Diet Manager / 1.3

Project Design Document

Group 2

Noa Mišić <nm8625@rit.edu>

Franko Fišter<ff1574@rit.edu>

Domagoj Dušić<dd8614@rit.edu>

Denny Lulak<dal4933@rit.edu>

Josip Pavlinić<jp3447@rit.edu>

# Project Summary

The Diet Manager project aims to provide a comprehensive software solution for users to monitor their diets effectively. The program allows users to manage a collection of basic foods and recipes, record daily food intake, track weight changes over time, and set desired calorie limits. By analyzing the collected data, the program provides insights into the user's dietary habits, helps in achieving weight management goals, and offers suggestions for maintaining a healthy lifestyle.

# Design Overview

The Diet Manager project is architected around the Model-View-Controller (MVC) design pattern, emphasizing a clean separation of concerns and facilitating scalability and maintainability. The system comprises three principal subsystems: Model, View, and Controller. The model part of the application is architected around the composite pattern.

Model:

The Model layer serves as the backbone of the application, housing the core business logic and data management functionalities.

It encompasses classes responsible for managing the food collection, daily logs, and user settings.

The composite pattern will be utilized in future versions.

Data persistence is achieved through CSV files.

Composite pattern is used for Basic food and Recipe configuration.

View:

The View layer focuses on user interaction and data presentation, employing graphical user interface (GUI) components for menus, input forms, and visual representations of dietary information.

JavaFX technology is used to develop the GUI, offering an intuitive UX.

Visualization tools are employed to render dietary insights and trends comprehensively, aiding users in understanding their eating habits and progress toward health goals.

Factory pattern is applied for the View part of the application.

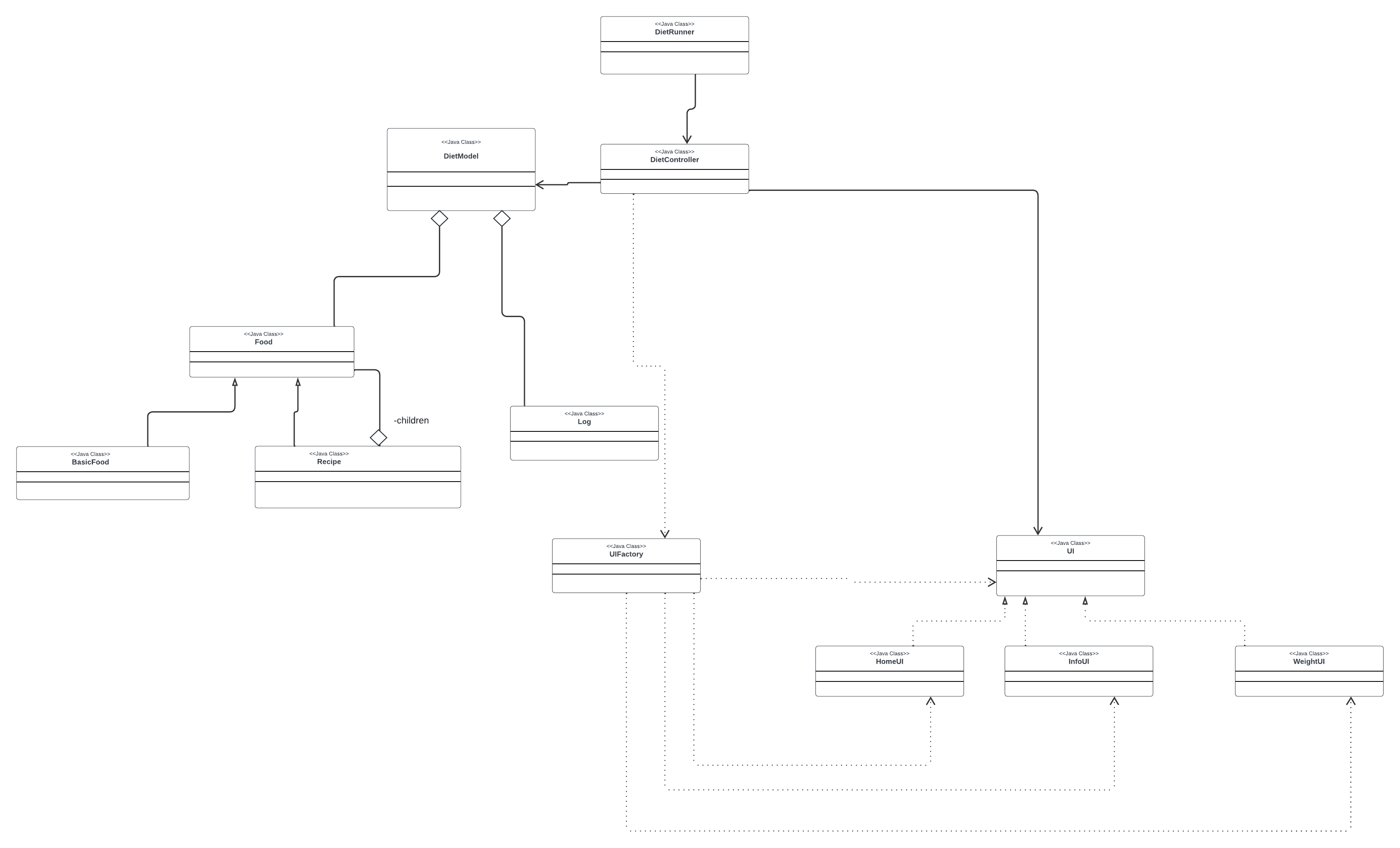
Controller:

The Controller layer acts as the mediator between the Model and View, acting as a communicator between the two.

