DIETMANAGERG5

PROJECT RELEASE

PROJECT DESIGN DOCUMENT

TEAM MEMBERS

Lamia Cero<lxc2877@rit.edu>

Paula Pufek<pxp1441@rit.edu>

Vjori Hoxha<vxh5681@rit.edu>

Ivor Baric<ixb6518@rit.edu>

Josip Vlah<jxv9840@rit.edu>

Project Summary

Diet Manager is a desktop – based application, which main focus is maintaining a user's diet program. The user will be able to monitor their daily intake of food, such as calories, proteins, carbs, fats and others. This application will allow users to add food which will then be organized into subcategories; furthermore, the application will have a feature where the client will be able to make recipes by using newly added food as well as by using sub-recipes.

The user will be able to select all the food consumed during the day as well as the number of servings for each food. After that, the Diet Manager application will be programmed to display all the necessary information about the food, such as carbs, fat intake etc. from the food that the user has consumed during the day, as well as the servings of each food.

Another important part of the Diet Manager application is the ability to record user's weight. For example, let's say that the user wants to set their weight goal during some period of time, Diet manager will be able to show the weight change over time and show user how much calories and other nutrients he/she might consume in order to achieve their desired weight goal.

Design Overview

Diet Manager will be divided into separate classes that will have different functionalities. When it comes to Separation of Concerns, the application will have its own classes that will address a separate concern. For example, as seen in the UML diagram Loader class will load foods from the CSV file and insert them into a data structure of Recipe or BasicFood, Diet Manager GUI class will only interact with the graphical user interface of the application, furthermore logger will only be used for logging the exceptions and errors that may occur during the run time of the application.

When it comes to cohesion, Diet manager is designed to have high cohesion, which means that the code inside of the application is closely related to each other for example, Food writer class as shown on the UML diagram, has only code related to that part. It contains write() for recipefood file; whereas, LogWriter contains write() method for the Log file. These two classes extend the Writer.

On the other hand, when we are talking about Coupling in the application, Diet Manager is created to have the lowest coupling possible, which in other words means that classes are independent from other classes. When one class is changed it does not affect other classes in the application. If we look at the UML diagram of our

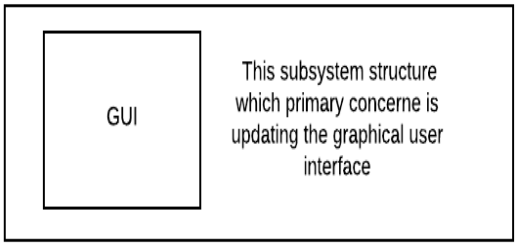
application Diet Manager GUI class only interacts with the Controller class, Recipe and Basic Food classes only Interact with the Food Interface.

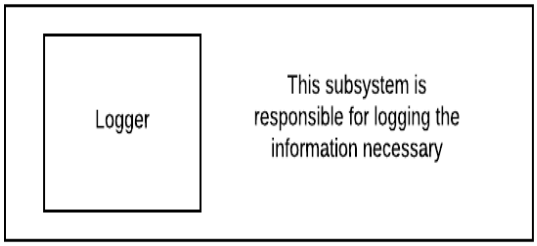
Extendibility as well as dependency via interface has been carefully designed to provide the best solution for future scalability of the application. Data, in this case Recipe and Basic food implement interface Food which then interacts with other classes. This means that Diet Manager Application supports program to interface not implementation principle.

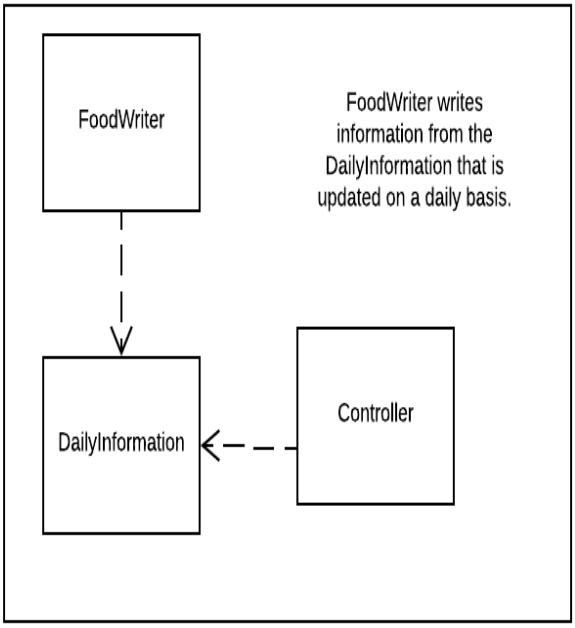
The design overview is the narrative that captures the thought process and evolution of the design from the preliminary design sketch through final implementation. The narrative should support how the project design addresses good design principles: separation of concerns, high cohesion, low coupling, dependency inversion via abstractions (interfaces), support for extendibility, etc.

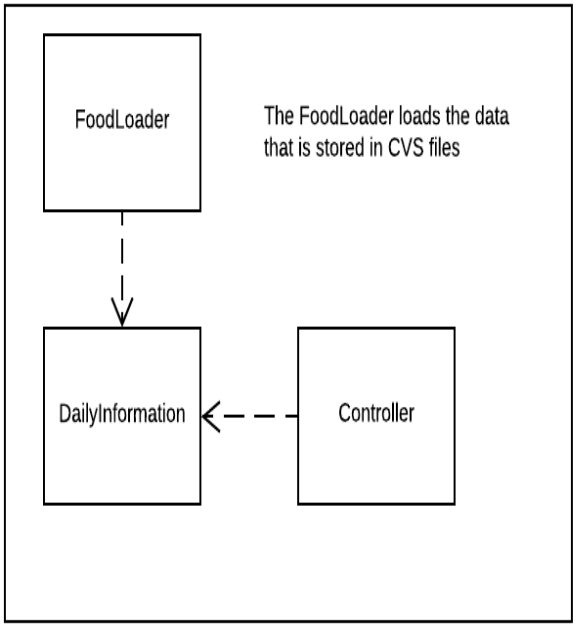
It is equally important to document design decisions that did not go as anticipated, as it is decisions that worked out well. This is extremely helpful background for future readers of the document to help them avoid solution paths that already had been attempted when extending the project with new or modified features. It is common for some design documents to have an entire section dedicated to “rejected alternatives”.

