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 developed using Java Swing and object-oriented design principles to help users track daily dietary intake, view nutrient breakdowns, and manage custom food entries. The final version of the application applies the Model-View-Controller (MVC) architectural pattern and the Composite Design Pattern to support both individual food items and complex recipes made of multiple ingredients.

At startup, the program loads data from foods.csv, which can include both basic foods and recipes. Users can view the food list, select one or more items, and log them for the current date or any date they choose using the "Change Date" feature. Logged entries are shown in a dedicated log section, and the application dynamically calculates total calories, fat, carbs, and protein. A pie chart visually presents the nutrient distribution, and all calculations update in real time as new entries are added.

Users can also create new basic foods by entering nutritional values or build recipes composed of any combination of foods with custom serving sizes. These additions are written back to the foods.csv file, and all food and log data is persistently saved to CSV files (foods.csv and log.csv) after any change. The app supports both today's log and historical or future logs, with the ability to view or add entries for any selected date.

The implementation separates concerns cleanly between the View (user interface), Controller (application logic), and Model (data structures and business logic). By using the Composite Design Pattern, both BasicFood and Recipe objects share a common interface, allowing uniform handling of all food types when logging, displaying, or calculating nutritional values. The final version demonstrates a scalable and extensible approach to diet tracking and provides a strong foundation for further enhancements.

# Design Overview

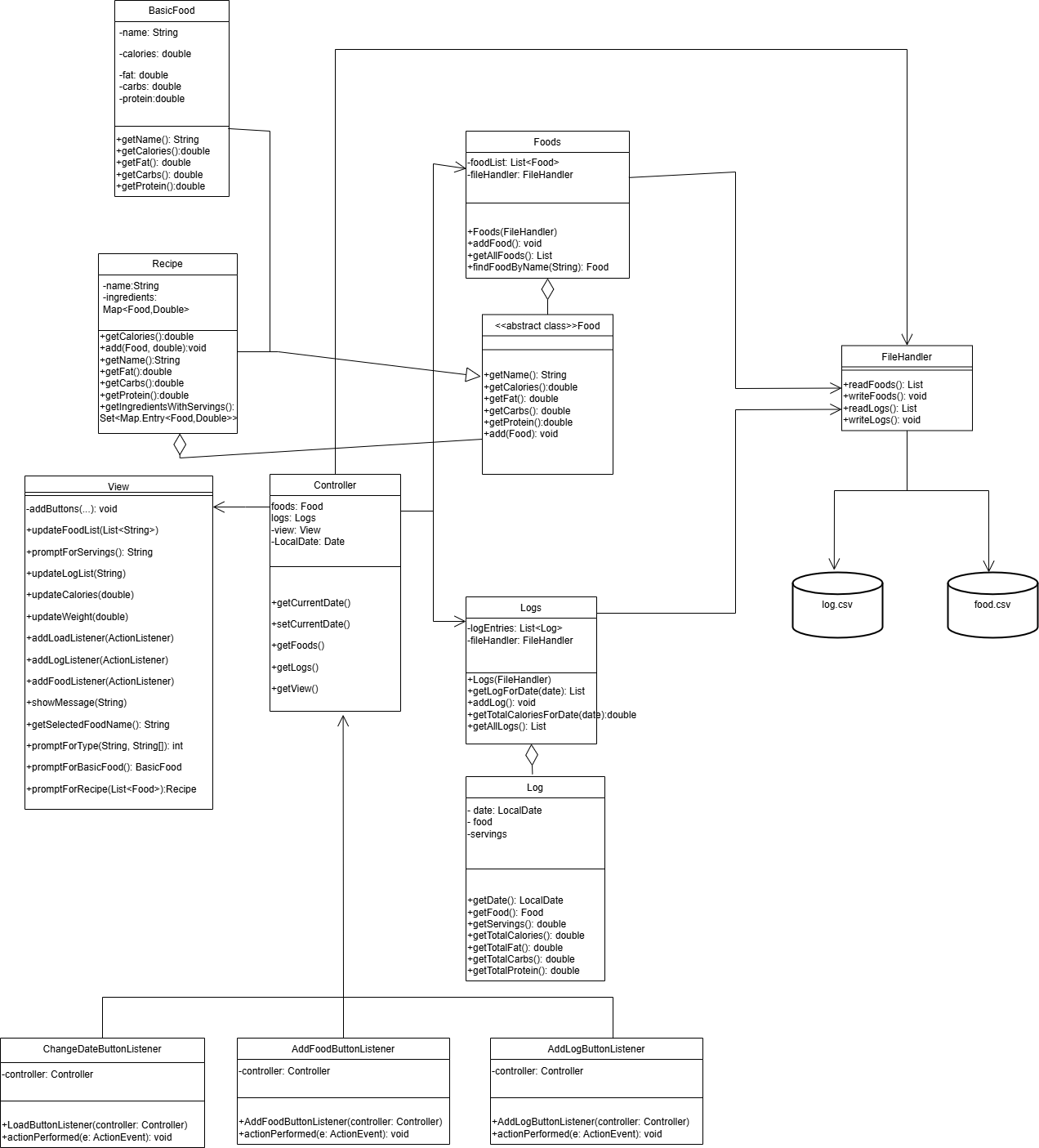
The design follows the Model-View-Controller (MVC) pattern. The **Model** contains classes such as Food, Foods, Log, and Logs, which manage food items and log entries. The **View** is implemented through a Swing-based View class that provides GUI components and handles user interaction. The **Controller** 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 ChangeDateButtonListener.

