PROJECT / RELEASE

Project Design Document

Group 2

Svebor Kamenski Bačun <[sk6309@rit.edu](mailto:sk6309@rit.edu)>

Luka Badžek <[lb6165@rit.edu](mailto:lb6165@rit.edu)>

Stipe Polić <[sp2181@rit.edu](mailto:sp2181@rit.edu)>

Marko Obšivač <mo3854@rit.edu>

Tena Vladović-Relja <tv8312@rit.edu>

# Project Summary

The project, Diet Manager, will be a helpful tool to its users as it will hold all necessary dietary information for the user. The program will let the user put in information such as weight and the maximum calorie intake for the day, so that the user can track his/hers goal and progress easily, as well as informing the user how much over or under they are regarding their calorie intake during that day. The user will have a list of foods they can select and input as intake during that day, each food will have specific dietary information related to it, such as calories, fats, protein and carbohydrates. Foods only containing that information are registered as “basic foods”, whereas more complex foods, which are made up of two or more basic foods are registered as “recipe”. When a recipe is chosen, the program automatically calculates the dietary information based on basic foods the recipe is composed from. The program also lets the user input new basic foods and recipes and their dietary information. For each intake, the user can also choose the number of servings for that food or recipe. Over time, the program will record weight change for the user and display the data.

# Design Overview

On the first day, we started with a class diagram to envision how the structure of the program might look. The diagram changed very fast as we implemented the MVC (model-view-controller) structure two days later. For the first activity, we separated into three groups of two, each pair was working either on the model, view or the controller part.

# Overall System Structure

The program has a runner class calling the main method and initializing the application. The rest of the program has a MVC structure, meaning model-view-controller. The model holds all data and methods, the view encapsulates the interface of the application and the controller connects the two and makes the rest of the code function.

To explain the application and its structure more in detail;

The model part of the application consists of three folders: database, food and log.

The database folder holds Database.java, FoodDatabase.java, and LogDatabase.java.

The food folder contains all code relevant for the food database, it is comprised of Food.java, BasicFood.java, and Recipe.java files.

The log folder contains all code relevant to the data being saved to the log database, it consists of CalorieLimit.java, FoodIntake.java, LogEntry.java, and Weight.java files.

The View folder is comprised of two files: GU.javaI, which instantiates the GUI for the application and the DataChosen.java file.

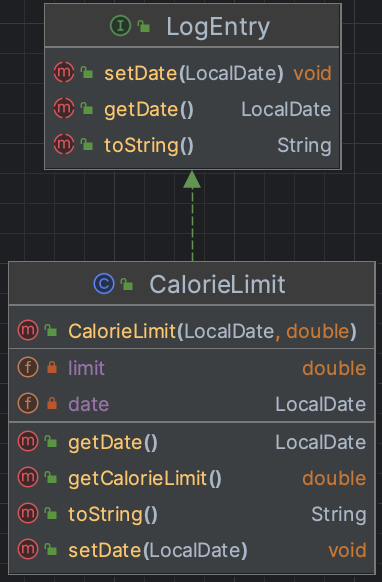
**Subsystems**

**Log sub-system**

**CalorieLimit.java**

The **CalorieLimit** class is a Java class that implements the **LogEntry** interface. This class represents a daily calorie limit entry in a log, which consists of a specific date and the calorie limit for that day. The class has two private member variables, **limit**, which is a double representing the calorie limit, and **date**, which is a Date object representing the date associated with the limit.

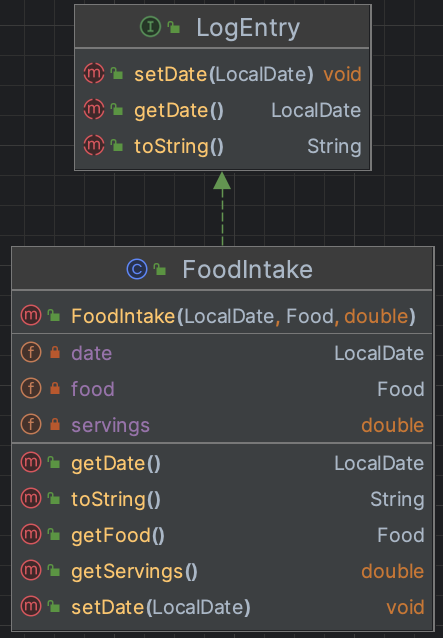
The class provides a constructor that takes a **Date** object and a **double** value for initializing the **date** and **limit** member variables. There are getter methods, **getCalorieLimit()** and **getDate()**, to retrieve the calorie limit and the date, respectively. The class also implements the **setDate(Date date)** method from the **LogEntry** interface, which allows updating the date associated with the calorie limit. **toString()** method returns CalorieLimit in a form of writable String that can be either saved to a file or presented to a user.



**FoodIntake.java**

The **FoodIntake** class, part of the **model.log** package, is a Java class that implements the **LogEntry** interface. This class represents a food intake entry in a log, which consists of a specific date, a **Food** object, and the number of servings consumed. The class has three private member variables: **food**, which is a **Food** object representing the food item consumed; **servings**, which is a double indicating the number of servings consumed; and **date**, which is a Date object representing the date associated with the food intake.

