Wellness Manager 2.0

Project Design Document

TEAM DeepBlue

Tianpeng Kuang <tk5433@rit.edu>

Evan Hiltzik <eh8319@rit.edu>

Peter Schwarzkopf <pjs4644@rit.edu>

Yongheng Mei <ym9276@rit.edu>

Vincent Sze <vs7050@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, an exercise class, and a new controller 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. The ButtonListener class acts as a controller for all the buttons in the GUI.

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

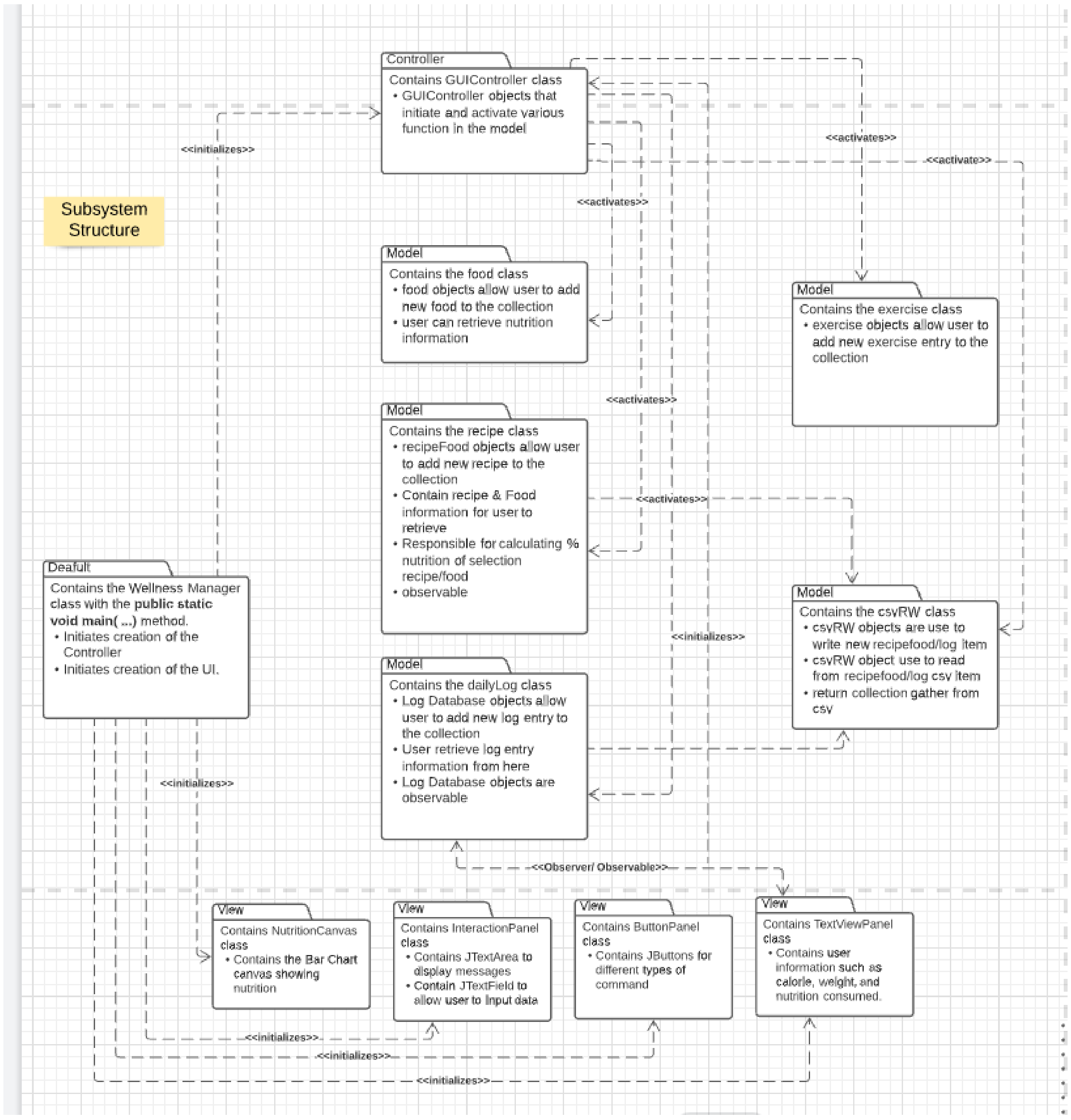

Subsystem Structure
Controller
Contains GUIController class
• GUIController objects that initiate and activate various function in the model
<<initializes>>
<<activates>>
<<activates>>
<<activate>>
Model
Contains the food class
• food objects allow user to add new food to the collection
• user can retrieve nutrition information
Model
Contains the exercise class
• exercise objects allow user to add new exercise entry to the collection
<<activates>>
Model
Contains the recipe class
• recipeFood objects allow user to add new recipe to the collection
• Contain recipe & Food information for user to retrieve
• Responsible for calculating % nutrition of selection recipe/food
• observable
<<activates>>
Deafult
Contains the Wellness Manager class with the public static void main(...) method.
• Initiates creation of the Controller
• Initiates creation of the UI.
Model