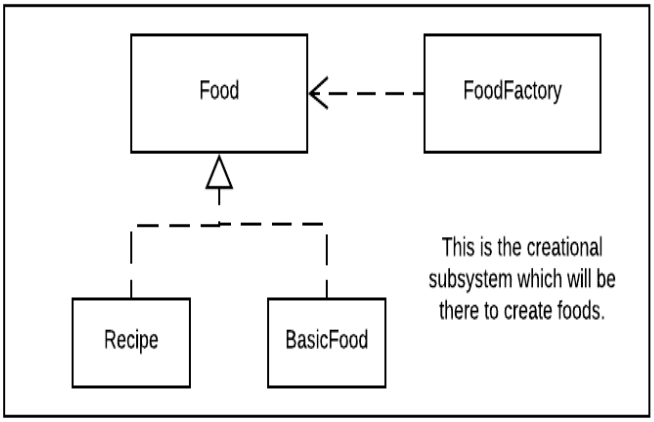
Subsystem Structure











Subsystem GUI

- Responsible for visuals and user communication
- Interacts with the controller class

Subsystem Logger

- Responsible for logging the necessary information to a file

Subsystem Writer

- Responsible for writing the data to the .csv file
- It can be used for writing either Daily Info or simply new foods the user comes up with
- Interacts with the controller

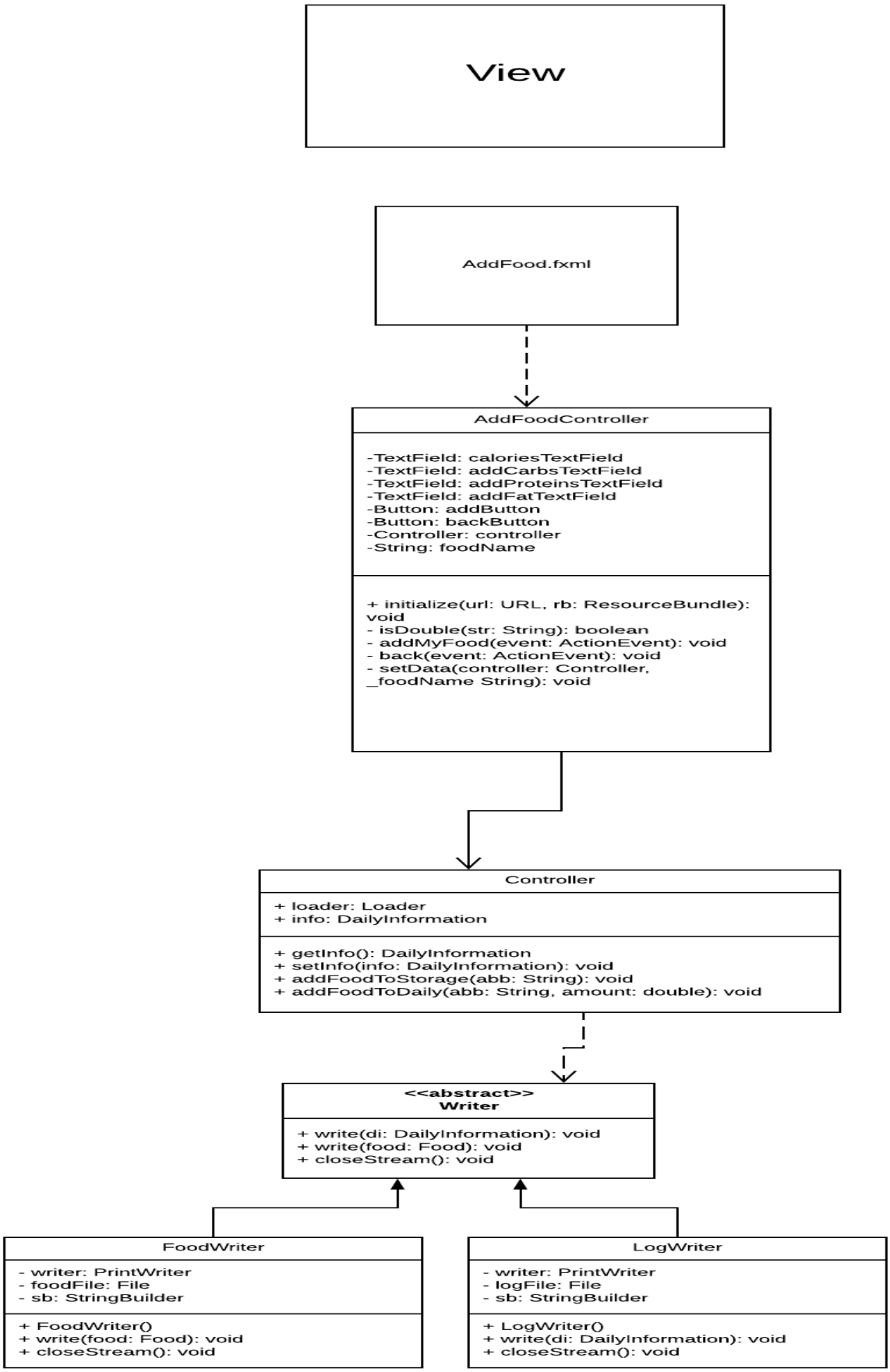
Subsystem Reader

- Responsible for filling out the data within our objects with the data from the .csv file
- Can be used for reading either Daily Info or the foods that the user came up with