The class provides a constructor that takes a **Date** object, a **Food** object, and a **double** value for initializing the **date**, **food**, and **servings** member variables. There are getter methods, **getFood()** and **getServings()**, to retrieve the food item and the number of servings, respectively. The class also implements the **getDate()** and **setDate(Date date)** methods from the **LogEntry** interface, which allow retrieving and updating the date associated with the food intake entry. **toString()** method returns FoodIntake in a form of writable String that can be either saved to a file or presented to a user.



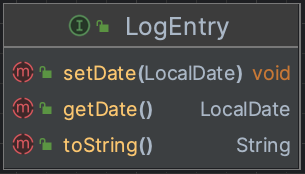
**LogEntry.java**

The **LogEntry** interface, part of the **model.log** package, represents a common contract for all classes used to log data based on user input. This interface aims to ensure that any class implementing it will have a standardized way to handle date-related information.

The **LogEntry** interface defines two methods:

1. **getDate()**: This method returns a **Date** object representing the date associated with the log entry. Implementing classes should provide their own implementation of this method to return the relevant date information.
2. **setDate(Date date)**: This method takes a **Date** object as a parameter and sets or updates the date associated with the log entry. Implementing classes should provide their own implementation of this method to update the date information as needed.
3. **toString()**: This method, when implemented by a concrete class, will return LogEntry in a form of writable String that can be either saved to a file or presented to a user.

By implementing the **LogEntry** interface, classes that represent different types of log entries can adhere to a consistent structure, making it easier to manage and work with the logged data.

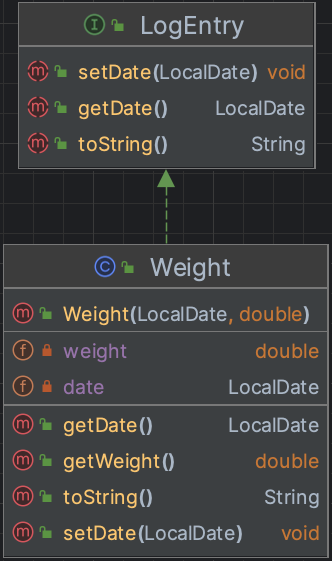


**Weight.java**

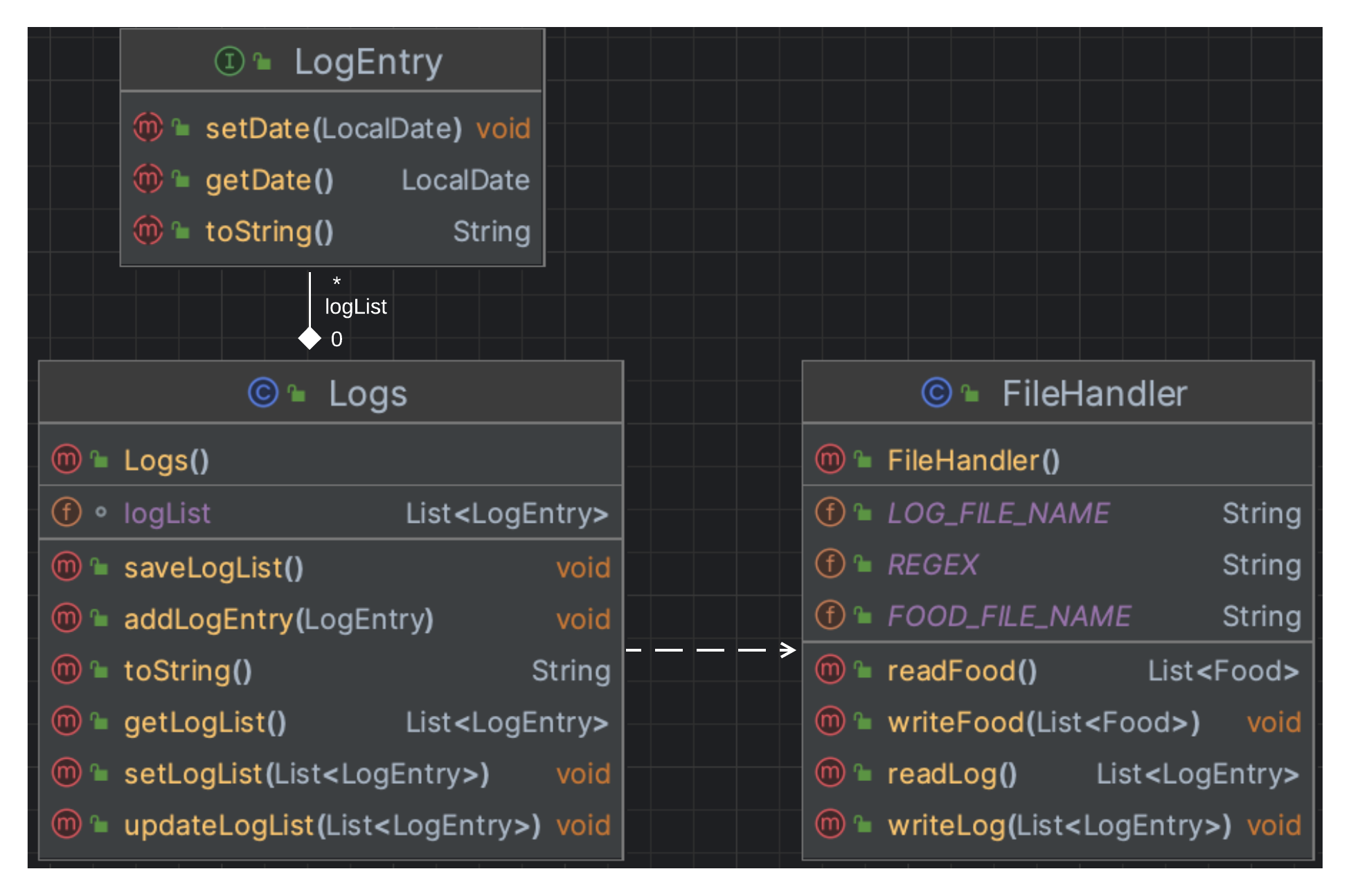
The **Weight** class, part of the **model.log** package, is a Java class that implements the **LogEntry** interface. This class represents a weight entry in a log, which consists of a specific date and the weight value for that day. The class has two private member variables: **weight**, which is a double representing the weight value, and **date**, which is a Date object representing the date associated with the weight entry.

