PROJECT / RELEASE

Project Design Document 2

G4

Enrik Cipa

Niko Galic Alviz

Rinor Bugujevci

Emanuel Ivan Mlikota

Bruno Mehle

# Project Summary

# The Diet Manager project is a Java application designed to help users manage their diets by tracking their food intake and monitoring their weight. The application is built using the Model-View-Controller (MVC) design pattern, ensuring separation of concerns among the data handling, user interface, and control logic. It allows users to add and track basic foods and recipes, displaying nutritional information such as calories, fat, carbohydrates, and protein. Users can log their daily food intake and weight, with the information being stored and retrievable for any specific date. The application also supports loading and saving user data to CSV files for persistence. Design Overview

The design of the Diet Manager application adheres to several key principles of good software design. The application employs the MVC architectural pattern, which separates the application into three interconnected components, improving modularity and facilitating easier maintenance and scalability.

* Model (DietManagerModel): The model component holds the business logic and data. It interacts with FoodCollection and LogCollection classes to manage foods and daily logs, respectively. The model is responsible for all data-related operations, such as adding foods, retrieving food information, managing daily logs, and handling CSV data storage and retrieval. The separation from the view and controller ensures high cohesion and low coupling, as the model doesn't depend on user interface details or control logic.
* View (DietManagerView): The view component handles all user interface and display-related responsibilities. It uses JavaFX to create a graphical user interface (GUI) where users can input and view data. The view is designed to be reactive, updating the interface in response to model changes, ensuring a dynamic and responsive user experience. This separation ensures that the user interface can be modified or replaced without affecting the underlying business logic.
* Controller (DietManagerController): The controller acts as an intermediary between the view and the model. It processes user actions (e.g., button clicks), updates the model accordingly, and then updates the view. This component ensures that the view and model are loosely coupled, and changes to the control logic can be made independently of the model and view.
* CSV Handling (CsvHandler): Data persistence is handled by the CsvHandler class, which reads from and writes to CSV files. This allows users to maintain their diet information across sessions, contributing to the robustness and usability of the application. This class demonstrates a single responsibility principle by isolating file handling logic from the rest of the application.
* Logging (Logger): The application includes a Logger class for error and information logging, improving the maintainability and debuggability of the application by providing a centralized way to log messages.

**Composite Pattern Integration:**

The adoption of the Composite pattern allows treating individual foods and recipes uniformly, simplifying interactions and calculations. This decision showcases the application's flexibility and future scalability.

Rejected Alternatives:

Direct manipulation of the view from the model was considered but rejected to maintain a clear separation of concerns and allow for easier testing and maintenance.

Initially, there was an idea to use a single food class without distinguishing between basic foods and recipes. However, this was rejected for lacking flexibility and not properly representing the domain, leading to the current design where Recipes consist of BasicFoods.

**A screenshot of a computer

Description automatically generatedOverall System Structure**\*\*\*attached image for better overview in folder

# Subsystems

This The Diet Manager application is organized following the Model-View-Controller (MVC) design pattern, which separates the application into three interconnected components. This organization enhances the separation of concerns, making the system more modular, easier to maintain, and scalable. The Model subsystem manages the application's data and business logic, encompassing classes such as DietManagerModel, FoodCollection, LogCollection, and User. This segregation ensures that the business logic and application data are encapsulated away from user interface concerns, facilitating easy updates and maintenance without affecting the UI.

The View subsystem, primarily represented by the DietManagerView class, handles all user interface and presentation logic. This separation allows the UI to be changed or redesigned without altering the underlying business logic. The Controller subsystem, through the DietManagerController, acts as an intermediary between the View and Model, handling user interactions, updating the model based on user actions, and updating the view to reflect changes in the model. This structure allows for a clear distinction between different aspects of the application, promoting a clean architecture where each part can be developed and tested independently.

Rationale for Object Interactions in a Task:

In the context of a task such as adding a new food item, the objects interact through well-defined pathways in alignment with the MVC pattern. The user inputs data in the View (DietManagerView), which captures the user actions and sends this information to the Controller (DietManagerController). The Controller validates this input and modifies the Model (DietManagerModel) accordingly. The Model updates its state and notifies the View of the change. The View then updates the user interface to reflect the new state of the Model. This sequence of interactions ensures a clear separation between the user interface, the application logic, and the data management.

