**Wellness Manager 1.0**

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 Szei <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. Keeping a log of daily nutrient intake will give the user a clearer picture of how their eating habits affect their wellness as well as keeping the user responsible for their health.

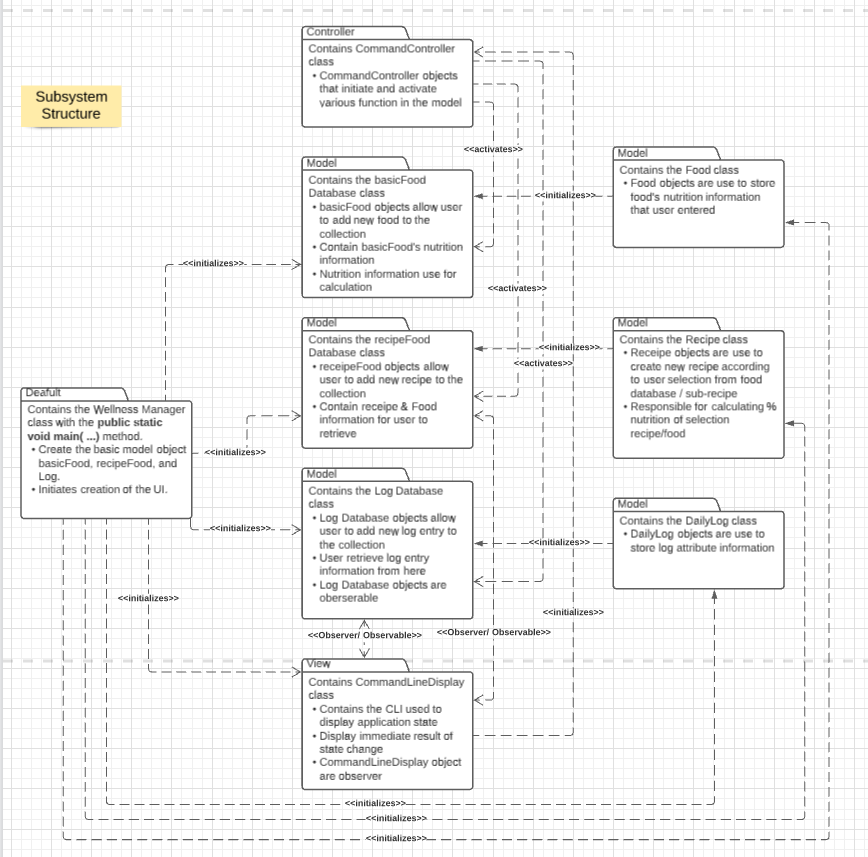
# **Design Overview**

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 final design sketch, 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, recipe, and food collection classes by making a food class that was stored in a database called BasicFood Database. The recipe class goes into a database called RecipeFood Database. An example of dependency inversion in our design is the Component interface. All of the databases in the design use the Component interface to manage items. Our separation of concerns is demonstrated by the food, recipe, and DailyLog classes having their own corresponding databases to do their own tasks. We also have the Wellness Manager class to initiate the view and the UI instead of putting that burden on a different class.

Our final design document 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 multiple databases make this diagram flow smoothly and effectively. 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**

