DietManager Design Document

Title Page

Project Title: DietManager

Team:

1. M. Bheema Siddartha

2. B. Srinivas

Date: 06-04-2025

Overview

The **DietManager** is a CLI-based Java application designed to help users manage their daily food intake. The application enables users to:

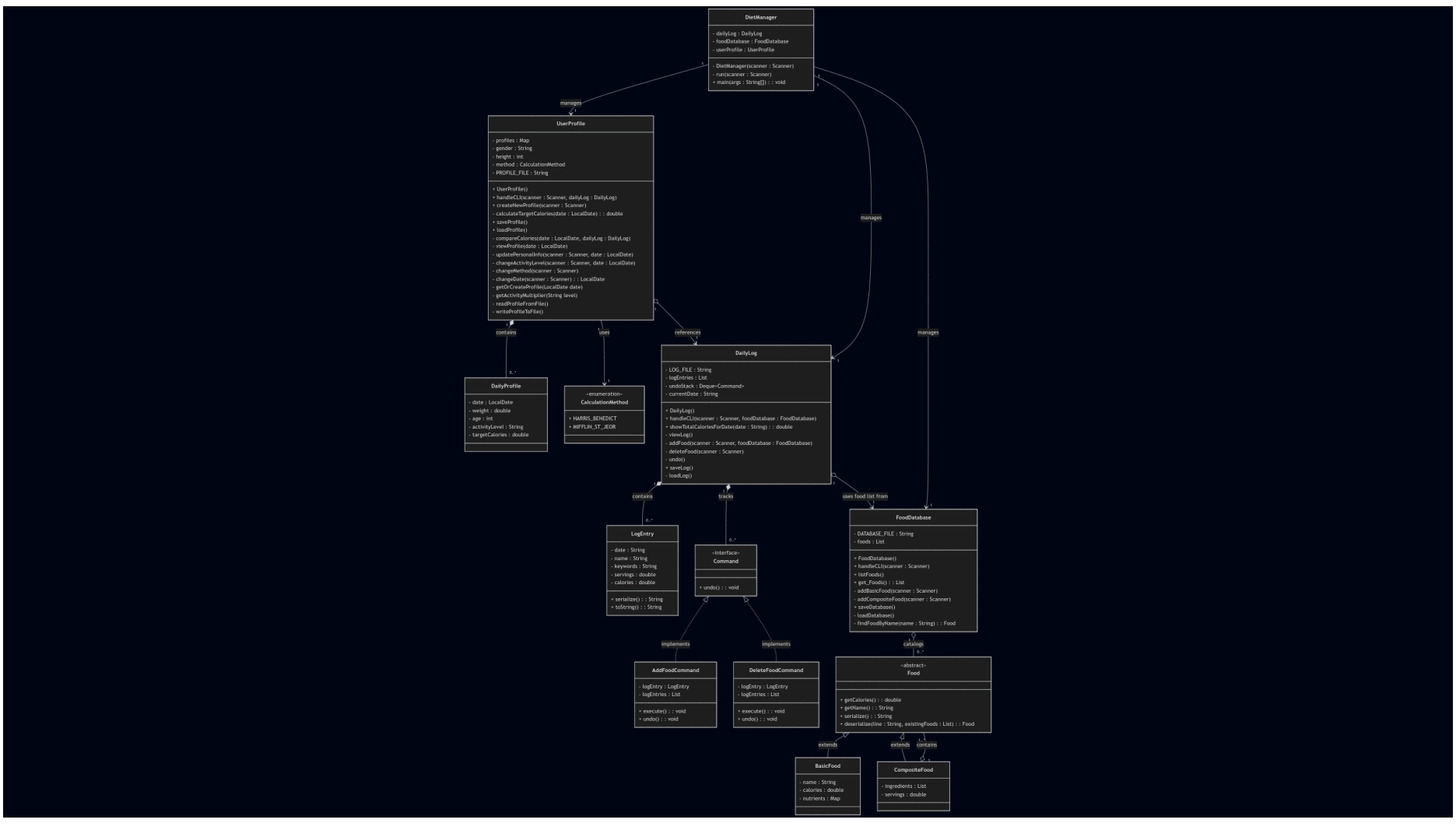
- Log daily food entries
- Manage a food database (basic and composite foods)
- Track calories and nutritional values

