PROJECT / RELEASE

Project Design Document

Group 1

Amra Rrahi <ar9393@rit.edu>

Arian Vucelić <av7189@rit.edu

Ejup Elezi <ee1641@rit.edu>

Janko Leskovac <jl8592@rit.edu>

Roko Hafner <rh9922@rit.edu>

# Project Summary

The Diet Manager Application is a software tool that is designed to assist users in managing their dietary habits effectively. It integrates various features including a vast collection of foods, recipes, daily intake tracking, and weight monitoring, all aimed at promoting healthier lifestyles and facilitating users’ progress toward their fitness goals.

Key features of our application include a food database that offers nutritional information such as calorie count and, a recipe repository that makes it easier for users to plan means that align with their fitness goals. Daily intake tracking is important since users can log their meals throughout the day, enabling them to monitor their calorie intake in real-time.

The benefit of using our Diet Manager is that it provides personalized recommendations based on users’ dietary preferences and health goals. Information given to the user will help them be more informed about their diet and how to get in shape as soon as possible.

# Design Overview

The project is done with a Model-View-Controller (MVC) design pattern in which we divide our application into three interconnected components, a model that manages the application’s state and responds to requests for information about the data, the view that represents the presentation layer of our application, and controller that is an intermediary between model and view. We chose this design since we separate the application logic which makes it easier to manage and modify each component independently. Additionally, it is easier to maintain the application since everything is divided into logical parts, therefore, we can add and remove new features more easily.

## **Directory Structure**

Firstly, we created a clear directory structure that will help us navigate through the project and make it easier to know what each directory contains.

## **View**

The view was implemented right after the directory structure since it represents the user interface of our application. We have decided to have four buttons and two text areas. Buttons can load the data, add food, add recipes, and add food to the logs. The first text area located on the left has foods and recipes in it, while the log is on the right side and represents what person ate at what time.

## **Model**

In the model, we have created an abstract class, csvModel, that consists of the constructor, and abstract methods that are used in the Foods and Logs classes. Foods class manages food data and provides methods for reading and writing. It interacts with the FileHandler class which is used for file input and output operations. The logs class manages log data, provides reading and writing methods and also interacts with the FileHandler class. Log class contains accessors and mutators for attributes that are used for their respective attributes. It also contains the toString method which prints out information about logs. Food class is an abstract class that is used in BasicFood class, and it contains abstract methods which will be used in BasicFood. Additionally, there is BasicFood class which extends Food class and it contains accessors and mutators for attributes, as well as, inherited methods from Food class.

## **Controller**

The controller class is responsible for handling user input, updating the model accordingly, and manipulating the view to reflect any changes in the model. Our constructor acts as an intermediary between the view class and foods model and the logs model. It loads the data from log CSV file within the array list and in that way, it adds it to the log area as well. From foods CSV file it reads basic foods and recipes which are also displayed in the area in the application.

**Factory**

The Factory subsystem is tasked with the creation of Food objects. It utilizes the createFood method to generate a Food object based on the data extracted from the foods.csv file. This method constructs a Food composite object and distinguishes between BasicFood and Recipe objects based on the attributes read from the file. These objects serve as composed component elements within the composite structure.