Contains the csvRW class
• csvRW objects are use to write new recipefood/log item
• csvRW object use to read from recipefood/log csv item
• return collection gather from csv
<<initializes>>
Model
Contains the dailyLog class
• Log Database objects allow user to add new log entry to the collection
• User retrieve log entry information from here
• Log Database objects are observable
<<initializes>>
<<Observer/ Observable>>
View
Contains NutritionCanvas class
• Contains the Bar Chart canvas showing nutrition
View
Contains InteractionPanel class
• Contains JTextArea to display messages
• Contain JTextField to allow user to input data
View
Contains ButtonPanel class
• Contains JButtons for different types of command
View
Contains TextViewPanel class
• Contains user information such as calorie, weight, and nutrition consumed.
<<initializes>>
<<initializes>>
<<initializes>>

Figure #1 : Subsystem Structure

Subsystems

Default Subsystem

Class Wellness Manager	
Responsibilities	Initialize the view Initialize the controller
Collaborators **(uses)**	**controller.ButtonListener** - creation of ButtonListener object **view.HealthPanel** - creation of HealthPanel object **view.InteractionPanel** - creation of InteractionPanel object **view.ButtonPanel** - creation of ButtonPanel object **view.NutritionCanvas** - creation of Nutrition Canvas object **model.dailyLog** - creation of dailyLog Object

