PROJECT / RELEASE

Project Design Document

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

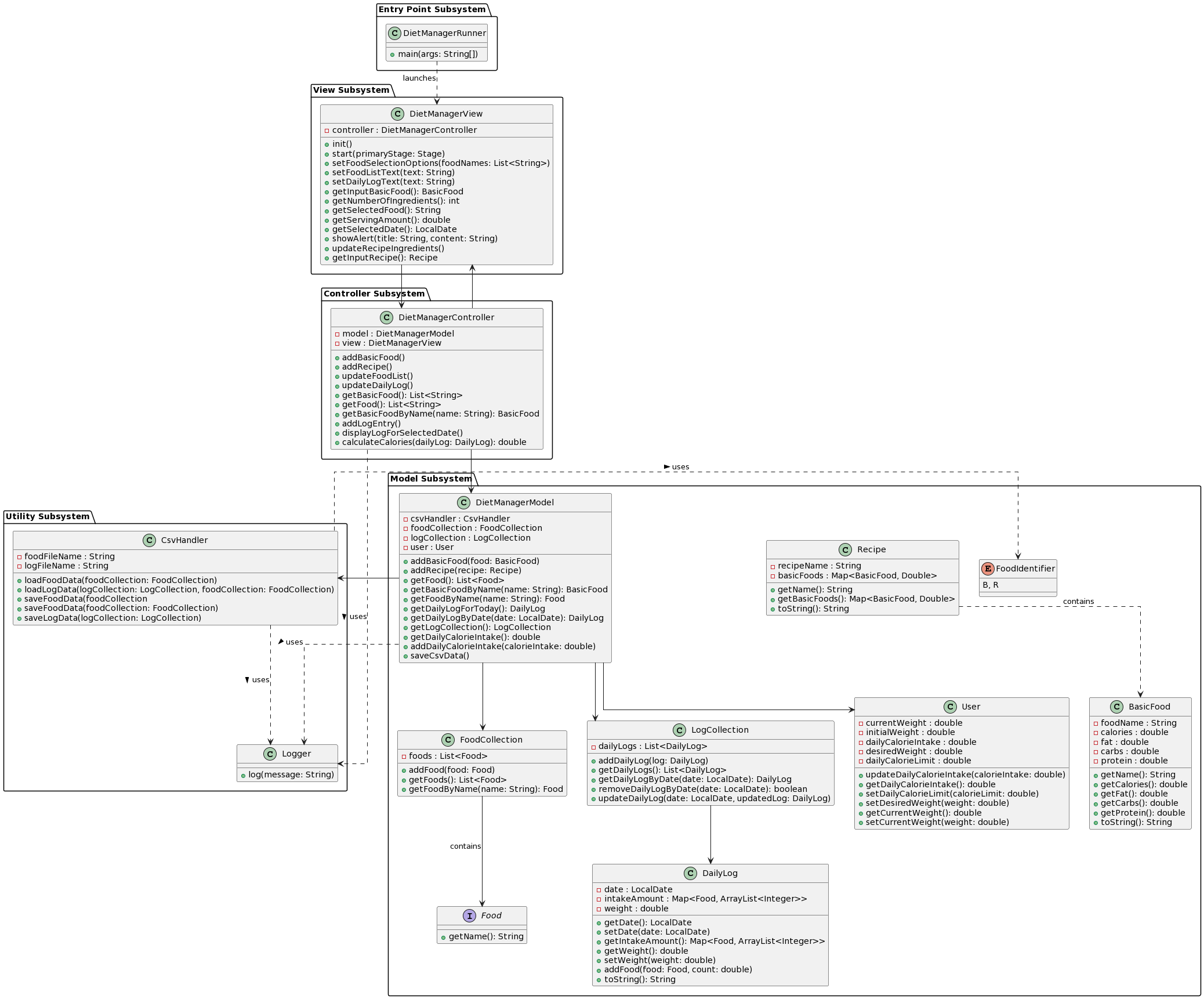
The design of the Diet Manager application adheres to several key principles of good software design. The application employs the MVC architectural pattern, which separates the application into three interconnected components, improving modularity and facilitating easier maintenance and scalability.

* 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.

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.

