Diet Manager / 1.1

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.

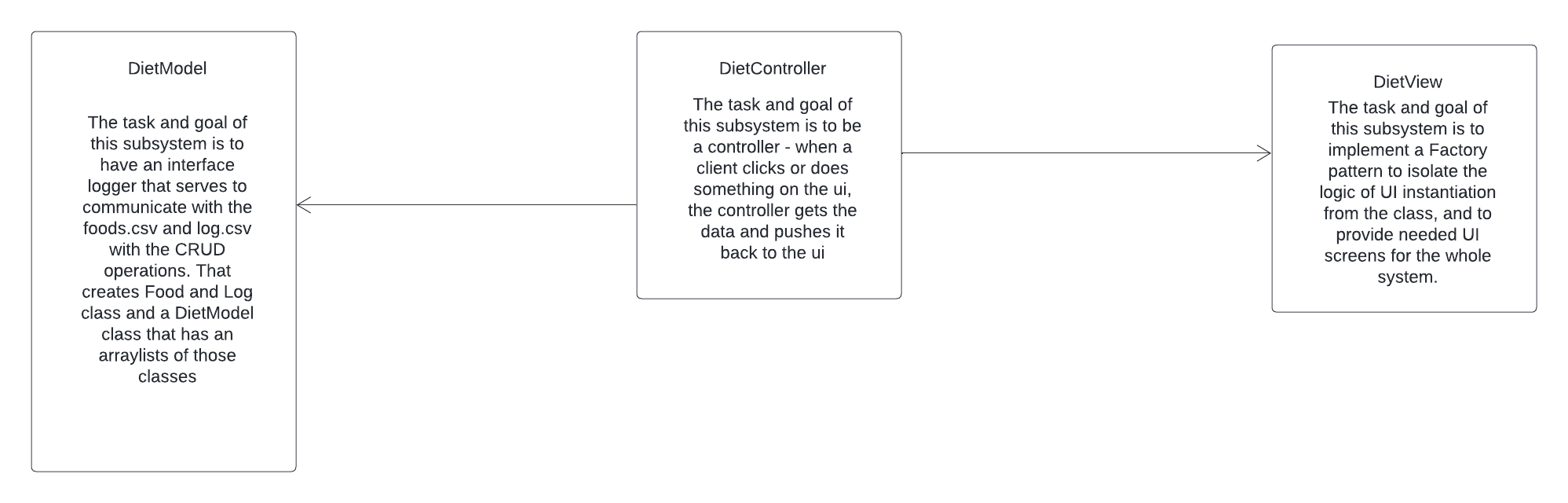
Controller:

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.

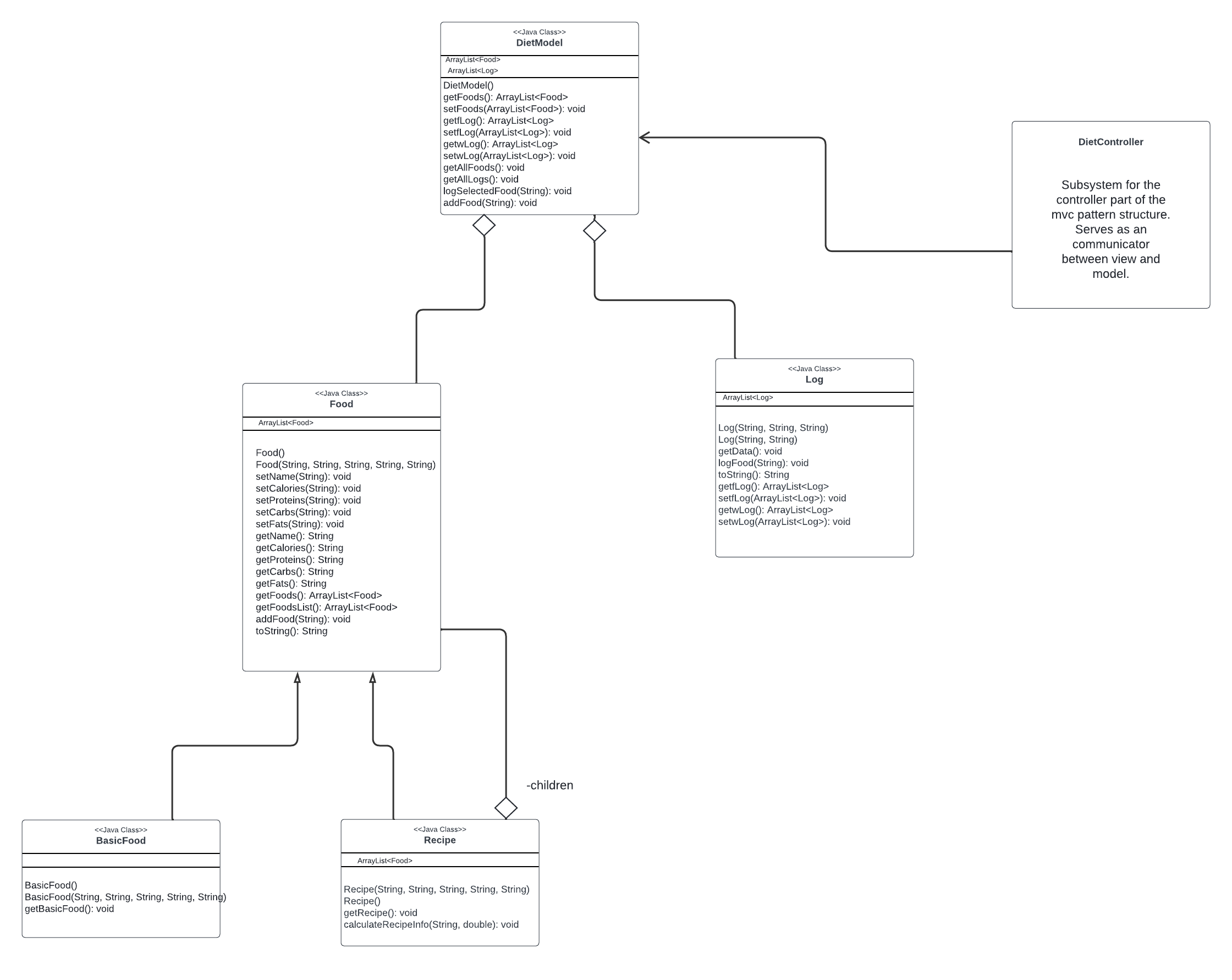
Real-time synchronization ensures that modifications to data are promptly reflected in the user interface.

