Diet Manager V2

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

**Project Summary (V2.0)**  
The Diet Manager V2.0 is a Java Swing desktop application built with a clean Model-View-Controller (MVC) architecture and the Composite design pattern to give users a unified, extensible interface for tracking both diet and exercise. On launch, the app simultaneously loads foods (basic items and multi‐ingredient recipes) from foods.csv and exercises (name + calories burned per kg per hour) from exercise.csv into in-memory models.

Users can

* Add or edit basic foods, recipes, and exercises—each persisted immediately back to their respective CSV.
* Select any date (default = today) via “Change Date” and log either foods (with servings) or exercises (with minutes).
* Record weight and set a daily calorie goal; both values are saved to log.csv and, if absent on a given day, automatically fall back to the most recent entry (or sensible defaults).

Every time a log is added, edited, or deleted, the application

1. Recomputes total calories, fat, carbs, and protein consumed,
2. Calculates calories expended in exercise (using the user’s weight),
3. Derives net calories and goal difference,
4. Renders an updated pie-chart of macronutrient breakdown,
5. Updates all textual stats (including weight and remaining calories),
6. And immediately persists the change to log.csv.

By decoupling data (Model) from UI (View) through the Controller and using Composite for uniform treatment of BasicFood and Recipe, Diet Manager V2.0 delivers a scalable, maintainable foundation for comprehensive diet & exercise tracking—and lays the groundwork for future enhancements such as user accounts, richer analytics, or mobile synchronization.

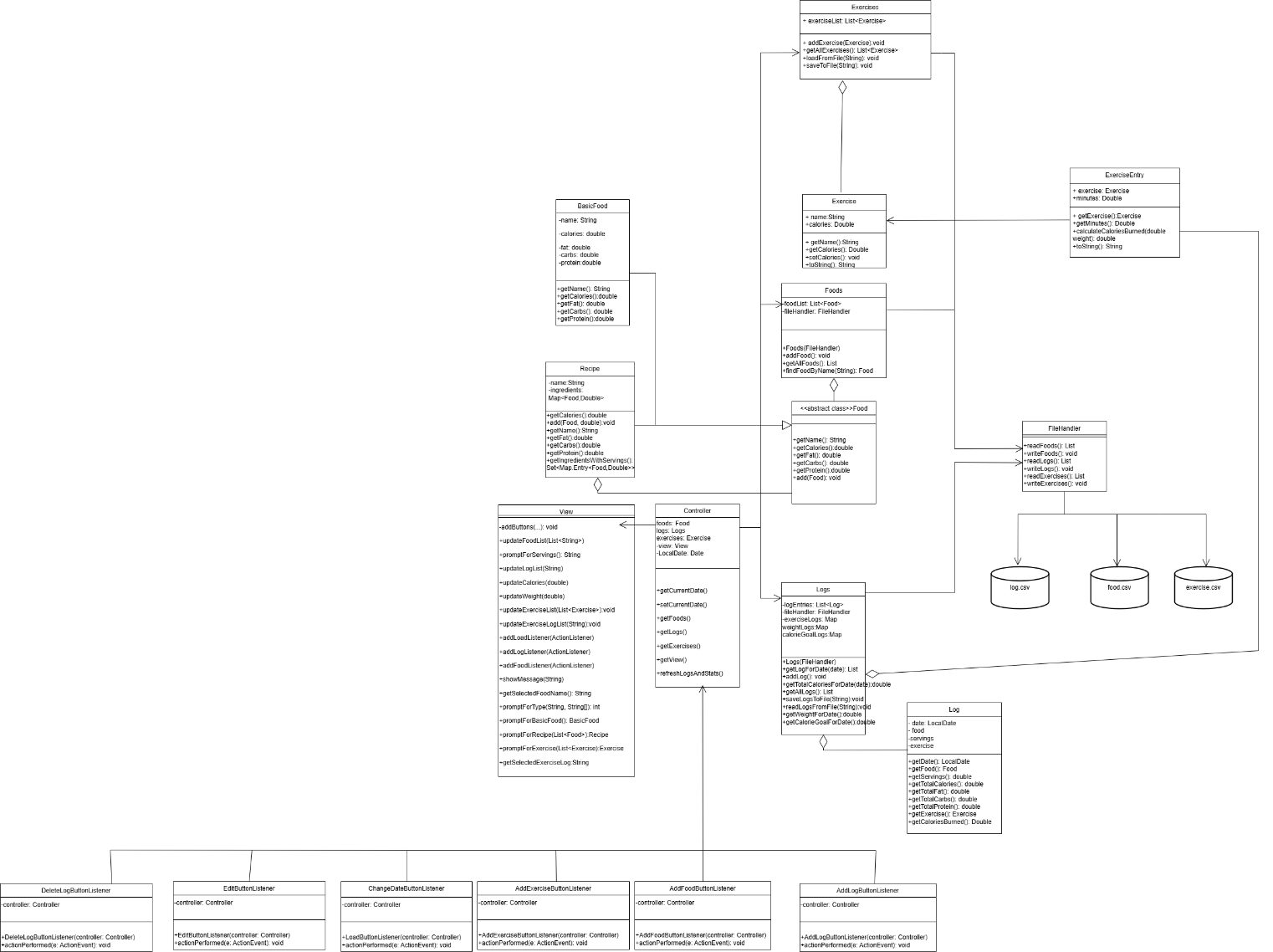
# Design Overview

Diet Manager V2.0 retains a strict Model–View–Controller separation, with each component shouldering a focused responsibility and communicating only through well‐defined interfaces.