The class provides a constructor that takes a **Date** object and a **double** value for initializing the **date** and **weight** member variables. There is a getter method, **getWeight()**, to retrieve the weight value associated with the entry.

The **Weight** class also implements the **getDate()** and **setDate(Date date)** methods from the **LogEntry** interface. The **getDate()** method returns the date associated with the weight entry, while the **setDate(Date date)** method allows updating the date associated with the weight entry. By implementing the **LogEntry** interface, the **Weight** class adheres to a consistent structure, making it easier to manage and work with the logged data. **toString()** method returns Weight in a form of writable String that can be either saved to a file or presented to a user.



**Logs.java**

****

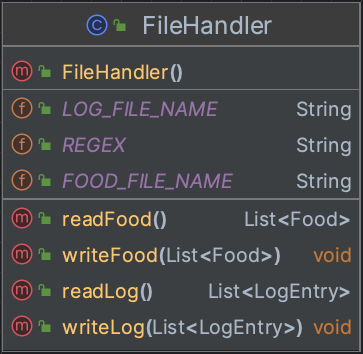
This is a Java class named Logs that belongs to the model.log package. The purpose of this class is to manage a list of LogEntry objects, which are assumed to be stored in a file. The class uses a FileHandler class to read and write the log data from/to the file. It provides an abstraction for managing a list of LogEntry objects and simplifies the reading and writing of the log data to/from a file. It also provides methods for adding new log entries, updating the list with multiple log entries, and retrieving the list of log entries.

The class has the following methods:

* a constructor method of the Logs class. When an object of the Logs class is created, it calls the readLog() method of the FileHandler class to read the log data from the file and initializes the logList instance variable with the data
* setLogList() sets the logList instance variable with the provided List of LogEntry objects
* updateLogList() adds all the LogEntry objects in the provided List to the logList instance variable
* addLogEntry() adds a new LogEntry object to the logList instance variable
* saveLogList() calls the writeLog() method of the FileHandler class to save the logList instance variable to the file
* getLogList() returns the logList instance variable, which contains a list of LogEntry objects
* toString() iterates through the logList collection and calls on their toString() methods, to return a collection in a writable form that can be either saved to a file or presented to a user.

**Database subsystem**

**FileHandler.java**

****

FileHandler is a class located in the model.database package. It has methods to read from and write to CSV files that store information about food and logs.

The class has two constant variables, FOOD\_FILE\_NAME and LOG\_FILE\_NAME, which store paths of the food and log files . static final REGEX is used to separate the values in the CSV files.

* readFood() reads food data from the CSV file specified by FOOD\_FILE\_NAME. It creates a list of Food objects, which can be either BasicFood or Recipe. Each line of the CSV file represents either a BasicFood or a Recipe. The method reads the file, splits each line into an array of values using the REGEX, and creates a BasicFood or Recipe object based on the values. It then adds the object to the list of Food objects and returns the list.
* writeFood() writes the list of Food objects to the CSV file specified by FOOD\_FILE\_NAME. It loops through the list of Food objects and writes each object to a line in the CSV file. It writes a BasicFood object beginning with the letter 'b', followed by the food name, calories, protein, carbs, and fat, separated by the REGEX constant. It writes a Recipe object as a line beginning with the letter 'r', followed by the recipe name, and then each ingredient and its amount, separated byREGEX. That writing is actually using a call to a Food’s toString method to write each row/ entry.
* readLog() reads daily log data from the CSV file specified by LOG\_FILE\_NAME. Each line of the CSV file represents a LogEntry, which can be a FoodIntake, Weight, or CalorieLimit. The method reads the fil, splits each line into an array of values using REGEX, and creates a LogEntry object based on the values. It then adds the object to the list of LogEntry objects and returns the list.
* writeLog() writes the list of LogEntry objects to the CSV file specified by LOG\_FILE\_NAME. It loops through the list of LogEntry objects and writes each object to a line in the CSV file, and writes a FoodIntake object as a line starting with the letter 'f', followed by the date, food name, and servings, separated by the REGEX constant. This method writes a Weight object starting with the letter 'w', followed by the date and weight, separated by REGEX. It writes a CalorieLimit object starting with the letter 'c', followed by the date and calorie limit, separated by REGEX.That writing is actually using a call to a LogEntry’s toString method to write each row/ entry.

