DIET MANAGER

*Version 1*

Software Design Principles and Patterns Project 2

Design Document

*by Group 3, Section 800*

**Diet Managers**

Luka Krpan  
 <[lk8175@rit.edu](mailto:lk8175@rit.edu)>

Matija Brnas   
<[mb8828@rit.edu](mailto:mb8828@rit.edu)>

Amar Zagorcic   
<[az4739@rit.edu](mailto:az4739@rit.edu)>

Doroteja Krtalic   
<[dk5399@rit.edu](mailto:dk5399@rit.edu)>

Toma Jeronim Culic  
 <[tjc6227@rit.edu](mailto:tjc6227@rit.edu)>

Matea Boderištanac   
<[mb4407@rit.edu](mailto:mb4407@rit.edu)>

# Project Summary

Diet Manager application is a program designed in a way that it tracks and records the user’s information about their diet and eating habits. The program will calculate the caloric intake based on the food and the quantity that has been added by the user, while additionally also taking the data on the user’s weight to fully track the changes related to their diet.

Inside the application, the data related to the basic food and the dietary information assigned to each individual entry will be stored. This also includes the recipes and subrecipes the user will have added to the program. By doing so, the user will extend the existing data and be able to adjust it to their habits and needs.

All input related to caloric intake and weight will be tracked on a daily basis. In order to perform the necessary calculations, the users will add the food they have consumed on that particular day, along with the information on the quantity that has been consumed and enter their weight periodically so as to be able to preview the change over time. The managing aspect of this application will take into consideration not only the input related to the food and the quantity consumed each day, aka the caloric intake calculation, but will also allow the users to set their desired caloric intake and be able to keep track on whether they have adhered to the limits they have set or have gone over them.

# Design Overview

We approached this issue from the perspective of MVC architecture. By focusing on splitting the responsibilities inside of our application based on the program’s requirements and specific functionalities, we decided on an MVC model in which the View and the Model do not communicate or know of one another’s existence within the application. With such a setup, the classes have been separated accordingly into logical subsystems which adhere to the rules of the MVC architecture, and thus we created a Model, View and a Controller. In this schema, the Controller is in charge of managing the communication that needs to occur between the model and the view. By doing so, it ensures that the changes which are being requested are initiated inside the application’s system and that the new state takes hold in order to ensure accuracy of the overall performance. Meanwhile, the view is in charge of updating itself based on the information passed from the controller, and the model will deal with the CRUD operations in order to ensure that the external files which serve as the application’s storage are updated according to the information that was passed to it from the controller, which forwarded the data that the user has provided through the interface. Such a schema somewhat resembles the observer pattern, where the responsibilities are being split between the classes so as to have the logic contained inside separate sections, and thus adhering to the separation of concerns programming rule.

By maintaining the business rules and the logic in separate subsystems, the responsibilities are being focused on individual areas inside the code, which enhances the performance of the program and makes it easier to scale, maintain, expand or alter in any way as time progresses. This way, we do not only centralize the efforts of our classes to perform their dedicated tasks, but also achieve such connections between classes where the chances of having high coupling within our code is significantly decreased.

**2.1 Skeleton Code - Approach**

Firstly, we defined the classes accordingly, by splitting them into the three subsystems of an MVC model, with our Runner class, which contains the main method that starts the program, being set outside this “triangle schema”. These help us group our classes into logical units which in return deal with different tasks and contain information that is meaningful and relevant to the processes they are set to perform.

**2.2 Model**

Inside the mode, we created a DMModel class which is concerned with the events regarding the external csv files inside the application storage. This involves all CRUD operations, where the model class will act as a “file handler”. This involves reading from the files and writing in them based on the information passed from the controller. Additionally, the model class will need to save the changes that have been made in the view to the existing csv files, as the user will need to be able to add, update and remove data from the files. For this reason, the model contains classes which map or reflect the data stored inside the csv files. This results in the Log, Recipe and BasicFood classes being added to the program. These classes contain attributes which will map the values stored inside the csv files, such as “String name, double calories, double fat, double carbs, double protein” inside the BasicFood class. These attributes are all set to private, as they are not to be accessed or altered by any other class within the program. In order for us to use instances of these classes, we will use the parameterized constructor and the mutators that were added after the toString method, which format the data from the attributes in the same way as they are stored inside the csv. The same approach is applied with the Log class and will follow in the Recipe class (fully implemented in later application updates).

One issue we encountered here was inside the Log file. Because the calculations of the daily calorie intake depend on the basic food entries on a particular date and on the quantity of all of those individual entries consumed on the same date, we first stored the values inside of a LinkedHashMap<BasicFood, Double>. Here, our map stored the food items added on that particular date along with the quantity that was consumed. However, in cases where an entry was repeated, our map would not display accurate data, as Maps do not allow duplicate key values. Therefore, in case we ate one pizza slice twice in one day, where one was in the morning and the next in the evening, the entries in the log would have to be displayed as “yyyy,mm,dd,f,Pizza Slice,1.0” twice. For this reason, the current solution involves the creation of two maps and a method which is connecting the food and the quantity that has been consumed by connecting these two maps. This data is then added to a log entry of a particular date inside the model, whenever the file is being read or written into.

Moreover, the selection of data structures used was done by considering the processes and calculations that are needed in the program, which is the calculation of the daily calorie intake. This was made easier through the usage of a map which will store the basic food items and the quantity consumed on a particular date, as we will need to make fewer calls to query all the data we need in order to calculate the daily calorie intake. Because the data for these calculations is stored inside each instance of the BasicFood class, our Log contains an attribute that is a map of the food and the quantity that has been consumed. The two are then connected through the index integer and mapped together in the addFood() method before being added to the Log item.

