Diet Manager

Project Design Document

Group 3

Helena Markulin <hm8842@rit.edu>

Michael Geljic <mg3178@rit.edu>

Zarko Zizic <zz6085@rit.edu>

Luka Boban <[lb1097@rit.edu](mailto:lb1097@rit.edu)>

Felicio Orlandini <fo4324@rit.edu>

**NOTE: Replace all the blue text in this document with your content.**

# Project Summary

The Diet Manager project is a Java desktop application that allows users to manage their dietary habits by logging food intake, viewing nutritional information, and tracking total calories. Users can also add new foods and view the food log for the day and the total calories. The application has a user-friendly interface built with Java Swing and uses CSV files for persistent storage of foods and logs. The primary goal was to build a basic but functional version of a personal diet tracker with support for food management and log entry operations.

# Design Overview

The design follows the Model-View-Controller (MVC) pattern. The **Model** contains classes such as Food, Foods, Log, and Logs, which manage food items and log entries. The **View** is implemented through a Swing-based View class that provides GUI components and handles user interaction. The **Controller** handles the business logic and connects the view with the model, implemented through the Controller class and multiple action listeners such as AddFoodButtonListener, AddLogButtonListener, and LoadButtonListener.

The aim is for each class to have a single clear purpose. Coupling is kept low by ensuring models and views communicate only through the controller. The design supports extendibility. For example, recipes and more advanced log features (like calorie goals and weight tracking) can be added later without the need for a lot of modification of the existing model classes.

One early design decision was to focus only on basic food logging and storage. Features such as reading logs on startup, calorie goal tracking, and weight tracking were noted but not implemented. This allowed easier progress on core functionality. A rejected idea included merging logs and food data into one file, which was abandoned for clarity and separation of concerns. Also, when adding the nutrients to different maps in our project, we found ourselves violaing the DRY principle on multiple occasions, which we had to go back and fix.

# Overall System Structure

A diagram of a computer code

AI-generated content may be incorrect.The UML class diagram shown above represents the overall architecture of the Diet Manager application. It reflects a clean and modular design that follows the **Model-View-Controller (MVC)** pattern, separating the responsibilities of data management, user interface, and user interaction logic into distinct subsystems.

Model Subsystem:

The Model layer is responsible for storing and managing application data. It includes:

• Food and Foods: Represent individual food items and collections of foods, with support for nutritional data and searching.

• Log and Logs: Represent daily food log entries, where each Log holds a reference to a Food and calculates total nutritional values based on servings. Logs provides filtering and aggregation by date.

• FileHandler: A utility class for reading from and writing to foods.csv and log.csv. It enables persistent data storage.

This subsystem has high cohesion, and each class focuses on a single responsibility, keeping the model independent of UI or event handling logic.

View Subsystem:

The View class manages the graphical user interface using Swing components. It handles all user input and output tasks—updating displays, prompting for user input, and responding to events like button clicks. It does not contain any business logic and depends on the controller to trigger meaningful operations.

Controller Subsystem:

The Controller class acts as the mediator between the view and the model. It contains references to the Foods, Logs, FileHandler, and View classes and serves as the central point of communication between subsystems.

Additionally, the three listener classes:

• AddFoodButtonListener

• AddLogButtonListener

• LoadButtonListener

implement ActionListener and encapsulate behavior triggered by user interaction with specific buttons. Each of them maintains a reference to the Controller and uses it to access the model and view as needed.

This structure keeps button-specific logic modular and makes the application easier to extend or maintain.

This design supports several key software engineering principles:

• Separation of concerns: UI, logic, and data are clearly separated into subsystems.

• Low coupling: Changes in the view or model are unlikely to break other components.

• High cohesion: Each class has a well-defined role, which keeps the code organized.

• Extendibility: New features like calorie limits, recipe management, or weight tracking can be added by extending existing models or adding new controller classes.

By isolating responsibilities, the design makes it easy to test, modify, and scale the application.

# Interaction Flow: Sequence DiagramA screenshot of a computer program AI-generated content may be incorrect.s

**Sequence Diagram 1 – Loading Food, Logging, and Calculating Calories**

This sequence diagram illustrates the full interaction required for completing the following scenario in the Diet Manager application:

1. Loading the food database (Action a)
2. Adding two servings of a basic food to the log for today (Action b)
3. Computing total calories for the current day (Action c)

It shows how objects in the system interact in a time-ordered sequence, following the Model-View-Controller (MVC) architectural pattern.

Interaction Breakdown

1. Application Initialization (Loading Food)

* The Main object creates the Controller, which initializes all components.
* The Controller calls readFoods("foods.csv") on the FileHandler.
* The FileHandler returns a List<Food> containing 3 items.
* The Controller adds each food item to the Foods model (inside a loop).
* The View is updated with the list of available foods and reset nutritional totals.

2. Adding Two Servings of Food

* The User clicks “Add Log Entry” in the UI.
* The View triggers AddLogButtonListener.actionPerformed() on the Controller.
* The Controller retrieves the selected food name from the View, then looks up the food in the Foods model.
* The View prompts the user to enter the number of servings ("2").
* A new Log object is created and added to the Logs model.
* The log display is updated.

3. Calculating Total Calories

* The Controller asks the Logs model to compute total calories for today.
* The Logs object internally loops over its log entries and calls getTotalCalories() on each Log.
* The final total is returned to the Controller, which updates the View with the calorie total.

Design Pattern Highlighted: MVC

This sequence strongly illustrates the Model-View-Controller (MVC) pattern:

* Model: Foods, Logs, and Log handle data and business logic.
* View: View handles all UI interactions and displays.
* Controller: Acts as the central coordinator, managing communication between the model and the view.