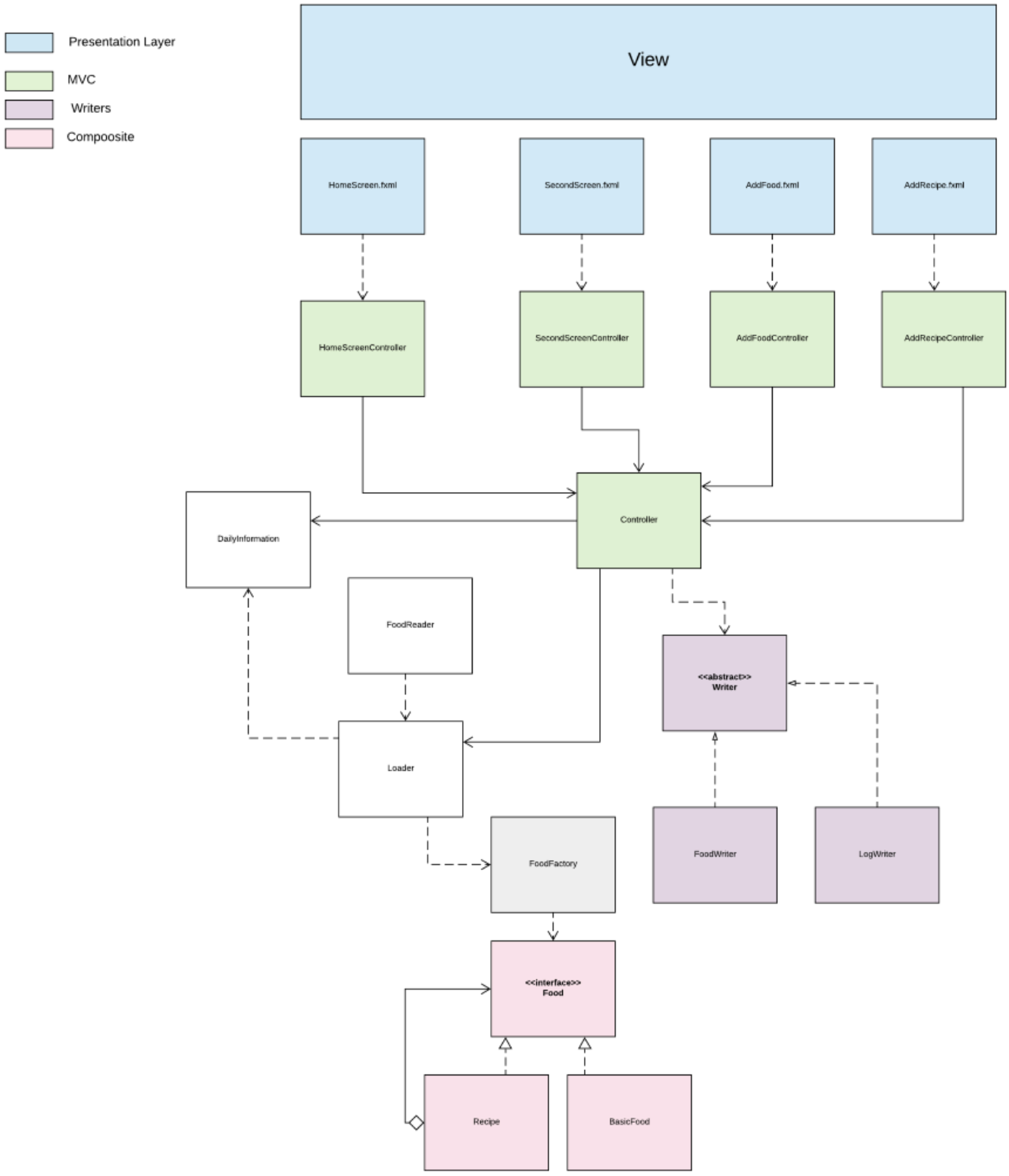
Subsystem Create

- Responsible for creating food objects for other classes that require their usage.
- Interacts with the controller