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.

The Composite Design Pattern was introduced to allow food items to be structured hierarchically. A new abstract Food class was created as the component, with BasicFood and Recipe acting as leaf and composite nodes, respectively. Recipes can contain other foods—including other recipes—and calculate nutritional values recursively. This change made the system more flexible and scalable, allowing users to build complex food combinations while reusing existing components. The use of the composite pattern also demonstrated clean separation of concerns and improved reusability of code within the model layer.

# Overall System Structure

 The UML class diagram illustrates the final version of the Diet Manager application. It follows the Model-View-Controller (MVC) architectural pattern and implements the Composite design pattern to manage both simple and complex food structures. Each class’s responsibilities are clearly defined to ensure maintainability, separation of concerns, and extendibility.

Model Subsystem:

* Food (abstract): Defines a unified interface for nutritional data access. It is the root of the Composite hierarchy, allowing both BasicFood and Recipe to be treated interchangeably.
* BasicFood: Stores information for individual food items including calories, fat, carbs, and protein. Implements Food methods by returning simple values.
* Recipe: A composite food that aggregates other Food objects (basic or recipe), each with a serving size. It recursively computes nutrient totals, supporting nested recipes.
* Foods: Maintains a list of all Food instances and provides operations to add, search, and persist them. It handles saving through internal calls to FileHandler.
* Log: Represents one consumption entry — a food item, the number of servings, and the date.
* Logs: Holds all log entries, provides filtering by date, and calculates total nutrient values (calories, fat, carbs, protein) for any day.
* FileHandler: Handles reading and writing of both foods and logs to CSV files. It performs two-pass parsing for correct reconstruction of basic and composite foods.

View Subsystem: The View class creates and manages the Java Swing interface. It displays the food list, current logs, nutrient totals, and a pie chart. It provides prompts for user input including recipe creation, calorie goals, and food servings. It is responsible for showing and updating GUI elements without containing any business logic.

Controller Subsystem: The Controller class orchestrates the application by connecting the View and Model. It initializes all components, sets up event listeners, and manages the current date for logging purposes. It provides centralized access to Foods, Logs, and the View, but delegates user interaction handling to dedicated listener classes.

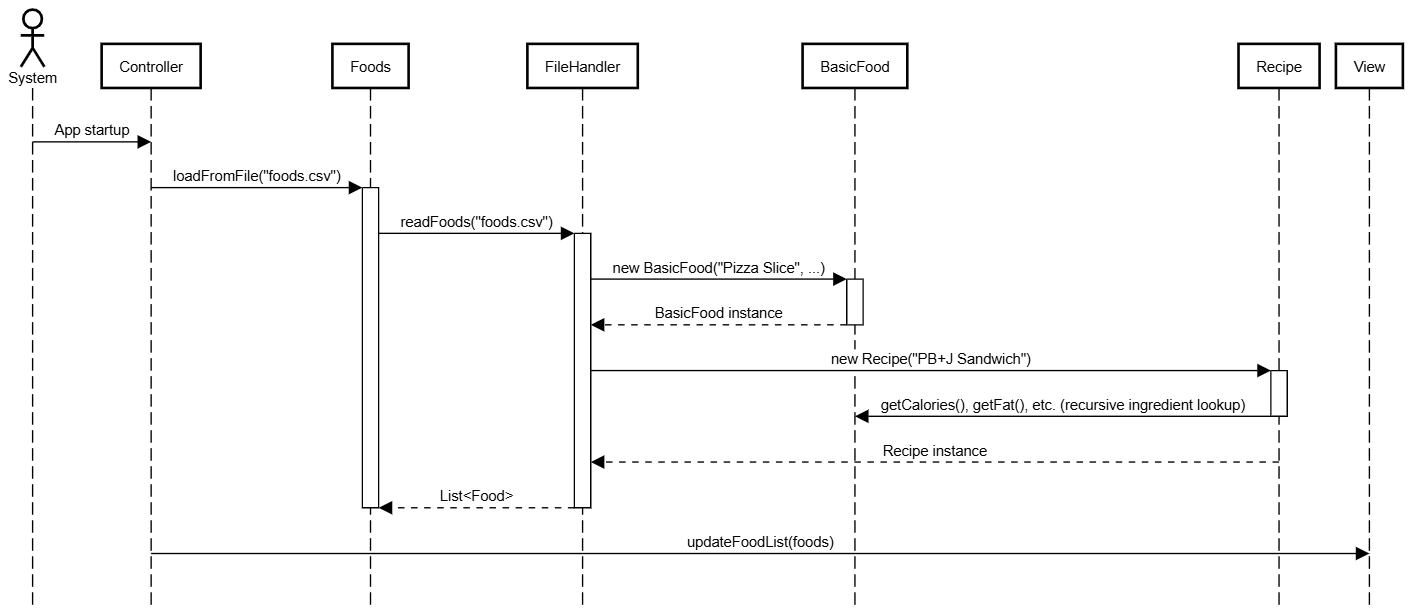
Listener Classes:

