Diet Manager / Version 2

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, exercise collection, daily logs, and user settings.

The composite pattern is utilized.

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.

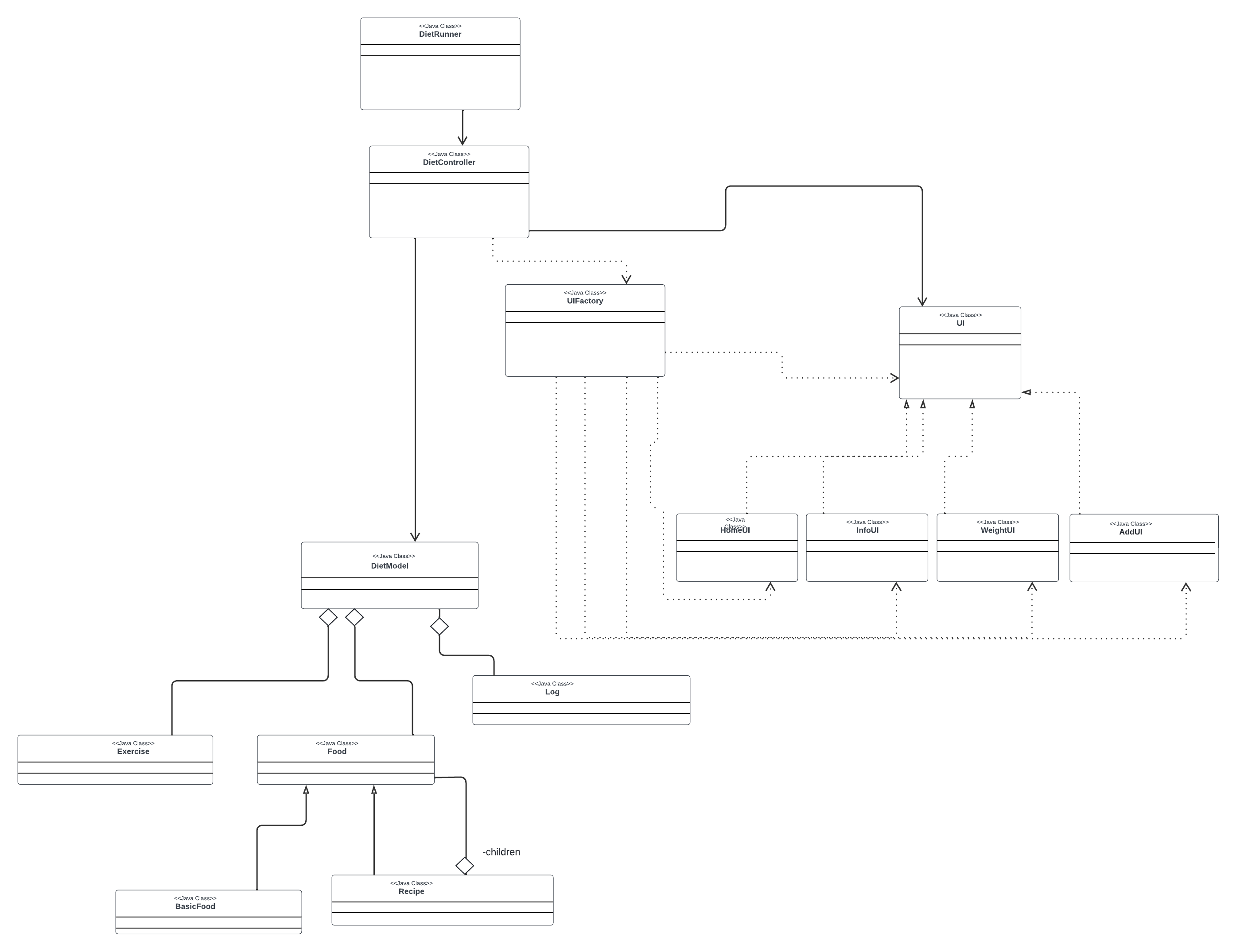
Controller:

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.