# Overall System Structure



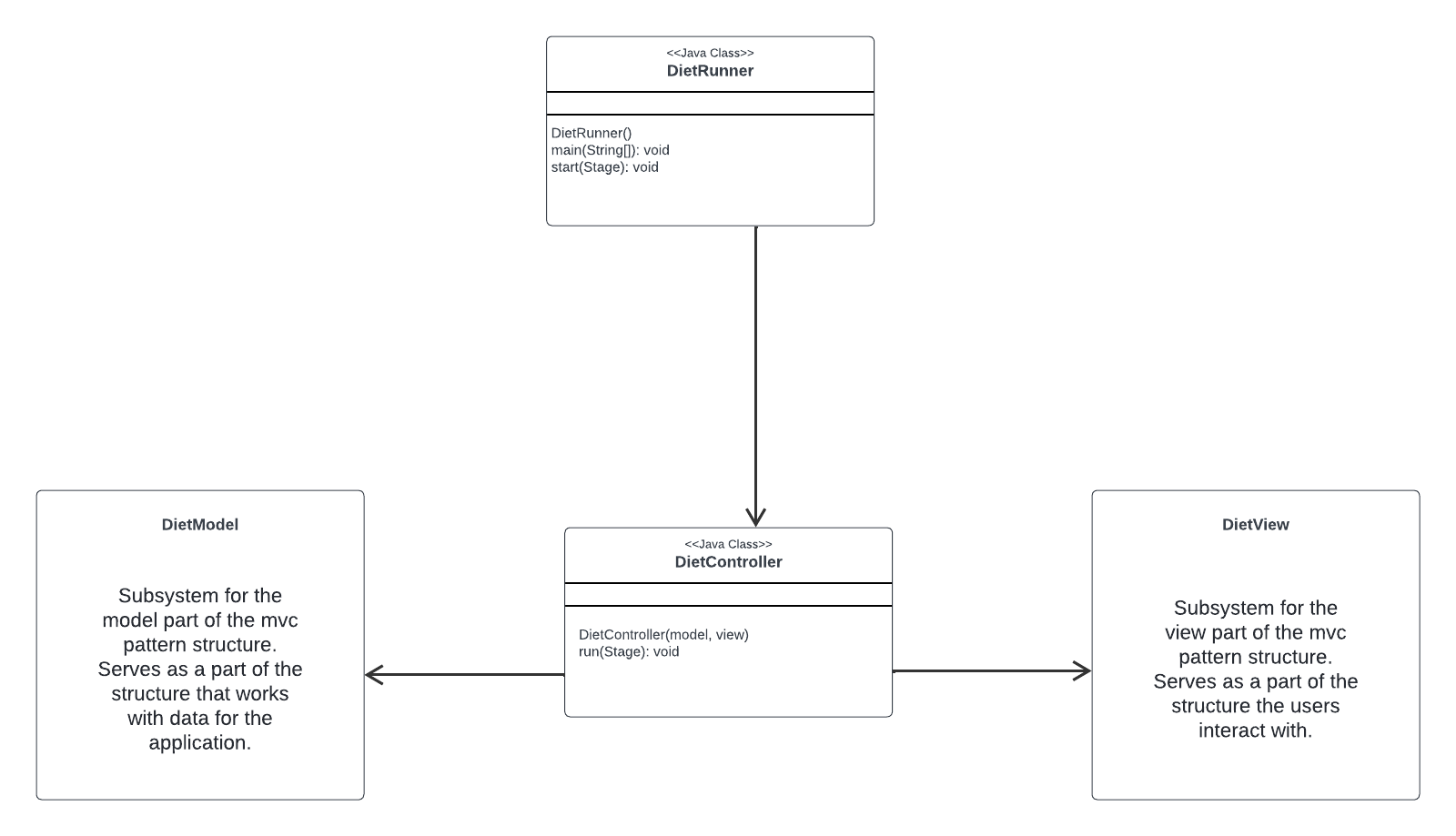
# Subsystems

## **Subsystem DietModel**



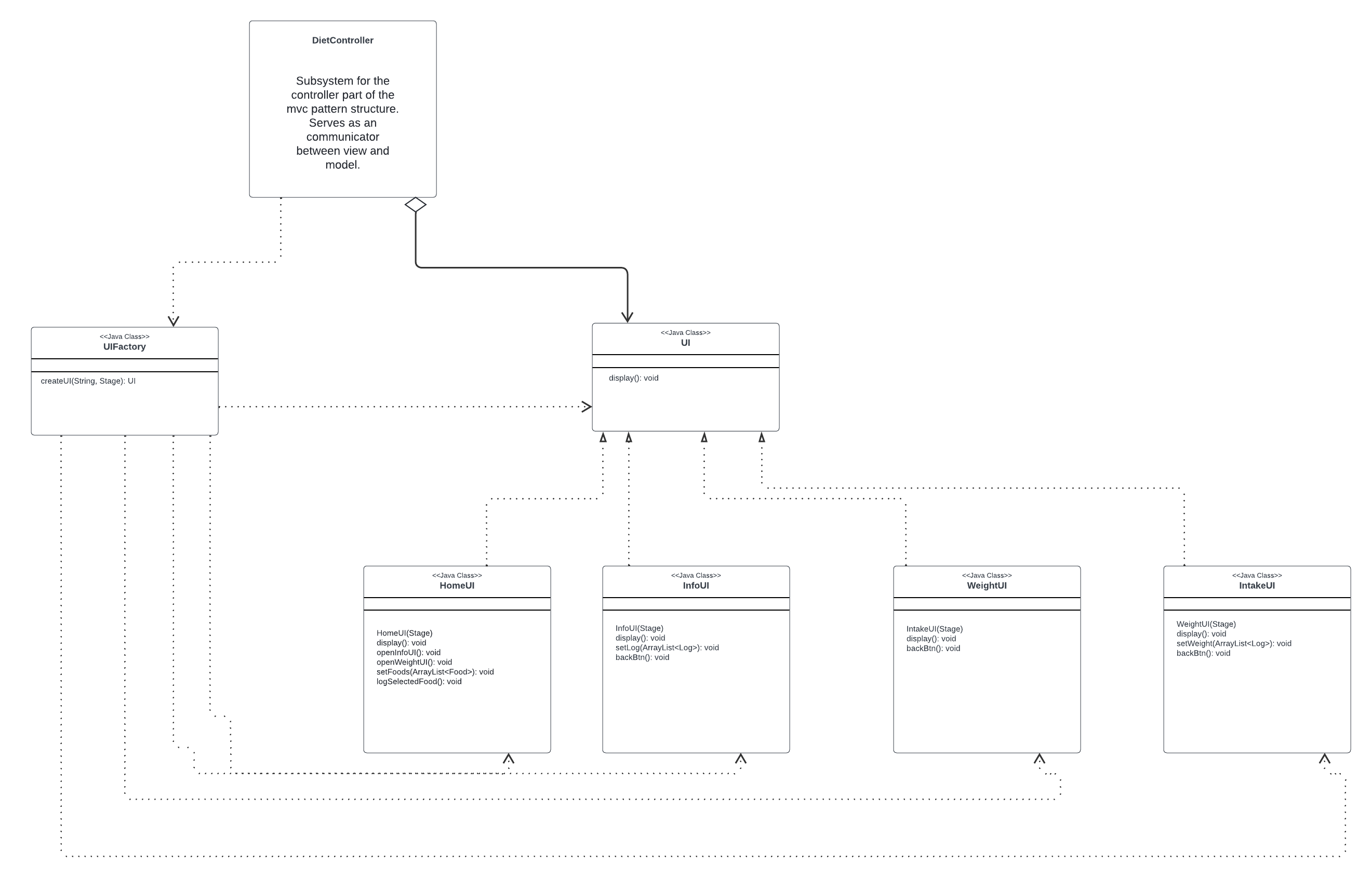
DietModel subsystem serves the purpose of getting the data from the food.csv and log.csv, as well as setting custom recipes and foods into the CSV File. It will send data from those CSV files to the DietController. DietModel class will have a collection of all Logs and Foods. The DietModel subsystem complies with a composite pattern. It has BasicFoods and Recipes which are made from BasicFoods. Recipe class has a collection of Foods and every time a recipe is loaded it uses the calculateRecipeInfo method to calculate the recipe information and store it into the collection of foods.