It encompasses classes responsible for processing user input, triggering appropriate actions in the Model, and updating the View correspondingly.

Real-time synchronization ensures that modifications to data are promptly reflected in the user interface.

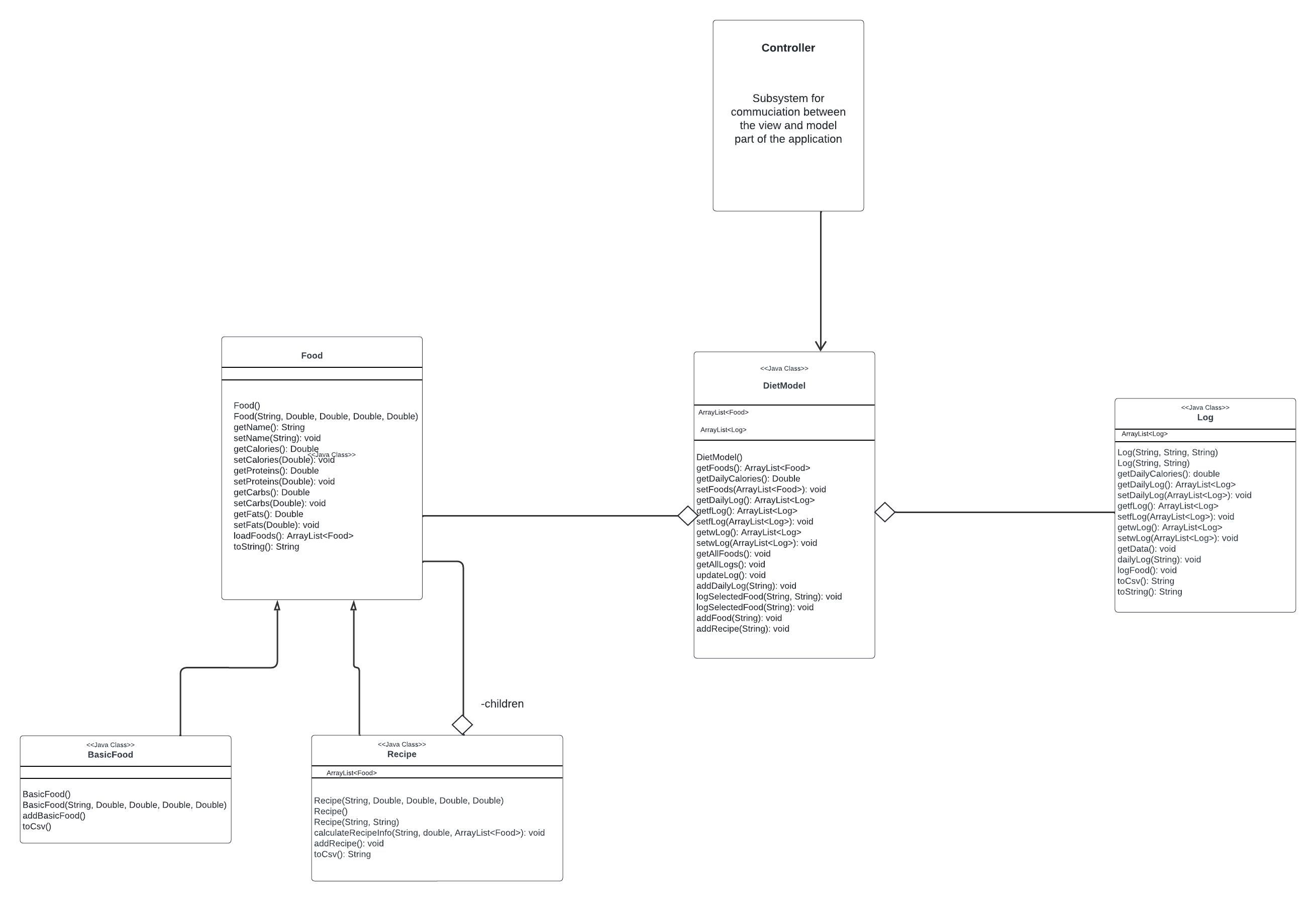
# Overall System Structure



For our project architecture, we implemented a factory pattern for UI creation, mvc pattern, and a composite pattern for Basic foods and recipes. Initially, we had factory and mvc, which serves our purpose great because we have multiple ui’s that the user can switch (factory). We added the composite pattern which allows us to have a collection of Foods, thus simplifying future expansions and maintenance of the model subsystem.