******

# 

# 

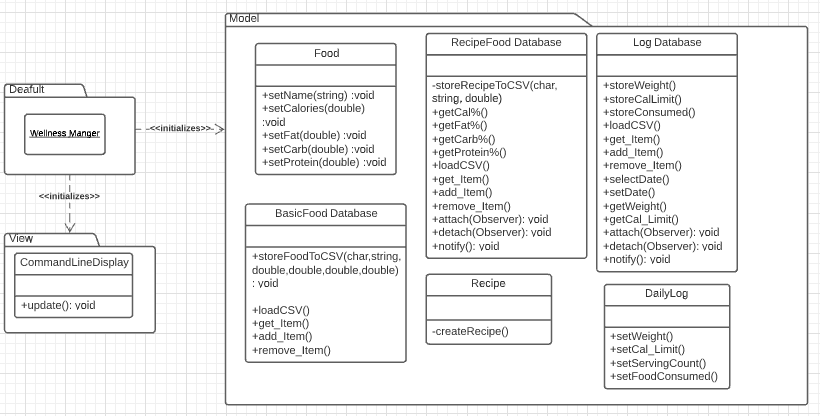
# 

# 

# **Subsystems**

## **Default Subsystem**

|  |  |
| --- | --- |
| **Class** Wellness Manager | |
| **Responsibilities** | Create the model object(s).  Initiates creation of CLI. |
| **Collaborators**  **(uses)** | **model.Food**- creation of food object  **model.basicFood Database** - creation of basicFood object  **mode.recipeFood Database** - creation of recipeFood database  **model.recipe** - creation of recipe object  **mode.Log Database** - creation of log database object  **model.DailyLog** - creation of daily log object  **view.CommandLineDisplay** - creation of commandLineDisplay object |



**Model Subsystem**

|  |  |
| --- | --- |
| **Class** Food | |
| **Responsibilities** | Food objects are use to store food's nutrition information that user entered |
| **Collaborators**  **(uses)** | **model.basicFood Database** - pass food’s nutrition information by creating basicFood Database object and call storeFoodToCSV() method to store into CSV. |

|  |  |
| --- | --- |
| **Class** BasicFood Database | |
| **Responsibilities** | basicFood objects contain basicFood nutrition information that is stored in arraylist. |
| **Collaborators**  **(implements)** | **model.component (interface)** - implement methods defined in the component interface. |

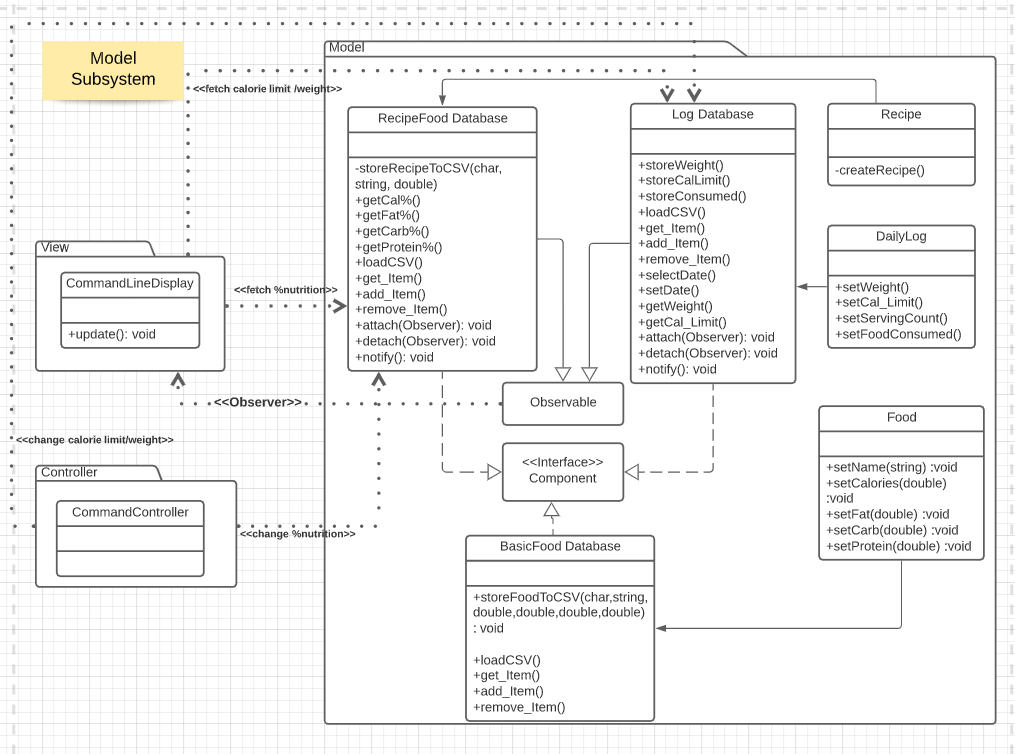
|  |  |
| --- | --- |
| **Class** Recipe | |
| **Responsibilities** | Recipe objects are use to create new recipe according to user selection from food database / sub-recipe  Pass information to recipeFood database class to store information in csv |
| **Collaborators**  **(uses)** | **model.recipeFood Database** - retrieve food(s) & recipe(s) from database to form a new recipe. call storeRecipeToCSV() method to store into CSV. |