## **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.

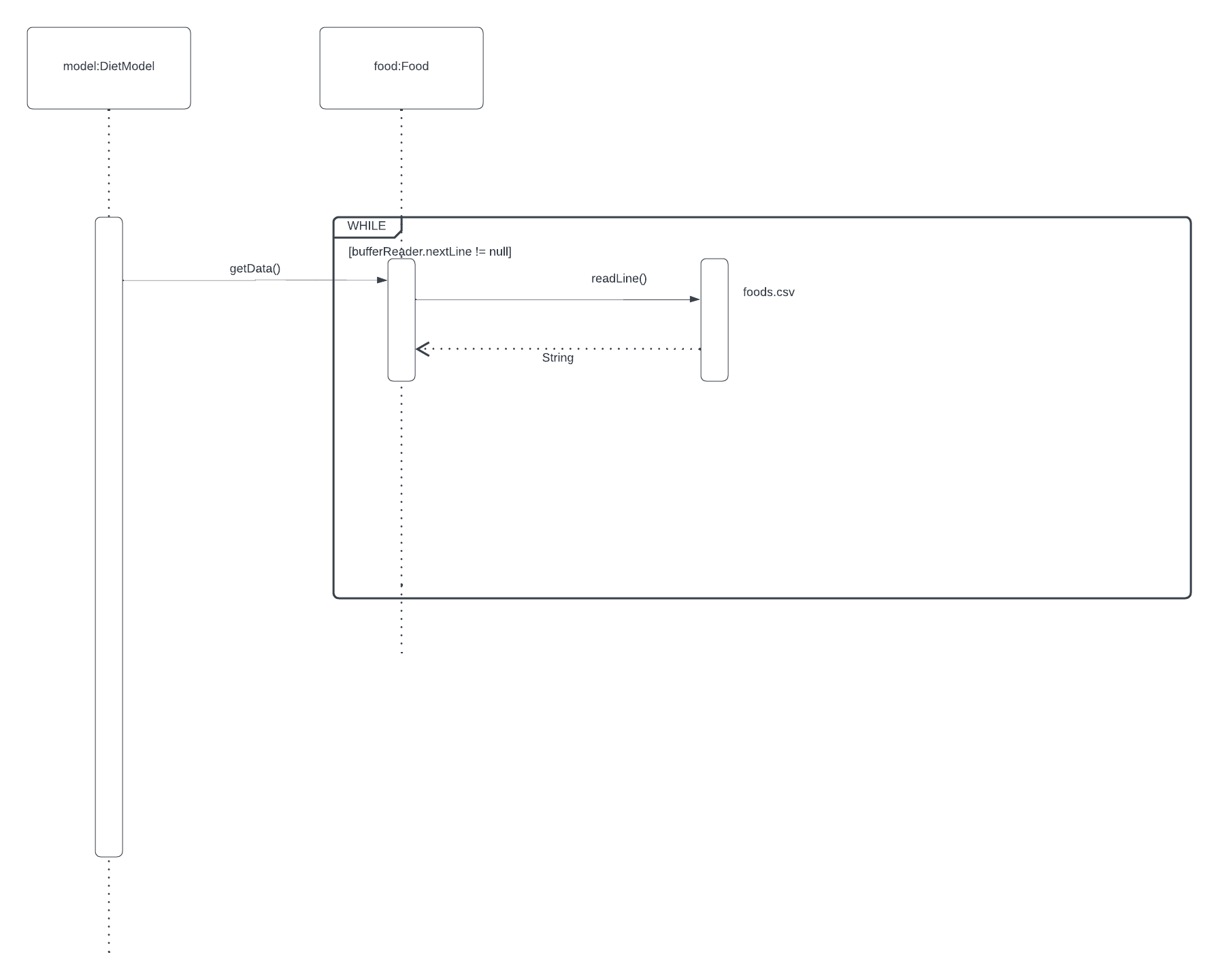
## **Subsystem DietView**



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. DietView 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.