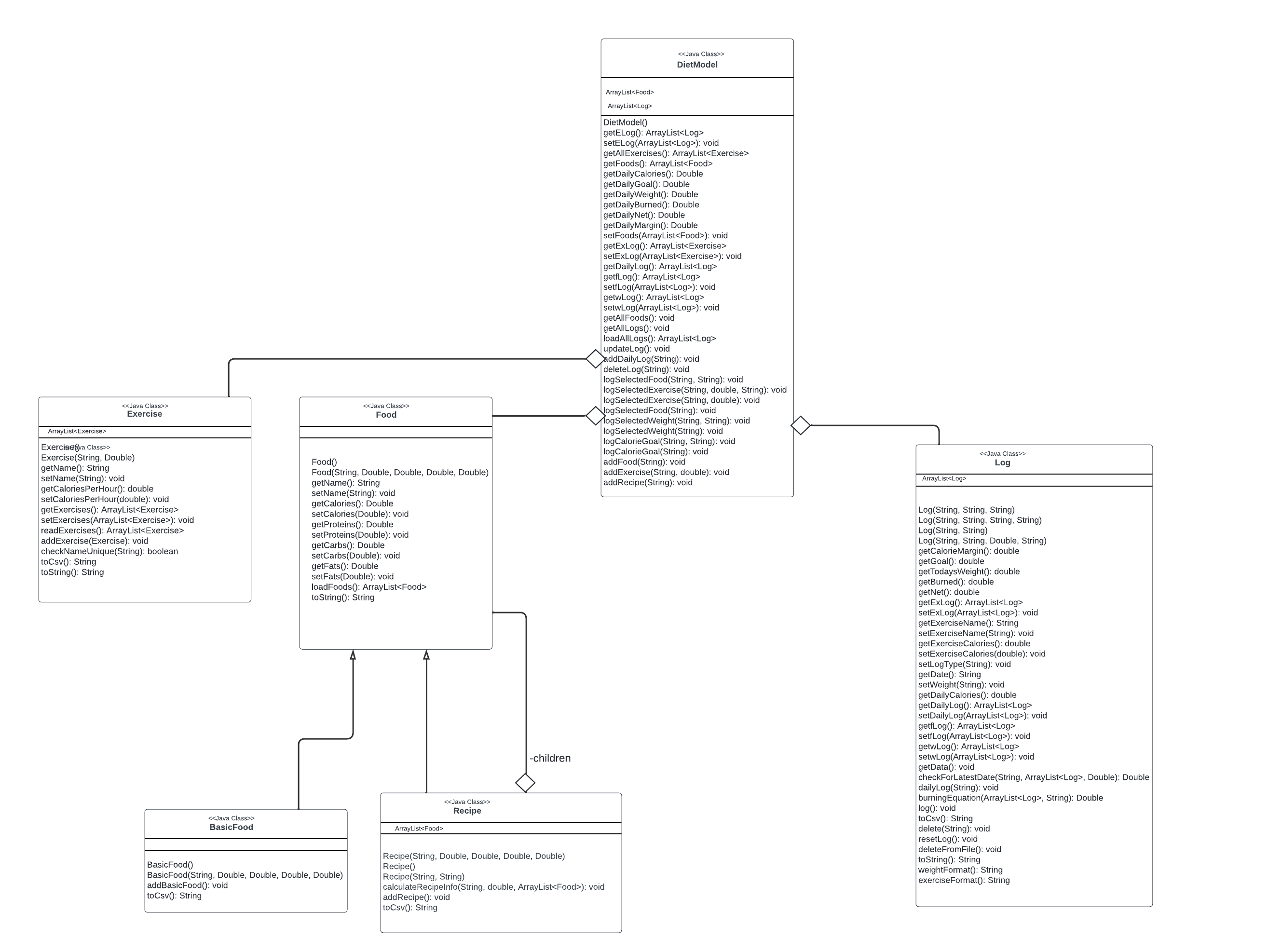
# Overall System Structure



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.

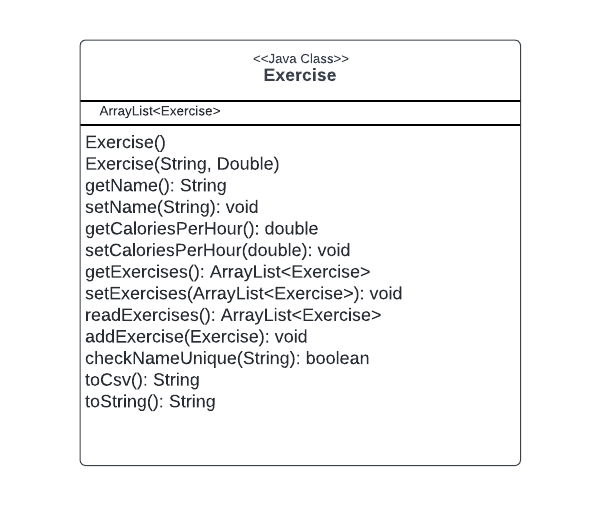
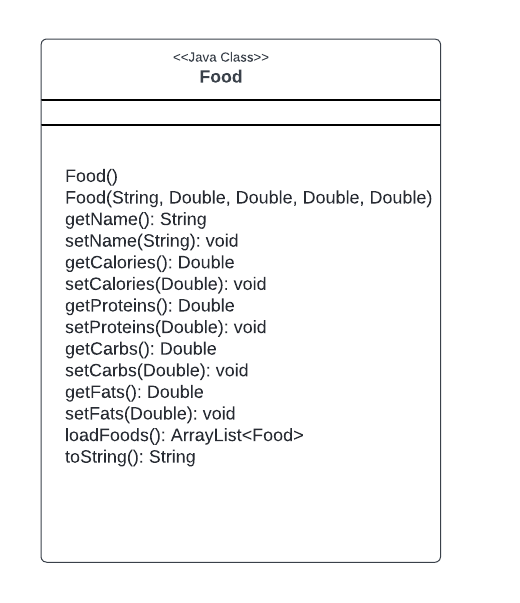
# Subsystems