* Model
  + Food hierarchy: an abstract Food interface is the root of a Composite pattern. BasicFood represents atomic items; Recipe composes any combination of Food instances (including other recipes) and computes all nutrient totals recursively.
  + Exercises: a new Exercise class encapsulates name and calories-burned-per-kg-per-hour, managed by an Exercises collection that handles deduplication, two-way CSV persistence, and lookup.
  + Logs: the Logs class now tracks four parallel data streams—food entries (Log), exercise entries (ExerciseEntry), daily weight, and daily calorie goals—providing both exact and “fall‐back” queries (most recent earlier date or sensible defaults). It computes total calories, nutrients, and net calories on demand.
  + FileHandler: orchestrates all CSV I/O—foods (two-pass parse for basic + recipes), exercises, and logs (food, exercise, weight, goals)—ensuring date formatting, zero-padding, and adherence to “no forward references” in recipes.
* View
  + A single Swing‐based View class lays out two side-by-side panels:
    - Left: selectable lists of foods and exercises, plus buttons for “Log Food/Exercise,” “Add Food/Recipe,” “Add Exercise,” “Edit Exercise,” and “Delete Log Entry.”
    - Right: a pie-chart of macronutrient breakdown, textual daily stats (calories, fat, carbs, protein, weight, remaining calories), date controls, and split lists for food and exercise logs.
  + All user prompts (adding/editing foods, recipes, exercises, servings/minutes, weight/goals) are rendered in single‐dialog forms for streamlined data entry.
* Controller & Listeners
  + The central Controller initializes models and view, loads all CSV data, and wires up six dedicated listeners:
    - AddFoodButtonListener
    - AddExerciseButtonListener
    - AddLogButtonListener
    - ChangeDateButtonListener
    - DeleteLogButtonListener
    - EditExerciseButtonListener
  + Each listener handles its own user‐interaction flow, updates the appropriate model(s), refreshes the view, and immediately persists changes—keeping UI logic out of models and persistence out of views.

This design preserves low coupling and high cohesion: models know nothing of Swing, views know nothing of file I/O, and controllers coordinate everything through clear listener interfaces. Composite ensures that BasicFood and Recipe are interchangeable, while the enhanced Logs and Exercises abstractions support the new exercise‐tracking, weight/fallback, and edit/delete features with minimal impact on existing code.

# 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.
* Exercise: Represents an exercise with calories burned per kilogram per hour. This supports calculating calories burned during exercise logging.
* Foods: Maintains a list of all Food and Exercise instances, providing operations to add, search, and persist them. It handles saving through internal calls to FileHandler.
* Log: Represents one consumption or exercise entry, storing a Food or Exercise, servings or minutes, and the associated date.
* Logs: Holds all log entries, provides filtering by date, and calculates total nutrient values (calories, fat, carbs, protein) and calories burned for any day.
* ExerciseEntry: Represents a specific instance of an exercise log, storing the exercise, minutes performed, weight at the time, and calories burned.
* FileHandler: Handles reading and writing of both foods and logs to CSV files. It manages foods.csv, log.csv, and exercise.csv, ensuring correct reconstruction of basic and composite foods, as well as exercise logs.

View Subsystem:

* The View class creates and manages the Java Swing interface. It displays the food list, exercise list, current logs, nutrient totals, calories burned, and a pie chart. It provides prompts for user input, including recipe creation, exercise logging, calorie goals, and food servings. It supports deletion and editing of exercise logs, reflecting the extended logging functionality. The View remains 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, manages the current date for logging, and handles coordination between data updates and the GUI. The Controller 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, persisting the data to foods.csv.
* AddLogButtonListener: Allows the user to log one or more food or exercise entries for the current date. It updates logs, exercise logs, and stats in the view.
* AddExerciseButtonListener: Enables users to add a new exercise (name and calories burned per kg per hour) to the system, saved in exercise.csv.
* ChangeDateButtonListener: Allows navigation between different dates. When the date changes, logs and stats refresh automatically for the selected day.
* DeleteLogButtonListener: Supports deletion of food or exercise logs from the selected date, ensuring the logs and stats are updated accordingly.
* EditExerciseButtonListener: Allows users to edit existing exercise definitions, such as adjusting calories burned per kg per hour.