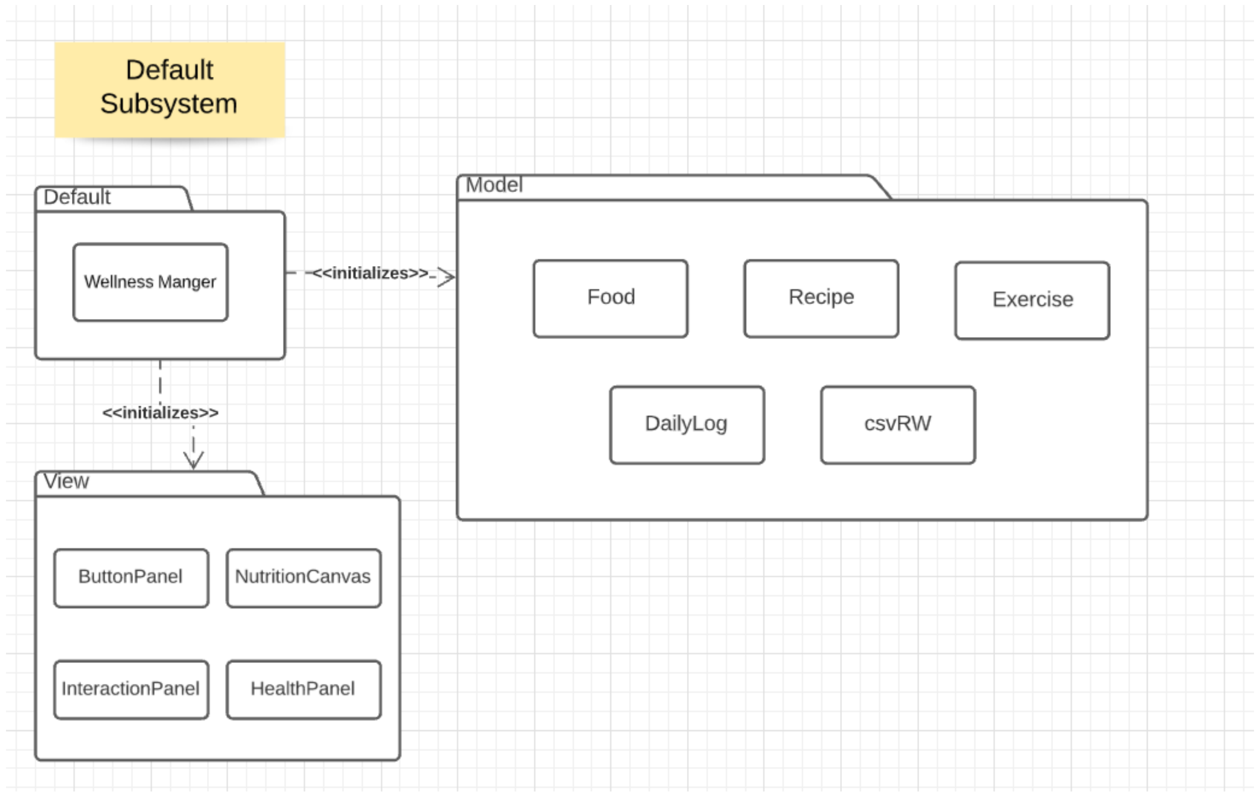


Figure #2 : Default Subsystem

Model Subsystem

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

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

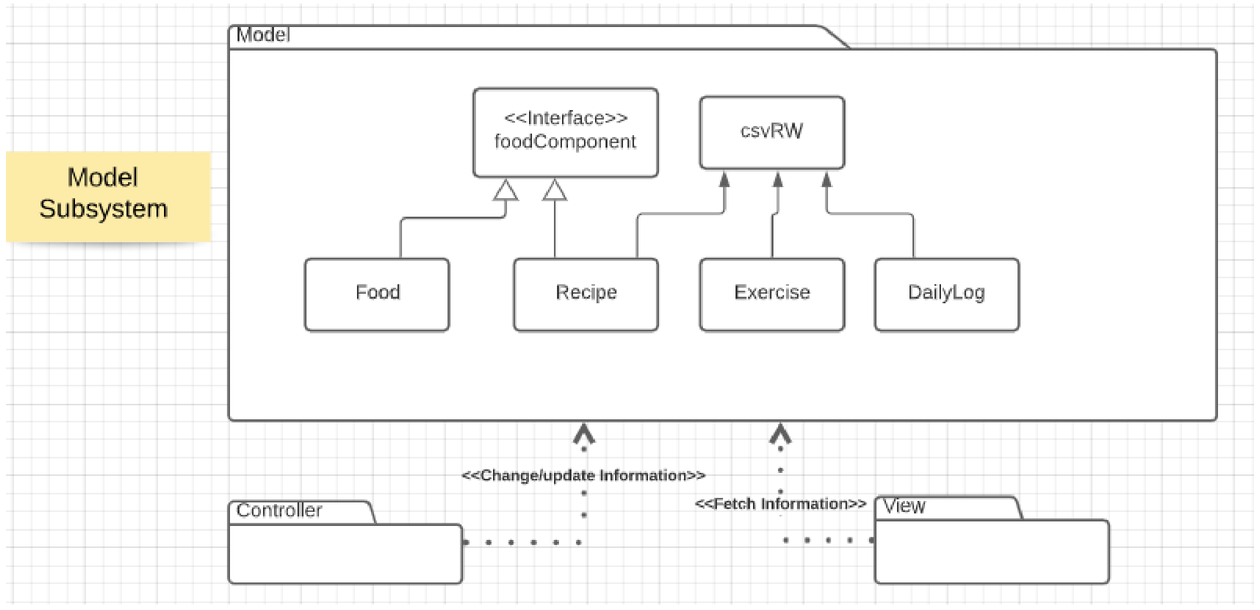


Figure #3 : Model Subsystem

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. **model.dailyLog** - creates a dailyLog object.