# Overall System Structure

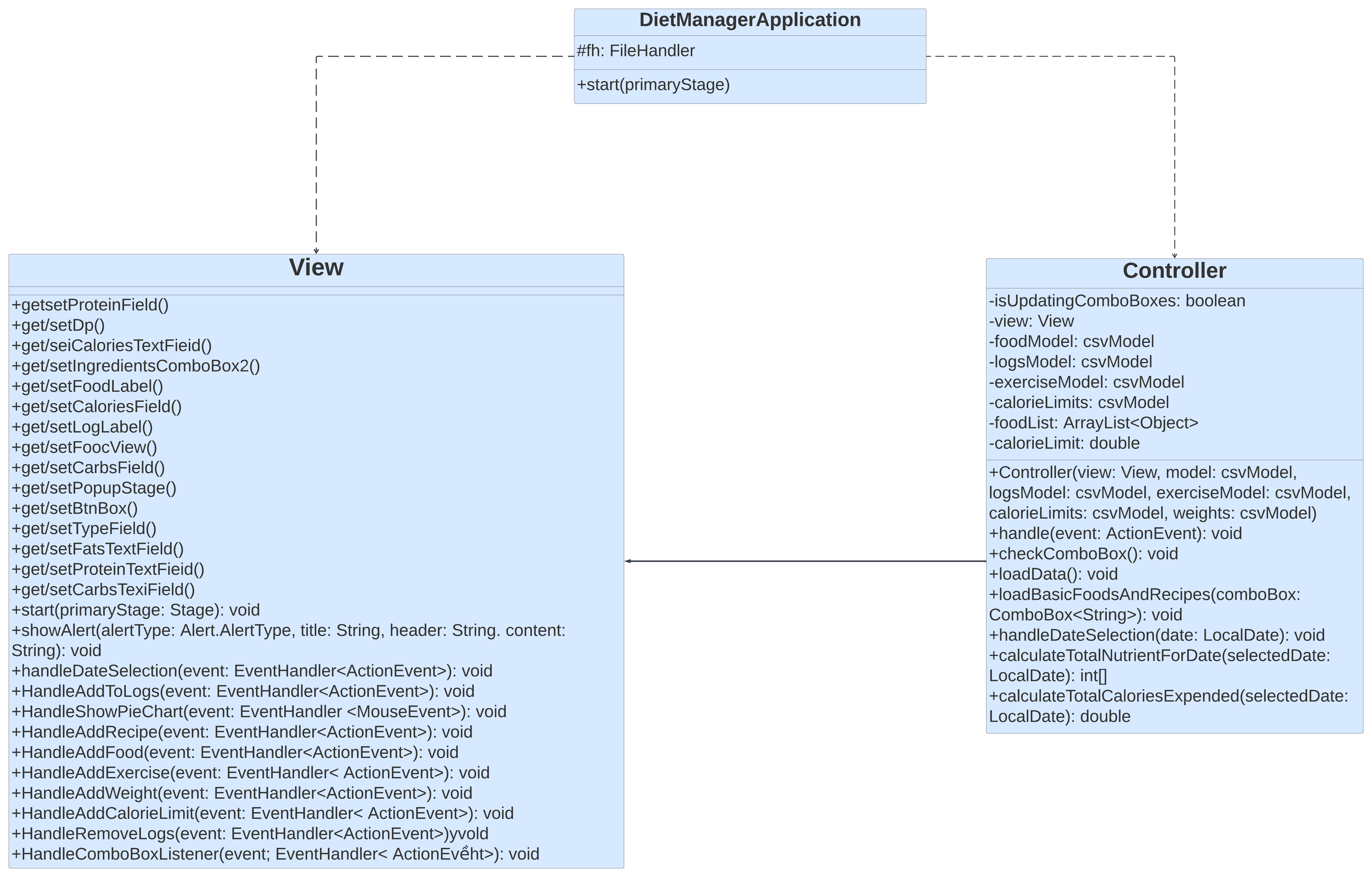
# A diagram of a computer flowchart Description automatically generated

In this class diagram, there are three main subsystems, model, view, and controller. The view is responsible for the graphical user interface and creates it. The model consists of thirteen classes out of which two are abstract classes. CsvModel is an abstract class with a constructor and abstract methods. Also, Food is an abstract class that contains only abstract methods. Foods class has the responsibility of accessing and modifying the food, as well as, reading and writing the foods into files. Similarly with log, exercise, weight, calorieLimit classes contain information and have accessors and mutators, while logs, exercises, weights, colorieLimits have reading and writing methods. Abstract class Food has its abstract methods that are inherited into BasicFood class which has a constructor for making basic foods and accessors and mutators for them. Additionally, read, and write methods are part of the csvModel abstract class that is inherited into Foods and Logs classes. The recipe class consists of ingredients and how many ingredients are needed for the recipe. The last class in the model is FileHandler which has two methods for getting a reader and writer. Additionally, there are handle methods for all buttons which make them functional and manipulate data.

Finally, the DietManagerRunner class is used for running the application which uses façade pattern to run DietManagerApplication.

# Subsystems

## **A black and white line with words Description automatically generated with medium confidenceSubsystem View**



The view subsystem creates the user interface for the application. It interacts with the controller class and handler classes since the controller needs a view to function. Data is changed through the user’s click on a button which functions through the handle methods that are part of the controller.