**2.2 View**

The view subsystem contains a single class, which is the DMView itself. It contains the user interface and ensures that the user is able to interact with the program in order to make all the changes the program’s functionalities enable, as well as to display all necessary data to the user. The class extends JFrame in order to use all the features needed to create a functional interface which will display the necessary data and enable the user to add proper input. The JFrame class is an extended version of java.awt.Frame that adds support for the JFC/Swing component architecture (Oracle).

The general overview of the created window is the clear separation in two sections. To the left, we have created input fields which enable food selection from the loaded foods.csv file which was sent from the model, below which we provide text fields and input fields that further specify the daily data. This includes a specific date, quantity of the selected food that has been consumed on that particular day, the setup of a daily calorie limit (which is automatically set to two thousand in case the user fails to provide input), and weight (which is automatically set to sixtyeight in case the user fails to provide input). Once all the information has been collected, the “Add” button displays the activity that has just occurred inside the text area that shows the log and saves the data into the external csv files. The data that was provided by the user is then processed and a calculation is returned, which informs the user of their daily calorie intake based on the food that has been consumed thus far on that day and shows if the consumed calories have surpassed the set calorie limit or not, and by how much. Upon the button-click action, the input fields are checked to ensure none are left empty.

To the right, the window takes input related to food entries. These input fields allow the user to create new food entries by providing the new item’s name, and the amount of calories, carbohydrates, proteins and fat it contains. Once all fields have been filled, the entry is saved to the csv file on the press of the “Save” button at the bottom of that column. Before being saved, the button triggers a check of all fields in order to ensure that all have been filled and that the provided name is unique, since the foods.csv file must not contain duplicate entries.

One important element here is the DatePicker. In addition to being used for the sake of enabling the user to select specific dates, it is essential for the program to log the activities in an ordered and organized way, which clearly separates the inputs based on the date. This means that the calorie intake calculations are performed on each specific date, without mixing in other entries which would hinder the correctness of displayed data. Hence, the date is being used as the key within the Map that is used for the calorie calculations and logging.

**2.3 Controller**

The desired performance and functionality of our MVC model is for the controller to act as a “bridge” between the model and the view. The Controller class handles user interaction with listeners. It ensures that the Model and the View are called when needed. Therefore, the controller subsystem consist of ActionListeners which would navigate between the subsystems and process the data, thus adhering to the separation of concerns, DRY, high cohesion, and low coupling rules, and maintain a hierarchy which ensures maintainability, sustainability, consistency, extendibility and scalability. The subsystem currently contains the following inner classes located inside the DMController: DateListener, ChangeListener, SaveFoodListener. All of these classes implement the ActionListener interface in order to process the registered action events and catch all changes taking place for as long as the user is interacting with the UI.

Inside of these classes are methods which all return void and where each takes in an ActionEvent as its parameter, upon which it then calls different methods to be executed within the model and/or the view. Through these listeners, communication is established between the model and the view, which ensures the data flow and that all application functionalities work as expected. These inner classes are to be extracted from the DMController and each be given its own file within the “Listeners” package. This way, the overall code composition is much more readable and easier to maintain, while still sustaining the same functionality and performance.

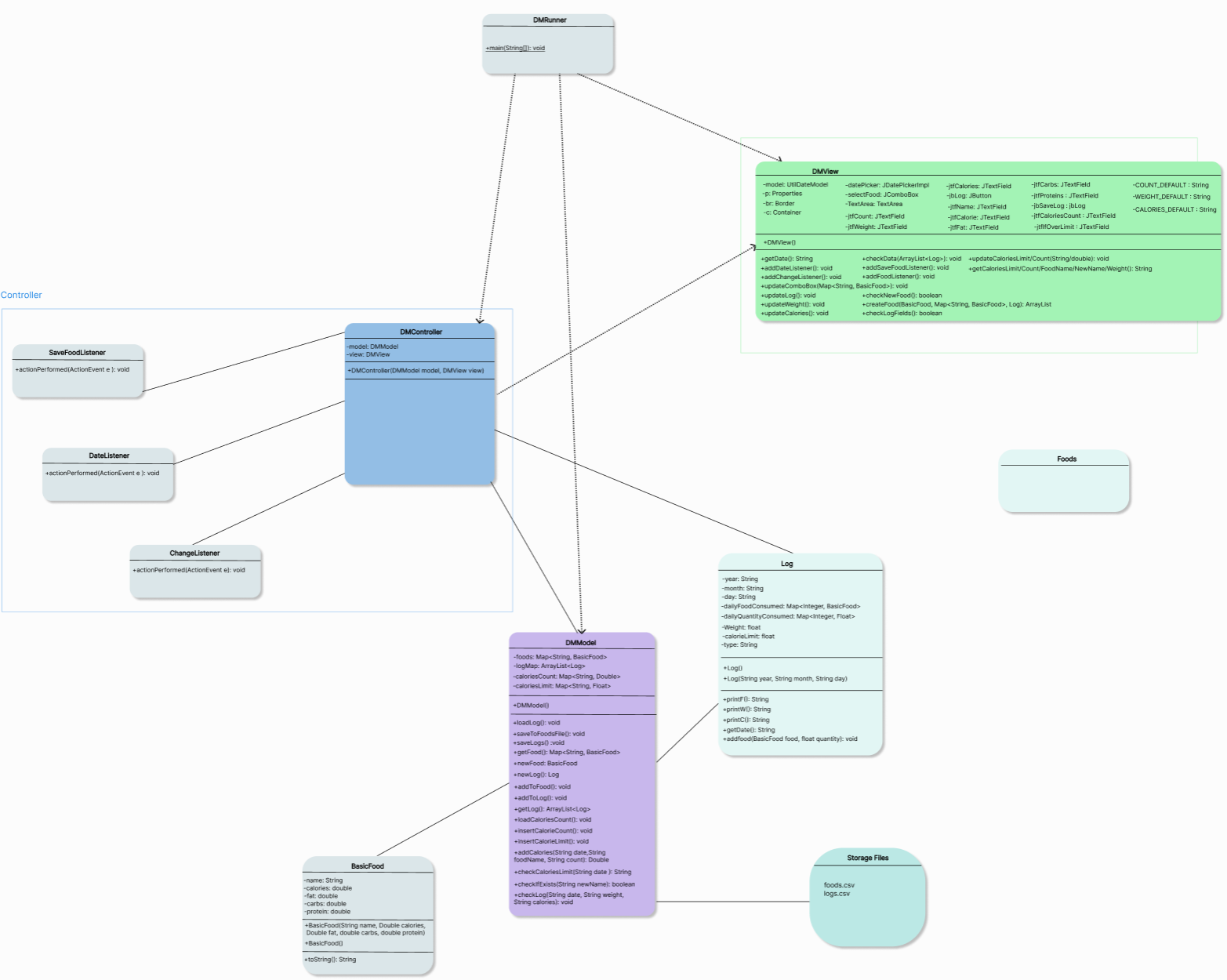
**2.4 Architecture Improvements and Refactoring**