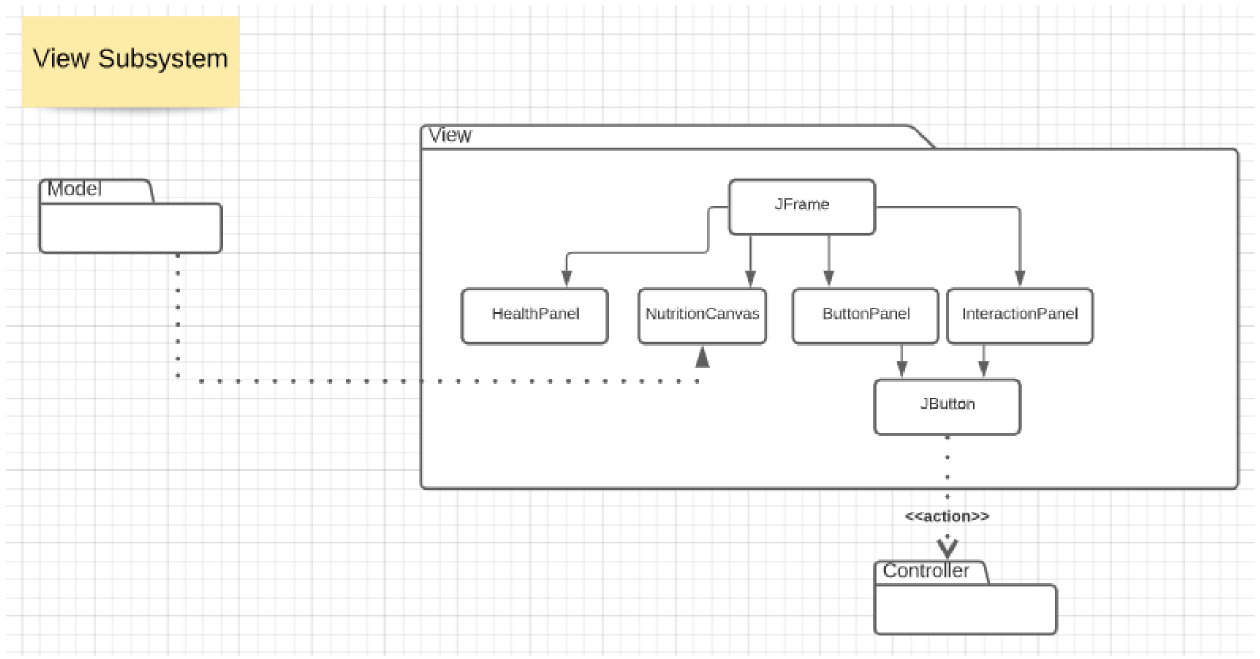


Figure #4 : View Subsystem

Controller Subsystem

Class ButtonListener	
Responsibilities	ButtonListener 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.*** - creates all model objects **view.HealthPanel** - creates HealthPanel object **view.InteractionPanel** - creates InteractionPanel object **view.ButtonPanel** - creates ButtonPanel object