The system is organized into three main modules:

- `DailyLog` for daily tracking of food intake
- `FoodDatabase` for storing and retrieving food items
- `UserProfile` for personal user metrics (not included in this code sample)

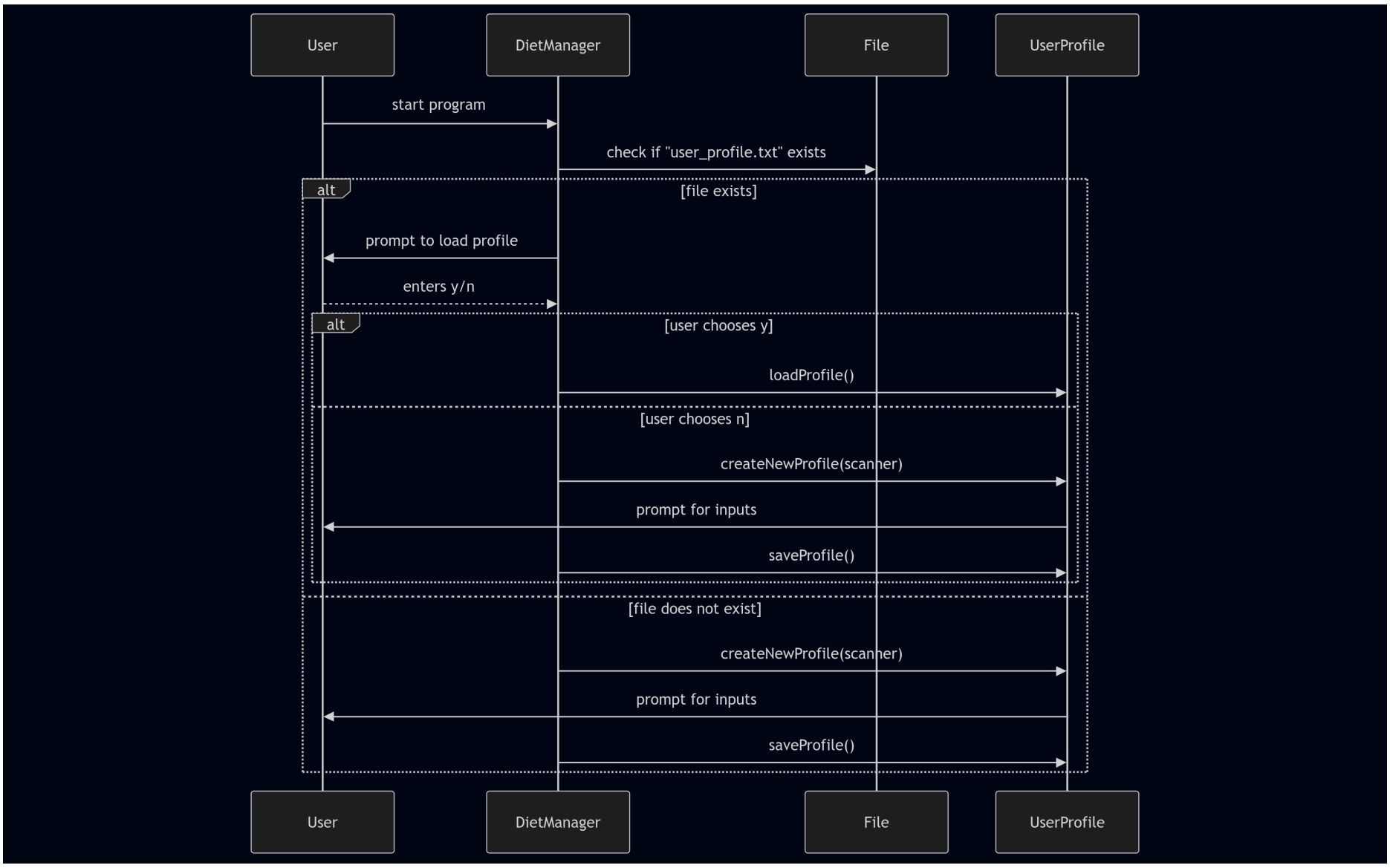
Data is persisted in plain text files to retain information between sessions.