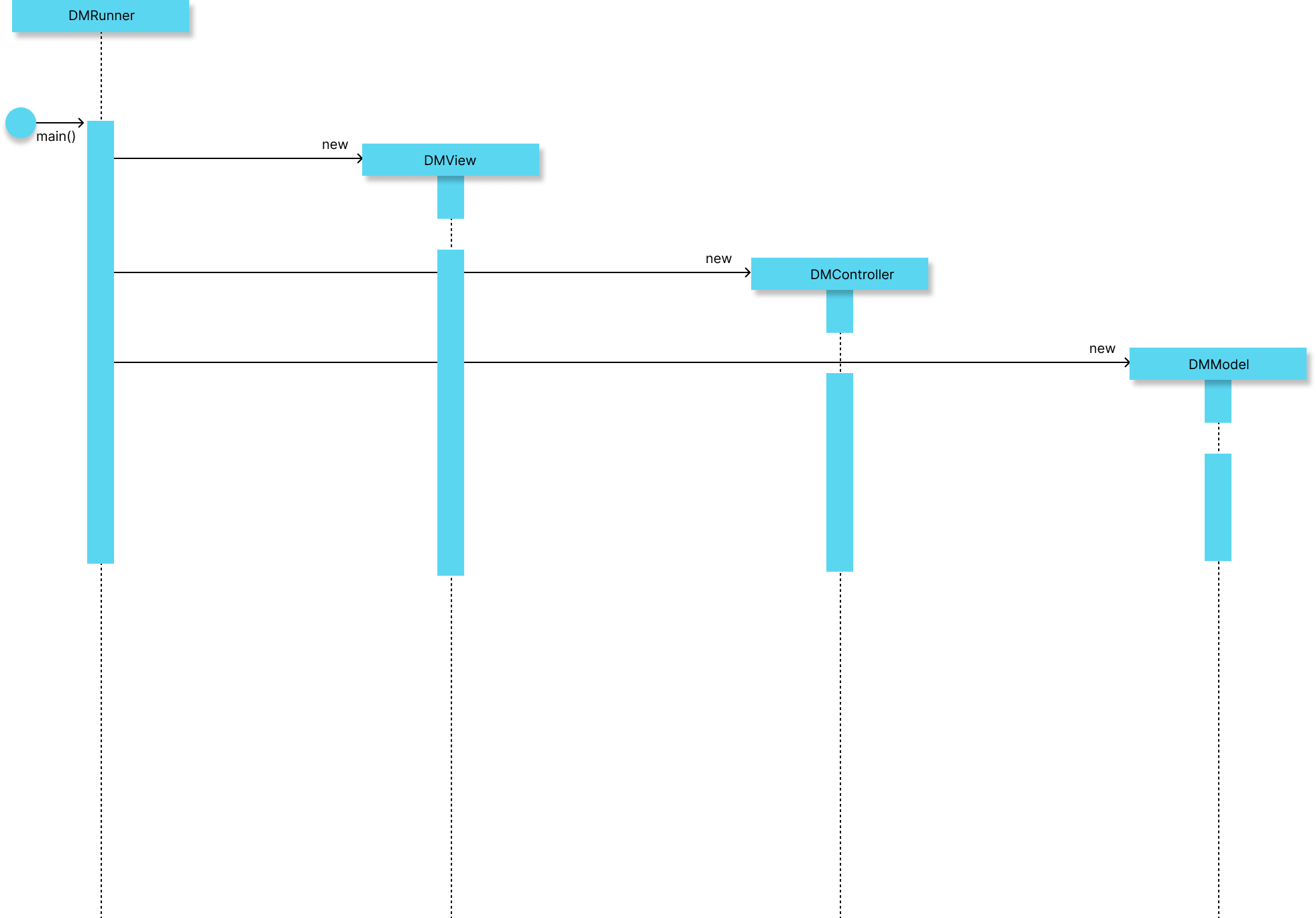
Given the current running program, the MVC architecture can be further improved and new patterns can be implemented which would enhance the performance of the program. In particular, the composite can be introduced to manage the addition of recipes to the application, in a way that an abstract class “Food” will be introduced. Here, BasicFood and Recipe classes will extend this newly introduced Food class, due to the fact that both require the data from the foods.csv. Additionally, the factory pattern can be implemented here, since we will choose which item is to be instantiated based on the type element inside each stored string in the external csv files. Meaning, in case the char that has been read indicates a basic food object, aka the char that was read was the letter “b”, the program shall proceed to instantiate an object of BasicFood class. Otherwise, the char will be the letter “r” which stands for “recipe”, in which case an instance of Recipe will be created.