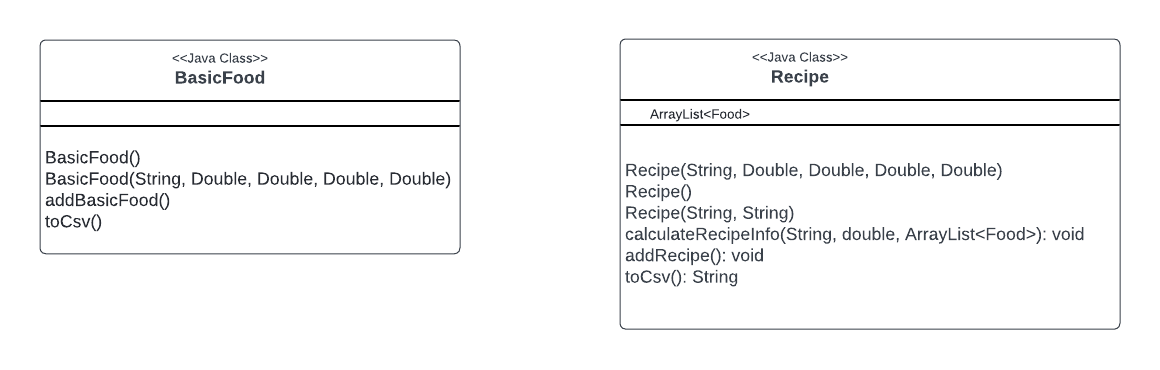
## **Subsystem DietModel**

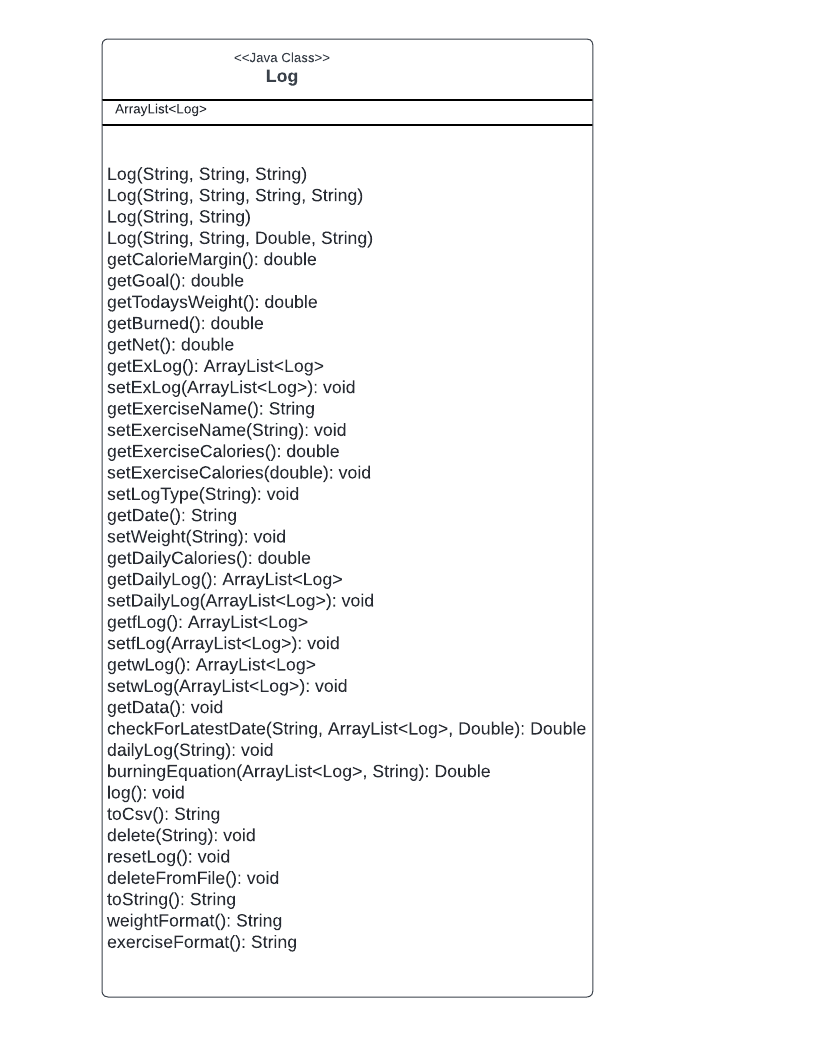


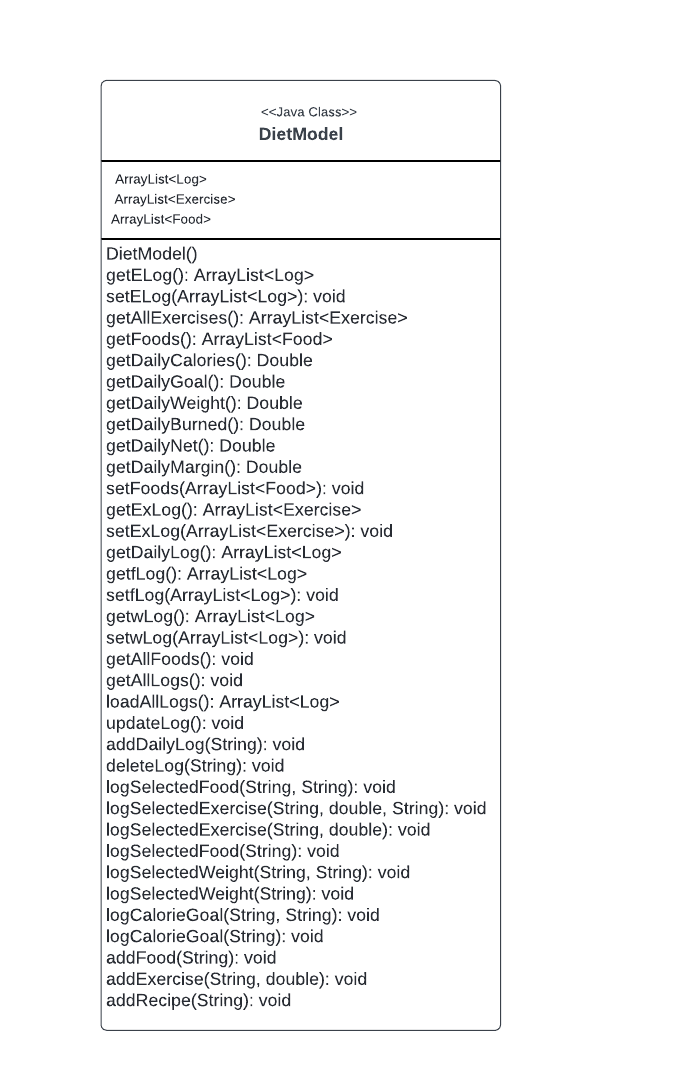
DietModel subsystem serves the purpose of getting the data from the food.csv, log.csv, and exercise.csv as well as setting custom recipes, foods, and exercises into the CSV File. It will send data from those CSV files to the DietController. DietModel subsystem is designed with the composite pattern. DietModel class is a class that communicates with the controller and holds the collections of foods, exercises, 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. Exercise class is a class responsible for talking to the exercise.csv file with CRUD operations.

Model classes individually (for better method visibility)