For sequence interactions involving different subsystems, such as saving data to a CSV file, the Controller might invoke methods in the Model to gather current state data. The Model, upon needing to persist this data, interacts with the Utility subsystem, specifically the CsvHandler, to save the data externally. This interaction clearly delineates the boundaries between application logic (Model and Controller) and auxiliary services (Utility), ensuring that each component remains focused on its responsibilities, enhancing maintainability and testability.**Subsystem name**

Each subsystem has a UML class diagram showing relationships between classes and interfaces.

In the simplest case the interfaces and classes will simply have class boxes with the appropriate name. In the final document, and preferably in intermediate documents, classes will also include the public methods provided by the class. Clearly highlight and explain here how a pattern is being used

If a class in this subsystem collaborates with a class in a different subsystem, simply include link to a box with the ***other subsystem's name***.

## 

## **Subsystem**

## **Model Subsystem:**

## **Classes: DietManagerModel, FoodCollection, LogCollection, User, BasicFood, Recipe, Food, DailyLog, FoodIdentifier.**

## **Description: This subsystem is the heart of the application's data management and business logic. It handles operations on food items (BasicFood and Recipe), user data (User), and daily intake logs (DailyLog), with FoodCollection and LogCollection managing collections of foods and logs.**

## **Patterns: Incorporates the Composite pattern through Food, BasicFood, and Recipe to manage food items flexibly. Singleton pattern considerations apply to central management components like DietManagerModel.**

## **different types of consumables. The DietManagerModel acts as the central class that orchestrates operations among these classes.**

## **Patterns: The Model subsystem uses the Singleton pattern for classes that should only have a single instance, such as DietManagerModel if implemented as such, ensuring centralized management of the application state.**

## **View Subsystem:**

## **Classes: CsvHandler, Logger.**

## **Description: Supports the application with logging (Logger) and CSV file operations (CsvHandler), performing essential tasks that are auxiliary to the core business logic.**

## **Patterns: The Adapter pattern is exemplified by CsvHandler, which bridges data storage needs with application data structures without entangling the Model with specific storage implementation details.**

## **Controller Subsystem:**

## **Classes: DietManagerController.**

## **Description: Acts as the intermediary between the View and the Model. It processes user actions, manipulates the Model, and updates the View.**

## **Patterns: Utilizes the Command pattern to encapsulate user actions into objects that can be executed and undone, providing a clear structure for user interaction management.**

## **Utility Subsystem:**

## **Classes: CsvHandler, Logger.**

## **Description: Supports the application with logging (Logger) and CSV file operations (CsvHandler), performing essential tasks that are auxiliary to the core business logic.**

## **Patterns: The Adapter pattern is exemplified by CsvHandler, which bridges data storage needs with application data structures without entangling the Model with specific storage implementation details.**