## **Subsystem Model**

## **A diagram of a diagram Description automatically generated**

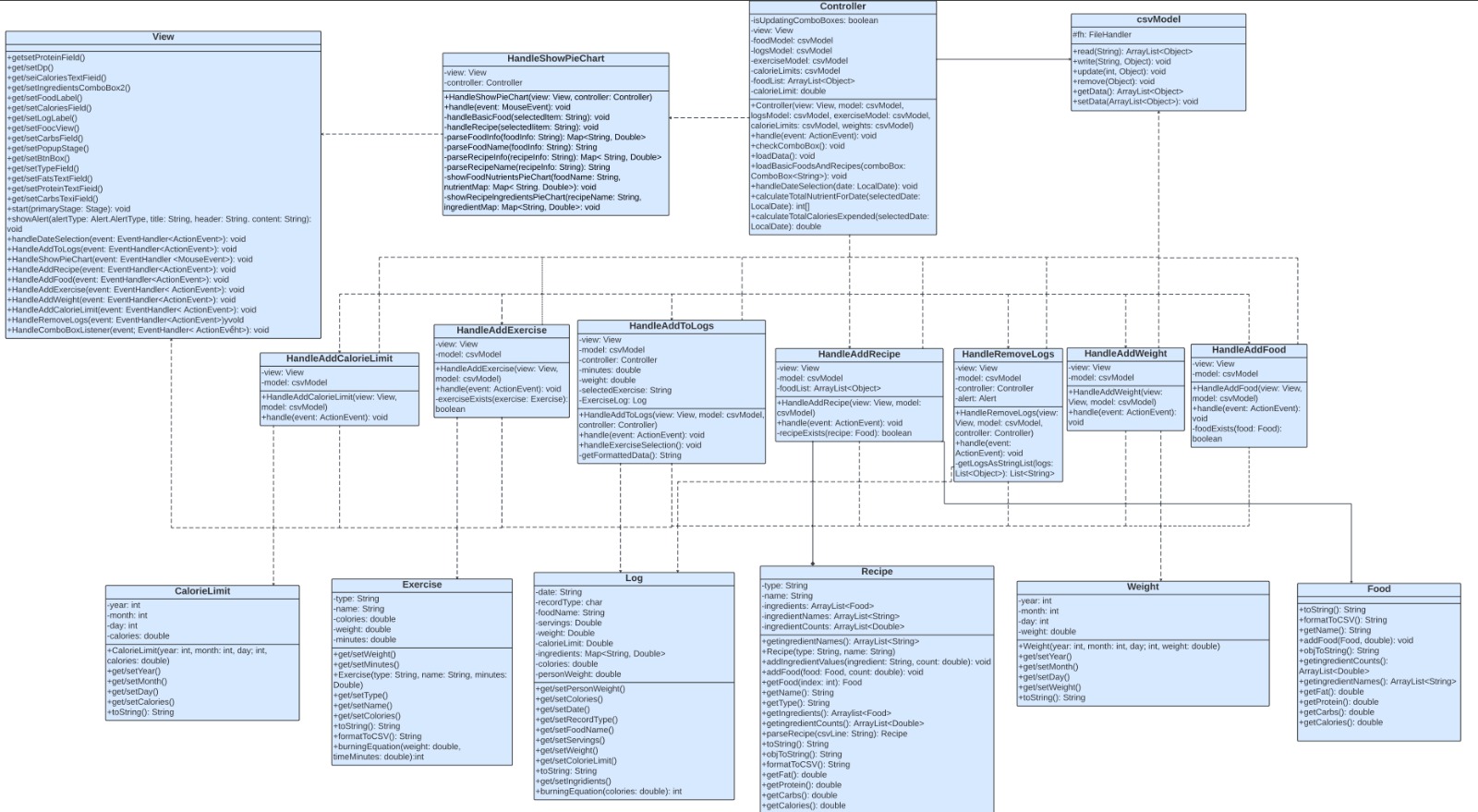
## A diagram of a computer Description automatically generated with medium confidence

The model subsystem manages the application’s state and responds to requests for information about the data. In our application, it reads values from the CSV file. It interacts with the controller since it needs a model to function. Methods from the abstract class, csvModel, are inherited which is in the Foods, Logs, Exercises, Weights, CalorieLimits classes. Logs class uses the Log class to access the logs. The Logs are visible in the text areas which are part of View and therefore need to interact with them for our application to be accurate. Abstract class Food is used in BasicFood class that inherits methods from Food class and creates a new instance of food object. Similar to logs all the classes have a collection of objects such as Exercise, Weight, CalorieLimit.