**Runner.java**

The Runner class contains the main method, which serves as the entry point for the program. In the main method, an instance of the Controller class is created, passing in a parameter of type gui.

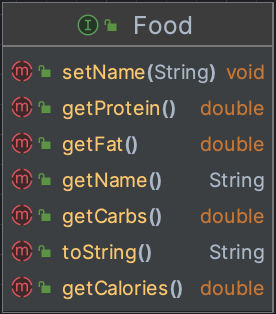
This instantiation creates a connection between the user interface (gui) and the backend logic (Controller),

allowing the user to interact with the program and perform various operations.



# Food subsystem

# Food.java



An interface which provides a template for objects that represent BasicFood and Recipe entries. The interface declares five methods, setName(), getName(), getCalories(), getFat(), getCarbs(), and getProtein().

The setName() method takes in a String values as a parameter and sets the name property of the food object. The getName() method returns the String value of the name of the food object.

Methods getCalories(), getFat(), getCarbs(), and getProtein() return the property values for the food object (all of which are of double type).

Method toString(), when implemented by a concrete class, will return Food in a form of writable String that can be either saved to a file or presented to a user.

Any concrete classes that implement this interface will have to implement methods of this interface and that will make sure that each kind of food behaves in a way we intended. Having this super type will make it easier to work with both BasicFoods and Recipes in the same way. Also, even recipe can have a sub-recipe, and abstraction (having Food super type) makes it easier to imagine Recipe subclass (which can now have a list of foods as ingredients, not a list of basic foods and a separate list of sub-recipes).

# 

# BasicFood.java

# 

BasicFood implements the Food interface. It has private attributes: String name, double calories, double fat, double carbs, and double protein, properties of BasicFood objects. BasicFood objects are Foods that do not have more Food ingredients (than themselves).

The constructor BasicFood(String name, double calories, double fat, double carbs, double protein) is used to create a new instance of a BasicFood object, with provided values assigned to corresponding properties.

Getter methods getName(), getCalories(), getFat(), getCarbs(), and getProtein() are used to access the values of BasicFood object’s properties.

Setter methods setName(String name), setCalories(double calories), setCarbs(double carbs), and setProtein(double protein) serve the purpose of augmenting property values of a BasicFood object after it has been created.

toString()method returns BasicFood in a form of writable String that can be either saved to a file or presented to a user.

BasicFood class will also serve in a Recipe class, as an ingredient of a Recipe (which has a list of Food as an ingredients property).

# Recipe.java

# 

Recipe implements the Food interface. It has private attributes String name and a Map of Food as a key and Double as a value (for servings), properties of Recipe objects. Recipes are objects that consist of other Food objects (along with how many servings of them in a form of a Double values associated with Food key). Those food objects can be both BasicFood of Recipe, and are treated the same (are paired with a Double value of servings inside the ingredients Map).

The constructor Recipe(String name, Map<Food,Double> ingredients) is used to create a new instance of a Recipe object, with provided values assigned to corresponding properties.

Setter methods setName(String name), setIngredients(Map<Food, Double> ingredients) serve the purpose of augmenting property values of a Recipe object after it has been created.

Getter methods getName(), getIngredients() are used to access the values of Recipe object’s properties.

Methods double getCalories(), double getFat(), double getCarbs(), double getProtein() calculate and return the values of corresponding properties of all BasicFood items inside (taking in consideration the amount of servings used in a recipe).

toString() method returns Recipe in a form of writable String that can be either saved to a file or presented to a user.

Recipe class uses both Food subclasses – BasicFood for basic ingredients, Recipe for sub-recipe (Recipe-type) ingredients.

**Foods.java**

# 

Foods is a class that belongs to the model.food package. The purpose of this class is to manage a list of Food objects. The list of Food objects foodList serves as a kind of a working memory - user is augmenting the list as he adds new food entries or logs, and when he is ready to commit changes, he can save to the file. The class uses a FileHandler class to read and write the food data from/to the file.  
Foods class provides an abstraction for managing a list of Food objects and simplifies the reading and writing of the food data to/from a file.