UML Class Diagram (Mermaid)

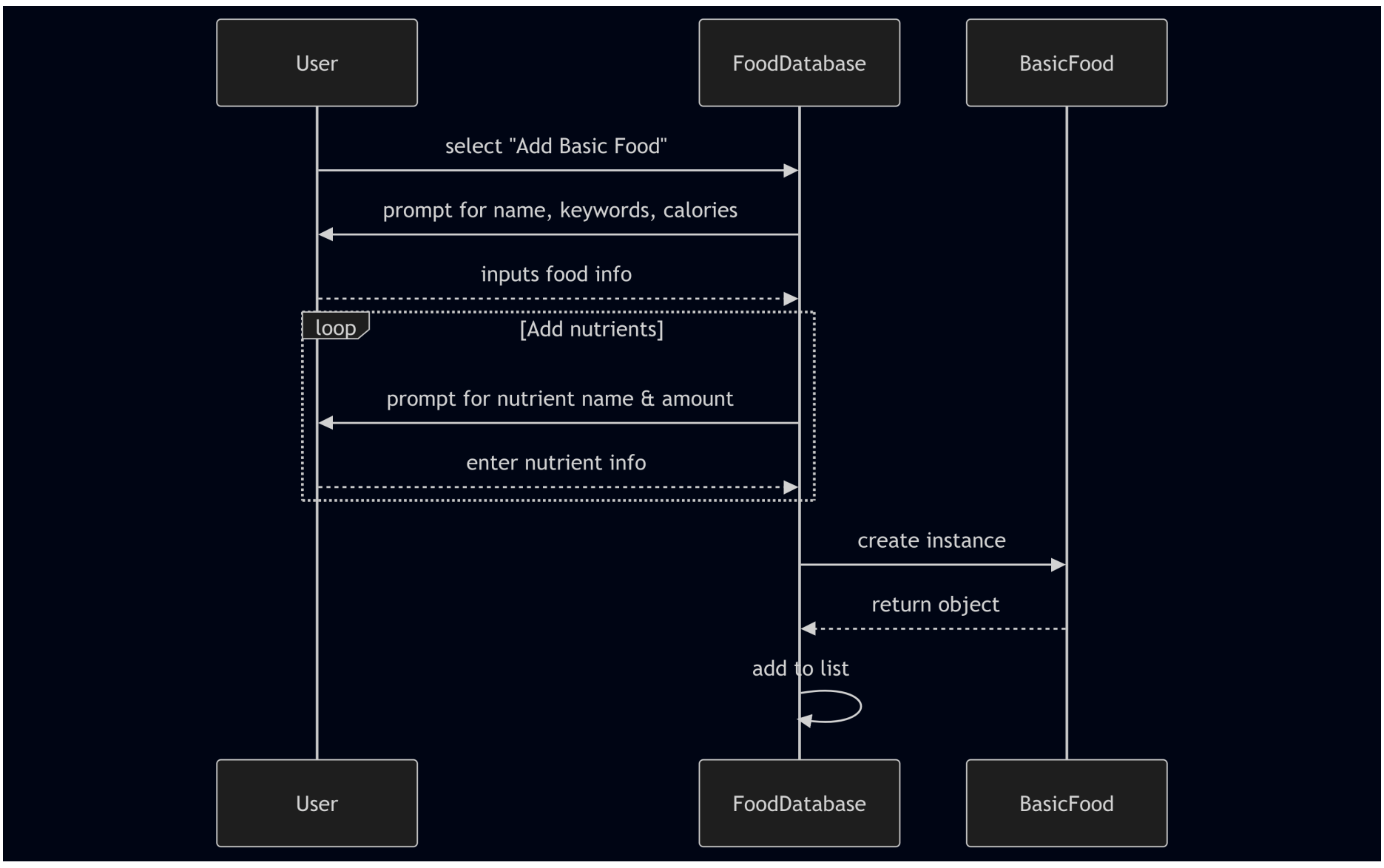


Sequence Diagrams (Mermaid)