|  |  |
| --- | --- |
| **Class** recipeFood Database | |
| **Responsibilities** | recipe Food objects allow user to add new recipe to the csv  Contain recipe & Food information for user to retrieve  Responsible for calculating % nutrition of selection recipe/food  Notify observer of any changes |
| **Collaborators**  **(inheritance)** | **java.util.Observable** - when % nutrition changes can be observed by others. |
| **Collaborators**  **(uses)** | **java.util.Observer -** for notifications of % nutrition changes to views. |
| **Collaborators**  **(implements)** | **model.component (interface)** - implement methods defined in the component interface. |

|  |  |
| --- | --- |
| **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.Log Database -** pass log information by creating a Log Database object and call store() method to store into CSV. |

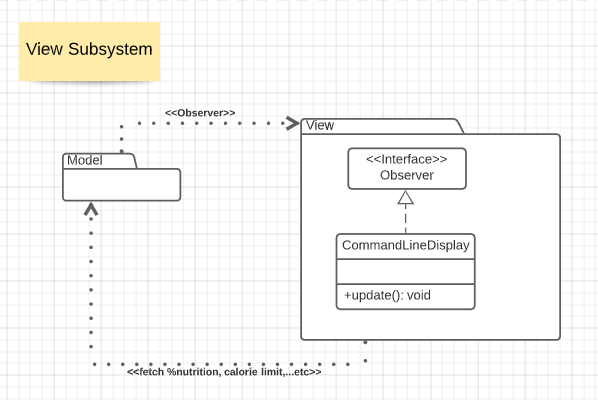
|  |  |
| --- | --- |
| **Class** Log Database | |
| **Responsibilities** | Log Database objects allow user to add new log entry to the collection  User retrieve log entry information from here  Log Database objects are observable |
| **Collaborators**  **(inheritance)** | **java.util.Observable** - when additions/deletion/modification of food consumed, changes can be observed by others. |
| **Collaborators**  **(uses)** | **java.util.Observer -** for notifications of calorie limit, weight, food consumed, changes to views. |
| **Collaborators**  **(implements)** | **model.component (interface)** - implement methods defined in the component interface. |

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



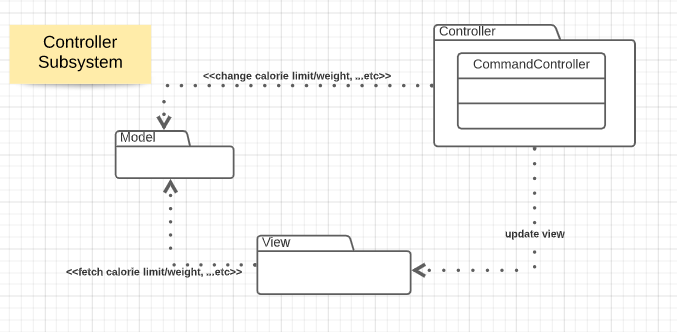
## **View Subsystem**

|  |  |
| --- | --- |
| **Class** CommandLineDisplay | |
| **Responsibilities** | Contains the CLI used to display application state  Display immediate result of state change  CommandLineDisplay object are observer |
| **Collaborators**  **(uses)** | **model.recipeFood Database -** fetch % nutrition information  **model.Log Database -** fetch calorie limit, weight information  **java.util.Observable -** to register for Log DataBase change notifications. |
| **Collaborators**  **(implements)** | **java.util.Observer** - to observe the calorie limit / weight / food consumed / % nutrition / ...etc |