Moreover, as mentioned previously, the ActionListeners ought to be placed in their own dedicated files and treated as separate classes, rather than being inner classes inside the controller. Following this logic, the handling aspect of the code which deals with the files could also be separated so as to simplify the current code. Because we adhered to the DRY principle, the methods include “switch case” and “if” statements in order to make a distinction between the external files and separate classes. This can be simplified by separating the handling aspect to LogHandler and FoodHandler, where the LogHandler could be further divided into FoodLogHandler and CaloriesLogHandler for easier management.

1. **Overall System Structure**

* (UML Class Diagram)
  + The architecture of the program follows the MVC pattern. This separates the concerts and functionalities of the program into three subsystems: Model, View and Controller. Here, the model is concerned with the CRUD operations, which ensures reading from and saving to the external files that act as the application’s storage. The view must update itself in order to display accurate data, as well as receive input from the user. The controller is the bridge that connects the two and enables communication and data flow. It contains inner Listener classes that have methods which trigger these calls and updates between the model and the view. Inside the model subsystem are the classes that reflect the data stored inside the csv files and allow creation of such objects, which accurately reflect the data inside the stored files - BasicFood and Log classes. Finally, the Runner class is the one that contains the main method and starts the application. The constructors for all three classes representing the model, view and controller are called inside the runner for the application to function appropriately.
  + This application requires the use of external storage files. These two csv files are necessary for the program to run and provide all functionalities the user requires. Because of the existence of this kind of storage and the necessity for a user interface the clients will interact with in order to interact with the program, we decided to follow the MVC architecture. Here, by separating the overall functionality into logical subsystems that have been predefined in the definition of the MVC architectural pattern, our code will be balanced and sustainable.   
    Thanks to such architecture, the code will also adhere to the separation of concerns, high cohesion, low coupling and DRY principles and patterns. The result will be a functional program which is easier to maintain, extend, scale and/or modify further in order to implement other potentially necessary patterns and design principles as our program develops further. Additionally, this approach ensures that the correctness and performance of our application are optimized, while also maintaining code which doesn’t require high complexity (Big-O is kept within the desired range).



* (UML Sequence Diagram)
  + The Runner class starts the program and creates an instance of the DMModel, DMView and DMController. The external storage files are first loaded into the application, which is the task of the model class that deals with the CRUD operations. Essentially, at the very start of the program’s runtime, the DMModel reads the data from the csv files in order for them to be passed to the view and available to the user. Furthermore, the view is responsible for creating the user interface, aka the GUI which displays and uses the loaded data and provides input fields which will take the user’s data. The changes being made inside the GUI are registered by the inner listener classes which exist within the controller.   
    These classes extend the ActionListener interface, which enables them to process the registered action events and cause the program to react accordingly. The examples provided in sequence diagrams show the interaction between these subsystems as it takes place inside the program for three separate actions, as well as the general behavior within each subsystem. The input is checked and forwarded from the view to the model through the controller, which then saves the data if the validation has been satisfied. The input that concerns itself with calorie calculation and logging is monitored by the controller with its action listener classes, and the calculation is performed by the system in order to display the desired information in the GUI in the view.
  + 

