DIET MANAGER

*Version 2*

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, eating habits and physical activity. The program will calculate the caloric intake based on the food and the quantity that has been added by the user, while also taking the data on the user’s weight, as well as other dietary information regarding the food (protein, fat etc.) into account when performing calculations in order 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. Additionally, while the program is tracking the caloric intake and the detailed information of the dietary information regarding the consumed food and meals, the users will also be able to log the activities they perform on a daily basis. These exercises will be added to the program by the user selecting the exercise they desire and specifying the time they spent performing it. Here, the so-called “burning equation” kicks in and calculates the amount of calories burned during that specific exercise, on top of which it will also display the net calories, aka the amount of calories the user has burnt subtracted from the calories the user has consumed.

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 goeal they have set. The same approach will be used for both basic food items and recipes. To provide an example, the caloric value will be accounted for regardless of the type of food that has been given, e.g. “chocolate” as a BasicFood item, and “chocolate cake” as a Recipe item. The data stored under a specific date will be returned and shown in the UI whenever that particular date is selected, which will allow the user to look back and analyze their past logged activity.

# 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; they have been grouped into three logical subsystems where each adheres to the rules of the MVC architecture, and thus we created a Model, View and a Controller subsystem. In this schema, the Controller is in charge of managing the communication that needs to occur between the model and the view for our program to run smoothly. 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 correctness 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. Here, we introduced a separate “io” package that the model uses in order to manipulate the external files by calling the specific handler that is located inside that folder. 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 subsystems 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 model subsystem, which is found inside the “model” package, we created a DMModel class that is concerned with the events regarding the external csv files inside the application storage. This involves all changes that need to occur inside the storage files, as well as getting the files to be sent to the view. The class utilizes the methods from FoodHandler and LogHandler classes which are focused on each separate file so as to adhere to the SRP and DRY principles. This involves reading from the files and writing in them based on the information passed from the controller. Additionally, the model 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, Exercise, Food, 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 formats the data from the attributes in the same way as they are stored inside the csv. The same approach is applied with the Log, Exercise and Recipe classes. The Food class is an abstract class that, together with BasicFood and Recipe classes, makes for a tree structure of the composite pattern. Here, we will end up dealing with a collection of all Foods, meaning both Recipe and BasicFood objects stored inside of a single collection simply as foods, which is inside the DMModel class as an ArrayList<Food>. The BasicFood represents a leaf, while the Recipe represents a composite that consists of two or more BasicFood objects. Inside the DMModel class are collections of all the objects necessary for the program to manipulate all three external files based on user input.

One particular 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.

Finally, when implementing the necessary functionalities regarding the exercises, the burning equation required that the input regarding the duration of the activity was to be converted between minutes and hours per requirement. Ultimately, we have set the UI to ask for input to be in minutes, after which the program calculates the calories burned and stores the logged entry in hours instead.

**2.2 View**

The view subsystem contains two classes: the DMView itself and DateLabelFormatter class that extends the AbstractFormatter. The DMView contains the user interface and ensures interaction between the user and the program and displays 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). Meanwhile, the additional class is used to format the date provided by the DatePicker used in the view class.

The general overview of the created window is the clear separation of 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. Finally, we implemented two pop-up windows that each hold the sections that provide the input fields necessary to accept the data to create new exercise entries, and food and recipe entries that will be added to the csv file inside the application storage. One of which is the list of the existing foods in the storage that are to become ingredients in the newly created recipe. These input fields allow the user to create new entries by providing all the necessary information for the program to store the item correctly. For instance, when adding a new food entry, the user is prompted to provide 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 contains the following external listener classes located inside the controller package: AddExerciseListener, AddFoodListener, DateListener, ExerciseListener, LogListener, NewFoodEntryListener, QuantityIngredientListener, QuantityListener, RecipeNameListener, RemoveExerciseLogListener, SaveExerciseListener, SaveNewFoodListener, IngredientListener, SaveRecipeListener. 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, where the DMController class acts as the center class of this subsystem.