* AddFoodButtonListener: Prompts the user to create either a basic food or recipe and adds it to the model. Automatically persists to foods.csv.
* AddLogButtonListener: Allows the user to log one or more food entries for the current date and updates stats and logs in the view.
* ChangeDateButtonListener: Enables users to navigate between different dates. When the date is changed, logs and stats are automatically refreshed for the selected day.

The system is organized around the MVC architecture to cleanly separate user interface, data management, and control logic. This structure ensures that each component focuses on its own responsibility. The Model encapsulates all data handling and business logic, the View focuses solely on GUI presentation and input prompts, and the Controller coordinates communication between the two. This separation makes the system easier to debug, test, and extend. For example, updates to the food storage format or recipe logic can be made without touching the GUI, and new views or UI libraries could be integrated with minimal changes to the model layer.

The use of the Composite design pattern allows recipes to contain any number of ingredients, including other recipes, and still be treated as a single Food object. This makes it possible to compute nutritional values recursively and uniformly, simplifying logic in the logging and stats components. Listener classes further isolate user interaction logic from both the controller and view, making the system modular and adaptable. While this design introduces slightly more overhead initially, it significantly improves code maintainability and scalability, enabling future features like user accounts, nutrition goals, or more advanced analytics to be added without major rewrites.

# Interaction Flow: Sequence Diagrams

**Sequence Diagram 1 – Loading Data for Pizza Slice and PB+J Sandwich**

# This sequence diagram illustrates how the application loads food data at startup. When the program begins, the Controller calls loadFromFile() on the Foods model, which delegates to the FileHandler.readFoods() method. This method first creates a BasicFood object for “Pizza Slice” and adds it to a list. In a second pass, it constructs a Recipe object for “PB+J Sandwich” by linking it to the already loaded ingredients like peanut butter and bread. The Recipe recursively calculates its nutrition by aggregating the values of its ingredients. Once the complete list of Food objects (both basic and composite) is returned to the model, the Controller calls View.updateFoodList() to display the full list to the user. This process showcases the Composite design pattern, where recipes and basic foods are treated uniformly, and highlights clear separation between the controller, model, view, and file handling.

**Sequence Diagram 2 – Logging Pizza Slice and PB+J Sandwich for current date**

# This sequence diagram shows how the application logs multiple food entries for the current date. When the user clicks "Add Log Entry", the View calls AddLogButtonListener.actionPerformed(). The listener first gets the selected food names and retrieves the corresponding Food objects using Foods.findFoodByName(). It then prompts the user for servings of each item through View.promptForServingsForMultipleFoods(). The selected date is obtained via Controller.getCurrentDate(), and for each food, a new Log object is created and passed to Logs.addLog(). Once all logs are added, the listener requests the updated list of logs for the date and computes nutrition totals for calories, fat, carbs, and protein. These results are passed to the View for display, and finally, the updated logs are saved to the CSV file.diagram, it is an essential part of the logging logic handled internally.

# 

**Sequence Diagram 3 – Computing Total Calories for the Current Date**

This diagram illustrates how the total number of calories is computed for the currently selected date. The sequence begins when the Controller requests the total calories for the active date from the Logs model using getTotalCaloriesForDate(currentDate). Inside this method, Logs calls getLogForDate() to retrieve all Log entries that match the selected date. It then loops through each Log, calling getTotalCalories(), which multiplies the number of servings by the food's calorie value. The Food.getCalories() method provides the per-serving calorie count, supporting both BasicFood and Recipe due to the Composite pattern. The sum of all log entries is returned to the Controller, which then calls View.updateStats() to display the total calorie count in the GUI. This process ensures that nutrition stats always reflect the current date's log data accurately.

**Design Pattern Highlight – Composite Pattern in Activity 2**

In Activity 2, the **Composite Design Pattern** was introduced to unify the treatment of basic foods and recipes. The Food class was refactored into an abstract superclass, allowing both BasicFood and Recipe to inherit from it. This structure allows recipes to contain other Food objects (including other recipes or basic foods), forming a tree-like hierarchy. With this approach, the client code (such as the Log or View) can treat basic foods and composite recipes the same way — calling shared methods like getCalories() or getName() without needing to check the food type. This simplifies the logic in controllers and views, improves extensibility, and ensures consistency across operations involving food items.