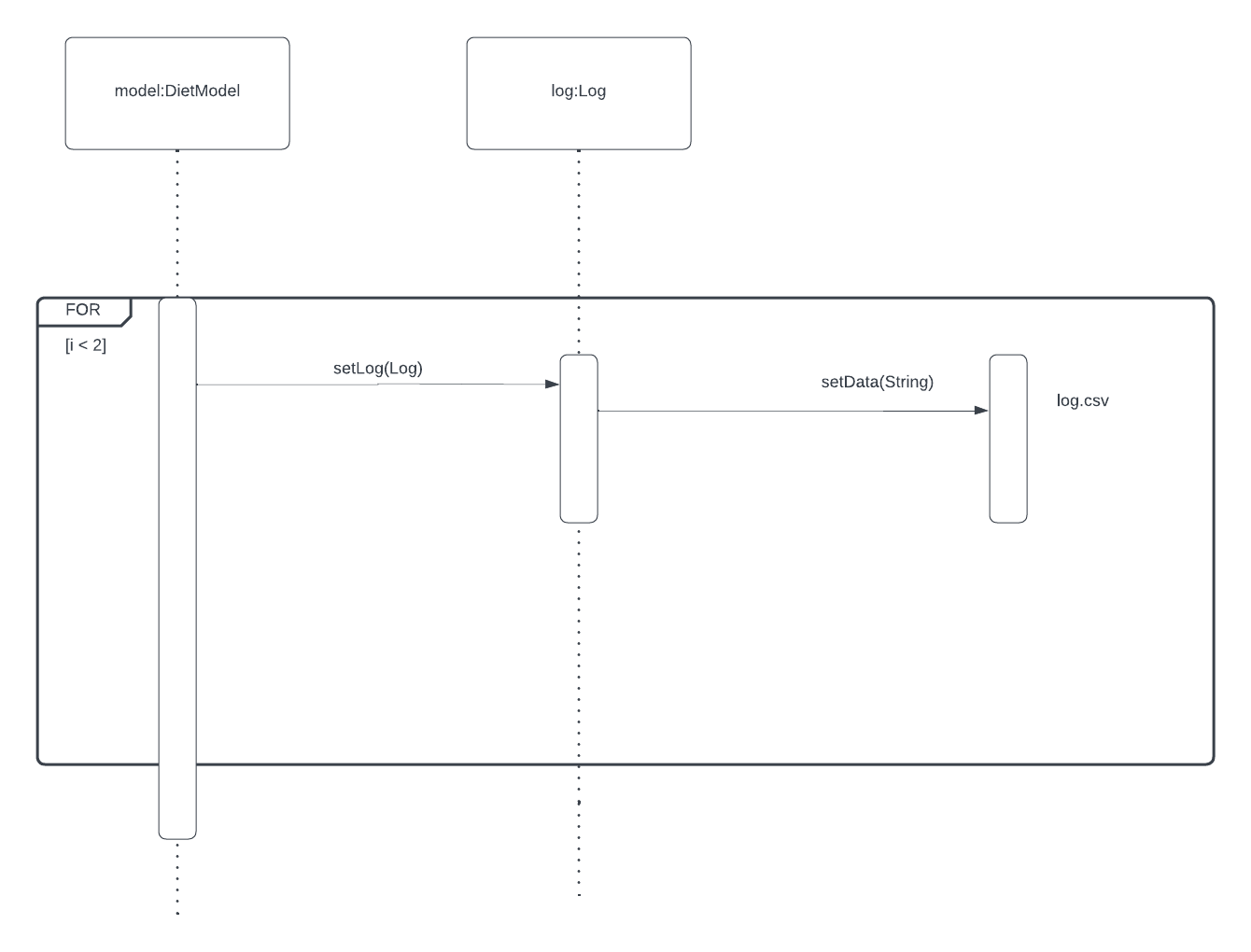
# Sequence Diagrams

## **Sequence Diagram 1: reading the foods.csv database**



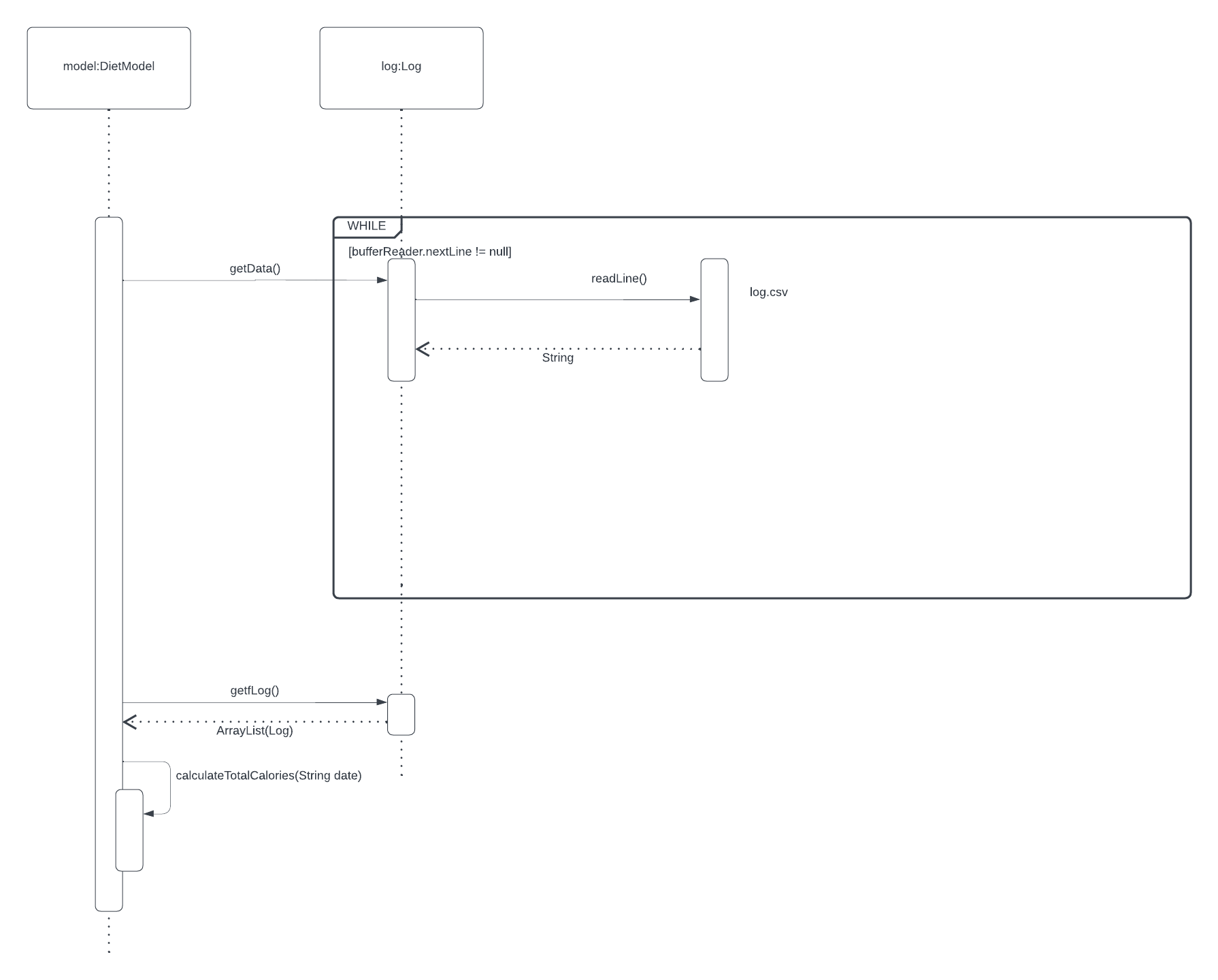
This sequence diagram shows reading an entire food database (currently has 3 rows). The model is calling the getData method of the Food class. The getData method has a while loop that is reading every line in a database and creating a Food object, and populating the arraylist of Foods. If the DietModel can access that arraylist with getter method.