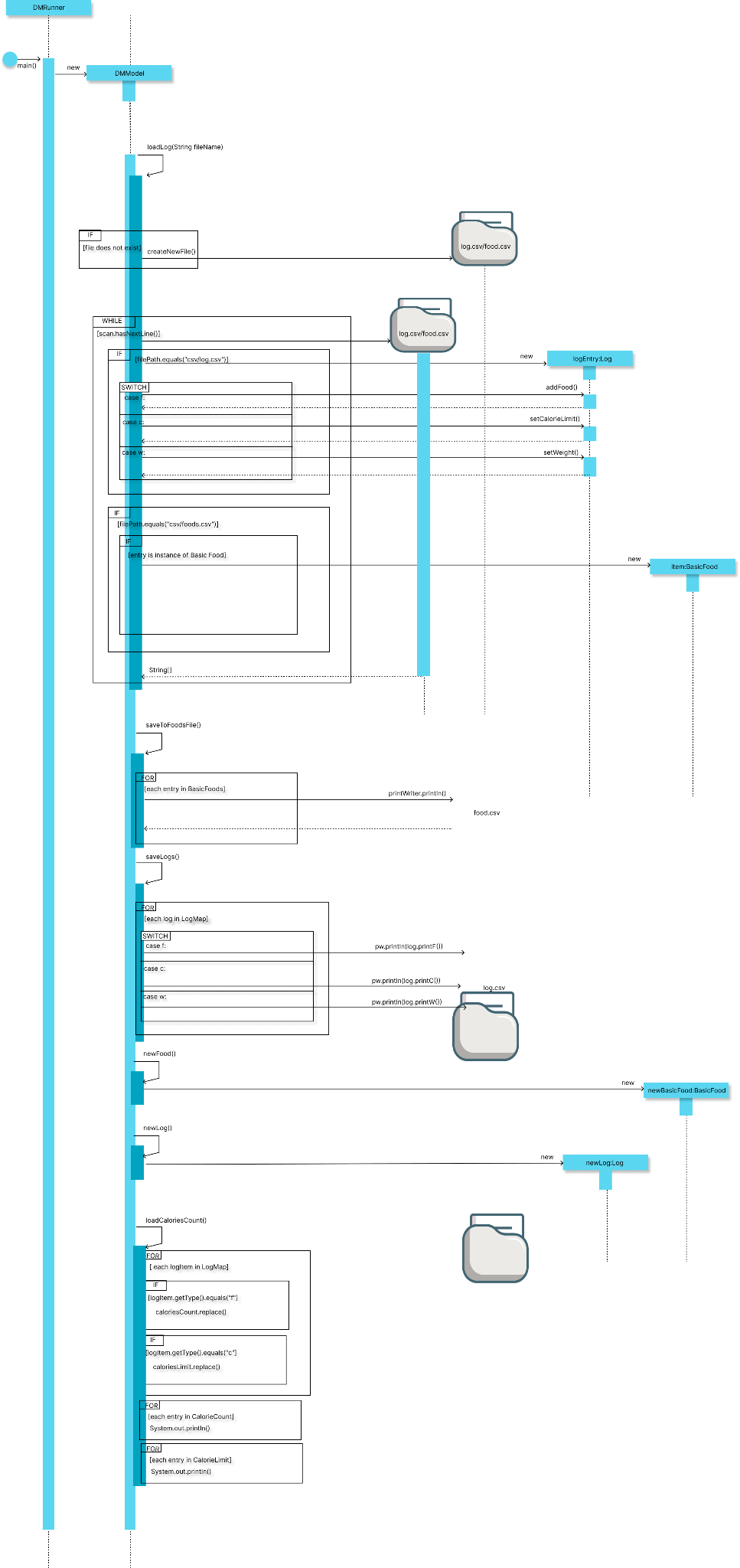
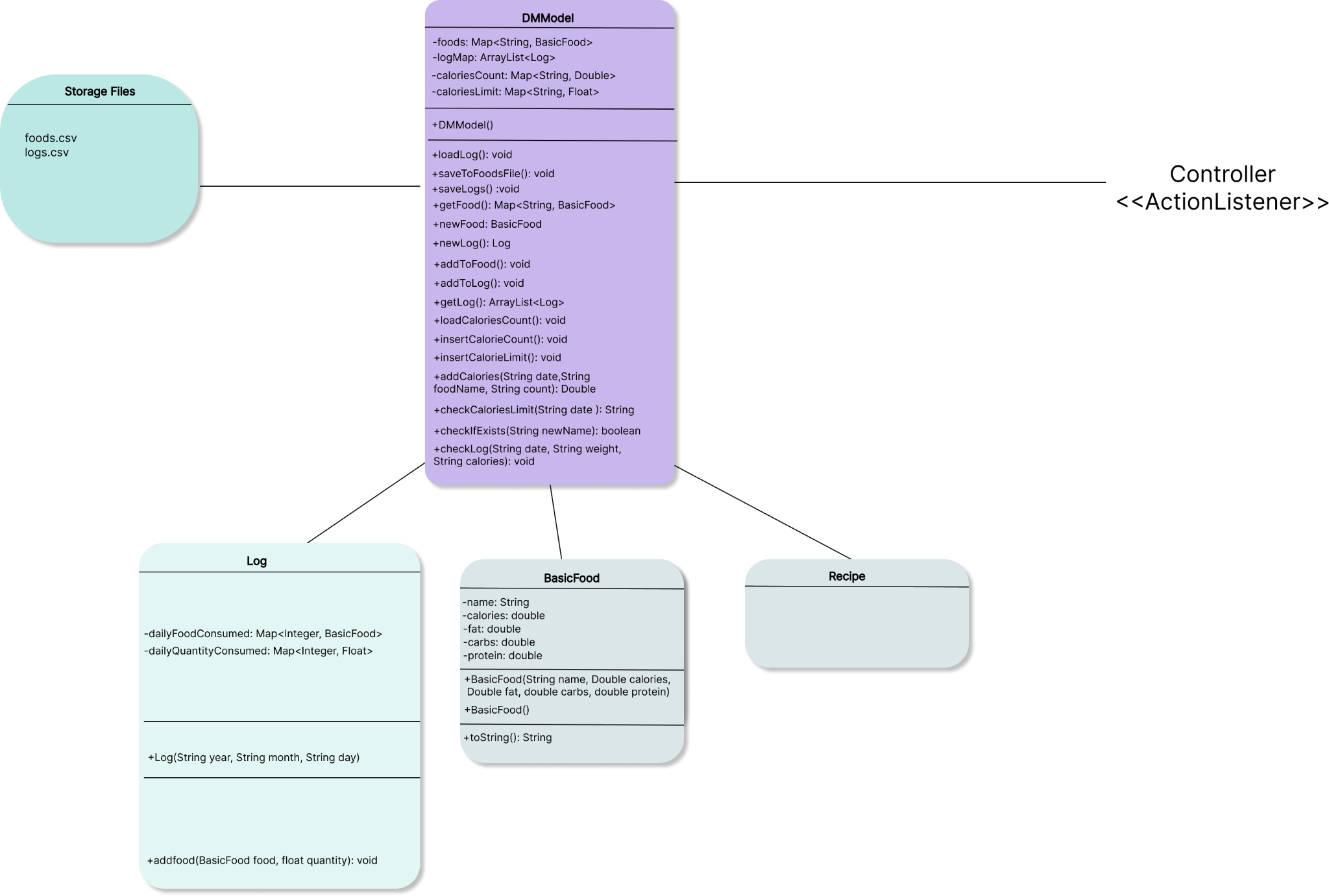
# Subsystems