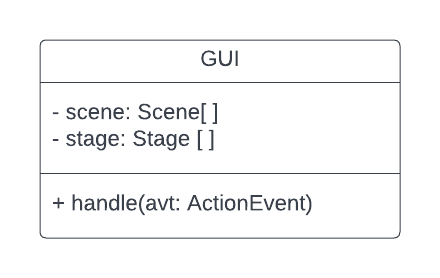
The class contains the following methods:

* a constructor Foods() that initializes it and sets the foodList with the values read from the file
* setFoodList() sets the foodList instance variable with the provided List of Food objects
* addFood() adds a new Food object to the foodList instance variable
* updateFoodList(): adds all the Food objects in the provided List to the foodList instance variable
* saveFoodList(): calls the writeFood() method of the FileHandler class to save the foodList instance variable to the file
* getFoodList(): returns the foodList instance variable, which contains a list of Food objects
* toString(): iterates through the foodList collection and calls on their toString() methods, to return a collection in a writable form that can be either saved to a file or presented to a user.

# model package:

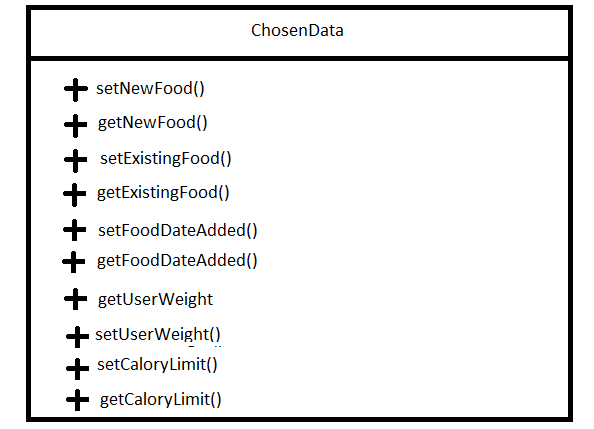
This subsystem serves as a model for objects that users will select, add, remove from list, log – in other words, work with. The Food interface is there to serve as a template for BasicFood and Recipe classes. It allows for abstraction and cleaner code (DRY principle). BasicFood and Recipe classes implement Food interface, Recipe uses Food (both BasicFood and itself – Recipe) as ingredients of its ingredients Map<Food,Double>.

**GUI.java**

The GUI class belongs to the View part of the file structure. It has private attributes stage and scene. it has a main method that initiates the GUI. The scene has a size set and is made up of three different parts: first part containing components for inserting daily intake, second part consisting of two text fields for listing available foods and previous logs, third part holds components for adding new foods to the database. 

**ChosenData.java**

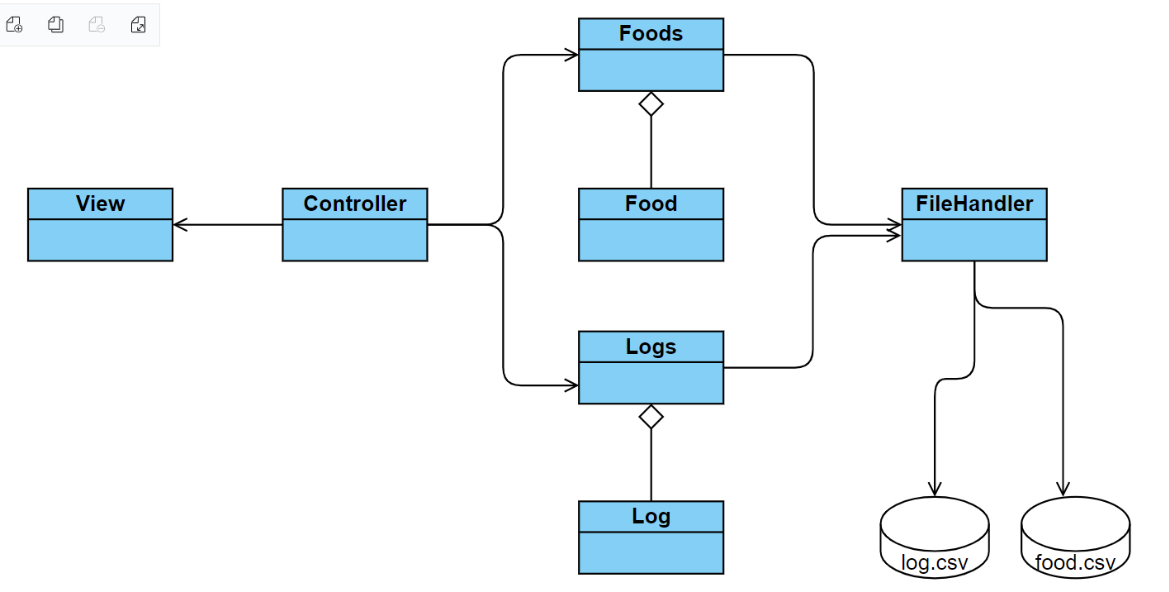
ChosenData is the View part of Diet manager. It has getters and setters for new food, existing food, date of adding the food, user weight, and calory limit. It will set and get the data that is imported into the GUI class.



**Controller.java**

The Controller class has two Strings that contain two csv files, where our information is being kept. There is also a constructor that uses one parameter and that is of type GUI, and it is called gui. There is also a method that reuters complete food with all of the statistics of what is inside of that food.