## **Sequence Diagram 2: adding two servings of food into a log from model**



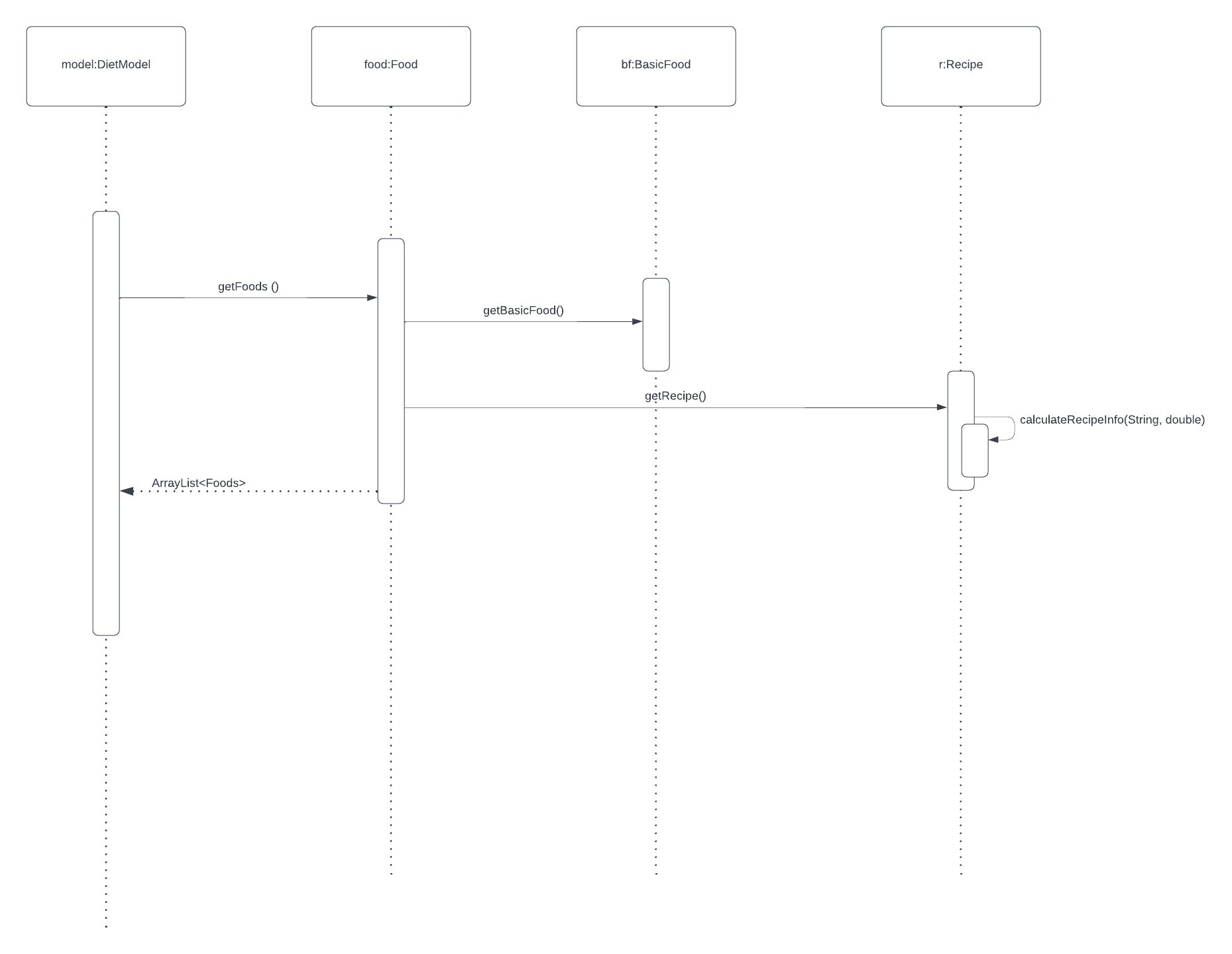
This sequence diagram is showing addition of two servings of food into a log. The DietModel calls setLog method which is using the BufferWriter to write into the database log.csv. That method is then settingData as a string into the database, string will be formatted depending on the intake log or weight log. This sequence diagram only shows the model part. Model will get from the controller info on what food and what amount of food should be inserted.