**Overall 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: Manages application data and rules. Classes like FoodCollection and LogCollection manage collections of foods and logs, respectively. User holds information about the application user, while BasicFood and Recipe represent 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: DietManagerView.**

## **Description: Responsible for displaying the user interface and gathering user input. It reflects the current state of the Model and sends user commands to the Controller. The subsystem uses standard UI components to construct the interface.**

## **Patterns: Observer pattern could be observed here as the View updates when the Model changes state, maintaining synchronization between the UI and the application data.**

## **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: Provides auxiliary functionalities such as logging (Logger) and CSV file operations (CsvHandler). These support the primary subsystems by performing specific tasks not directly related to the core business logic.**

## **Patterns: The Adapter pattern can be seen in CsvHandler, adapting the interface of data storage to the needs of the application while keeping it decoupled from the Model.**

## **Entry Point Subsystem:**

## **Classes: DietManagerRunner.**

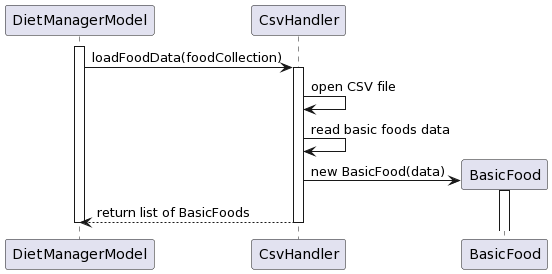
## **Description: Contains the main method to initiate the application, setting up the MVC components and starting the user interface.**

## **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

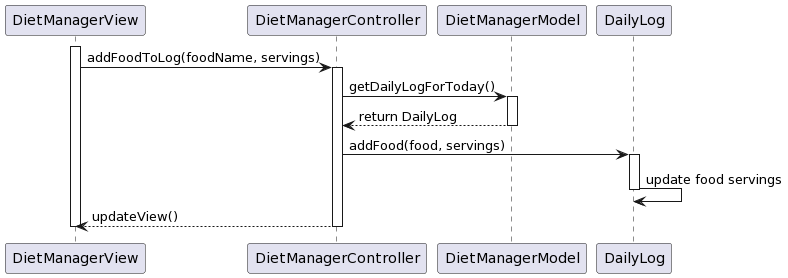
## **Sequence Diagram 1 (what feature / operation / scenario the diagram shows).**

1. **Reading in a Food Database:**



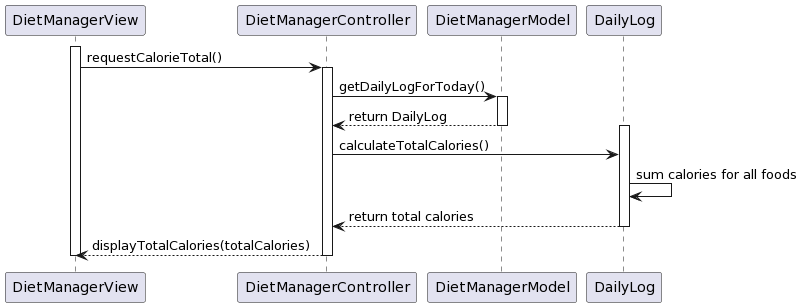
In this sequence, the **DietManagerModel** requests the **CsvHandler** to load food data. The **CsvHandler** opens the CSV file, reads the basic food data, and returns this information back to the **DietManagerModel**. This interaction showcases the use of the Singleton and Adapter patterns. The **CsvHandler**, acting as a Singleton, ensures that there's only one instance manipulating the CSV files, while its role as an Adapter makes external data formats usable within the system without changing the system's core logic.

1. **Adding Two Servings of a Basic Food to the Log Entry:**



1. In this scenario, the DietManagerView receives user input to add food servings and requests this action from the DietManagerController. The controller fetches the current DailyLog from the DietManagerModel, then adds the specified servings of food to the log. Finally, it signals the DietManagerView to update the display. This sequence exemplifies the MVC pattern, clearly separating concerns: the View handles UI, the Controller manages user input, and the Model maintains the data.

3. **Computing the Total Number of Calories for the Current Date**

  
  
Here, the **DietManagerView** requests the total calories from the **DietManagerController**, which then retrieves the **DailyLog** for the current day from the **DietManagerModel**. The **DailyLog** calculates the total calories by summing up the calories for all food entries, and this total is then returned through the layers back to the view for display. This demonstrates the MVC pattern and highlights the encapsulation principle, as the logic for calculating total calories is kept within the **DailyLog**, separate from the UI and control logic.