MVC Pattern UML Class

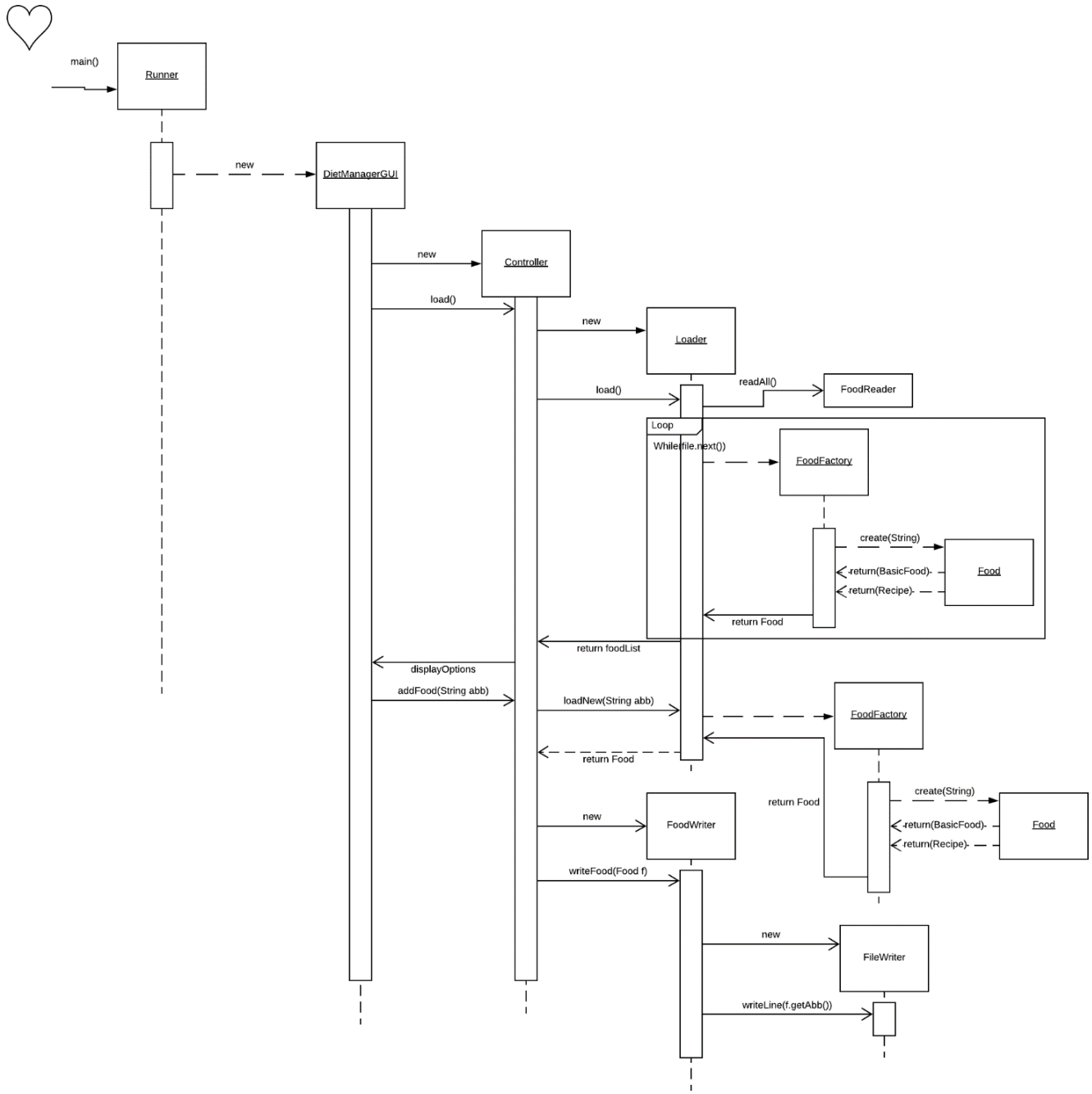

View
AddFood.fxml
AddFoodController
-TextField: caloriesTextField
-TextField: addCarbsTextField
-TextField: addProteinsTextField
-TextField: addFatTextField
-Button: addButton
-Button: backButton
-Controller: controller
-String: foodName
+ initialize(url: URL, rb: ResourceBundle): void
- isDouble(str: String): boolean
- addMyFood(event: ActionEvent): void
- back(event: ActionEvent): void
- setData(controller: Controller, _foodName String): void
Controller
+ loader: Loader
+ info: DailyInformation
+ getInfo(): DailyInformation
+ setInfo(info: DailyInformation): void
+ addFoodToStorage(abb: String): void
+ addFoodToDaily(abb: String, amount: double): void
<<abstract>>
Writer
+ write(di: DailyInformation): void
+ write(food: Food): void
+ closeStream(): void
FoodWriter
- writer: PrintWriter
- foodFile: File
- sb: StringBuilder
+ FoodWriter()
+ write(food: Food): void
+ closeStream(): void
LogWriter
- writer: PrintWriter
- logFile: File
- sb: StringBuilder
+ LogWriter()
+ write(di: DailyInformation): void
+ closeStream(): void