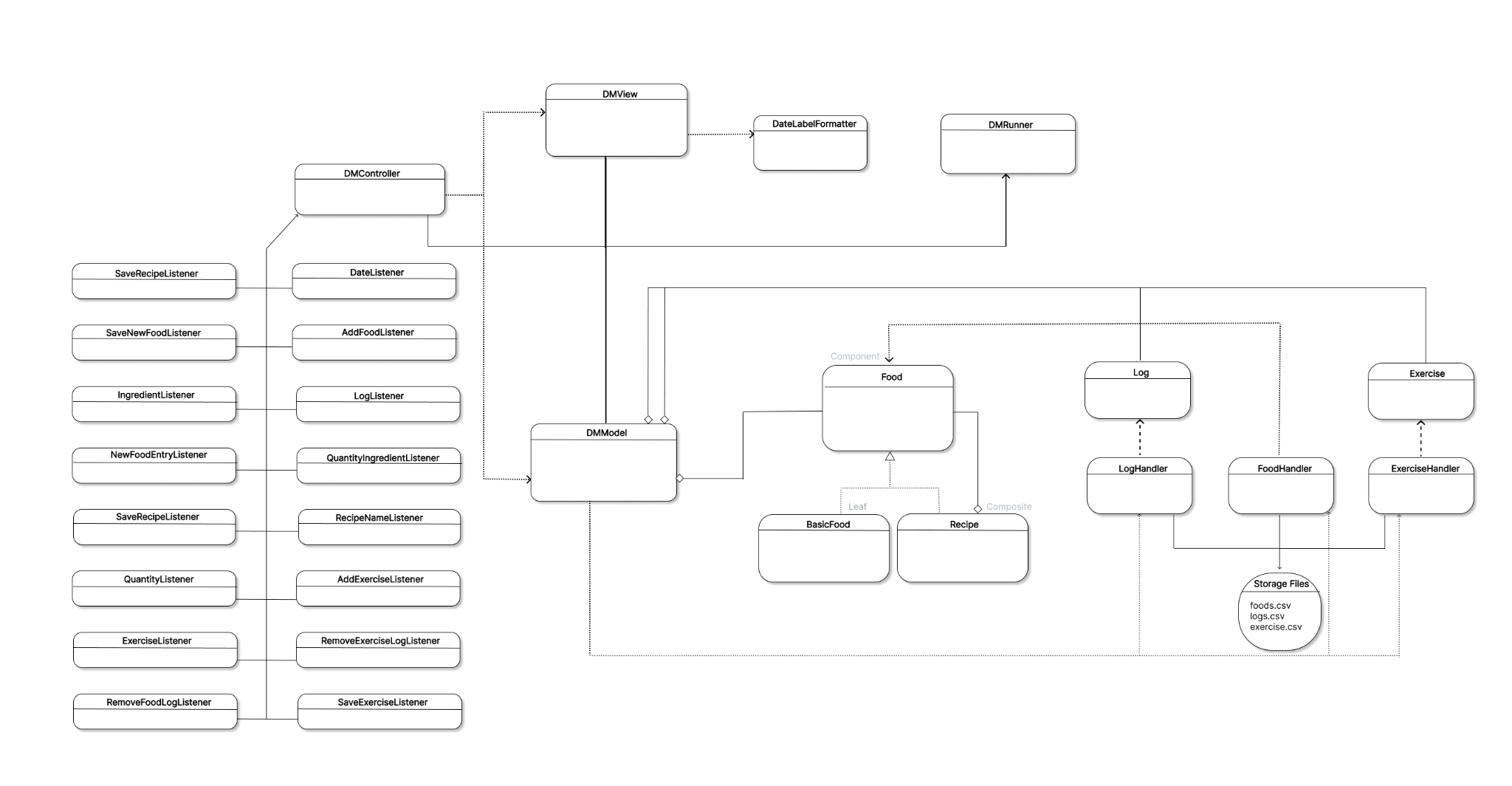
Inside of these classes are methods which all return void and where each takes in an ActionEvent as its parameter, upon which it then calls different methods to be executed within the model and/or the view. Through these listeners, communication is established between the model and the view, which ensures the data flow and that all application functionalities work as expected.

**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 our separation of handlers, the collections of Logs and Foods are formed and contained inside the DMModel class. While this follows the DRY and SRP principle and contributes to low coupling, the architecture of the model subsystem could be further improved by introducing collections as classes. This would warrant only one FileHandler that would only use a reader and a writer to perform the final actions in regards to the stored files, aka read from the csv files or write new data in them and would simplify our application architecture. In such a case, we would add “Logs.java”, “Exercise.java”, “Foods.java” while introducing the necessary alterations to the existing structure.