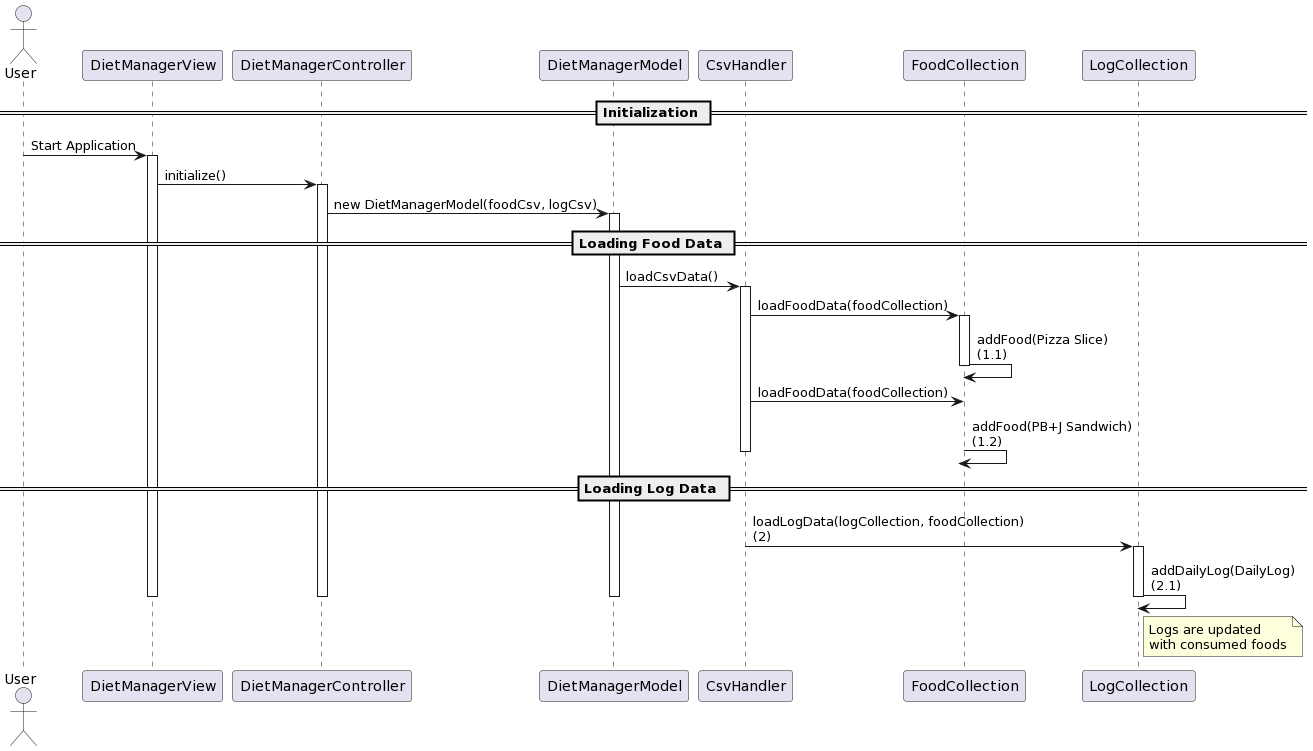
## **Entry Point Subsystem:**

## **Classes: DietManagerRunner.**

## **Description: Contains the main method to initiate the application, setting up the MVC components and starting the user interface.**

## **Patterns: This could represent a part of the Factory pattern if it involves creating instances of the MVC components based on specific configurations or environments.**