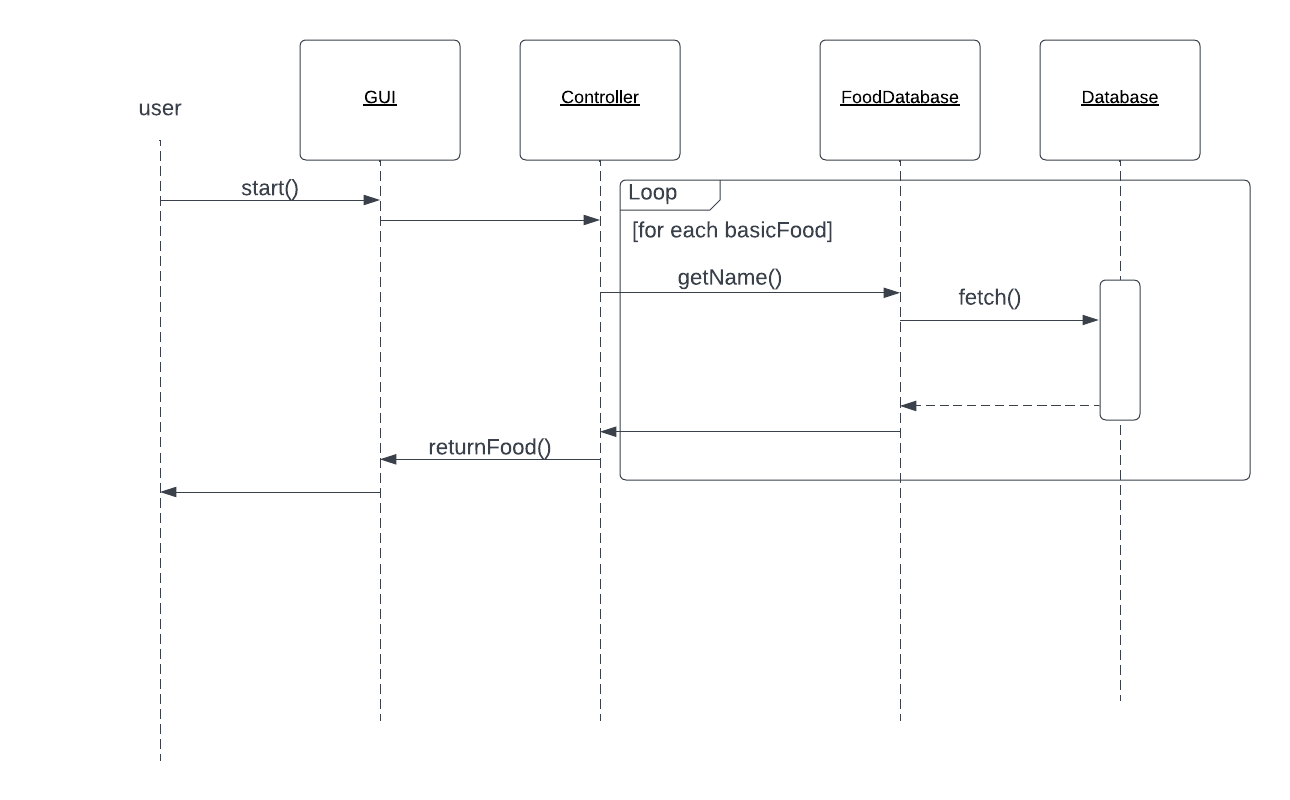
**dietmanager800g2 - class diagram**



We have decided to use this MVC model. Controller uses View and Foods and Logs classes. Foods and Logs serve as a kind of local storage or working memory. They are updated when inputs from View are passed by the controller to the corresponding methods from Foods and Logs classes. Foods and Logs classes use FileHandler’s methods for augmenting and reading log.csv and food.csv files, that serve as a kind of a database (or permanent storage) for the application.