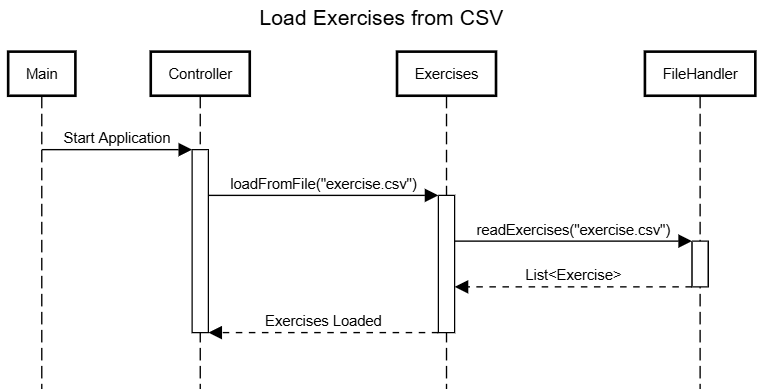
The system is organized around the MVC architecture to cleanly separate user interface, data management, and control logic. This structure ensures that each component focuses on its own responsibility. The Model encapsulates all data handling and business logic, the View focuses solely on GUI presentation and input prompts, and the Controller coordinates communication between the two. This separation makes the system easier to debug, test, and extend.

The Composite design pattern allows recipes to contain any number of ingredients, including other recipes, while being treated as a single Food object. This simplifies nutritional calculations and supports nested recipes.

The system has been extended to incorporate exercise tracking. Exercises can be logged alongside food consumption, and their calorie burn contributions are reflected in the daily stats. The introduction of ExerciseEntry and dedicated exercise logging ensures the same flexibility and uniform handling for physical activities.

The use of dedicated listener classes isolates user interaction logic, making the system modular and adaptable.

Interaction Flow: Sequence Diagrams

**Sequence Diagram 1 – Exercise records being loaded into the model**

This sequence diagram illustrates the process of loading exercises from a CSV file when the Diet Manager application starts. The interaction begins with the Main component, which triggers the Controller by sending a *Start Application* signal. The Controller initiates the loading process by calling the loadFromFile("exercise.csv") method on the Exercises class. The Exercises class, responsible for managing all exercise data, delegates the file reading task to the FileHandler by invoking readExercises("exercise.csv"). The FileHandler reads the exercise data from the CSV file and returns a List of Exercise objects back to the Exercises class. Once the exercises are loaded into memory, the Exercises class signals back to the Controller with an *Exercises Loaded* message, completing the loading process and ensuring that the exercise data is ready for use within the application.

# .

**A diagram of a log-in

AI-generated content may be incorrect.Sequence Diagram 2 – Logging Pizza Slice and PB+J Sandwich for current date**

When the user selects "Log Food/Exercise" from the interface, the system first prompts them to choose between logging a food or an exercise. If the user selects exercise, a dropdown appears listing all available exercises along with a field to enter the number of minutes performed. Once the user provides this information, the AddLogButtonListener creates a new ExerciseEntry containing the selected exercise and minutes. This entry is passed to the Logs model via the addExerciseLog() method, where it is stored internally under the selected date. Immediately after, the saveLogsToFile() method is triggered, which calls FileHandler.writeLogs() to persist all logs—including food entries, exercise entries, weight, and calorie goals—into log.csv. For exercises, the system writes lines formatted as yyyy,mm,dd,e,ExerciseName,Minutes. This ensures that the log is consistently saved with the correct structure. Finally, the controller invokes refreshLogsAndStats() to recalculate nutrient totals, calories burned, net calories, and to update the view, including the food and exercise log lists, nutrient statistics, and the pie chart, providing immediate feedback to the user.

# 

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

When the controller needs to refresh the daily stats, it begins by requesting the total calories consumed for the selected date from the Logs model using getTotalCaloriesForDate(currentDate). The Logs model filters all food logs for that date and, for each one, calls getTotalCalories() from the Log class to retrieve the calories for that food entry, summing them to get the total consumed. Once returned to the controller, the process continues by calculating the total calories burned via exercise through getTotalCaloriesBurnedForDate(currentDate, weight), passing the user's weight. The Logs model retrieves all ExerciseEntry logs for that date, and for each entry, it calls calculateCaloriesBurned(weight) to compute the calories burned. This calculation uses the exercise’s calories burned per kg per hour, retrieved via getCaloriesPerKgPerHour() from the associated Exercise object. The controller then calculates the net calories (consumed - burned) and the goal difference (calorie goal - net). Finally, the controller calls updateStatsPanel() on the View, passing the calorie goal, calories consumed, calories burned, net calories, goal difference, and total fat, carbs, and protein for that day. The view updates the nutrient stats and pie chart, providing the user with the current day's dietary summary.

**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.