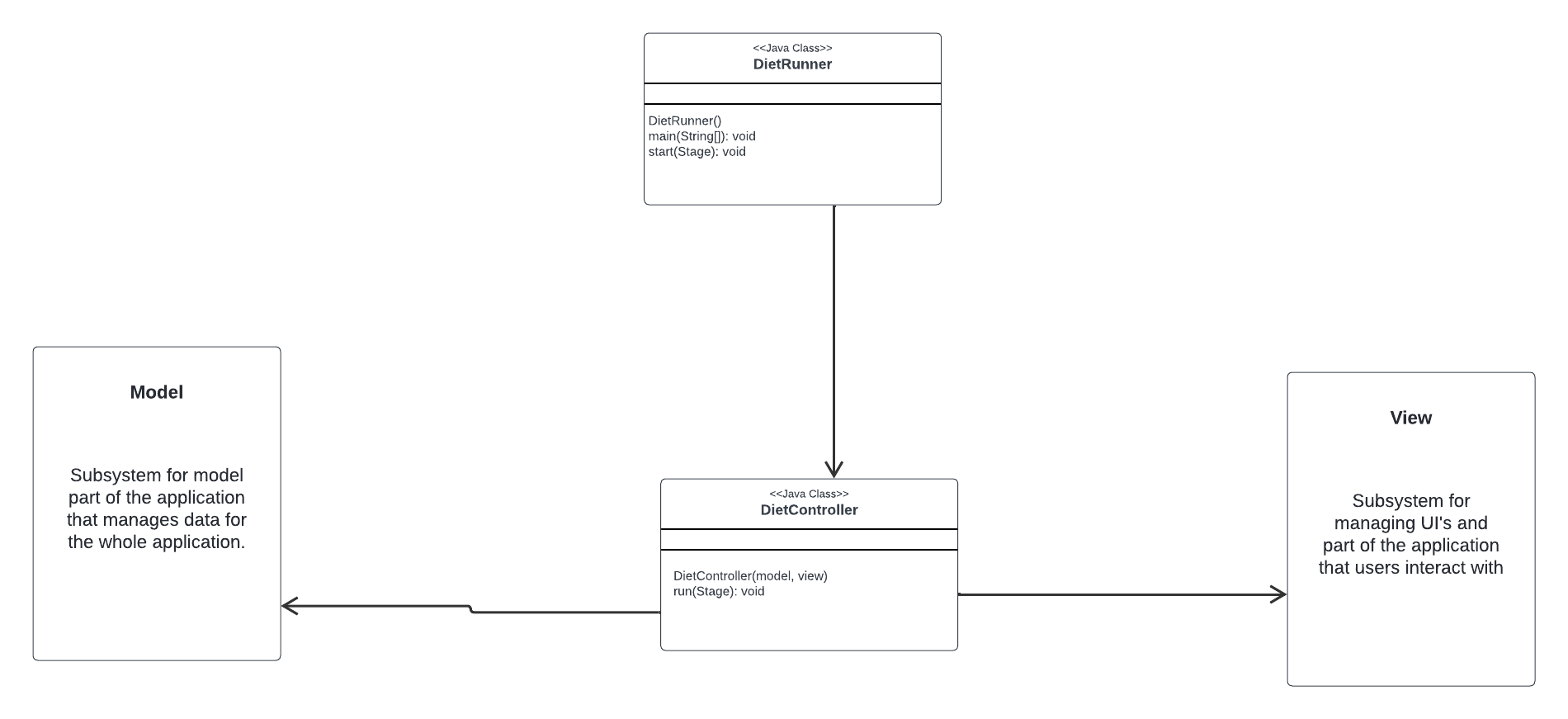






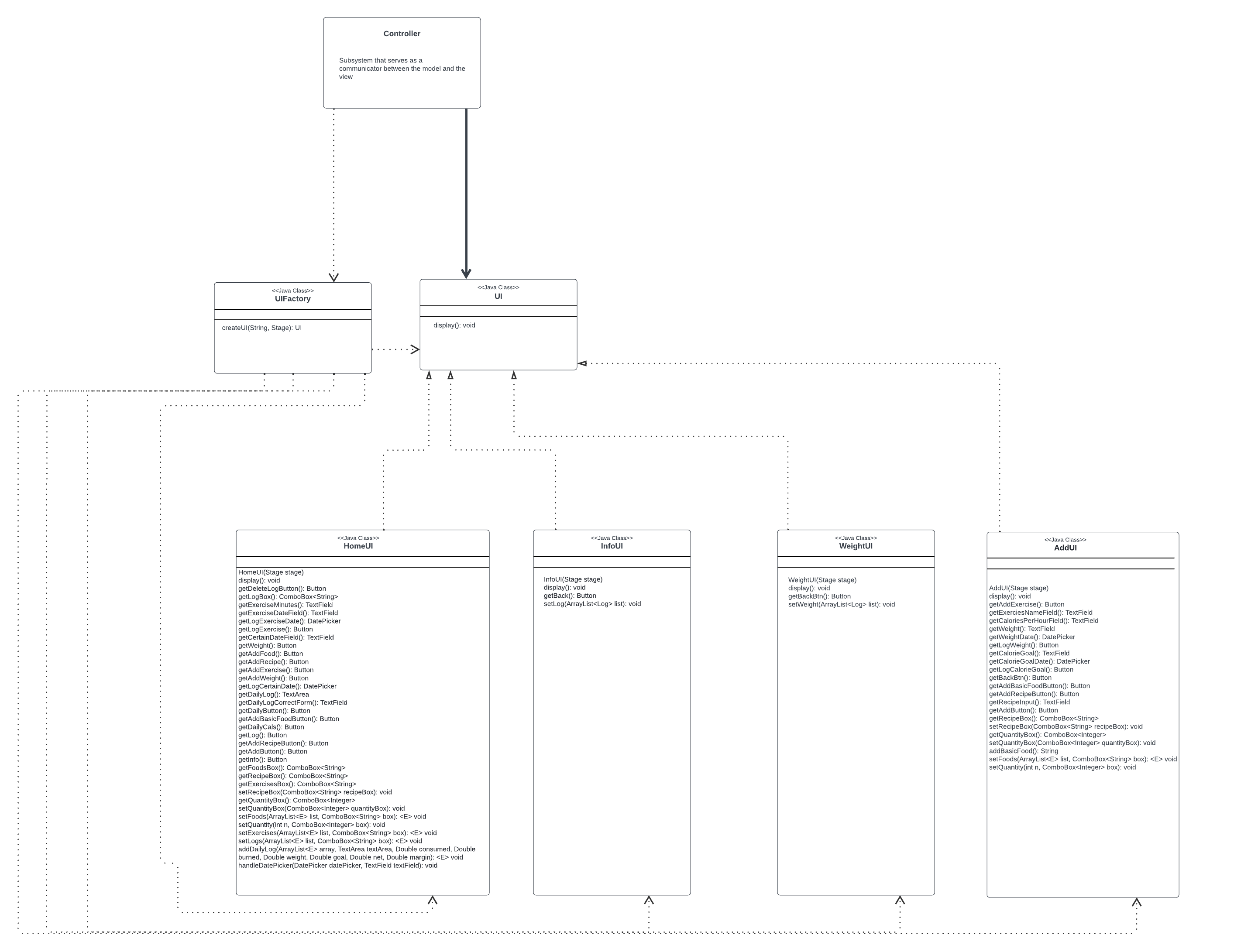


## **Subsystem DietController**



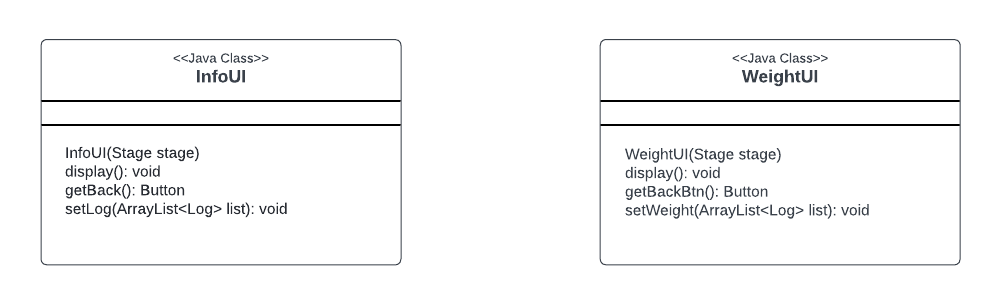
DietController subsystem serves the purpose of communicating between the view and the model. When the client does something on the UI, it notifies the controller, and the controller notifies the model subsystems to do the task. DietRunner is responsible for the main method and running the project, and the DietController class is the controller class that serves for communication between the view and the model. DietController knows about the view and the model, but view and model do not know about the controller.