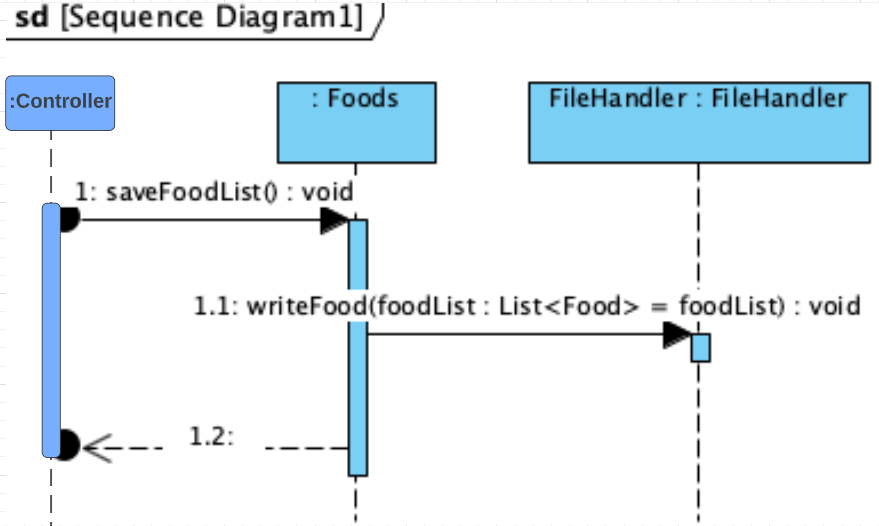
# Sequence Diagrams

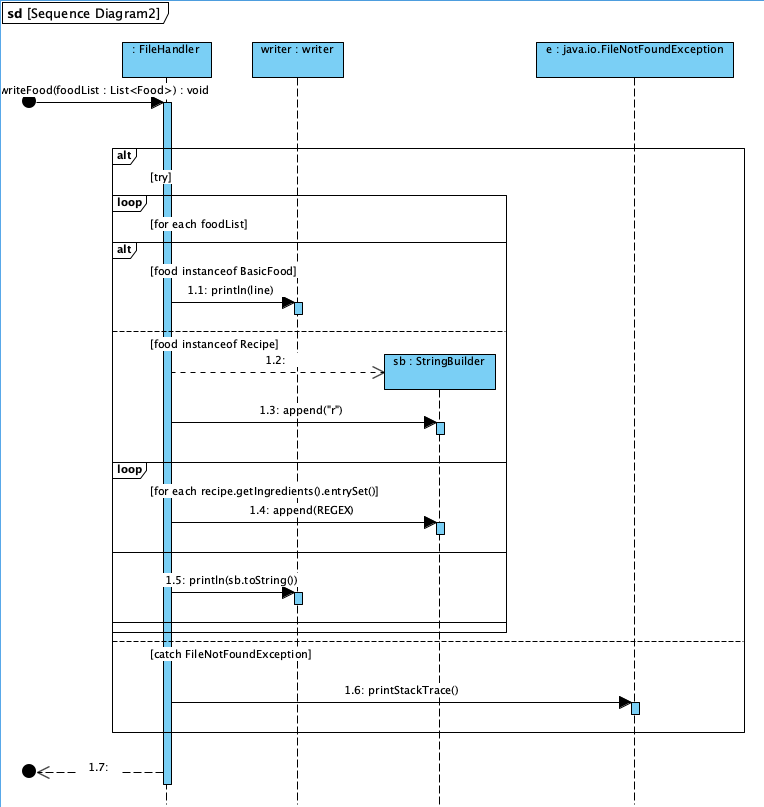
## **Sequence Diagram a.Read in a food database consisting of three basic foods.**



This sequence diagram shows the controller initiating the action of FoodDatabase fetching basic foods from the database and returning that data back to the controller to show them in the GUI in a textArea together with their dietary information associated with each food by the food name.

## **Sequence b. - Add two servings of a basic food to the log entry for the current date.**





In the first part of a diagram we can see how the method saveFoodList of Foods class is called from the Controller. That method passes values from the object it is belonging to (Foods), to FileHandler’s writeFoodList() method (Sequence Diagram 2). There is a for each loop that iterates through foodList We can see that there is a logic involved that decides which scenario will take place, as both Recipe and Basic food can be on the foodList (value that is passed to FileHandles’s writeFoodList()). There is also an inner for each loop in a case of a Recipe, that will iterate through ingredients of the Recipe. Values are written to the csv file using the PrintWriter writer and are separated with “,”. Also, in a case of BasicFood “b” is added, and in a case of Recipe “r” is added.