1. **Overall System Structure**

* (UML Class Diagram)
  + The architecture of the program follows the MVC pattern. This separates the concerns and functionalities of the program into three subsystems: Model, View and Controller. Here, the model is concerned with manipulation of the external files in the application. In order to successfully achieve its tasks, it uses external Handlers which are located in the “io” package. These handlers adhere to separation of concerns, as each file has its own dedicated and specialized Handler which is concerned with the CRUD operations that ensure reading from and saving to the external files that act as the application’s storage. This allows for logical separation of code that results in simplified “central” model class, in this case the DMModel class, which contains the collections of items that reflect the data inside the external files. 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 Listener classes that have methods which trigger these calls and updates between the model and the view. As stated previously, 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, those being - Exercise, BasicFood, Recipe and Log classes. The composite pattern here allows us to create a single collection of Foods which entails both basic food and recipe objects, and this was done by adding an abstract Food class that plays the role of a component in the implementation of the composite pattern. Meanwhile, as can be previewed in the Model subsystem diagram, the DMModel class contains an array list of foods, logs, and exercises (reflecting the foods.csv, log.csv, and exercise.csv files). It then returns them and performs the required operations which will allow the files to be altered as per the user’s needs. Finally, the Runner class is the one that contains the main method and starts the application, while the handlers inside the “io” package deal with the CRUD operations as necessary. 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 csv files are necessary for the program to run and provide all functionalities the user requires. If they do not already exist, all must be created by the program when started. By following the MVC structure, we separated the overall functionality into logical subsystems that have been predefined in the definition of the MVC architectural pattern, and achieved that our code is well-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. Additionally, the composite pattern was implemented in order to enable easier manipulation and modification of the classes inside the model which reflect 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 continues to develop. 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).



* Program Overview
  + 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 dependent on the handlers that deal with the CRUD operations (FoodHandler and LogHandler), in addition to the DMModel class. Essentially, at the very start of the program’s runtime, the handlers read 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 listener classes that are part of the controller subsystem.  
    These classes extend the ActionListener interface, which enables them to process the registered action events and cause the program to react accordingly. The examples show the interaction between these subsystems as it takes place inside the program, 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. The view updates itself through the update methods defined within its dedicated class, upon being notified and/or triggered by the controller’s action listeners.