**Subsystem Controller**

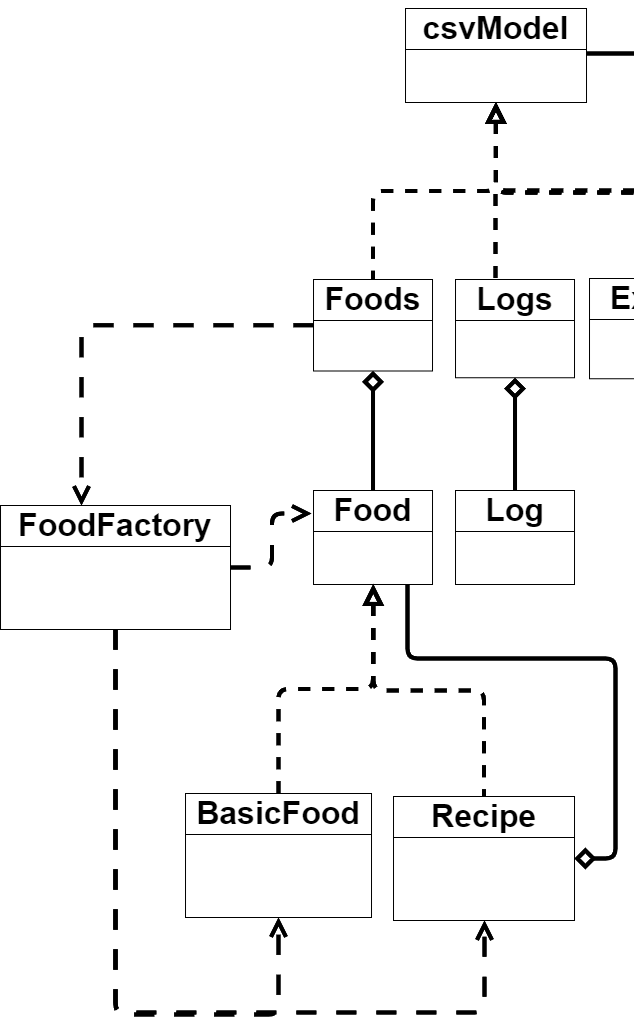
**A close-up of a document

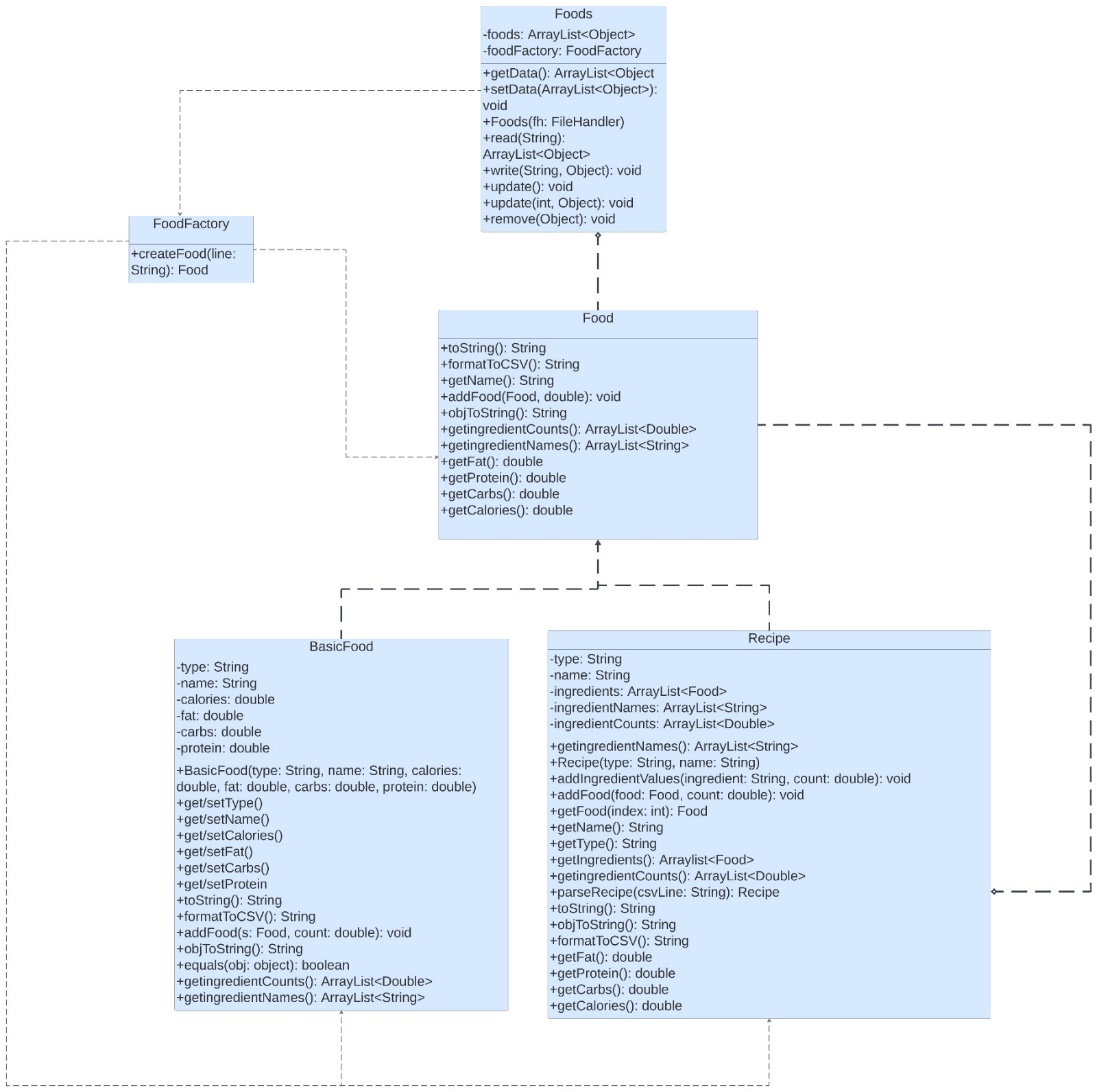
Description automatically generated**



The controller subsystem consists of the Controller class and all handle classes that make the buttons from view work. Controller object has both view and csvModel and it is used as an intermediary for model and view. It cannot exist without those classes. It is responsible for handling user input, updating the model accordingly, and manipulating the view to reflect any changes in the model. Handle classes use View’s buttons and csvModel to make buttons work and manipulate data displayed in the text area.

**Subsystem Factory**

****

****

The Factory subsystem is responsible for creating Food, it creates a Food object based on what is read from the foods.csv file using the createFood method, the method creates a Food based on the attributes that are read from the file. It differentiates BasicFood and Recipe objects. The recipe is a composite of Food.

# Sequence Diagrams

**Sequence Diagram 1 – Shows how exercise records are loaded from the exercise.csv file into the internal data structure (model).**

Once the user runs the program, the DietManagerApplication calls the start method which instantiates the main controller which contains the view and all the models. Then it also starts the view. Once the controller is instantiated, in its constructor the loadData method is called by default, to add the information to the view without the need of a event to trigger it. This method calls the read() method of the Exercises, with reference to the path of exercise.csv file as an argument, to read and return that information in an arraylist of type exercise, where each objects contains information for 1 exercise in the file. Once we receive this information, we then use a for each loop to load the data of each exercise object to the view. We first get the user friendly format of the data by calling exercise.toString() method, and then we add that string to the view.