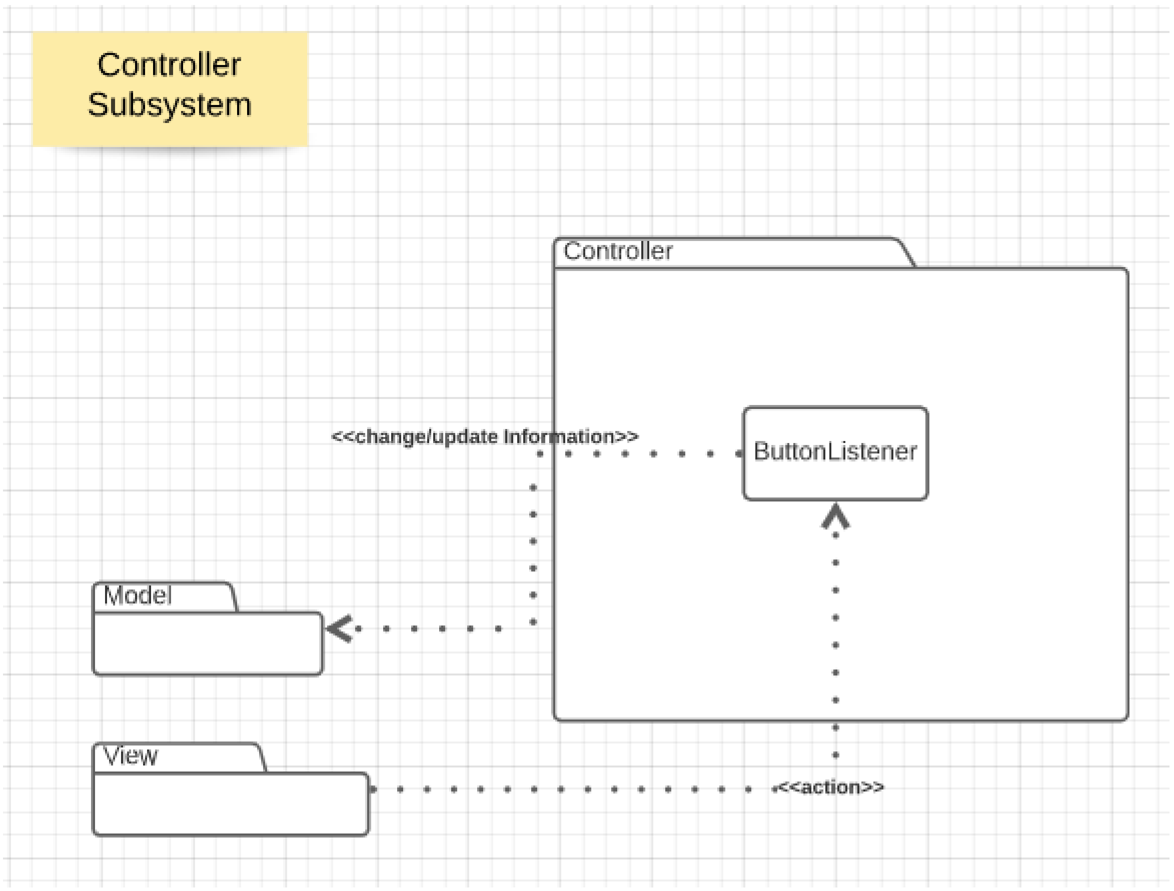
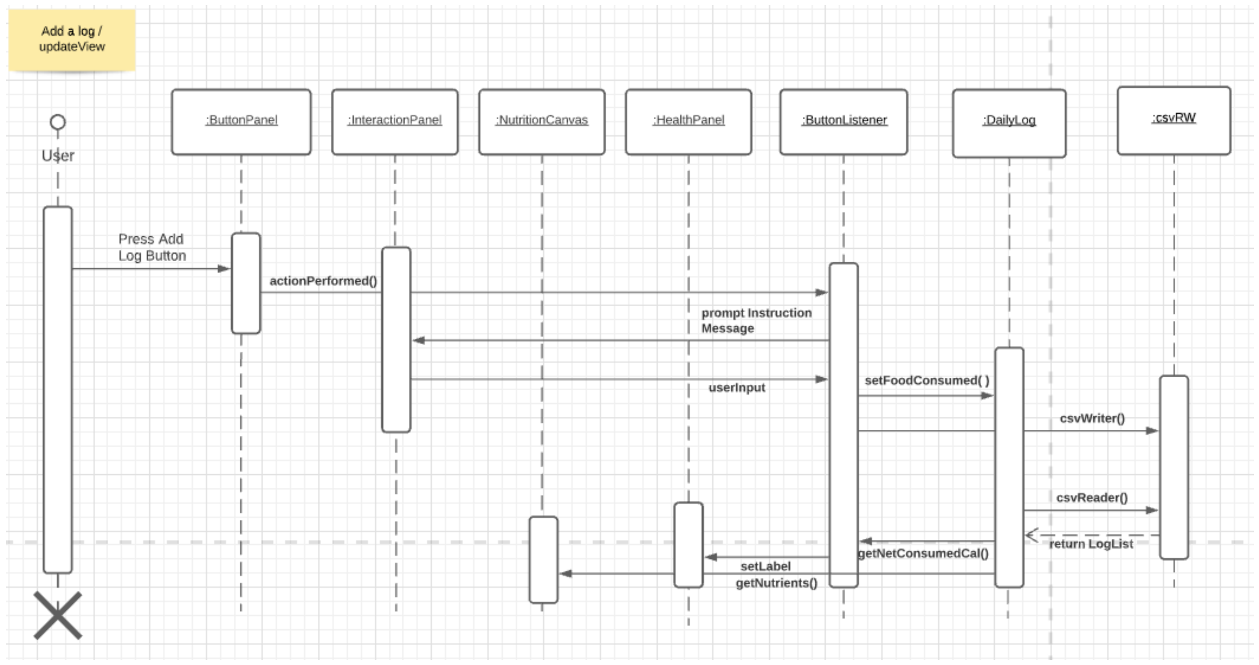