UML Class of the overall program

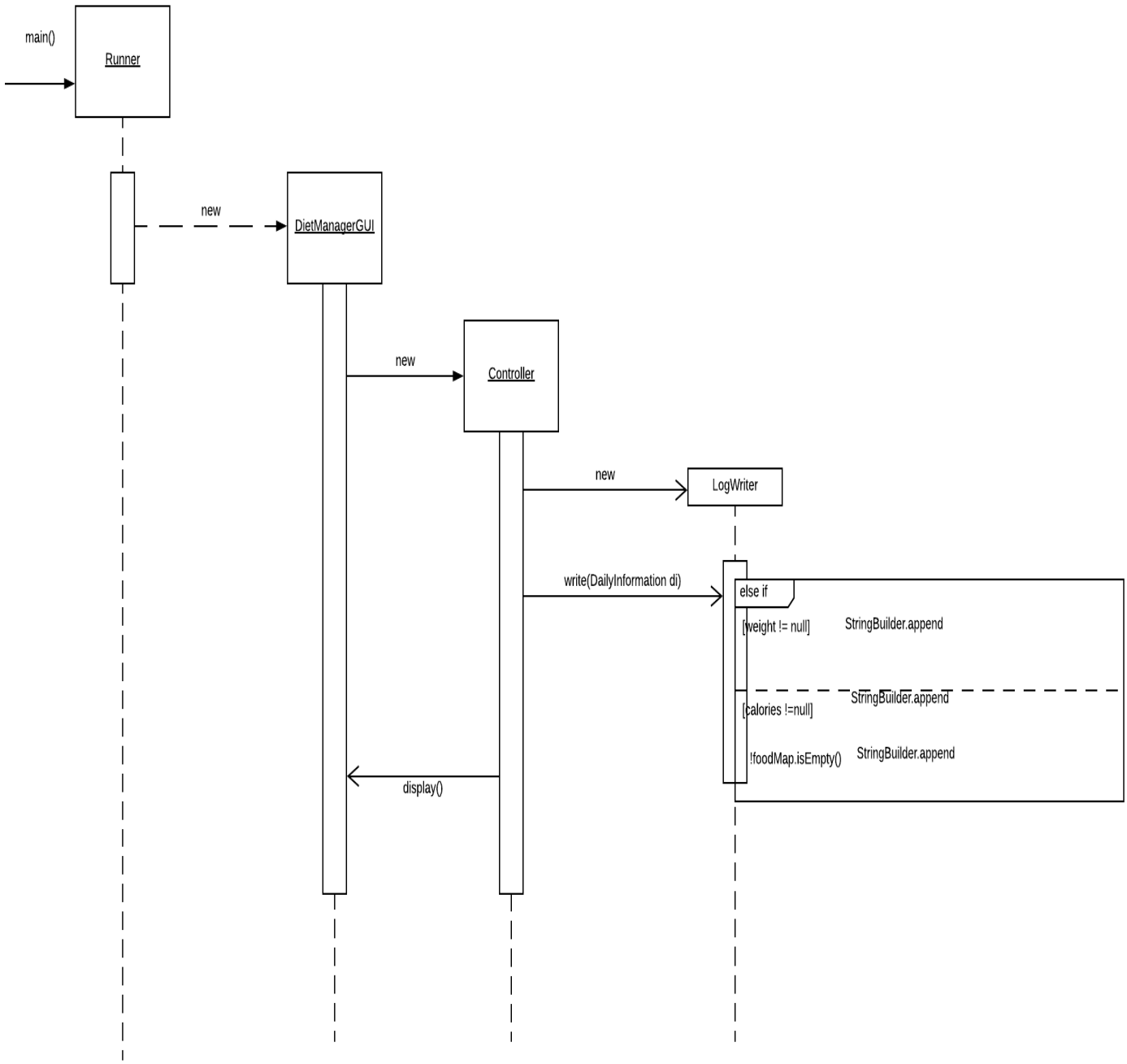


Sequence Diagrams

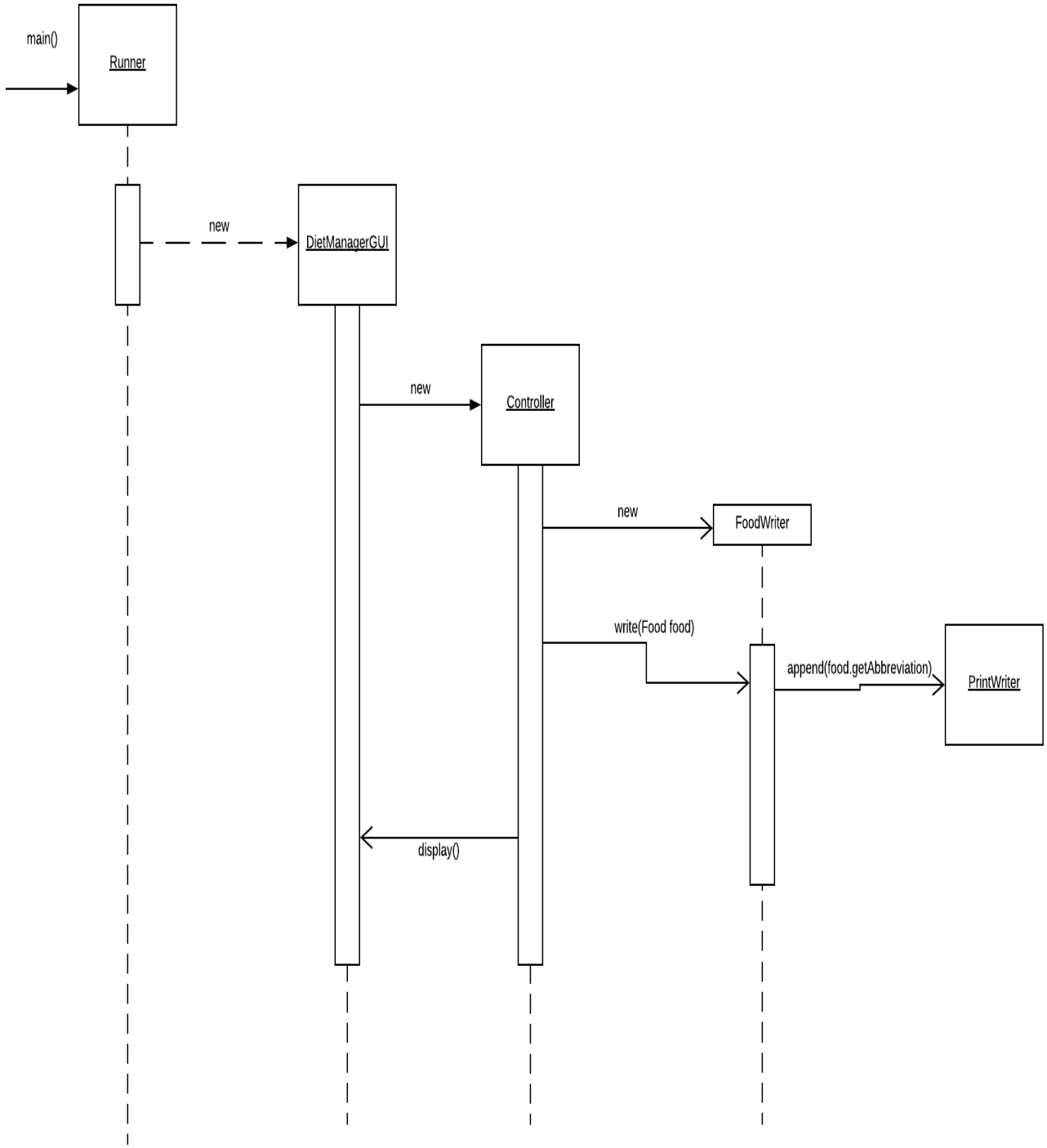
Loading and writing foods – shows MVC (DietManagerGUI stands for the VIEW - FXML)