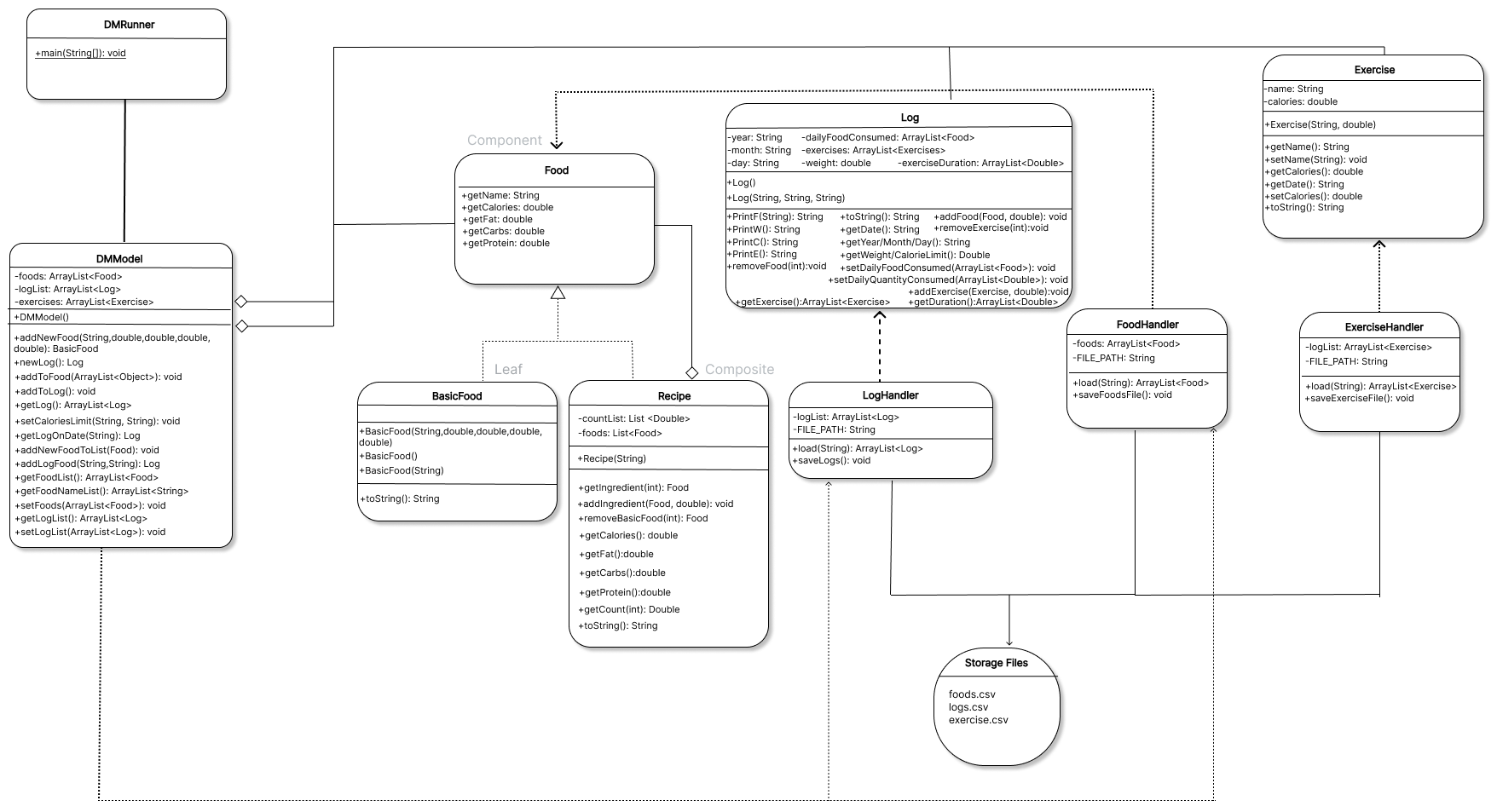
1. **Subsystems**

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**

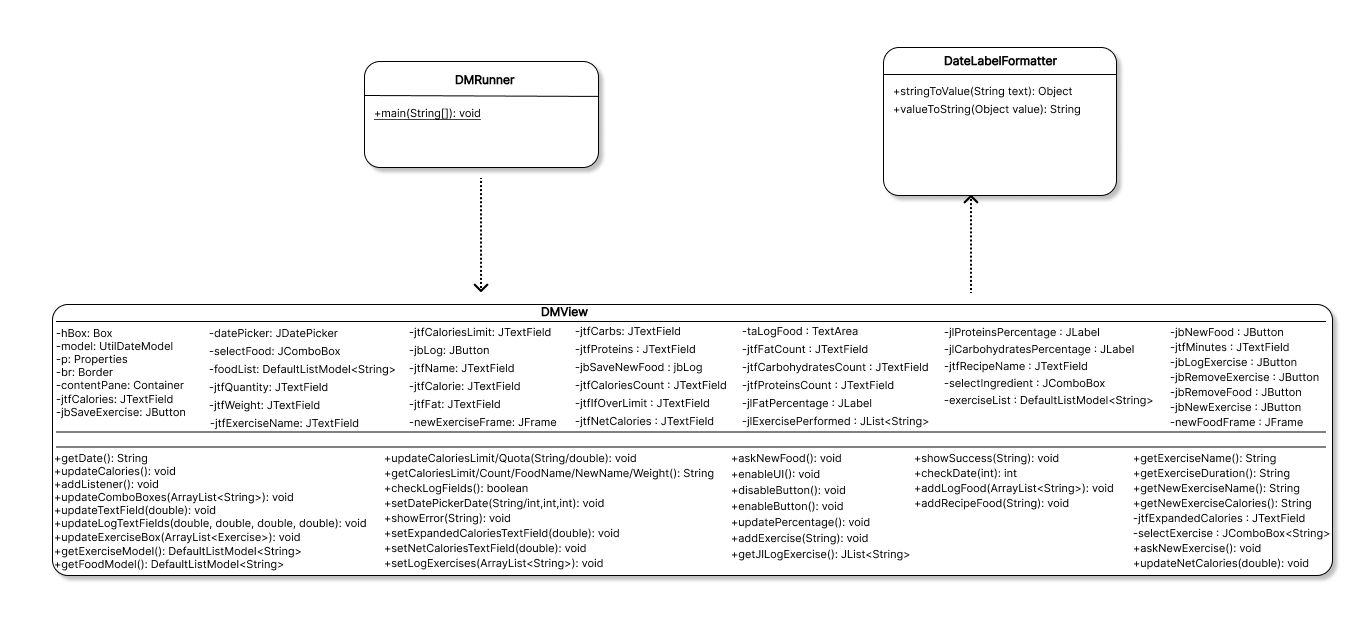
The model subsystem is represented by the DMModel class which utilizes the remaining classes essential for receiving and maintaining information within the program: Log.java, Exercise.java, Food.java, BasicFood.java and Recipe.java (all inside the model package). In addition to the handlers that monitor the input and output flow, it deals with operations meant to modify the data through its methods in order for the external csv files inside the application’s storage to match the latest updates made by the user. 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. Moreover, the composite pattern is implemented by connecting the basic foods and recipes into a collection of Food objects for simplicity.