## A diagram of a software development Description automatically generated

**Sequence Diagram 2 – Shows how an exercise is selected and recorded into the log.csv file.**

Once the user runs the program, the DietManagerApplication calls the start method which instantiates the main controller which contains the view and all the models. Then it also starts the view. Once the controller is instantiated, in its constructor will contain handlers to handle events on button click. In our case, HandleAddToLogs is triggered when the button add to logs is clicked. This handler checks if the selected item is of type Exercise. If so, it then calls the HandleExerciseSelection method. This method makes a pop-up window for the user to input the minutes invested on the selected exercise, and the date the exercise was done. On button apply click, a new log object is created. This object will contain the information about the exerciseSelected as well as minutes invested and date this exercise was done. Then the same method uses the Exercises model to write to the csv file by calling the write(exerciseLog) method that takes a Log object as an argument and writes the information of that log to the csv file. Once this method is finished, It calls the loadData() method from the controller, to update the view so that it contains the recent information that was added to the logs.

A diagram of a software system

Description automatically generated

**Sequence Diagram 3 – showing how the log view is updated in relation to the**

**calories expended via the exercise.**

The sequence diagram begins with the application starting up and instantiating a new Controller, passing along components like the view, logs, and data related to exercises, food, calorie limits, and user weight. The Controller handles user interactions, such as date selections, which influence the display of logs or exercises relevant to that date. It also calls a function to calculate the calories expended on the chosen date. The Controller retrieves data by calling a getData() method on two separate csvModel instances, one for logs and another for exercises. These models read and parse CSV files, returning the data as array lists. The Controller then enters a nested loop, iterating over the exercises and logs. Within this loop, it checks if the exercise and log entries match by type, name, and date. If they do, it calculates the calories burned using the burningEquation() method and stores the result in a variable. After processing, the application updates the user interface to reflect the new calculations, ensuring that the user sees the latest information based on their interactions.