## **Controller Subsystem**

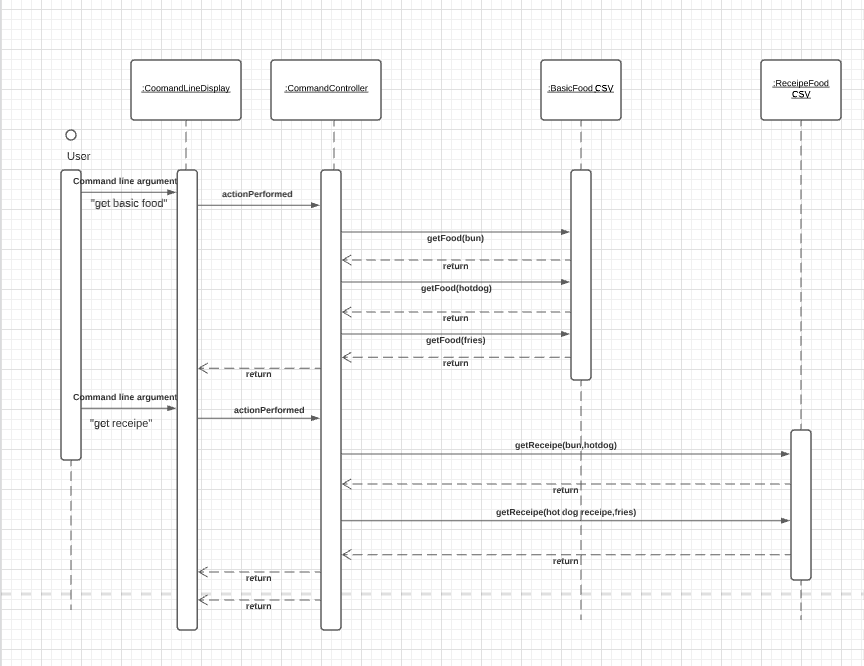
|  |  |
| --- | --- |
| **Class** CommandController | |
| **Responsibilities** | CommandController objects that initiate and activate various function in the model |
| **Collaborators**  **(uses)** | **model.BasicFood Database -** add a basic food  **model.recipeFood Database -** add a recipe, compute total calorie  **model.Log Database -** set weight , set calorie limit, add food consumed, add serving, add log entry  **view.CommandLineDisplay -** update display, prompt user corresponding information |