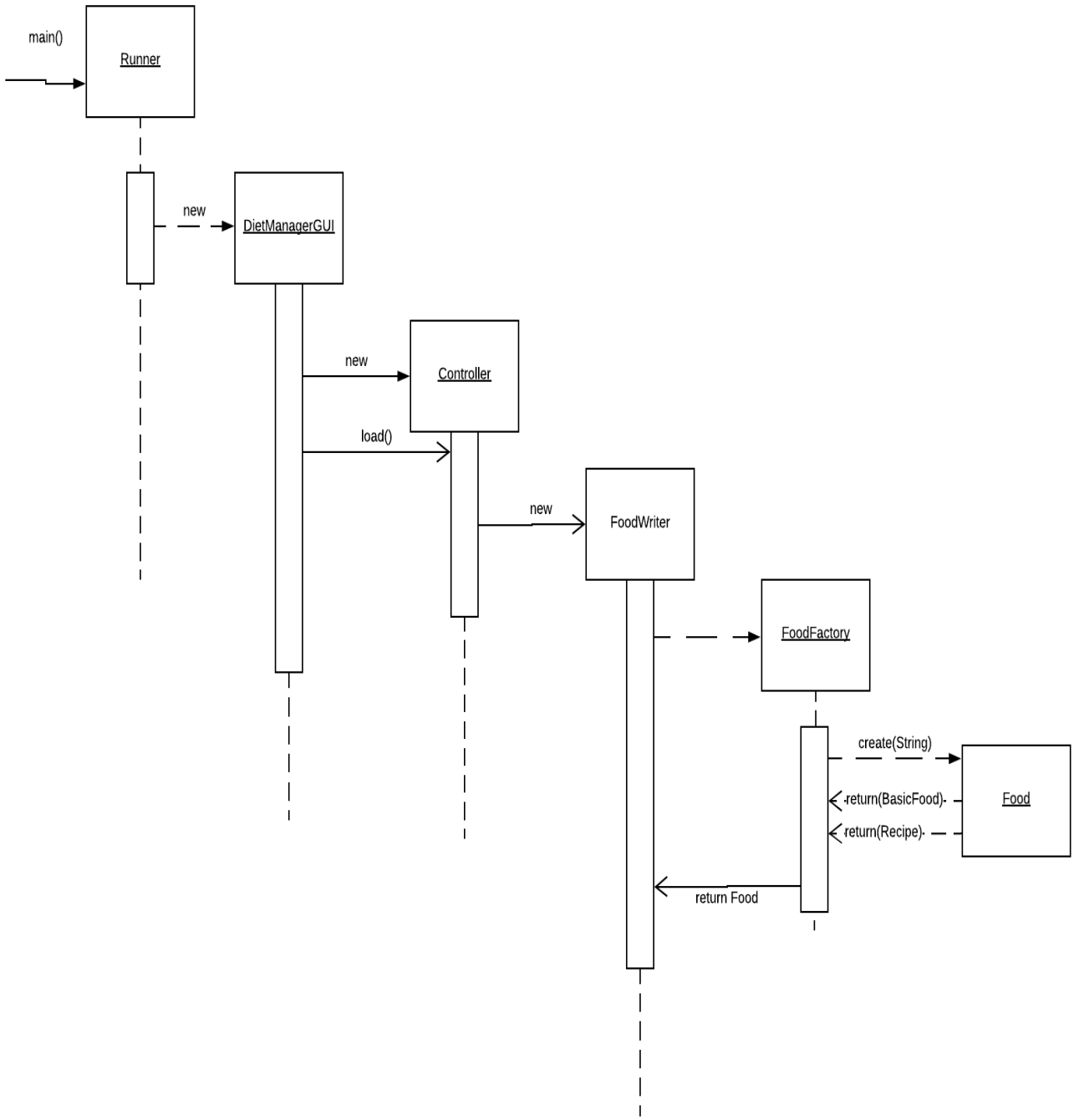
Write from LogWriter - Used for the daily intake in the log.csv which extends the Writer abstract class and appends into the .csv file via PrintWriter.