The DMModel class interacts with the controller subsystem and the two handler classes inside the “io” package: FoodHandler and LogHandler.



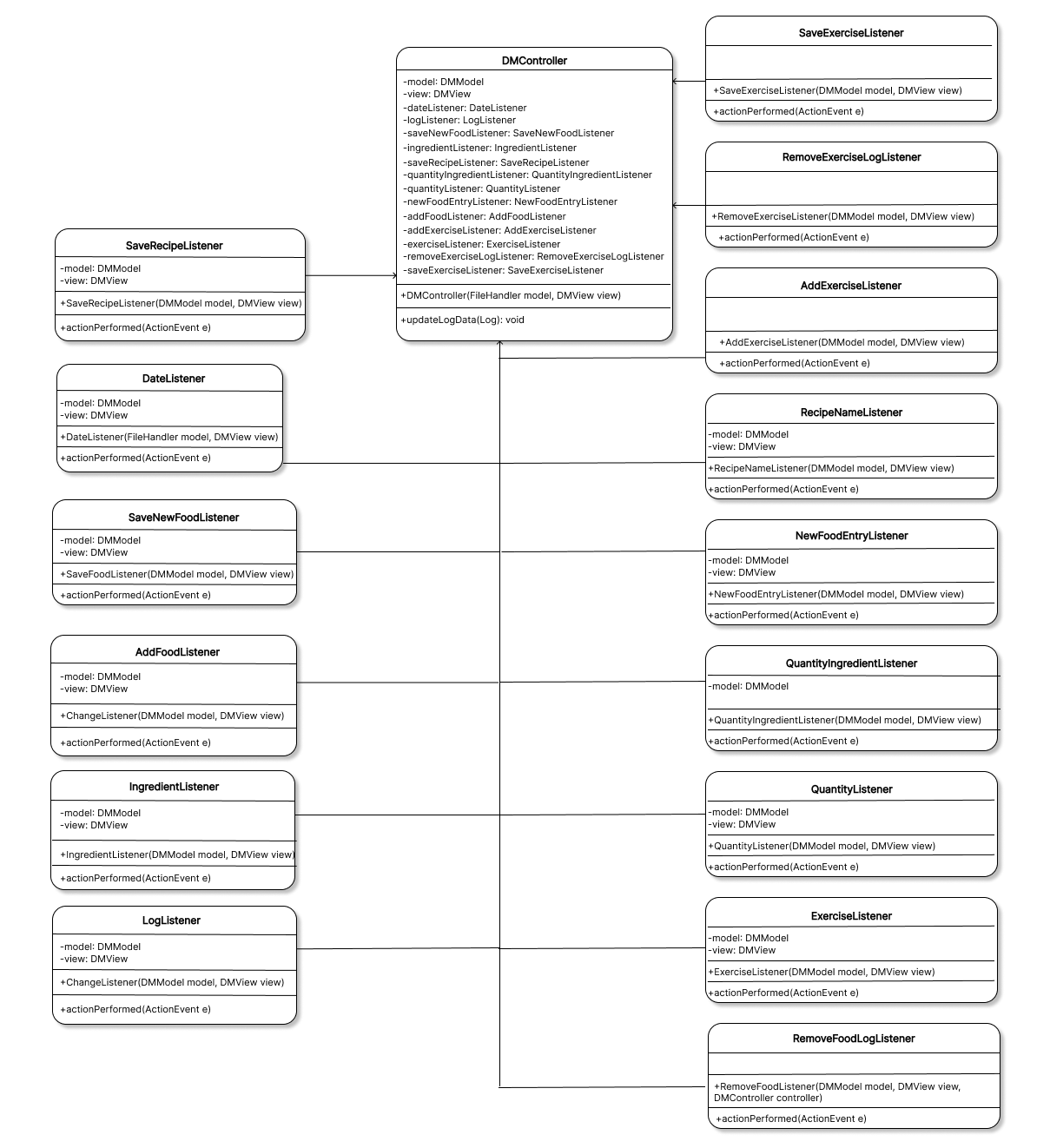
## **4.2 Subsystem View**

The view subsystem consists of the DMView class which is responsible for the creation of the GUI, and DateLabelFormatter class. Here, the user can both preview and alter the application’s data by interacting with the user interface displayed. In 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. The DateLabelFormatter class, much like its name suggests, is used to format the date according to the format inside the csv storage file. Finally, the view updates itself accordingly through the update methods contained within the class, so as to have all the GUI elements display accurate and valid information.