As stated previously, the three subsystems included in this application are model, Controller and View. They are each concerned with a separate aspect of the application, as defined below.

## **4.1 Subsystem Model**

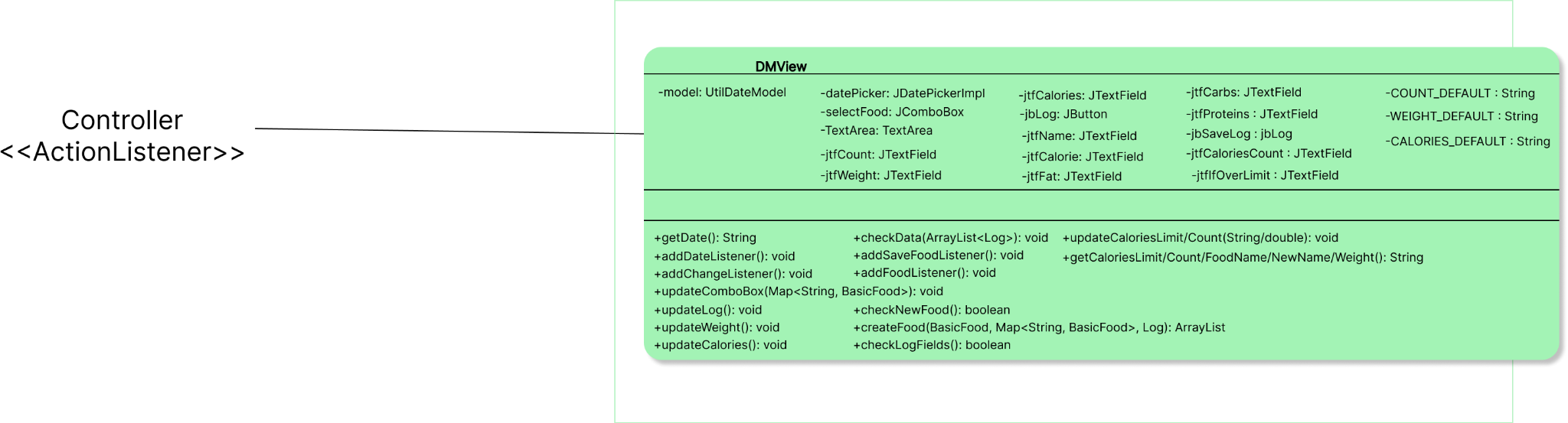
The model subsystem is represented by the DMModel class. It deals with the CRUD operations through its methods that read and write from and to external csv files inside the application’s storage. The model subsystem therefore includes additional classes which serve as blueprints that reflect the data stored in each csv file, in order for the program to be able to manage and manipulate the data through Java objects. The DRY principle, high cohesion, low coupling and the separation of concerns are all reflected here, as the methods deal with the data processing in a way that does not require code repetition or creation of duplicate objects, and the classes used by the model are private so as to not be used by any other class inside the program.

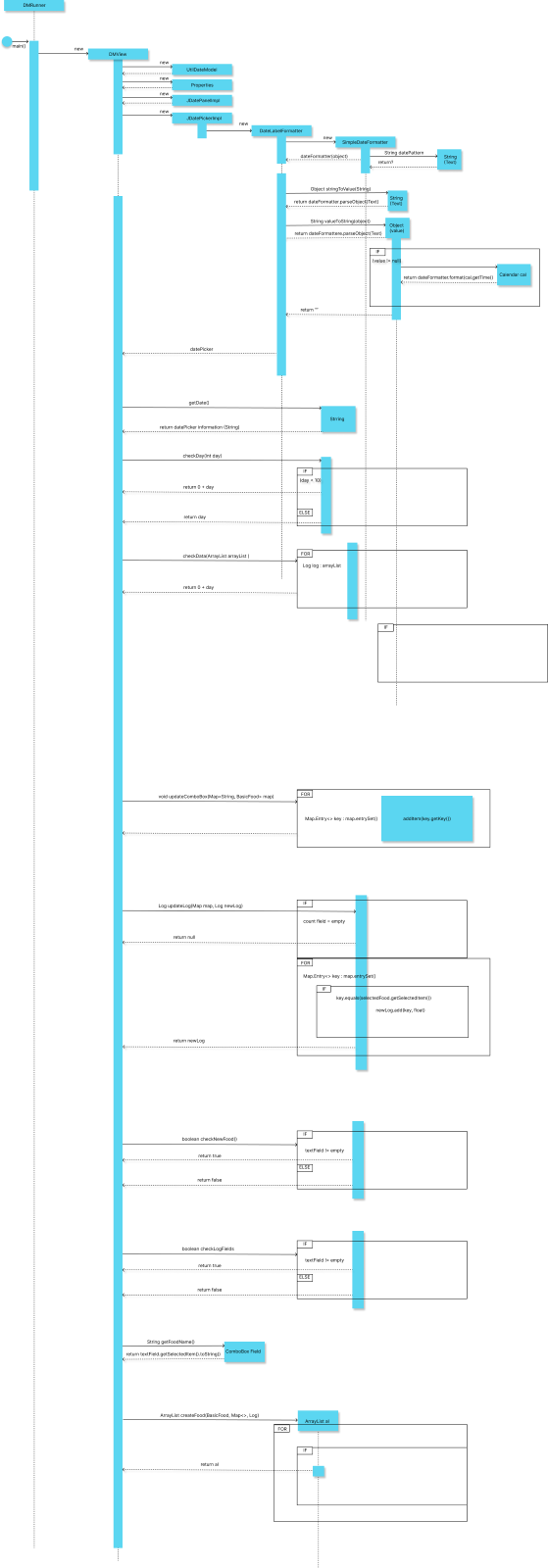
This class interacts with the controller subsystem.