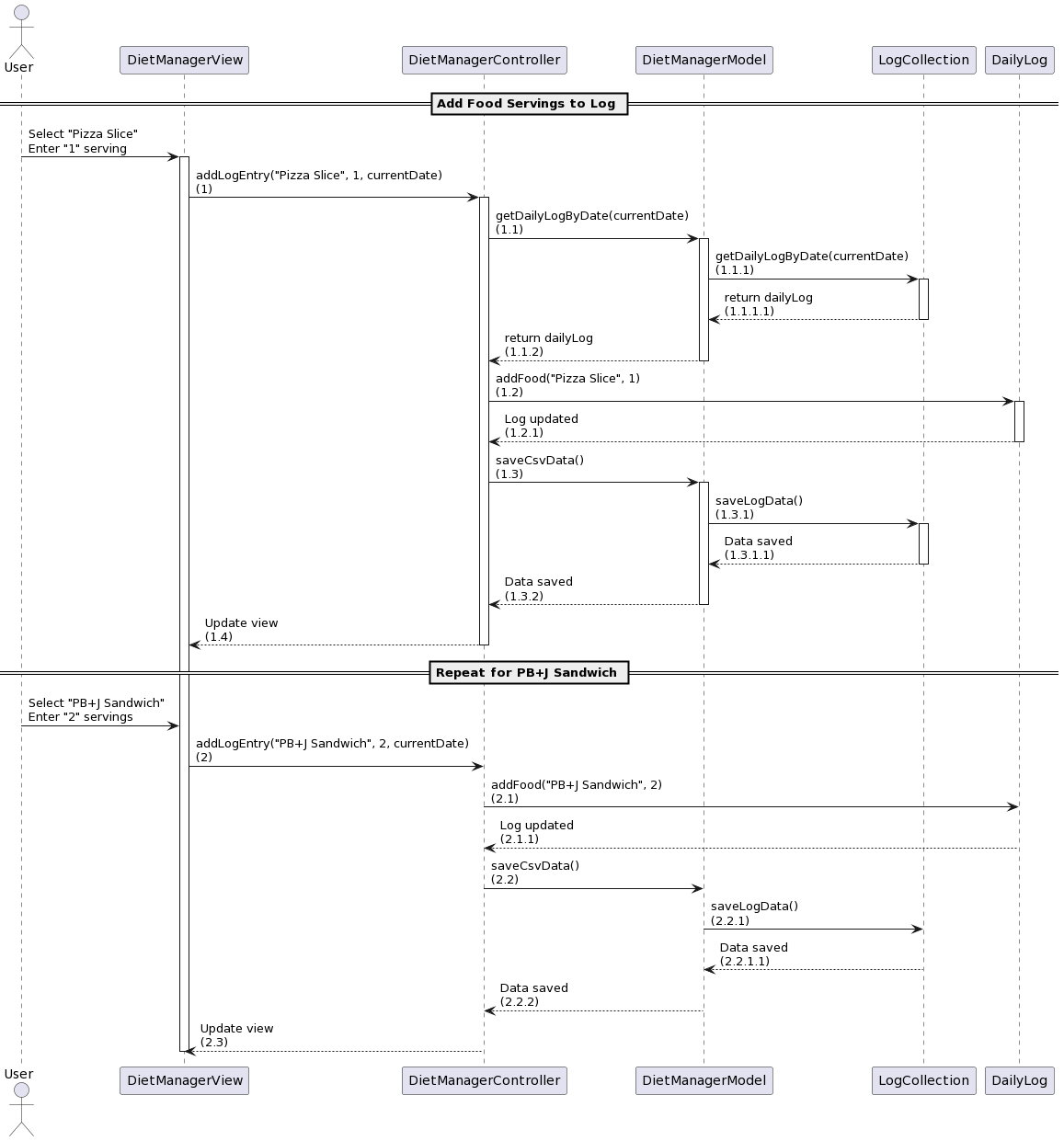
# Sequence Diagrams

**1.Loading data for 1 basic food (Pizza Slice) and 1 recipe (PB+J Sandwich)  
  
**

Upon starting the application, the **DietManagerView** initiates the **DietManagerController**, which then creates a **DietManagerModel**, potentially utilizing a Factory pattern to manage object creation. The model kicks off data loading by directing the **CsvHandler** to read and interpret CSV files.

During the loading phase, the **CsvHandler** acts as a Builder, constructing **Food** objects and adding them to the **FoodCollection**. It adds basic food items such as "Pizza Slice" and composite items like the "PB+J Sandwich" recipe, showcasing how the application employs the Composite pattern to handle both simple and complex food items uniformly.

Lastly, the **CsvHandler** updates the **LogCollection** with data on consumed foods, likely using the Singleton pattern to ensure that a single instance of each collection is used throughout the application. This setup guarantees that all parts of the application are referencing the same data, maintaining consistency and integrity across the system.

1. **Add 1 serving of the Pizza Slice & 2 servings of the PB+J Sandwich to the log entry for the current date.**
2. 