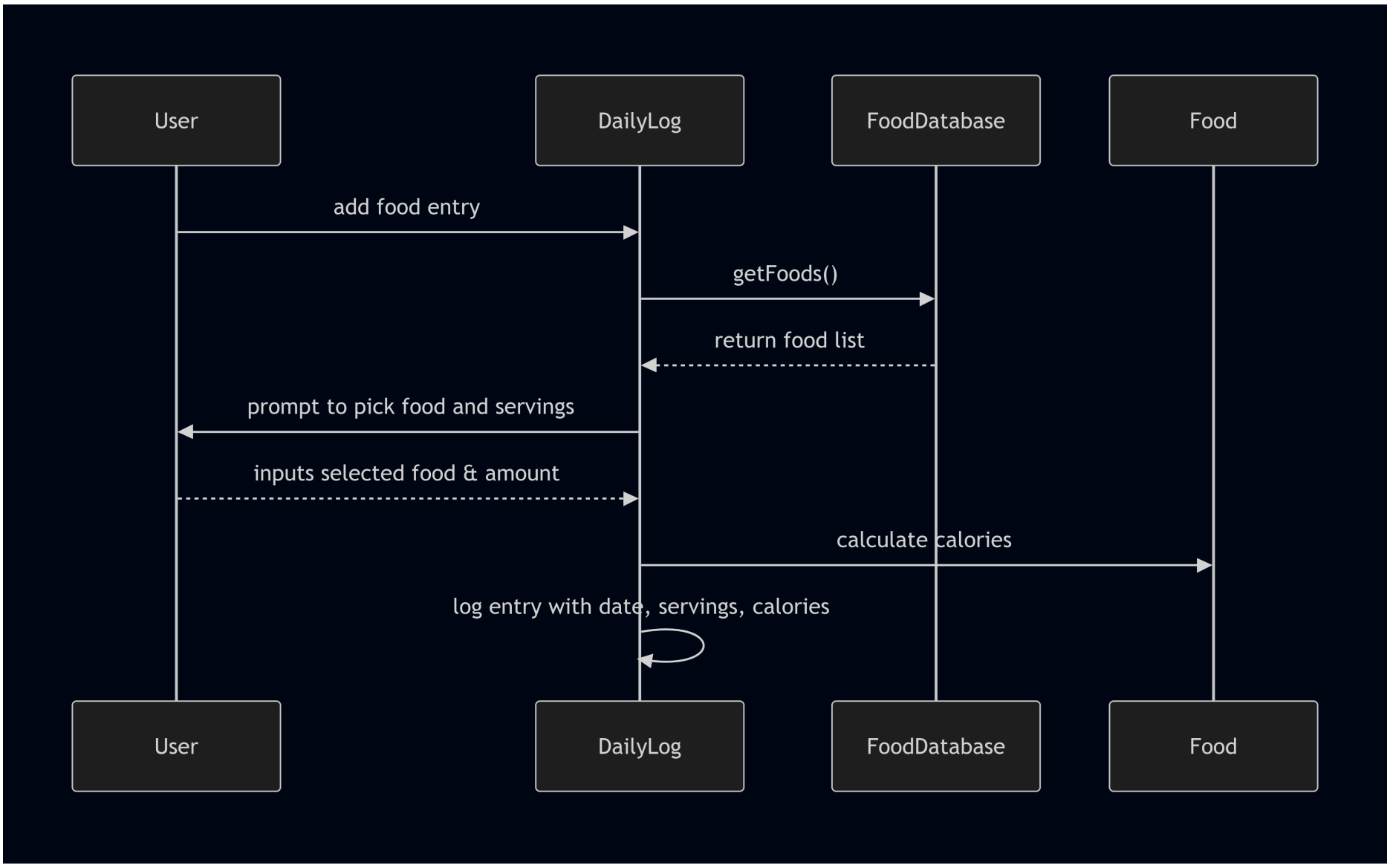
1.Registration



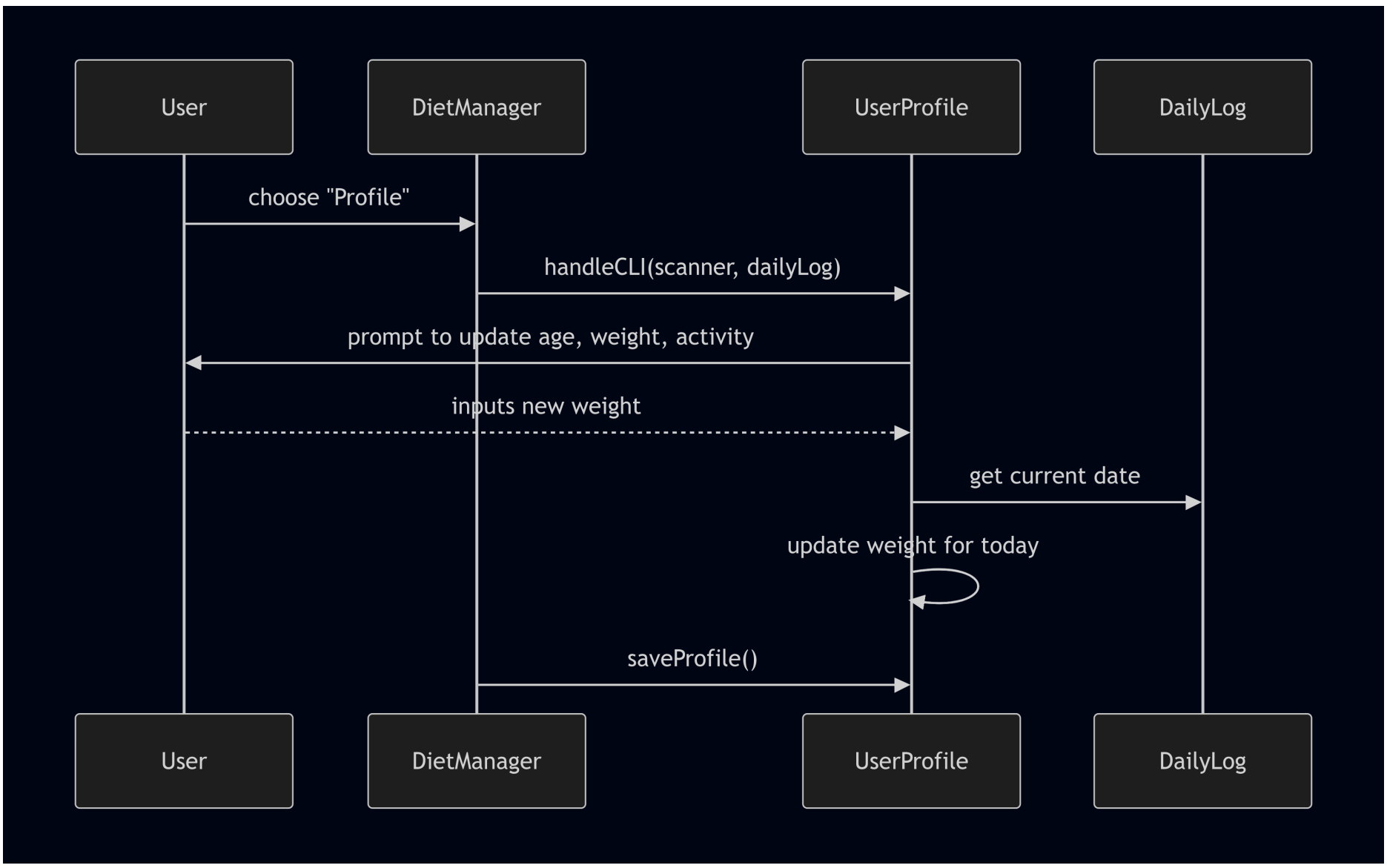
2. Adding a Basic Food



3. Food entry in Daily Log



4. Updating profile Details



Design Principles Narrative

The design of **DietManager** demonstrates a thoughtful balance among several core software design principles:

1. Low Coupling

Each major module (`DailyLog`, `FoodDatabase`, `UserProfile`) interacts through well-defined method calls and avoids tight interdependencies. For example, `DailyLog` accesses food items through the `FoodDatabase`'s `get_Foods()` method, rather than directly manipulating its internal list. This allows the modules to be modified or tested independently, supporting easier maintenance and future extension.

2. High Cohesion

Classes in DietManager are internally focused and cohesive. For instance, `DailyLog` is solely responsible for managing the user's daily food log—viewing, adding, deleting, and saving logs—while `FoodDatabase` exclusively handles food-related storage, retrieval, and categorization. This single-purpose focus improves readability and testability.

3. Separation of Concerns