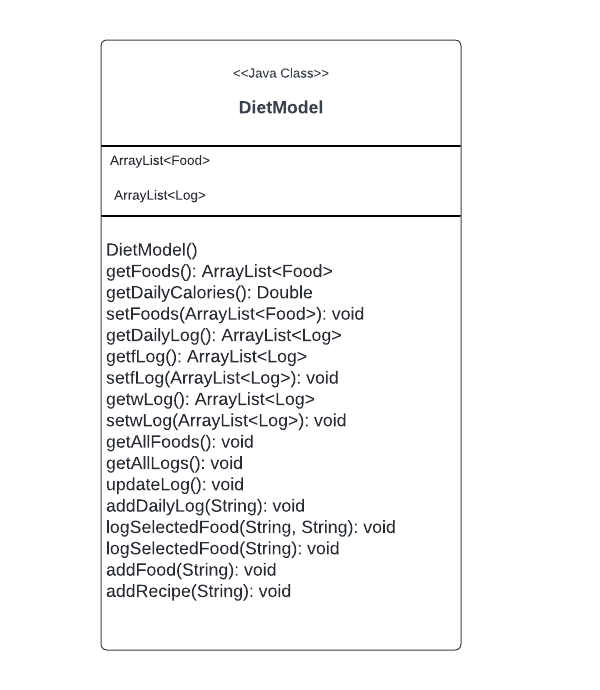
# Subsystems

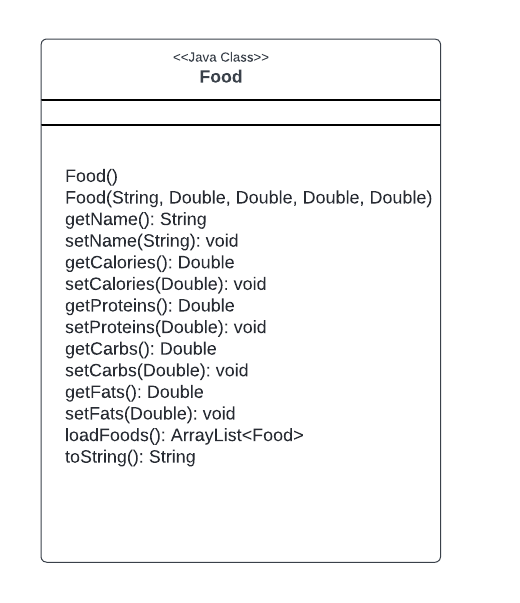
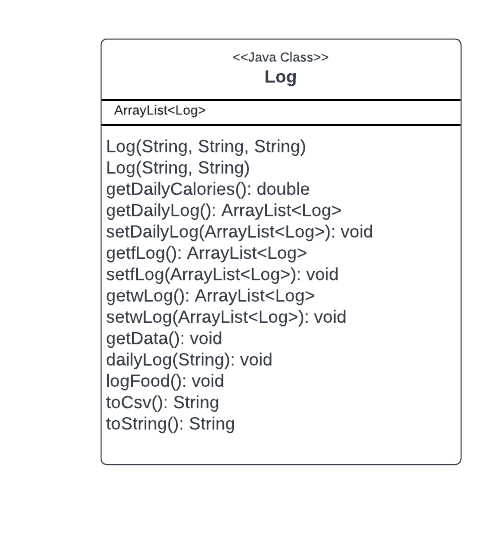
## **Subsystem DietModel**

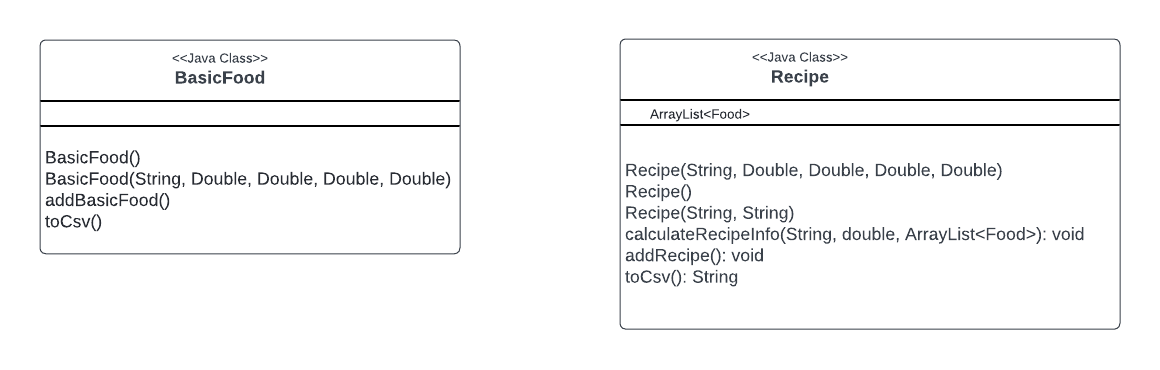


DietModel subsystem serves the purpose of getting the data from the food.csv and log.csv, as well as setting custom recipes and foods into the CSV File. It will send data from those CSV files to the DietController. DietModel subsystem is designed with the composite pattern. DietModel class is a class that communicates with the controller and holds the collections of foods and logs. Log class is responsible to talk to the log.csv file, to calculate calories for particular date and to provide a list of logs to the DietModel class. Food class servers as abstract class for composite pattern that is reading the foods.csv file and sending the collection of food items to the DietModel class. BasicModel is a leaf class, and Recipe has collection of all Foods and creates a Recipe in a Food like format, thus aligning with the composite pattern.

