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 designed to help users track their dietary intake by logging foods, viewing nutrition details, and monitoring total calories for the day. In Activity 2, the application was enhanced using the Composite Design Pattern to support both basic foods and recipes. A recipe is composed of multiple food items (basic or other recipes), each with customizable serving sizes. Users can now add either a single food or a composite recipe, and the application correctly calculates total nutritional values recursively. The interface remains user-friendly, built with Java Swing, and all data is stored and loaded via CSV files. The project demonstrates key object-oriented design principles and extends the original functionality with more flexible and scalable food management.

# 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.

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

A diagram of a computer

AI-generated content may be incorrect. The UML class diagram above represents the updated architecture of the Diet Manager application, enhanced in Activity 2 with new functionality and the use of the Composite design pattern. The structure continues to follow the Model-View-Controller (MVC) pattern, cleanly separating concerns between data handling (Model), user interface (View), and interaction logic (Controller).

Model Subsystem:

The Model is responsible for storing and managing application data. In this version, it includes:

* Food (abstract): Acts as the root of the Composite hierarchy. Both BasicFood and Recipe extend from it, allowing consistent treatment of single food items and combinations of items (recipes).
* BasicFood: Represents an individual food item with calories, fat, carbs, and protein.
* Recipe: A composite food that contains other Food objects (either BasicFood or other Recipe instances), along with serving sizes for each. It recursively calculates total nutrition.
* Foods: Manages a list of all Food instances (basic and composite), and allows adding and searching foods.
* Log and Logs: Log stores a food consumed on a specific date along with the number of servings. Logs stores all log entries and includes methods to filter logs and compute total nutrition for a date.
* FileHandler: Handles reading and writing from food.csv and log.csv. It supports serialization and deserialization of both basic foods and recipes using structured formatting.

These model classes are tightly focused, cohesive, and independent of GUI logic, aligning with separation of concerns and supporting future extensibility.

View Subsystem:

The View class manages the Swing-based graphical interface. It presents a list of available foods, displays the current day's log, and shows nutrition totals. It allows users to add basic foods or build recipes interactively, using dialog prompts to collect input and display error or status messages.  
New methods like promptForRecipe() and promptForType() allow the view to support composite food creation while staying decoupled from business logic.

Controller Subsystem:

The Controller class serves as the central mediator. It holds references to all model classes (Foods, Logs, FileHandler) and the View, wiring them together during initialization. The controller does not directly handle button actions but delegates them to listener classes.

Listener Classes:

There are three ActionListener implementations:

* LoadButtonListener: Loads foods from the CSV file and updates the view.
* AddLogButtonListener: Prompts for servings and adds a log entry for a selected food.
* AddFoodButtonListener: Handles the creation of both basic foods and composite recipes.

Each listener encapsulates logic tied to a specific GUI button, allowing clean, testable, and reusable code. They depend on the controller for access to shared application state and services.

Design Patterns and Principles:

* Composite Pattern: Allows treating both BasicFood and Recipe as a unified type (Food). This simplifies operations like getCalories() and makes recipe nesting possible.
* Separation of Concerns: Each class has a clear and isolated responsibility—UI, data, or logic.
* Low Coupling & High Cohesion: Subsystems interact through well-defined interfaces, and internal responsibilities remain focused.
* Extensibility: Features such as recipe nesting, calorie targets, or nutritional charts can be added with minimal disruption to existing classes.

By building on the MVC foundation and introducing Composite-based flexibility, this design keeps the Diet Manager scalable, modular, and easier to maintain or enhance.

# Interaction Flow: Sequence DiagramsA diagram of a food system AI-generated content may be incorrect.

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

This diagram illustrates the system’s behavior during application initialization and food loading. When the program starts (Main), it creates a Controller object. The controller calls readFoods("foods.csv") on the FileHandler to read a list of foods, which includes one basic food (Pizza Slice) and one recipe (PB+J Sandwich). The controller iterates through the list and adds each Food (basic or recipe) into the Foods collection using addFood(). Afterward, it updates the GUI by calling updateFoodList, clears previous logs with updateLogList(""), and resets calories and weight to 0 using updateCalories(0) and updateWeight(0.0). This shows the system's setup process and how both types of food are added to the model and view.

# A screenshot of a computer AI-generated content may be incorrect.

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

This diagram captures the process of adding food log entries for the current date. The user clicks the "Add Log Entry" button twice. The first click triggers a search for "Pizza Slice", which is a BasicFood. The user inputs "1" serving, and a new log is created and added using addLog(). On the second click, the user selects "PB+J Sandwich", which is a Recipe. After the user enters "2" servings, another log is added. After both logs are stored, the Logs model is queried for today’s entries using getLogForDate(), and the total calories are computed with getTotalCaloriesForDate(). The GUI is updated with updateLogList() and updateCalories(). Finally, the logs are persisted to the CSV using writeLogs().

# A screenshot of a computer AI-generated content may be incorrect.

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

This diagram shows how total calories are computed. The controller calls getTotalCaloriesForDate() on Logs, which iterates through each Log entry. Each log retrieves its associated Food and calls getCalories(). If the food is a Recipe, it contains multiple ingredients, and each ingredient’s calories are retrieved in a nested loop. If it's a BasicFood, its calories are returned directly. The total calories for each log (servings \* food calories) are summed up and returned to the controller. Finally, updateCalories() is called to display the result in the GUI.

**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.