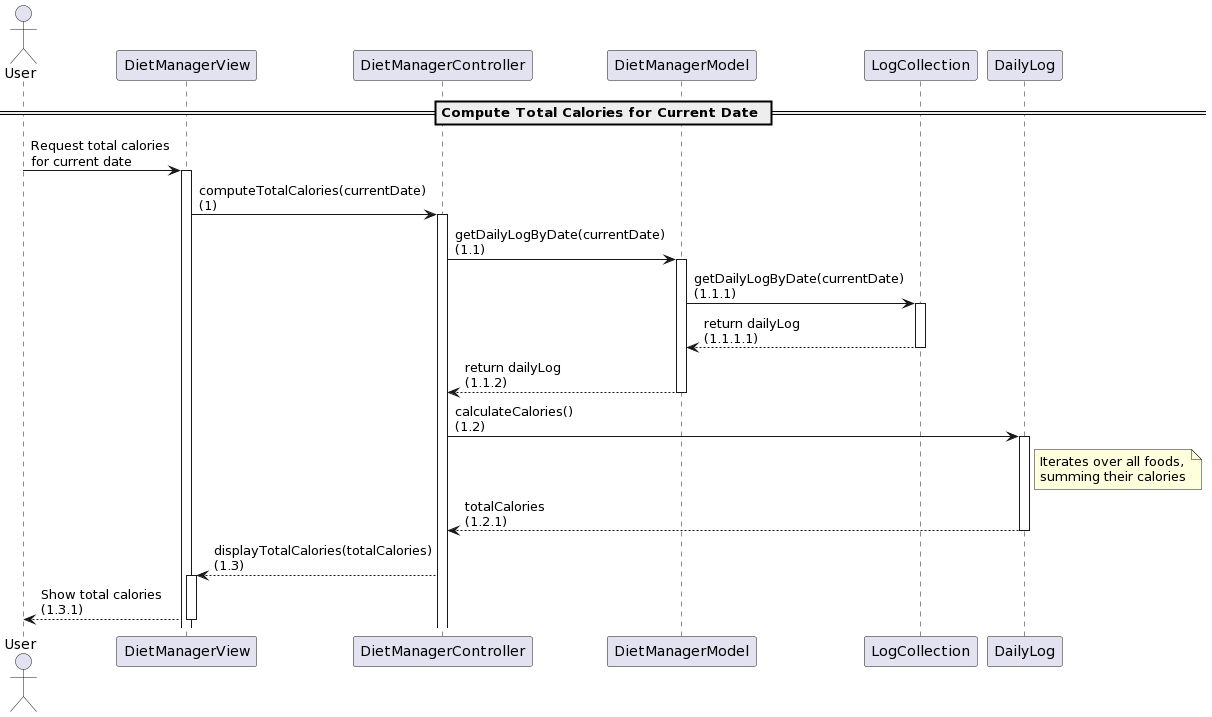
This sequence diagram describes the process of adding food servings to a log entry for the current date in the Diet Manager application, specifically for "Pizza Slice" and "PB+J Sandwich". The interaction follows the Model-View-Controller (MVC) architectural pattern, illustrating the flow from user action to data persistence and view update. Here's a compact narrative of the diagram:

A user selects a food item and specifies the number of servings through the **DietManagerView**, initiating two separate requests for adding "Pizza Slice" with 1 serving and "PB+J Sandwich" with 2 servings to the daily log. The **DietManagerView** communicates these actions to the **DietManagerController** by calling **addLogEntry** with the relevant food name, serving amount, and the current date.

The **DietManagerController**, upon receiving each request, queries the **DietManagerModel** to fetch the **DailyLog** for the current date, leveraging the **LogCollection** for retrieval. Once the **DailyLog** is obtained, the controller instructs it to add the specified food and servings. The addition of food to the log triggers a cascade of actions to save the updated log data: the controller calls the model to initiate CSV data saving, which in turn involves the **LogCollection** updating the CSV files.

After the data persistence actions are completed for both food items, the controller signals the view to update, reflecting the addition of the food servings to the user. This interaction showcases the MVC pattern in operation, with clear delineation of responsibilities: the view handles user interface, the controller processes user inputs and orchestrates updates, and the model manages the application data, ensuring data consistency and persistence.

3. **Compute the total number of calories for the current date.**

  
  
The sequence diagram for computing the total number of calories for the current date in the Diet Manager application showcases the Model-View-Controller (MVC) architectural pattern in action. The process starts with a user request in the View (**DietManagerView**), highlighting the View's role in interfacing with users. This request is then forwarded to the Controller (**DietManagerController**), demonstrating the Controller's function as the intermediary that processes user inputs. The Controller interacts with the Model (**DietManagerModel**), specifically requesting data from the **LogCollection** to retrieve the relevant **DailyLog**. This step emphasizes the Model's responsibility for managing business logic and data. The **DailyLog** calculates the total calories, a computation handled within the Model, underscoring the separation of concerns principle central to MVC. Finally, the calculated total is passed back to the View via the Controller, completing the cycle by updating the user interface with the requested information. This interaction flow exemplifies the MVC pattern's efficiency in separating concerns, enhancing modularity, and facilitating straightforward updates to the user interface in response to changes in the application's state or user actions.