## **Sequence c. - Compute the total number of calories for the current date.**

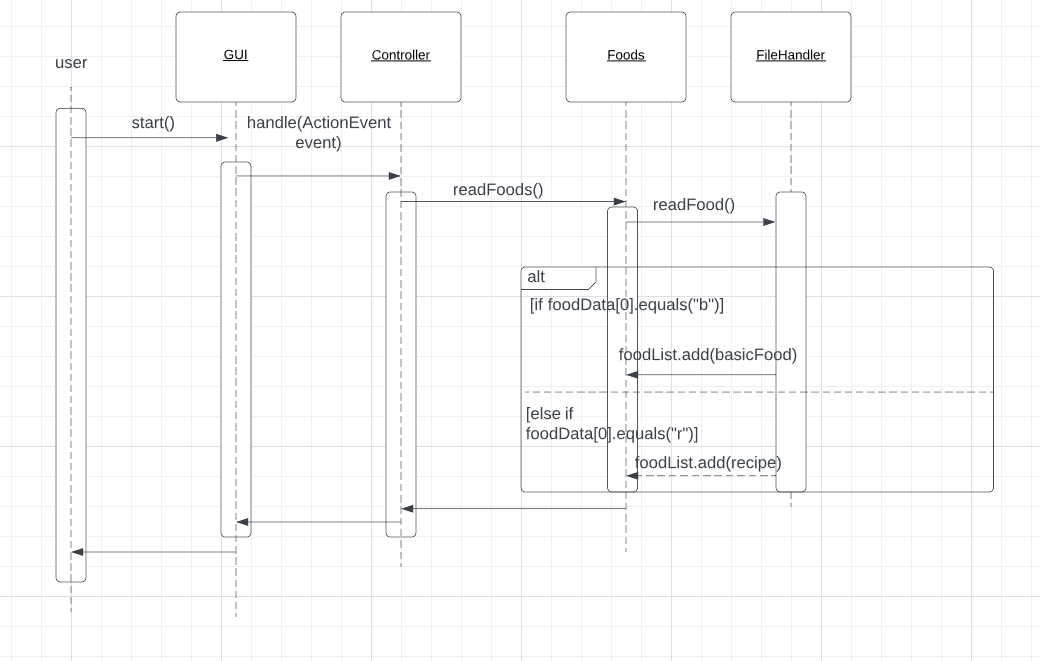
## 

The first part the sequence diagram shows that FileHandler reads log from the logs class and it returns the list log. The same happens when the file handler is reading logs from the food. That is how the total number of calories for the current date is computed.

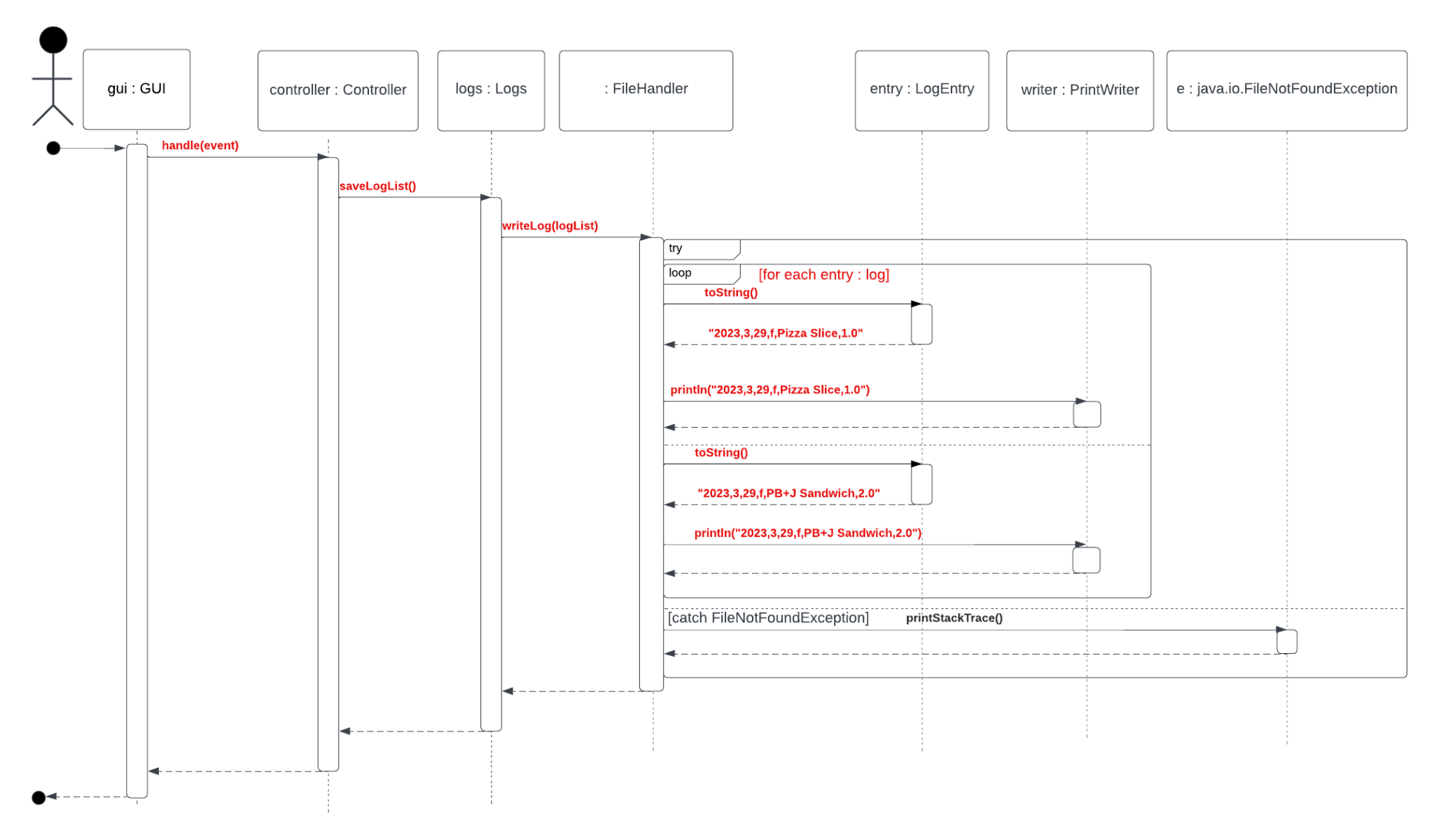
## 

## **Activity 2**

## **Sequence a. - Loading data for 1 basic food (Pizza Slice) and 1 recipe (PB+J Sandwich).**



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



## 

sequence c