## **Sequence Diagram 3: calculating calorie intake for a specific day**



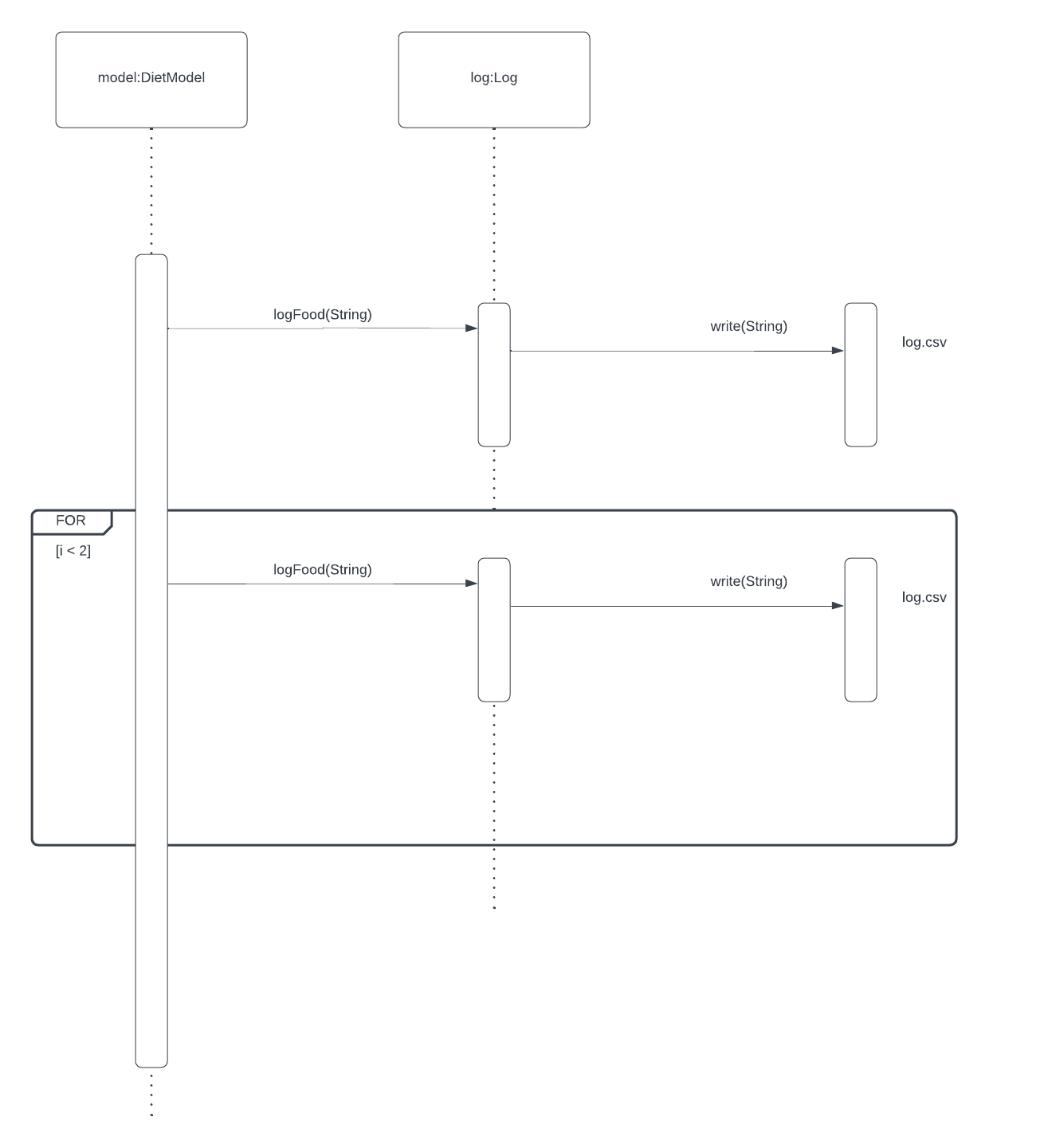
This sequence diagram is showing calculation of intake for a specific day. First we get the data from the log.csv, and we call the getfLog getter to get arraylist of all logs. We than call the calculateTotalCalories method which takes in a date parameter, and based on the date and calories from the log it adds them up.