**4.3 Subsystem Controller**

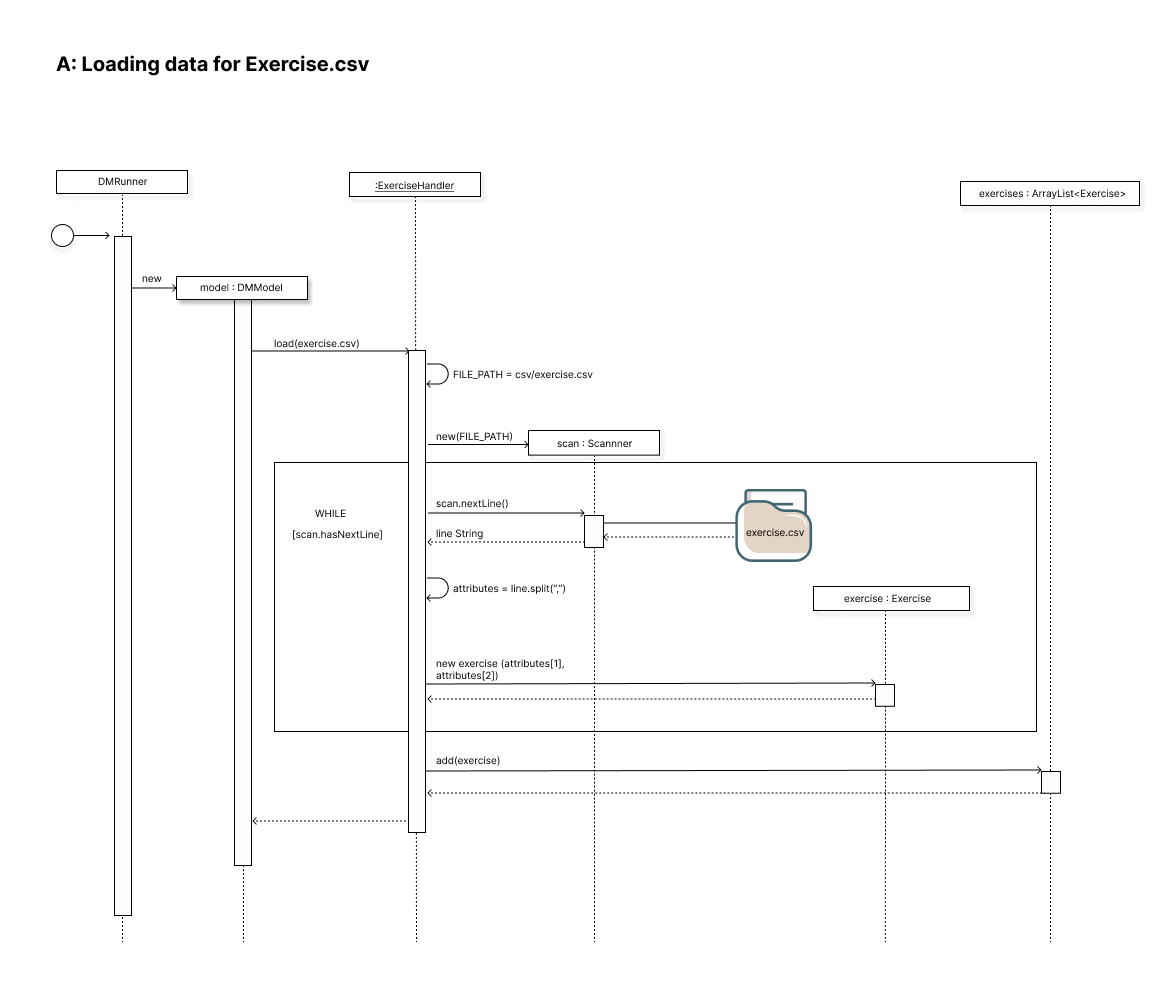
The controller subsystem consists of the DMController class which utilizes the functionalities of the listener classes that implement the ActionListener interface. This allows the controller to monitor the actions and changes that are occurring during the user’s interaction with the application and trigger 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. Hence, within the controller package, we created: DMController, AddExerciseListener, AddFoodListener, DateListener, ExerciseListener, LogListener, NewFoodEntryListener, QuantityIngredientListener, QuantityListener, RecipeNameListener, RemoveExerciseLogListener, SaveExerciseListener, SaveNewFoodListener, IngredientListener, SaveRecipeListener classes.