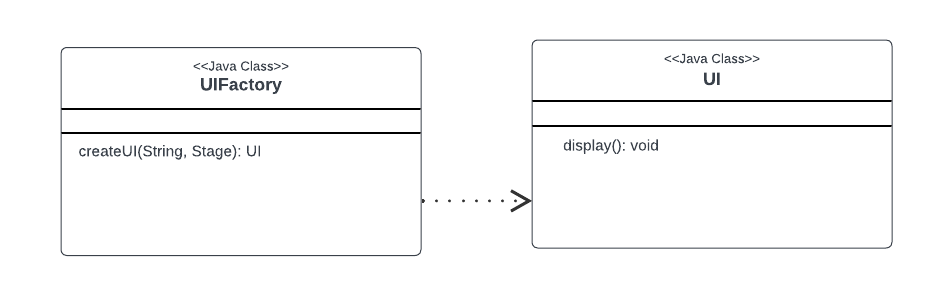
## **Subsystem DietView**

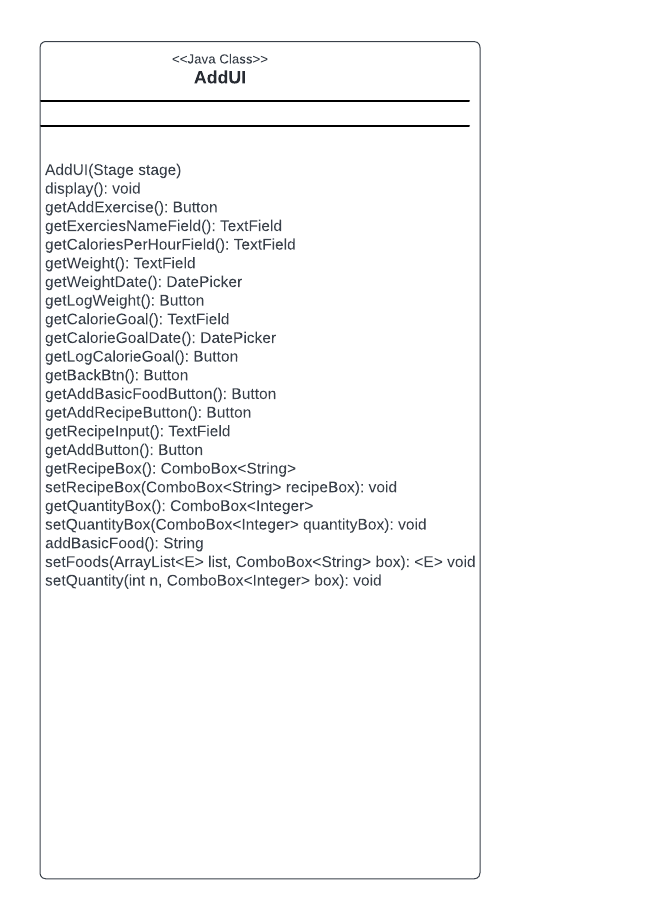


DietView Subsystem serves the purpose of creating UI’s or the view part of the project that the clients will be interacting with. The factory pattern is used in UI creation to isolate the logic of object instantiation, making the application easily expandable with other UI and view elements. The principles 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 does not know about model or the controller.

View classes individually (for better method names visibility)