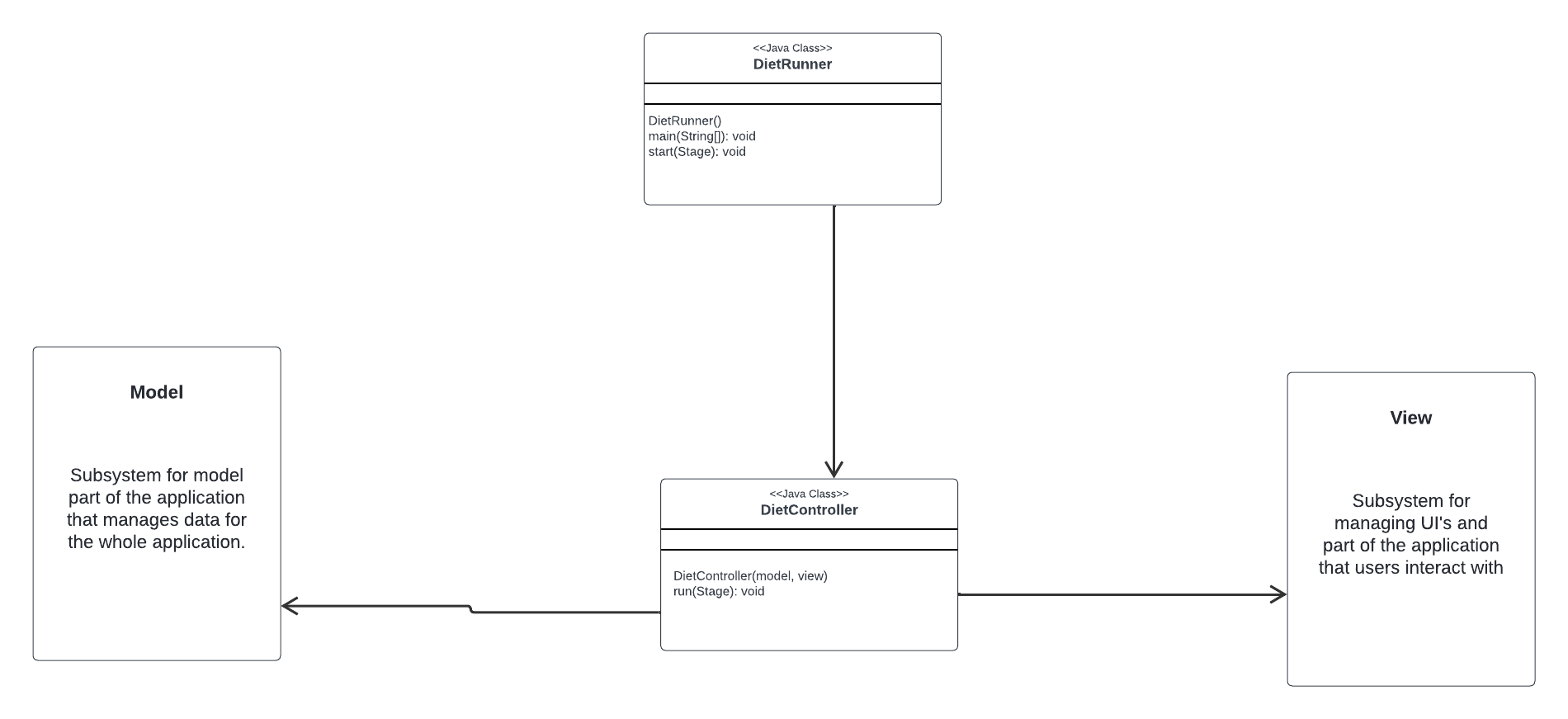
Model classes (for better method visibility)