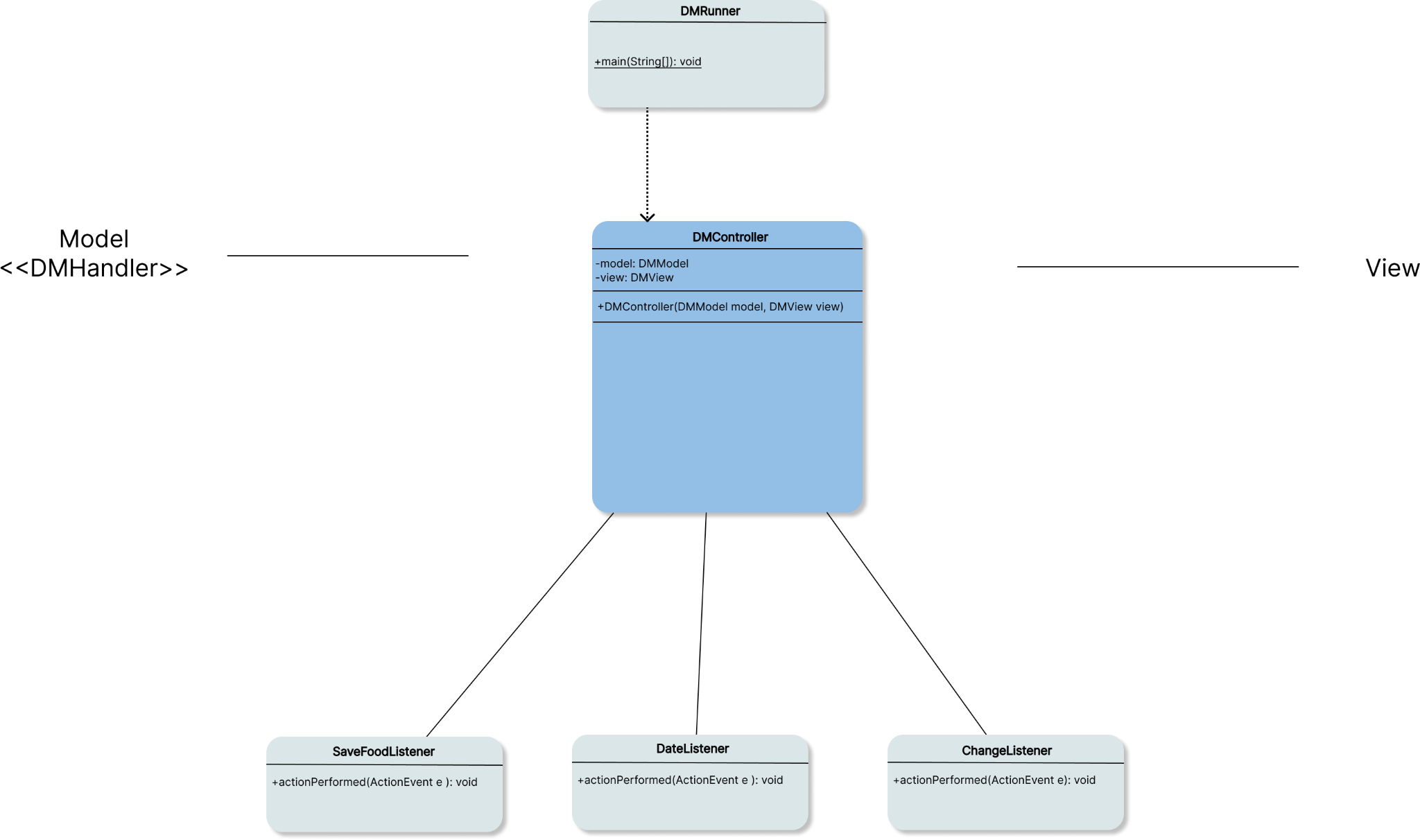
## **4.2 Subsystem View**

The view subsystem consists of the DMView class which is responsible for the creation of the GUI. Here, the user can both preview and alter the application’s data by interacting with the user interface displayed. IIn order to do this, the data needs to be loaded from the model system and passed through the controller system. Same as the model subsystem, the view is connected to the controller subsystem which then registers the actions being performed in the UI. One important aspect is the date picker, which is used as the key in order to perform the calorie intake calculations.