## **Sequence Diagram 4: loading data for 1 basic food and 1 recipe**



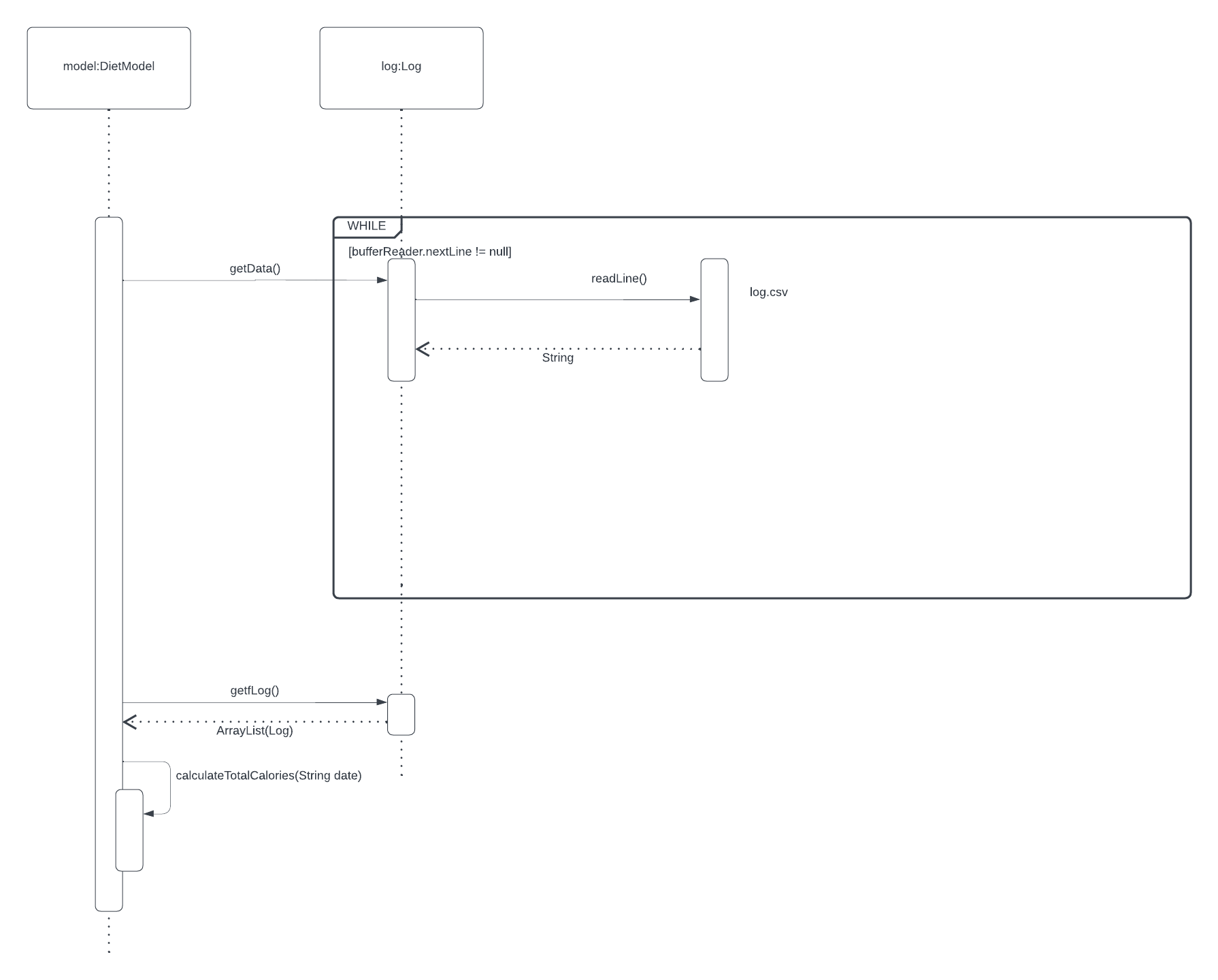
This sequence diagram is showing loading of data of basic foods and recipes into a DietModel. DietModel class calls the getFoods method from the Food abstract class which returns an arraylist of Foods (basic foods and recipes combined). That method before returning an arraylist first loads all of the basic foods with the getBasicFood() method into an arraylist, and then the getRecipe() method of the Recipe class gets all the recipes and calls the CalculateRecipeInfo(String, double) method which gets all of the names and amounts of basic foods in the recipe and calculates recipe calories, proteins, carbs and fats. The getRecipe then stores the recipe into an ArrayList in the format like the basic food has.

## **Sequence Diagram 5: add 1 serving of pizza and 2 servings of PB+J to the log entry**



This sequence diagram shows adding 1 basic food and 2 recipes to a log. Since both basic foods and recipes are in the same format, they are stored in the log the same way, as a comma-separated String with the date at the end, and the name of the food with the information needed.

## **Sequence Diagram 6: compute the total number of calories for the current date**



This sequence diagram is the same as Sequence diagram 3. Because both basic foods and recipes are store in the log at the same format, the diagram for calculating the total number of calories for a specific date is not changed. It will only now have both recipes and basic foods in the log and calculate both of those in the daily intake.