# Sequence Diagrams

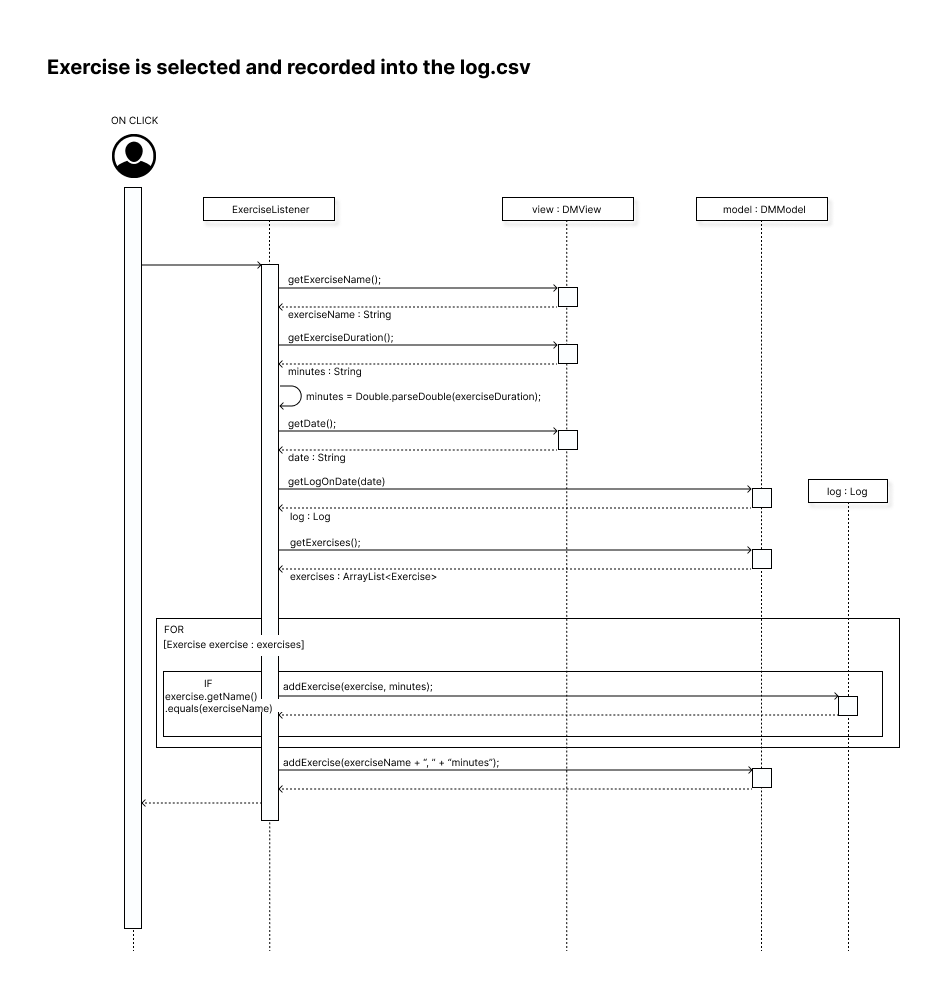
## **Sequence Diagram 1 (loading data exercise.csv / read the data and populate the model / running the program and reading the database).**