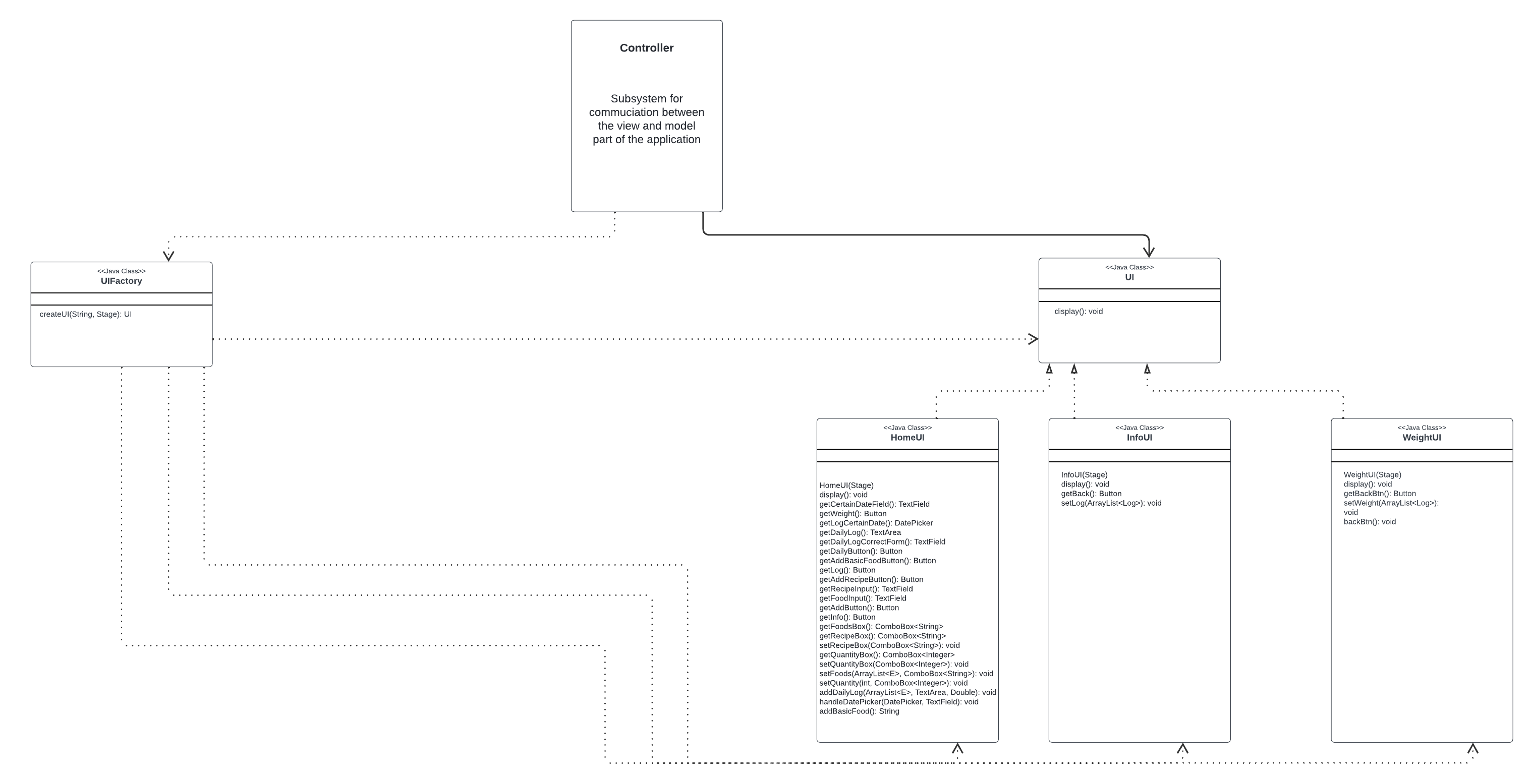


## **Subsystem DietController**



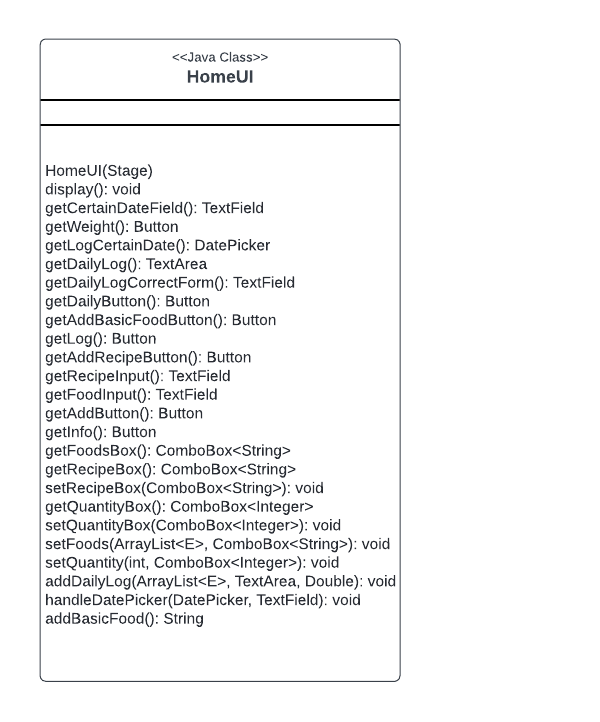
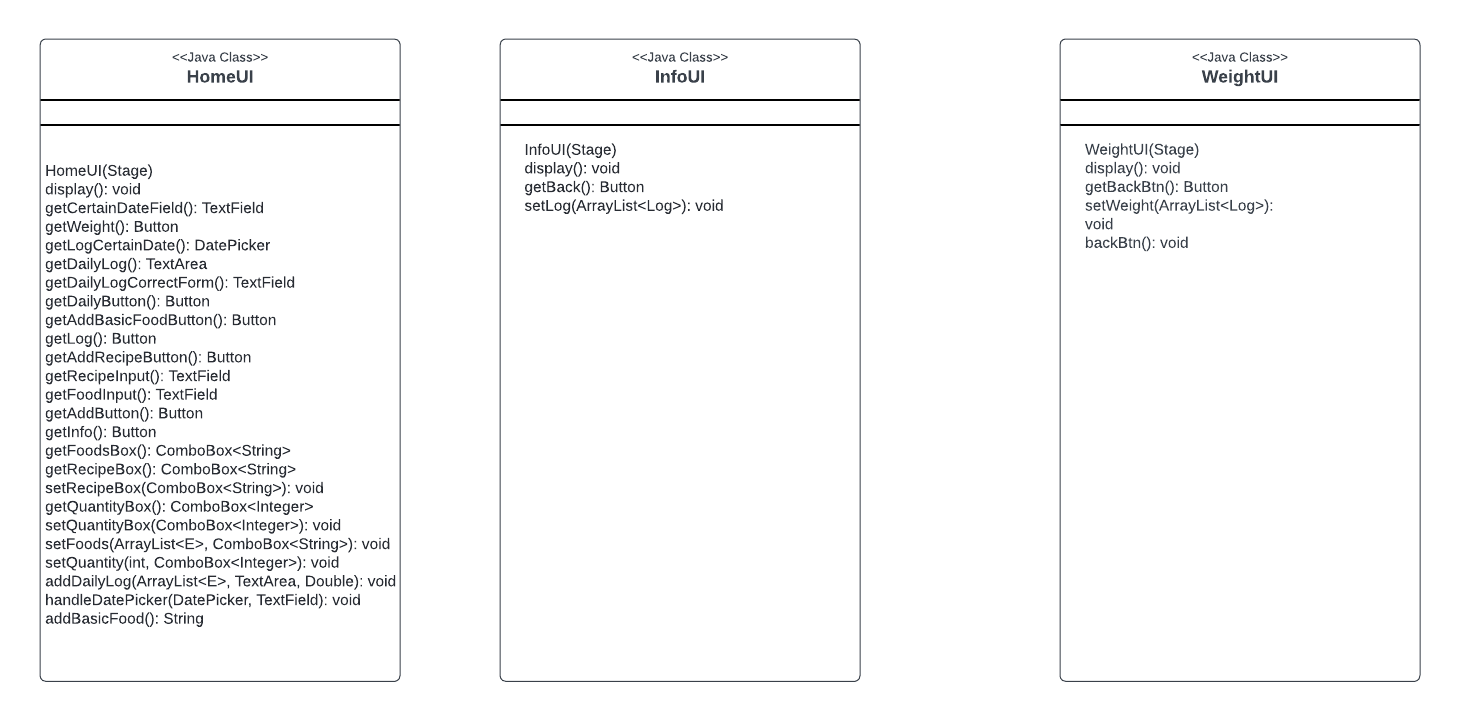
DietController subsystem serves the purpose of communicating between the view and the model. When the client does something on the UI, it notifies the controller, and the controller notifies the model subsystems to do the task. DietRunner is responsible for the main method and running the project, and