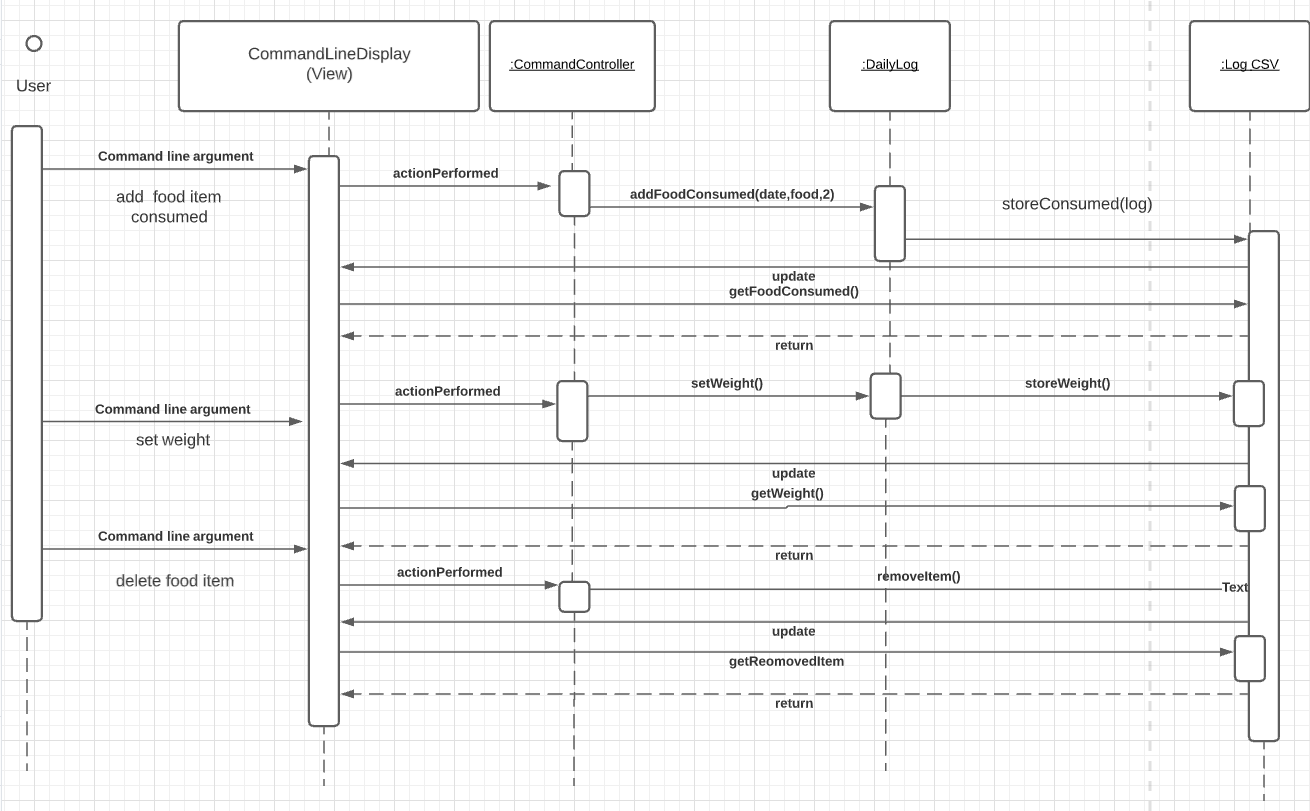
# 

# **Sequence Diagrams**

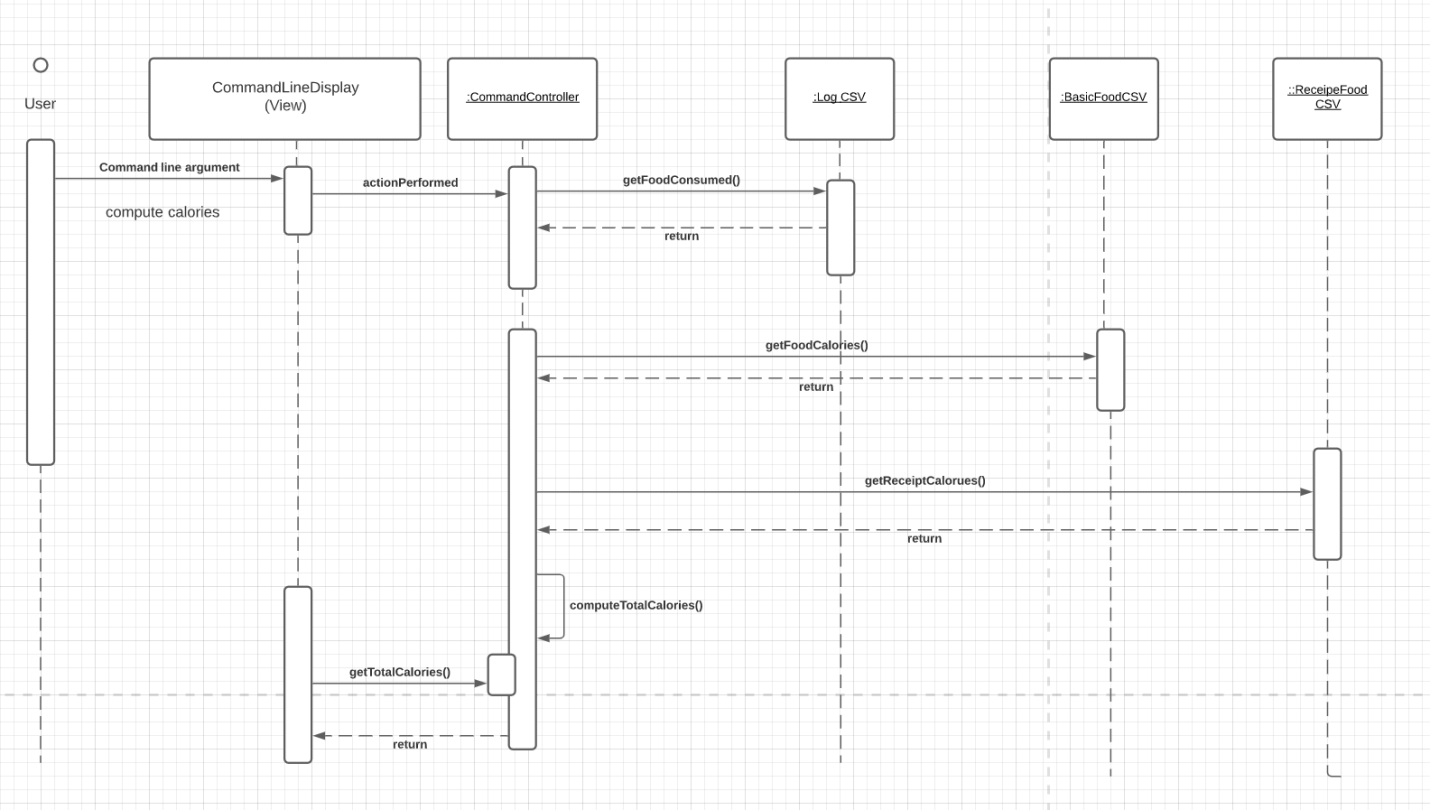
## 



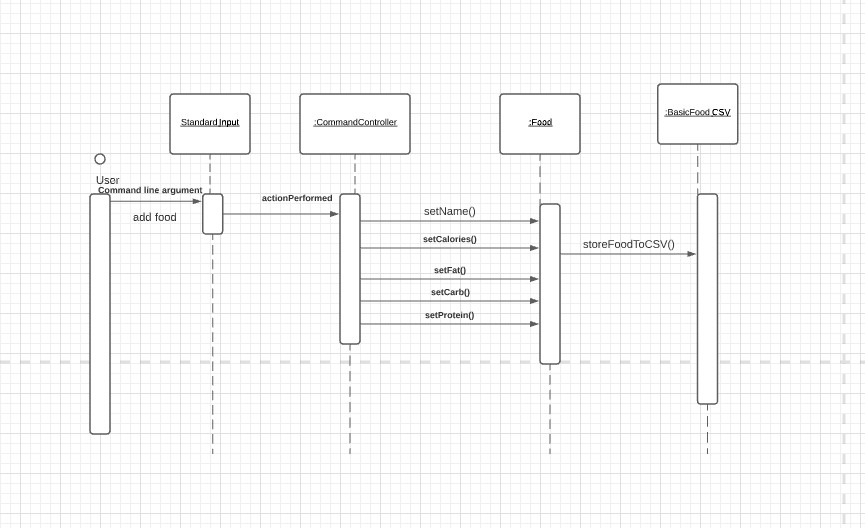
## **Sequence 1 - Get Basic Food and Get Recipe**



**Sequence 2 - Add food consumed & set daily weight & delete food**



**Sequence 3 - Compute Total Calories**



**Sequence 4 - Add Basic Food**

# 

# **Pattern Usage**

# **Pattern #1 Observer Pattern**

Description: The various display entities in the command line observe the log database and, when the database changes, update the information in the command line.

|  |  |
| --- | --- |
| **Observer Pattern** | |
| **Observer(s)** | CommandLineDisplay |
| **Observable(s)** | RecipeFood Database  Log Database |

## **Pattern #2 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)** | Component |
| **Composite(s)** | RecipeFood Database  Log Database  BasicFood Database |

## **Pattern #3 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, BasicFood Database  Recipe, RecipeFood Database  DailyLog, Log Database |
| **View(s)** | CommandLineDisplay |
| **Controller(s)** | CommandController |

## 

# 

# **Rationale**

**Observer Pattern**

We decided to use the Observer Pattern because it enables us to update the UI whenever there is a change in data such as calorie limit, weight, nutrition percentage, and more. The *CommandLineDisplay* (view) will be notified whenever there is a change in the *RecipeFood Database* and the *Log Database*.

**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 database, the controller will handle that interaction and update the model accordingly, which will then update the view accordingly.