The Runner starts the program and creates objects of all MVC elements - model, view and controller. The model triggers the ExerciseHandler which reads from the file by initializing a Scanner. All instances of the Exercise class are created upon the Handler reading and retrieving the data from the csv file, which is responsible for instantiating the correct item and adding it to the collection of exercises, which is then given to the model and forwarded through the controller, and all items created with the data that has been read is sent to the model as a collection of Exercise items. The model holds three collections - foods, exercises and logs.



## **Sequence Diagram 2 (MVC data processing / adding an exercise performed 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 triggers the ExerciseHandler which reads from the file by initializing a Scanner. All instances of the Exercise class are created upon the Handler reading and retrieving the data from the csv file, which is responsible for instantiating the correct item and adding it to the collection of exercises, which is then given to the model and forwarded through the controller, and all items created with the data that has been read is sent to the model as a collection of Exercise items. The data is sent to the view through the controller. This enables the user to interact with the application and use the loaded data. The user adds input in the GUI which is then processed and sent to be stored inside the application storage through the model’s CRUD methods and MVC architecture (communication between separate subsystems - model, view and controller - where the controller acts as a bridge between the two). The changes being made are registered by the appropriate action listener class, in this case the ExerciseListener. The listeners are activated on button press and call the needed methods. Here, the information from the view is registered and processed in order to be sent to the model and logged when a button is pressed. That being the case, the ExerciseListener gets the name of the exercise and the duration spent practicing it from the input fields the user has filled, after which it performs the parsing in order to be able to perform calculations (from string to double). Then, it takes the date from the view, after which it uses that date to get the log on that specific date from the model and get the exercise from the model that was stored on that specific date. Once we get the collection of exercises that are stored in the model, since we need we need to iterate through the whole list of exercises that are stored in the Exercise.csv, the program enters the for loop in which it iterates through that collection until it finds the one that matches the user’s input (in this case, selection, since the user chooses from a combo box dropdown menu). When a match is found, the exercise is added to the log and stored along with the duration. The method then adds the new exercise to the model along with the specified duration to be stored in the model.



**Sequence Diagram 3 (MVC data processing / compute the total number of calories for the current date / calculating calories burned and net calories).**

Here, the controller is the one that needs to perform the necessary calculation and trigger the update in the view. It takes the information from the model class instance - log - and from the Exercises collection filled with correct data in order to have all necessary values to perform the requested calculations inside the method located in the controller. The data is processed and the calculation is being done within the same method, after which it is sent to the view to be displayed to the user. Once the name of the exercise, its duration and the given date have been read from the view, the controller will loop through the collection of exercises stored inside the model until it finds the entry that matches the one selected by the user. Once found, it calculates the burned calories and adds it to the dateExercise array along with the duration. From that class, the data will be sent to the view to be displayed for that specific date. The method sends the desired calculation to the view, which includes the amount of calories burned while exercising, the net calories (calories burned subtracted from calories consumed), and calorie goal which shows the exact number of calories by which the user has gone over their set calorie limit or has remained below the set bound . The controller here performs its duty as a bridge between the model and the view. This method deals with both food and exercise calculations, that way we can access the calorieIntake as the local variable inside the method, which was assigned a value beforehand, when the intake calories calculations were being executed.

The diagram that shows calculation of the total number of calories for the current date displays the first part of the update method, and one that displays how the calorieIntake local variable is being created and has a value assigned to it.