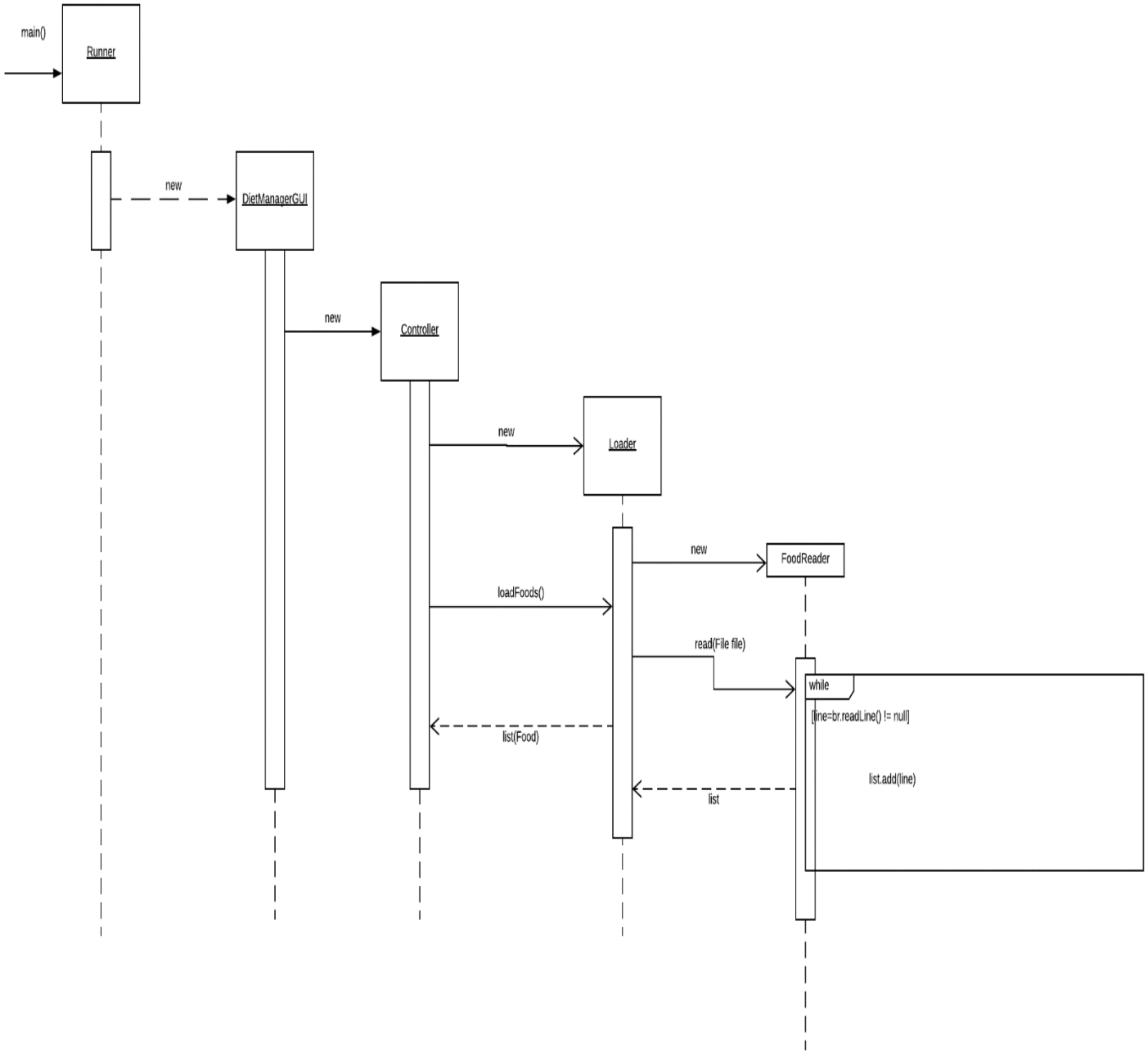
Write from FoodWriter which is used to write the Recipes or BasicFood into the recipefoods.csv . It Takes the Food and calls the abbreviation which returns a string with comma separated variables.



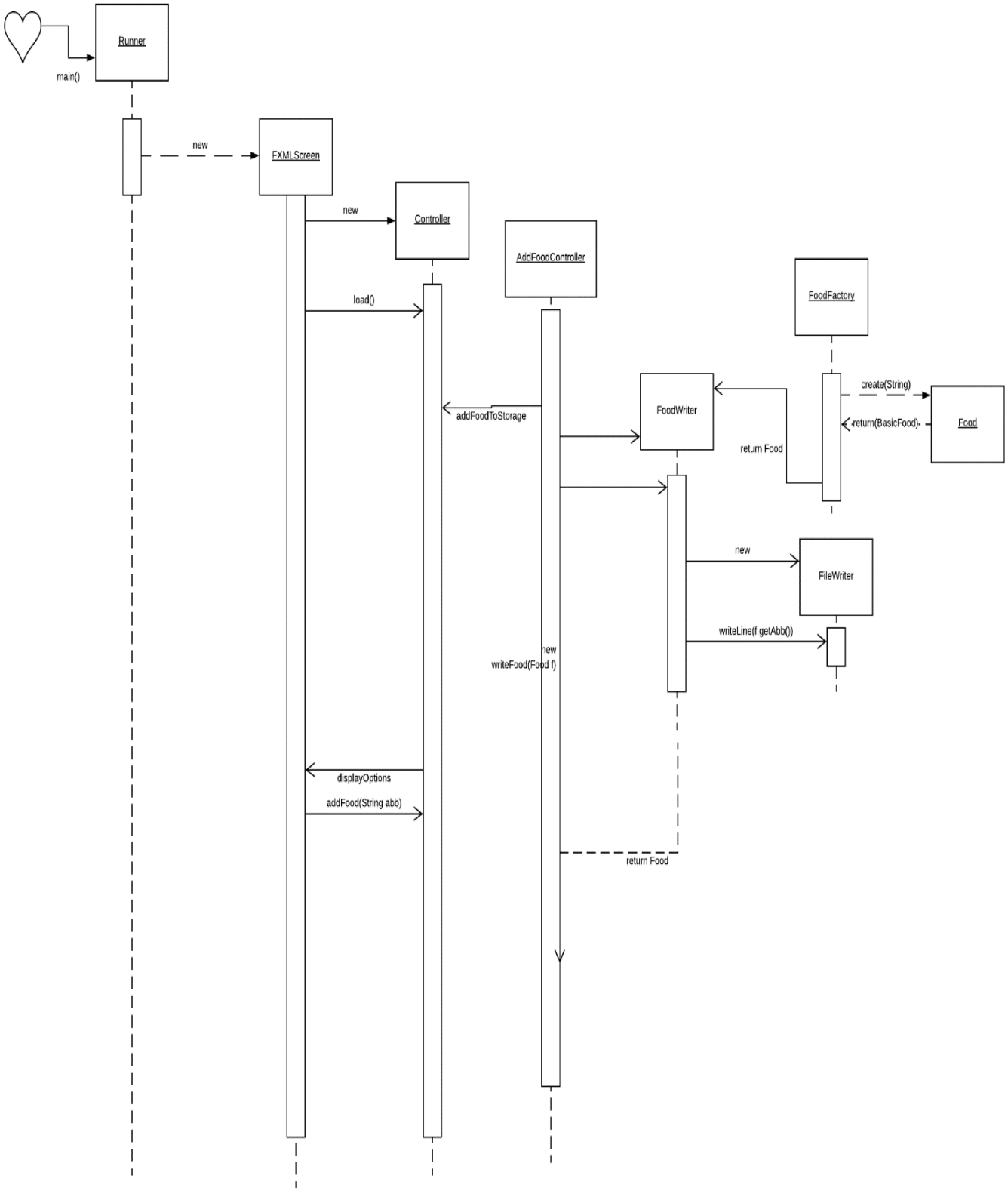
CreateFood - Used by the factory in order not to expose the concrete foods implementation.