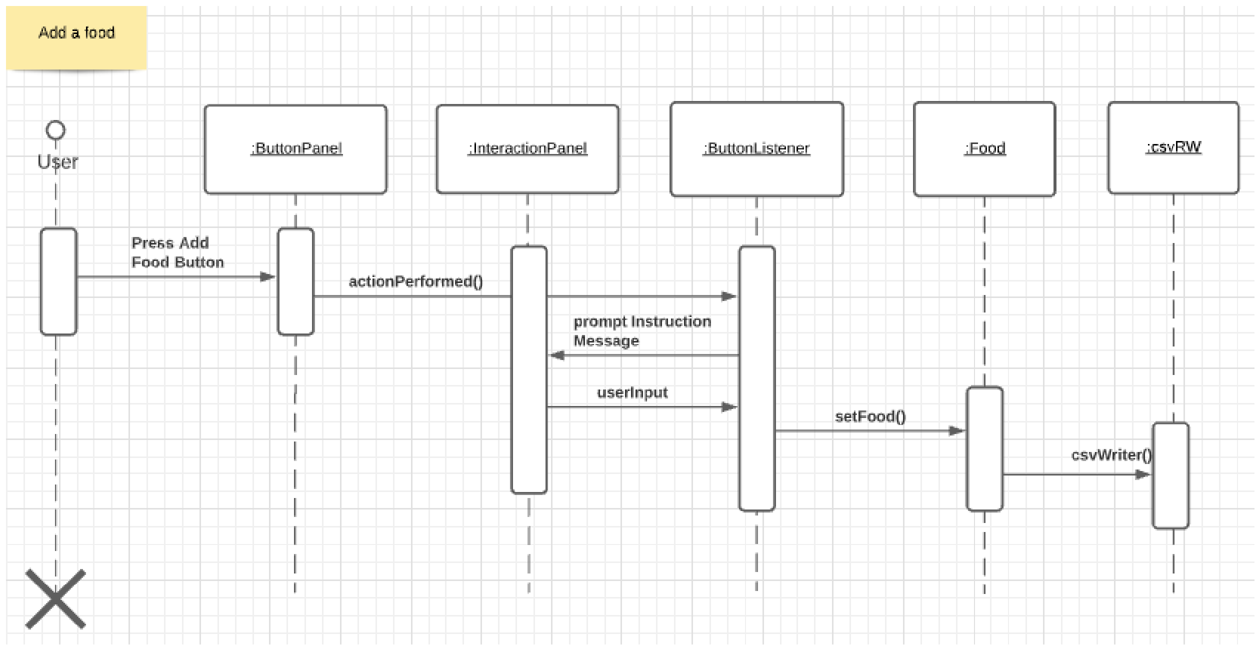
Figure #5 : Controller Subsystem

Sequence Diagrams



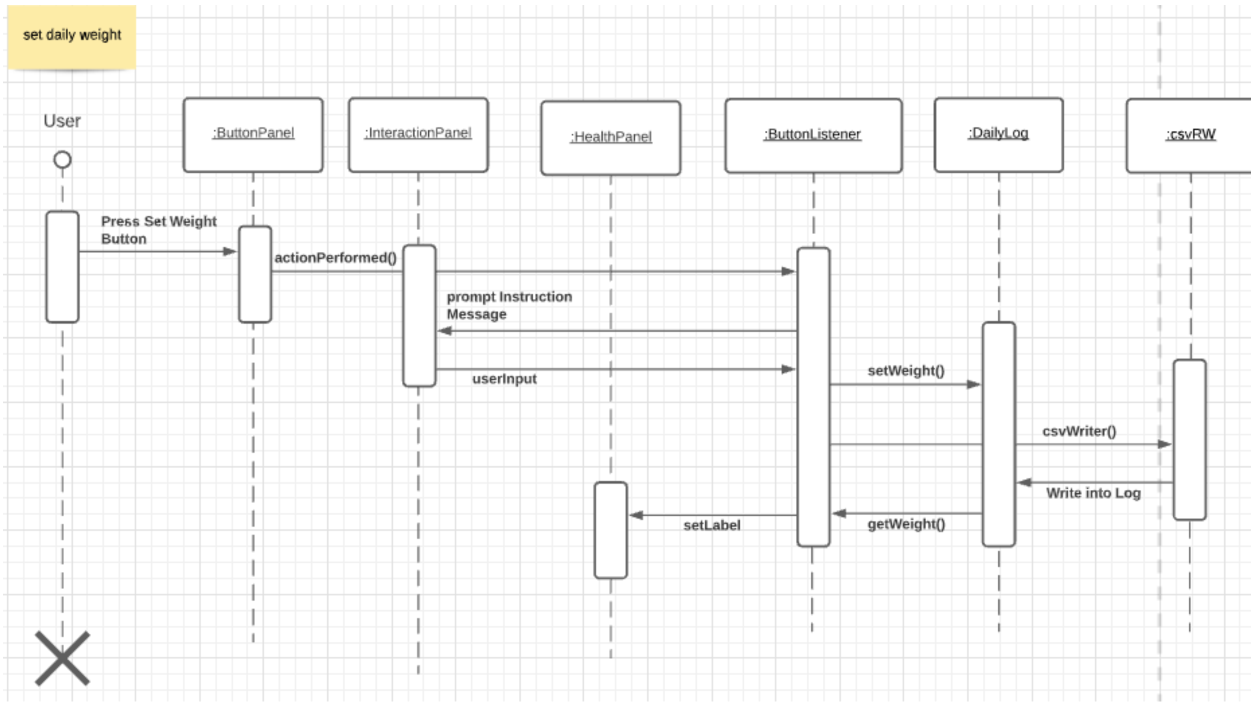
Sequence #1 : Add log entry and update view