the DietController class is the controller class that serves for communication between the view and the model.

## **Subsystem DietView**



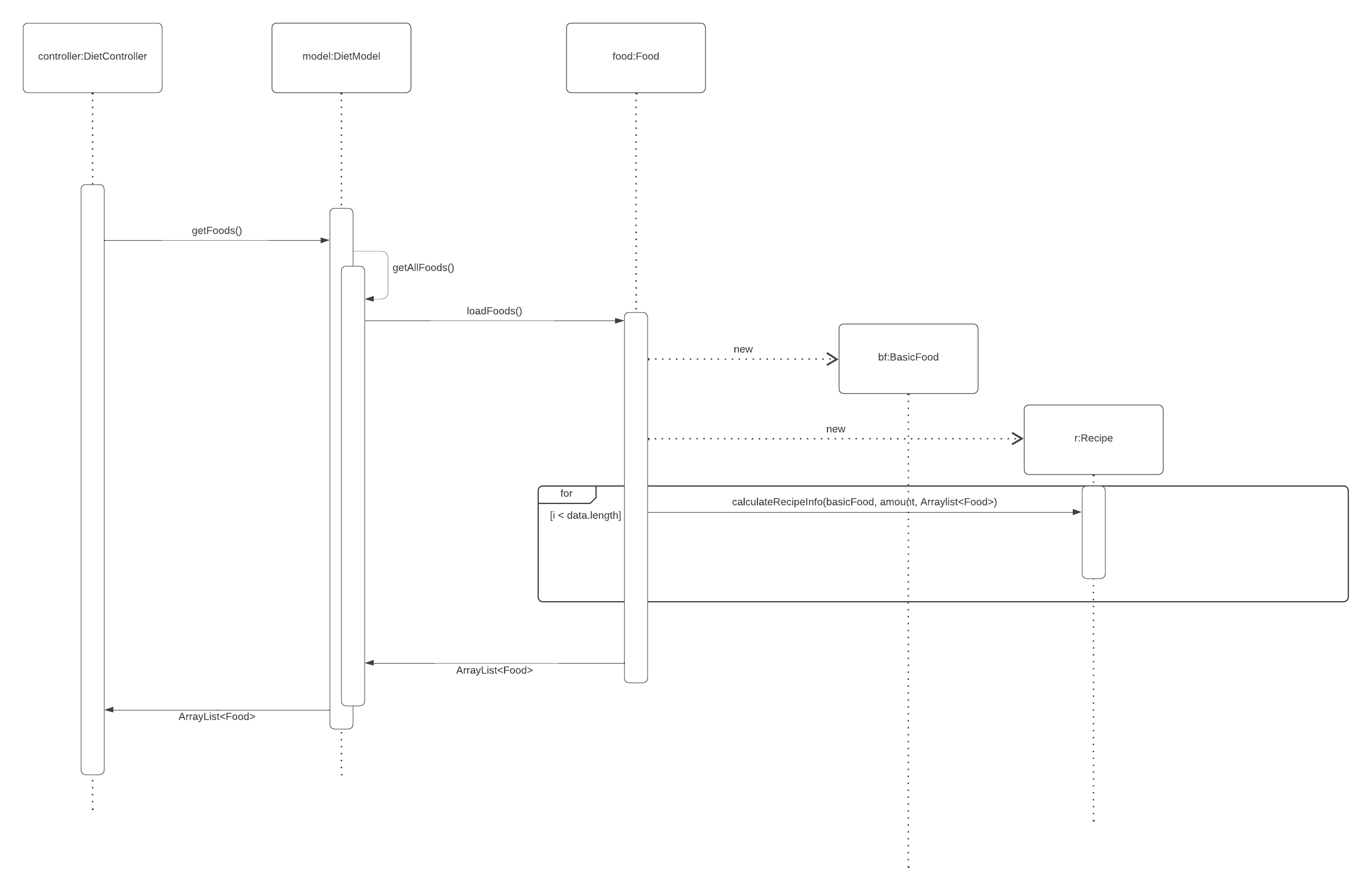
DietView Subsystem serves the purpose of creating UI’s or the view part of the project that the clients will be interacting with. The factory pattern is used in UI creation to isolate the logic of object instantiation, making the application easily expandable with other UI and view elements. The principles applied are Program to an interface, not an implementation and separation of concerns. DietView Subsystem will provide UIs for clients that they will interact with, and on interaction will notify the controller to update the model and get new data from the model to incorporate into the view. UIFactory is a class with a switch case that instantiates the UI, and it returns a UI interface. UI interface is an interface to all UI elements.