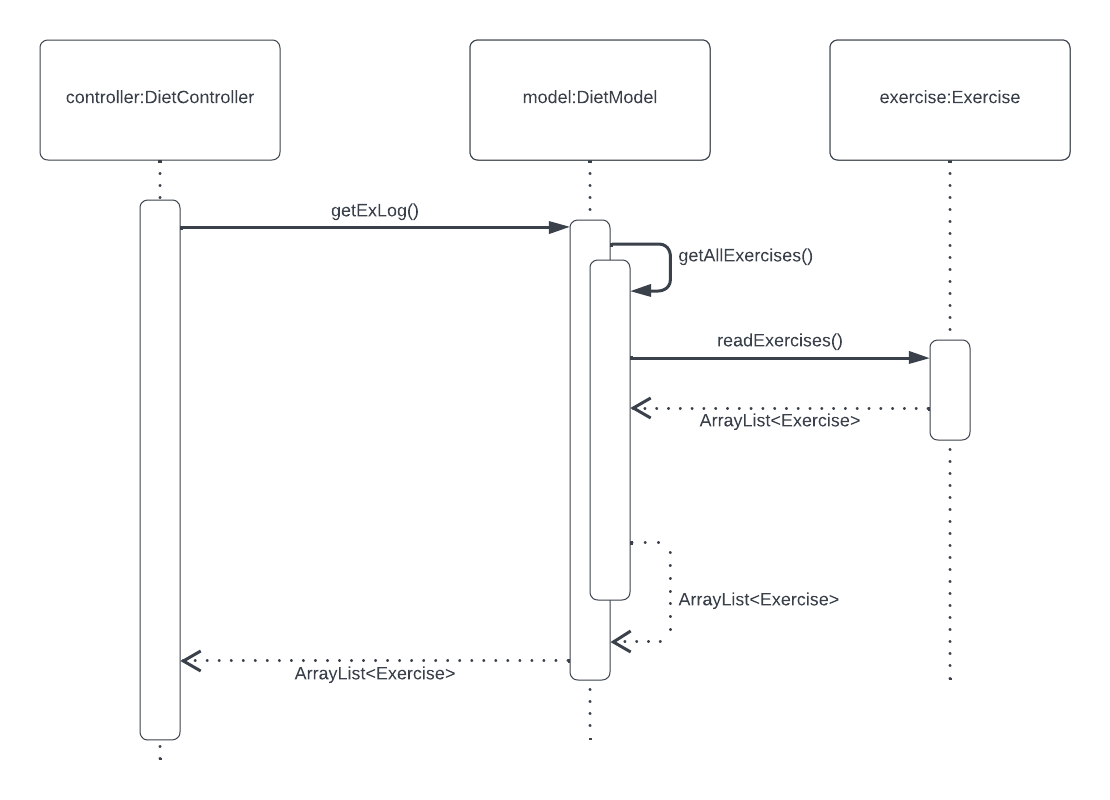






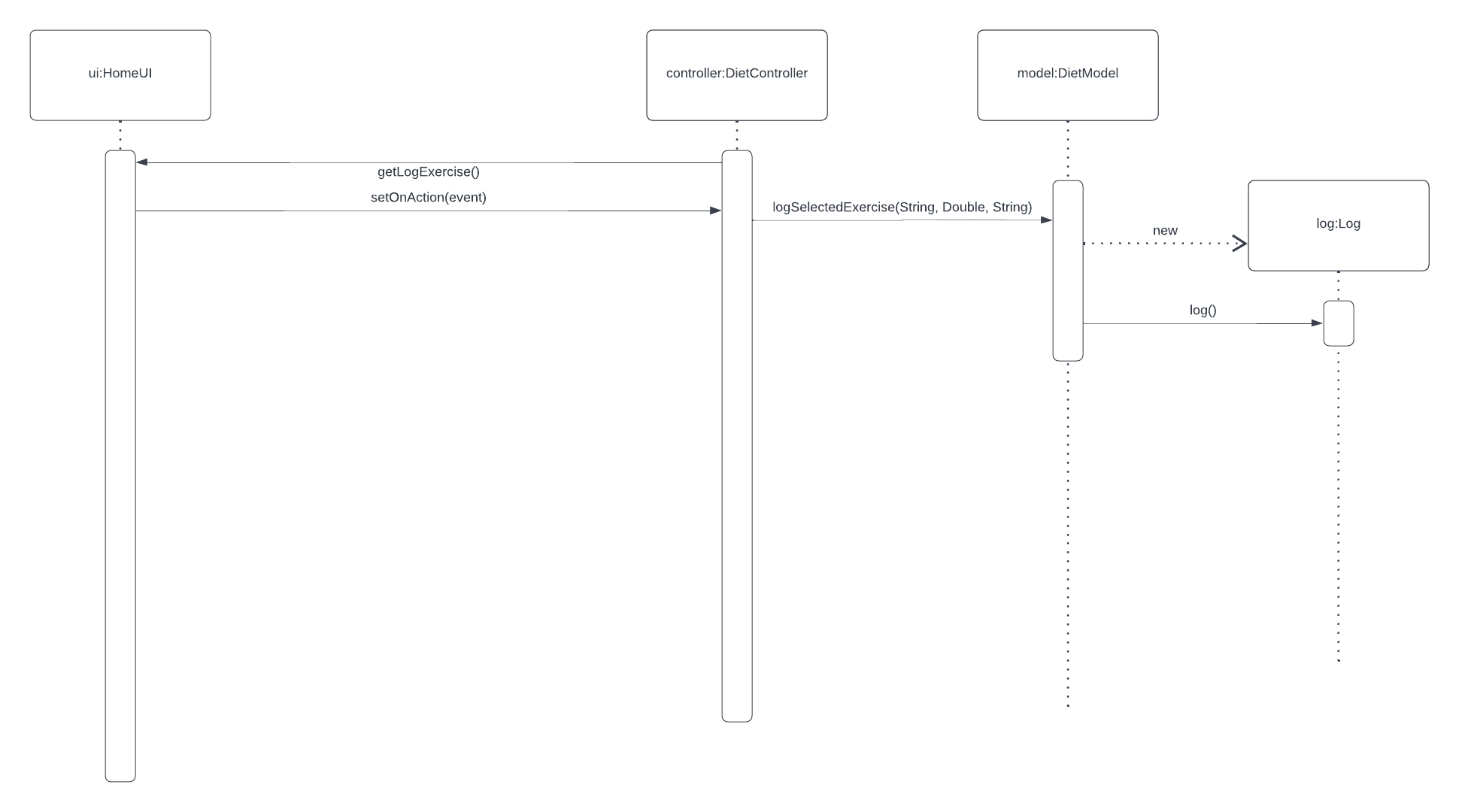
# Sequence Diagrams

## **Sequence Diagram 1: Exercise records are loaded from the exercise.csv file into the internal data structure**



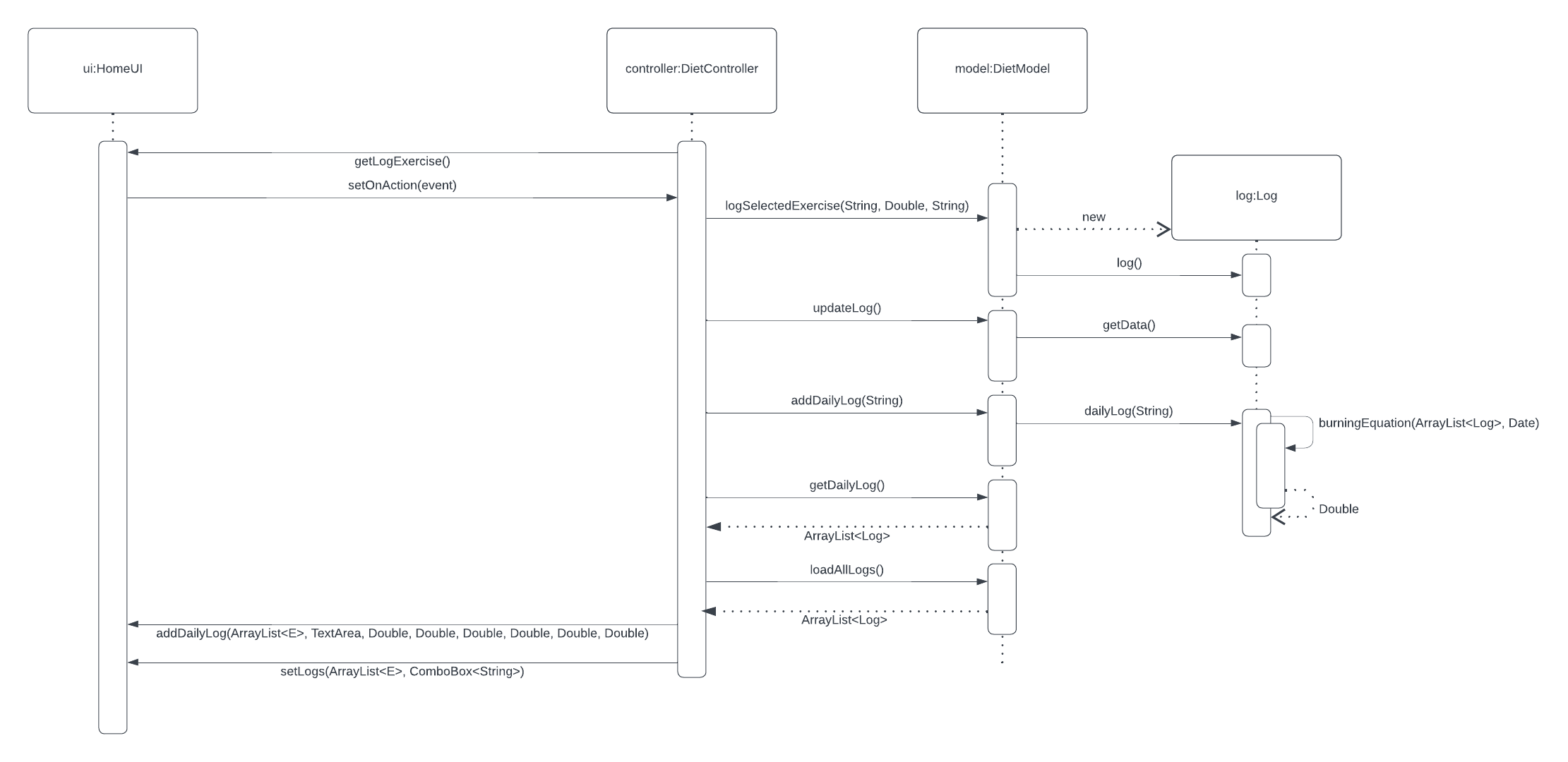
On start of the program, DietController calls getExLog() method from the DietModel, that returns the ArrayList of Exercises. The getExLog method is calling the getAllExercises() method, that is calling the readExercises() method from the Exercise class. readExercises() method is responsible for reading the exercises.csv file, load the Exercise classes into the ArrayList and return the ArrayList to the DietModel. That way, on the start of the program, exercises are loaded from exercises.csv into an internal data structure – ArrayList. That ArrayList is then passes to DietModel and DietController for further use, data manipulation, and injection into the view.

## **Sequence Diagram 2: Exercise is selected and recorded into the log.csv file**



When a log Exercise button is clicked on the view, the setOnAction method is triggered in the DietController class. That method is responsible for logging that exercise by calling the logSelectedExercise on the DietModel, passing the name of the exercise, the number of minutes exercised, and the date on which the exercise occurred as parameters. DietModel is then creating a Log object, and calling the log method on that object. The log method is logging the toCsv() method of that object to the logs.csv file using the BufferWriter.

## **Sequence Diagram 3: Log view is updated in relation to the calories expended via the exercise**



When the Log Exercise button is clicked, the exercise is logged into the csv file like explained in the previous sequence diagram. After that, DietController calls the updateLog method, that is calling getData method from the Log class. That updateLog method is responsible for loading the new data from the csv into the internal data structure. After that, DietController also calls addDailyLog from the DietModel, passing String of a date as a parameter, and the dailyLog() method is called on the Log, which is populating an ArrayList of Logs that happened on that particular date passed as a string. The dailyLog() method is calling the burningEquation() method to calculate the calories burned based on the weight, minutes, and calores per hour per kg of that exercise. After that, using getters getDailyLog(), and loadAllLogs() the DietController gets the ArrayLists of Logs with the updated exercises, and displays that data, along with the data from the burningEquation() method to inject into the view, and show that new logs to the user.