To begin, the client presses the Add Log button. A signal is detected by our ButtonListener, 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, ButtonListener 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.



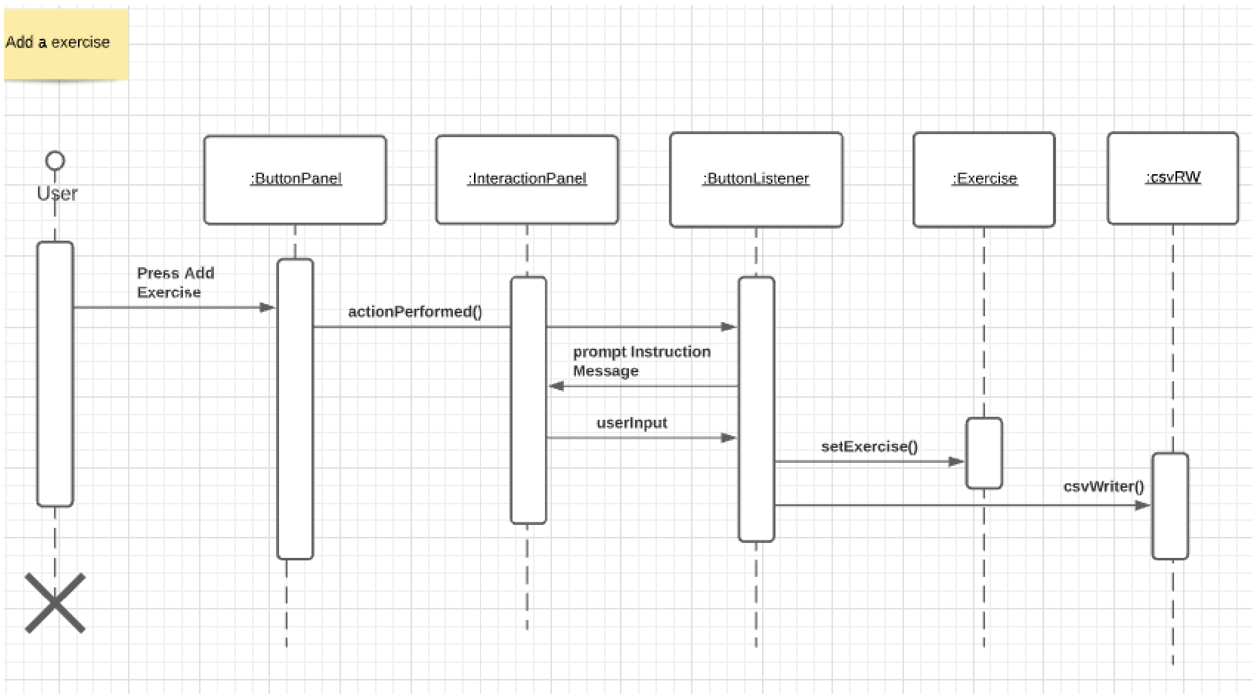
Sequence #2 : Add Basic Food

First, the user presses the Add Food button on our ButtonPanel. This, similar to the Add Log function, notifies our ButtonListener 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.