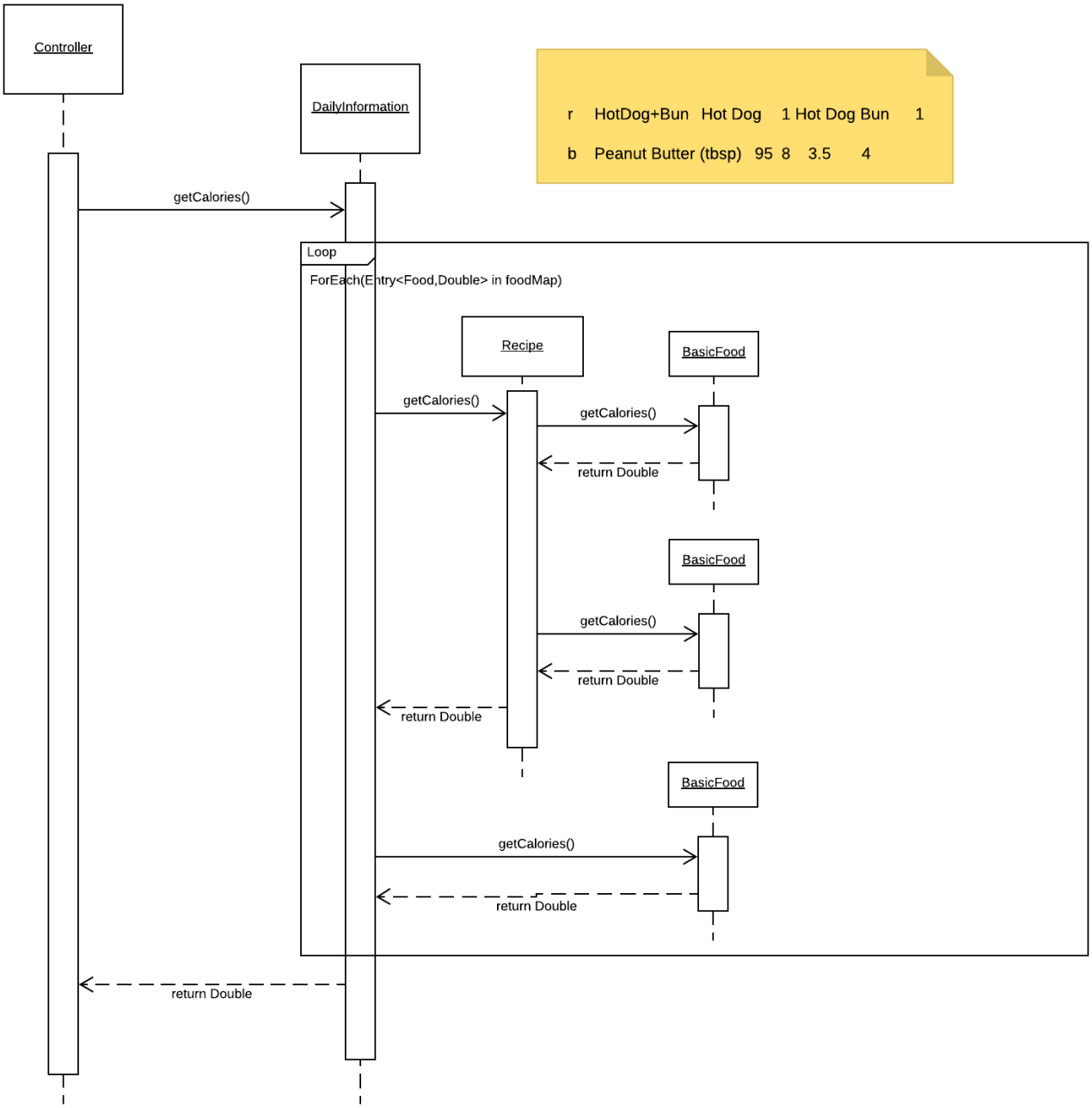
Load – Loader class interacting with FoodLoader which puts all the BasicFoods and recipes in a data structure of objects.



Sequence of MVC relationship



Getting the total amount of calories for the two specified foods in the yellow marker. Shows specific usage of the Composite pattern.



Pattern Usage

Pattern #1 The Composite Pattern

The Composite Pattern	
Component	Food
Composite	Recipe
Leaf	BasicFood

Pattern #2 The Model-View-Controller Pattern

The Model-View-Controller (MVC) Pattern	
Model	Food, FoodFactory, Recipe, BasicFood, DailyInformation, Loader…
View	FXML
Controller	Controller + FXMLControllers

Pattern #3 The Factory Pattern

Pattern	
Simple Factory	FoodFactory