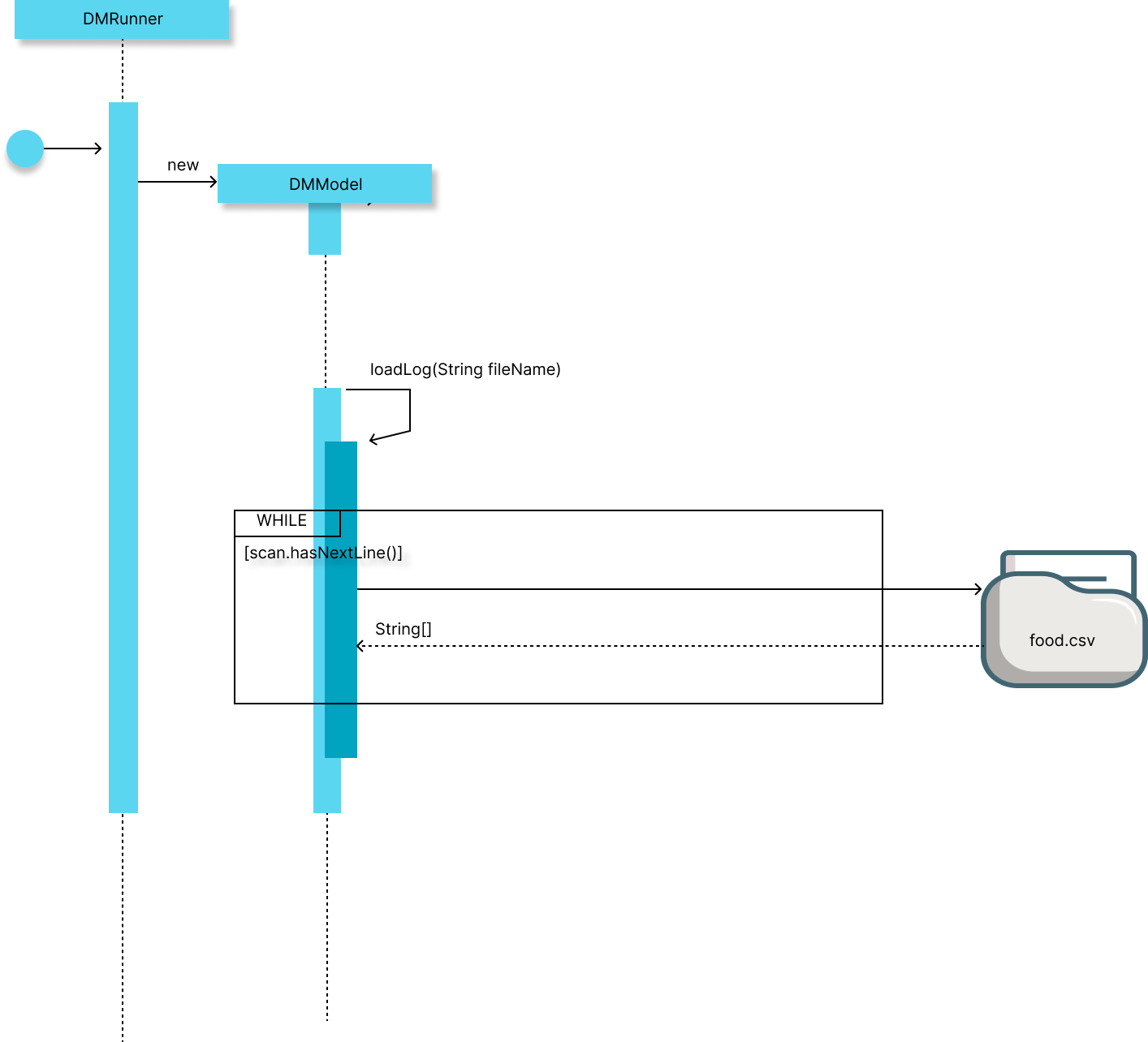
**4.3 Subsystem Controller**

The controller subsystem consists of the DMController class which contains inner listener classes that implement the ActionListener interface. This allows the controller to monitor the actions and changes that are occurring during the user’s interaction with the application and trigger an appropriate response from the program. In return, the result is an accurate display and management of the application’s data, as well as smooth program performance with low complexity.

# Sequence Diagrams

## **Sequence Diagram 1 (reading and loading data from an external file / read in a food database / running the program and reading the database).**

The Runner starts the program and creates objects of all MVC elements - model, view and controller. The model reads from the files and the read data is sent to the view. This enables the user to interact with the application and use the loaded data.



## **Sequence Diagram 2 (MVC data processing / adding servings of a basic food to the log entry for the current date / running the program and displaying interaction between the subsystems to perform the given operation).**

The Runner starts the program and creates objects of all MVC elements - model, view and controller. The model reads from the files and the read data is sent to the view. This enables the user to interact with the application and use the loaded data. The user then adds input in the GUI which is then processed and sent to be stored inside the application storage through the model’s CRUD method.

**Sequence Diagram 3 (MVC data processing / compute total number of calories for the current date / scenario the diagram shows).**

The Runner starts the program and creates objects of all MVC elements - model, view and controller. The model reads from the files and the read data is sent to the view. This enables the user to interact with the application and use the loaded data. The user then adds input in the GUI which is then processed and the calculation is performed by the application in order to display the desired data to the user based on the input they provided.