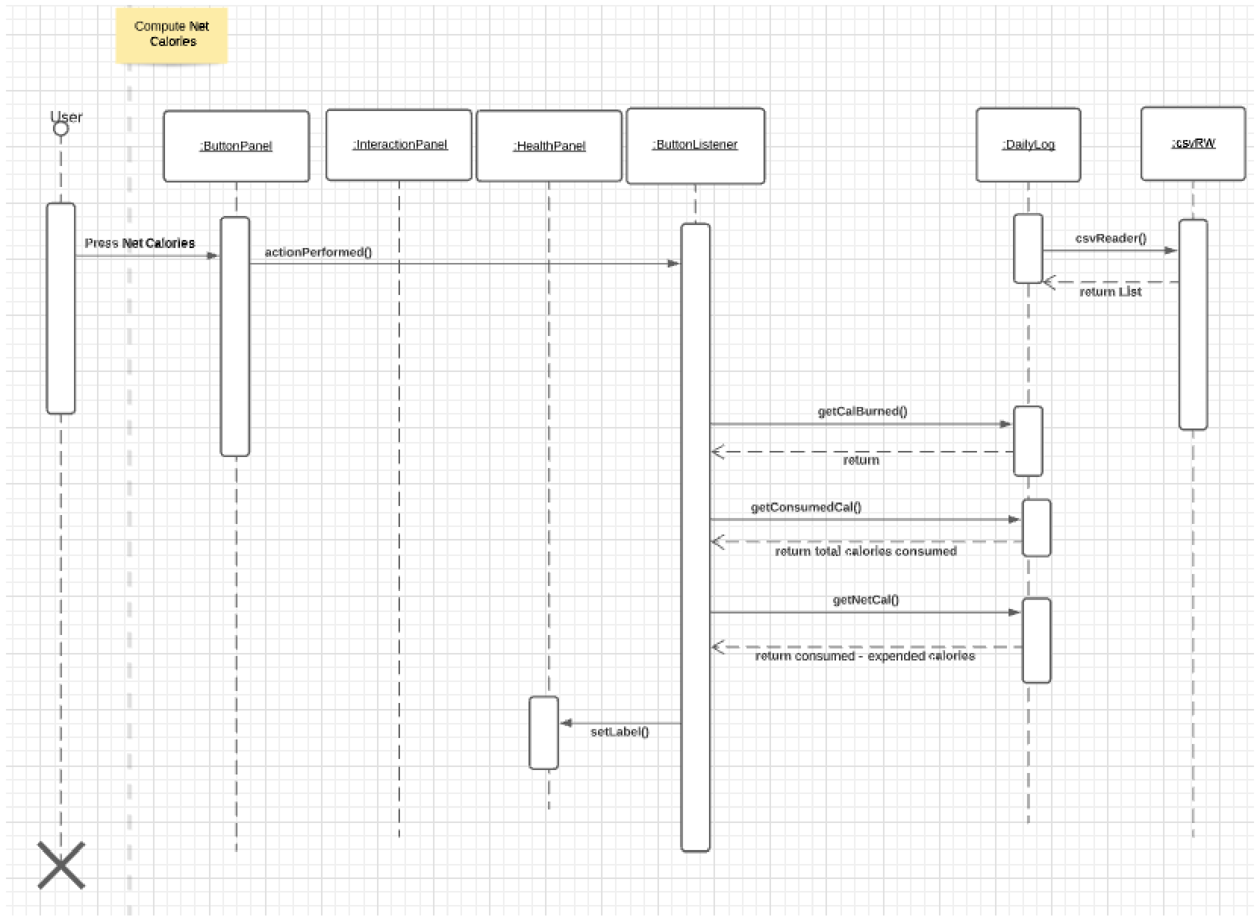
Sequence #3 : Set weight for the day

After pressing the Set Weight button, this will prompt the ButtonListener 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 ButtonListener. The ButtonListener 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 ButtonListener, 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 ButtonListener 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 called both getCalBurned and getConsumedCal() then passes these two values into getNetCal() so it can compute the net calories and return the value to the controller 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)	ButtonListener

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.