)** | GUIController |

## **Pattern #3 Observer Pattern**

Description: use to notify the view of a state change. This pattern helps to achieve less coupling and immediately update the view of state changes for the graph.

## 

## 

| **Observer Pattern** | |
| --- | --- |
| **Observer** | HealthPanel |
| **Observable** | DailyLog |

## 

# 

# 

# 

# 

# 

# 

# 

# 

# **Rationale**

**Composite Pattern**

We decided to use the Composite Pattern because it allows us to treat *leaf* and *composite* objects uniformly, making client classes easier to implement, change, test, and reuse. This allows for the user to interact with the *component* in order to refer to the databases uniformly.

**MVC Pattern**

We decided to use the MVC Pattern for its ability to promote cohesion and lessen coupling. This allows us to separate the concerns for the model, view, and controller into their own subsystems and manage what needs to be connected between them. Whenever the user interacts with the food class, the controller will handle that interaction and update the model accordingly, which will then update the view accordingly.

**Observer Pattern**

We decided to use the Observer pattern for its ability to notify a subject of state changes.This allows us to immediately provide an update to the GUI for the user. Whenever the user uses a button to add a food, remove a food, etc., the food’s calories will be calculated and output on a graph in the GUI. Using the Observer pattern will also loosen coupling between the classes.