The application separates CLI interaction from business logic. Each module's `handleCLI()` method deals with user interaction, while the rest of the class manages logic and data. Additionally, concerns like food composition, log management, and user profiles are handled in separate modules.

4. Information Hiding

Private fields (e.g., `foods`, `logEntries`) ensure that data structures are only modified through controlled interfaces. `FoodDatabase` returns copies of internal lists to avoid external modifications, preserving encapsulation and guarding against unintended side effects.

5. Law of Demeter

The design mostly adheres to the Law of Demeter (only talk to your immediate friends). For example, when a user adds a food to a daily log, the `DailyLog` module doesn't directly access internal fields of `FoodDatabase` or `Food`; instead, it uses methods like `findFoodByName()` and `getName()`.

There are a few minor violations, such as accessing nested structures in `CompositeFood` (e.g., `components.get(food)`), but these are constrained to appropriate places and don't significantly increase coupling.

6. Extensibility and Open/Closed Principle

The system is designed to be extensible—for example, `BasicFood` supports additional nutritional attributes via a `Map<String, Double>`, making it easy to include other values like fiber, sugar, or sodium. The use of the `Food` interface also allows for additional food types (e.g., `MealPlan`, `Recipe`) to be added without changing existing code.

Reflection

Two Strongest Aspects

1. **Modular Architecture with Clear Separation of Concerns**
   - Each core feature—logging, food management, and user data—is cleanly separated into different classes. This makes the system easier to navigate, test, and extend.
2. **Support for Composite Foods and Nutritional Flexibility**
   - The ability to combine basic foods into composite foods, along with a flexible nutritional map, showcases thoughtful extensibility. It models real-world complexity without bloating the codebase.

Two Weakest Aspects

1. **Lack of Persistent Undo Stack**
   - Although an undo feature is implemented using a stack, it is memory-resident only. Once the application closes, undo history is lost. This limits usability for longer sessions or accidental quits.
2. **Limited Input Validation and Error Handling**

- The system lacks robust handling for incorrect inputs (e.g., non-numeric servings or invalid dates). This could lead to crashes or invalid data being saved, which reduces the program's robustness and user-friendliness