View classes (for better method names visibility)



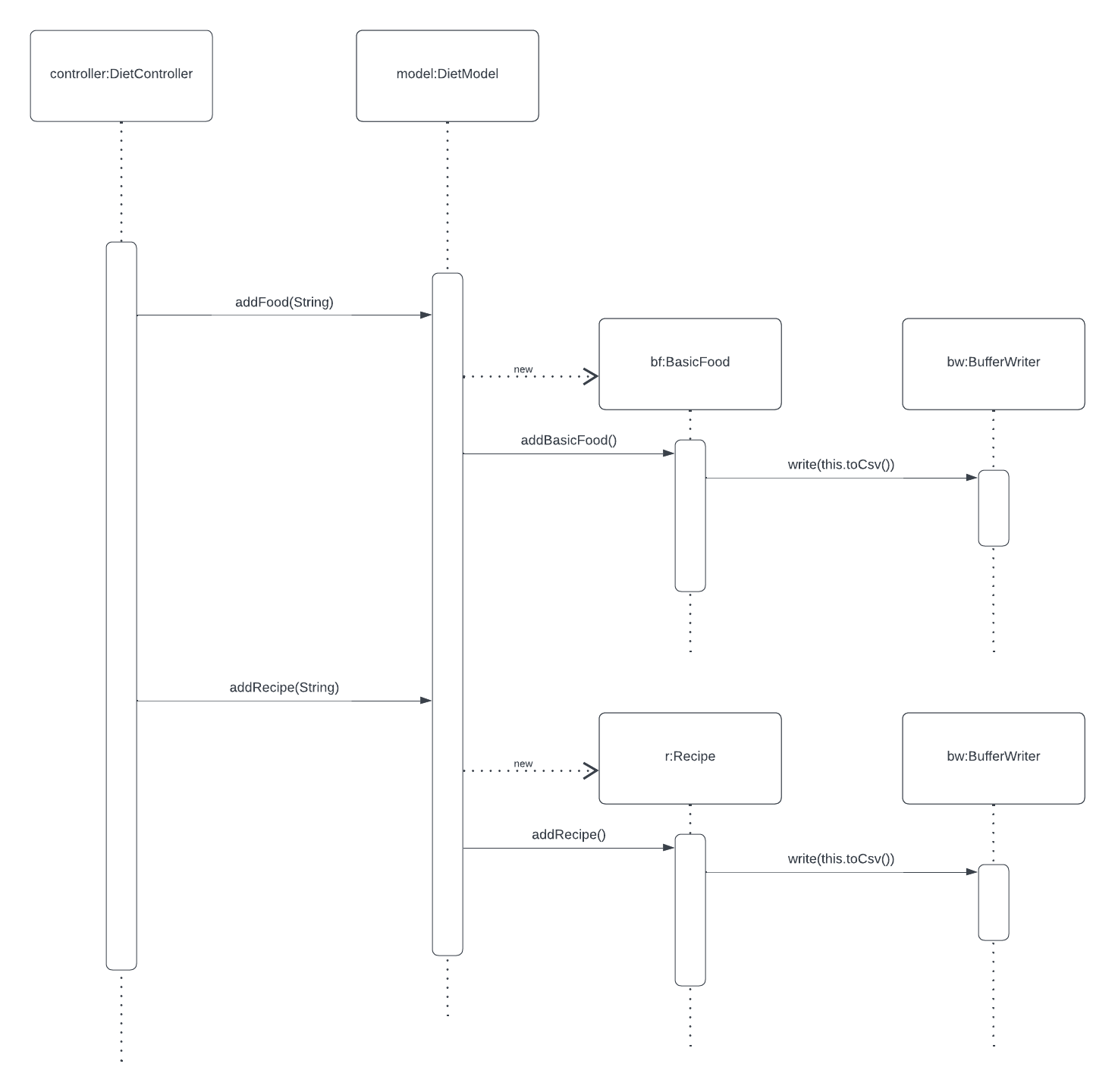
# Sequence Diagrams

## **Sequence Diagram 1: Loading data for 1 basic food (Pizza Slice) and 1 recipe (PB+J Sandwich)**



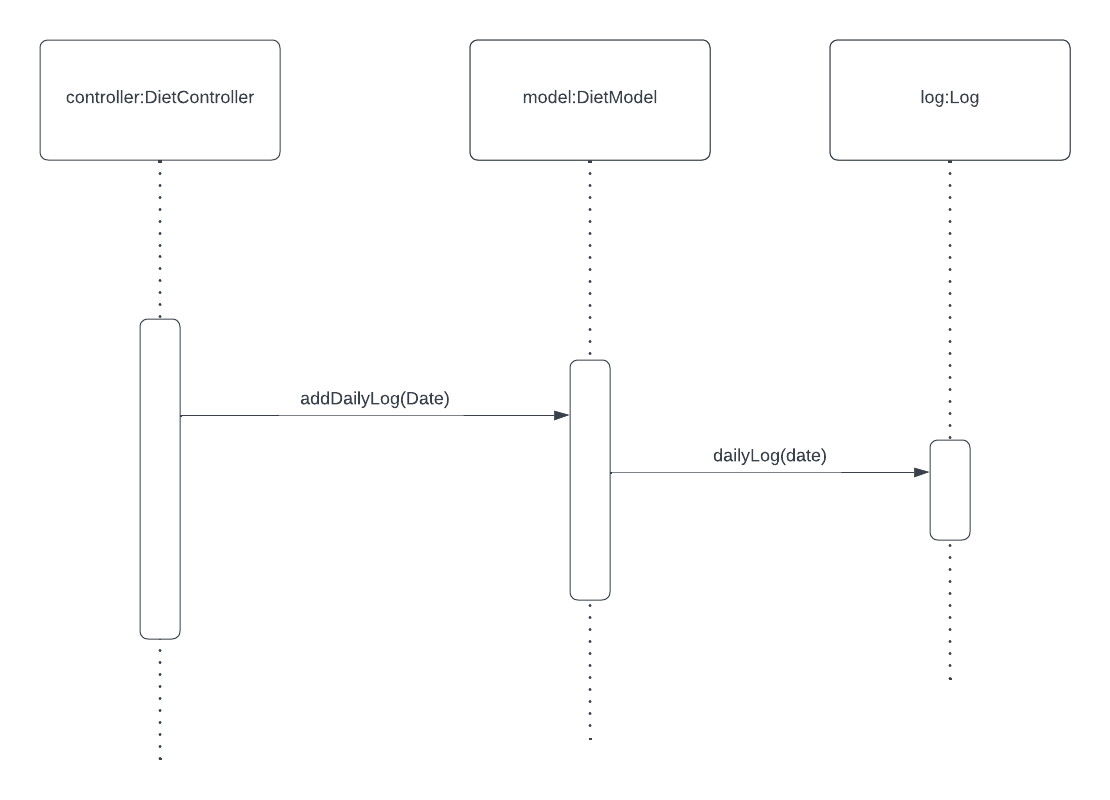
On start of the program, DietController calls getFoods method of DietModel. That method is calling getAllFoods method that is calling the loadFoods method. LoadFoods method is loading foods from the food.csv (not visible on the diagram because of lack of space). In the while nextLine that method is creating either BasicFood or Recipe and adding those foods to the ArrayList<Food> of foods. At the end, that method returns that ArrayList<Food> and the DietController gets a collection of all foods (1 basic food and 1 recipe in this case), later showing it in the view.

## **Sequence Diagram 2: Add 1 serving of the Pizza Slice & 2 servings of the PB+J Sandwich to the log entry for the current date**



When a button is clicked on the view, DietController is calling addFood method with the String as a parameter. That method is creating a new BasicFood object and calling addBasicFood method that is writing that food in the log.csv file using the toCsv method of BasicFood. For Recipe the process is similar, but it creates a Recipe object and also writes that in the foods.csv file using the toCsv method.

## **Sequence Diagram 3: Compute the total number of calories for the current date.**



For computing total number of calories, we call the addDailyLog with the Date as a parameter, that is calling dailyLog method of the Log class. That method goes through the collection of logs, and for every log that is logged on the date provided in the parameter, it adds it to the collection of logs for that date and adds ­(+=) the calories of that log to the totalCalories integer.