Design Document

Assignment - 3: Yet Another Diet Assistant

Kriti Gupta (2023101123) and Ananya Vishal Halgatti (2023101129)

Overview

YADA (Yet Another Diet Assistant) is a prototype diet management software designed to assist users in monitoring and improving their dietary habits. It addresses the widespread issue of overeating by allowing users to track their daily food intake, manage their personal diet goals, and understand their caloric balance over time. The system is implemented in Java and designed with extensibility, modularity, and data persistence in mind.

The prototype includes the following key features:

- **Basic and Composite Food Database:** Users can view, search, and add new basic foods by entering identifiers, keywords, and calories. Additionally, they can define composite foods by combining existing items, with calorie computation handled automatically.
- **Daily Log Management:** The system maintains daily logs of consumed foods, allowing users to add, delete, and view food entries by date. Logs support both keyword filtering (all/any match) and undo/redo functionality for command reversals.
- **User Profile and Calorie Goals:** The program stores personal details like gender, age, height, weight, and activity level, which can be updated daily. Caloric needs are computed using one of two supported methods - `Harris-Benedict and Mifflin-St Jeor`, switchable by the user.
- **Data Persistence:** All data (foods, logs, and user profiles) are stored in plain-text files, making them human-readable and editable. The system loads this data at startup and can save it on demand or at exit.
- **Extensible Food Import System (Design Concept):** To support future integration with a variety of online food data sources (e.g., McDonald's nutrition API or global diet databases), the system is designed to be **easily extensible** using the **Adapter Design Pattern**. This ensures that new sources can be added without requiring changes across the entire system.

    Design Motivation: Each food database or API has its own format for representing data — such as differing JSON structures, parameter names, or units. Hard-coding logic for each source into the core system would result in tightly coupled code that is difficult to maintain and extend.

The Adapter Pattern Solution :

To abstract away these differences, a **`FoodDataAdapter` interface** is defined:

```
public interface FoodDataAdapter {
    boolean supports(String sourceIdentifier);
          List<BasicFood>  fetchFoodData(String  query)  throws
FoodImportException;
}
```

Each food source implements its own adapter class by implementing this interface. For example, a `McDonaldsAdapter` would handle calls to the McDonald's nutrition API, parse the response, and return the result in a standardized `BasicFood` format used internally by the application.

Example:

```
public class McDonaldsAdapter implements FoodDataAdapter {
    @Override
    public boolean supports(String sourceIdentifier) {
        return "mcdonalds".equalsIgnoreCase(sourceIdentifier);
    }

    @Override
    public List<BasicFood> fetchFoodData(String query) {
          // Fetch from McDonald's API and parse into BasicFood
objects
    }
}
```

Future sources like USDA or Healthline can be added as new adapters by simply implementing the same interface, without touching any other part of the system.

Centralized Adapter Management

The application maintains a list of registered adapters. When a user selects a source, the system delegates the request to the appropriate adapter:

```
for (FoodDataAdapter adapter : registeredAdapters) {
    if (adapter.supports(sourceIdentifier)) {
        return adapter.fetchFoodData(query);
    }
}
```

This keeps the logic **centralized**, **modular**, and **easily extensible**.

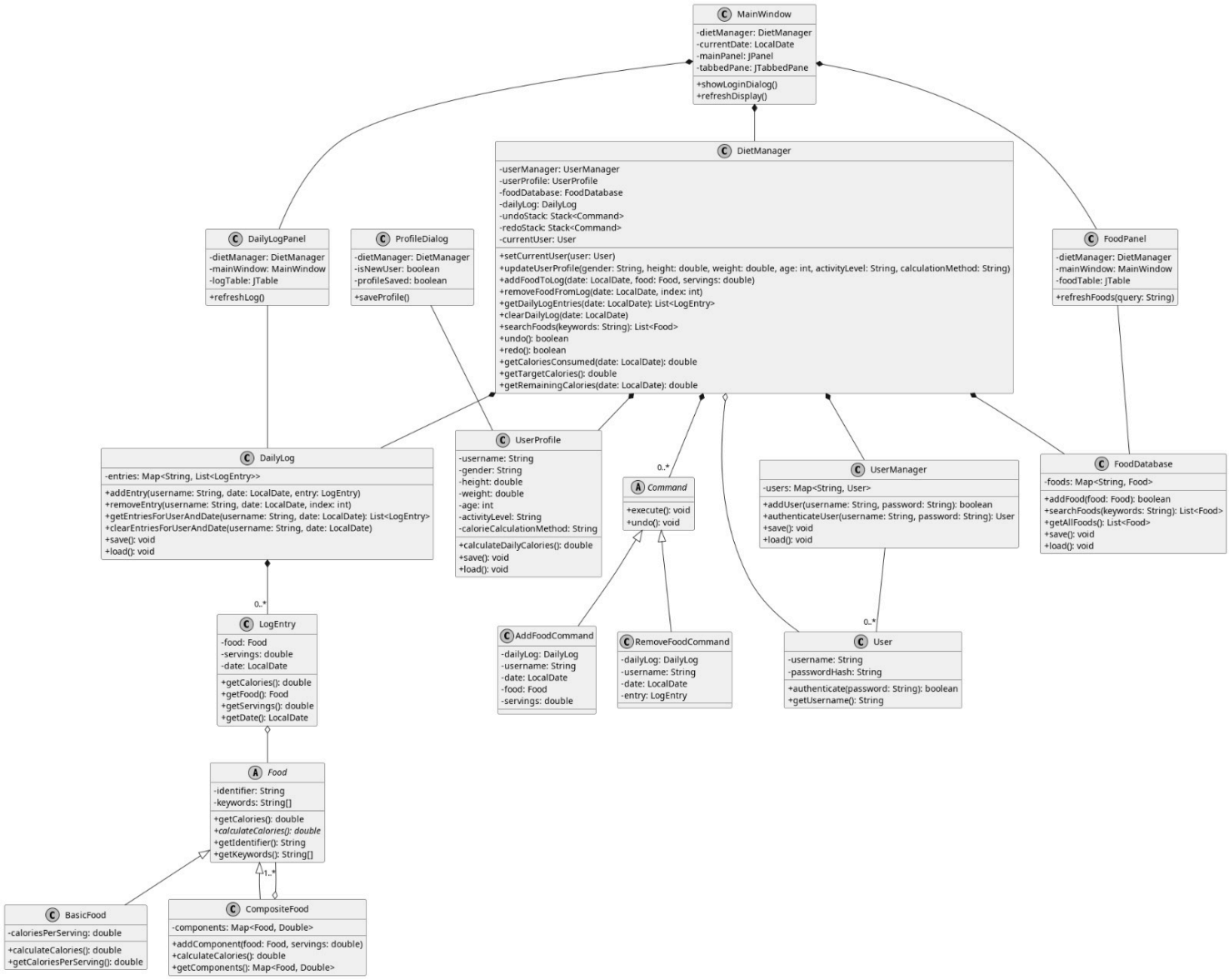
Benefits

- **Open/Closed Principle**: New data sources can be integrated without modifying existing code.
- **Separation of Concerns**: Each adapter handles only its own source's logic.
- **Future-Proofing**: Ready to support any number of websites or APIs.
- **UI Simplicity**: The UI can allow users to select a source, and the system routes the request transparently.

Prototype Note

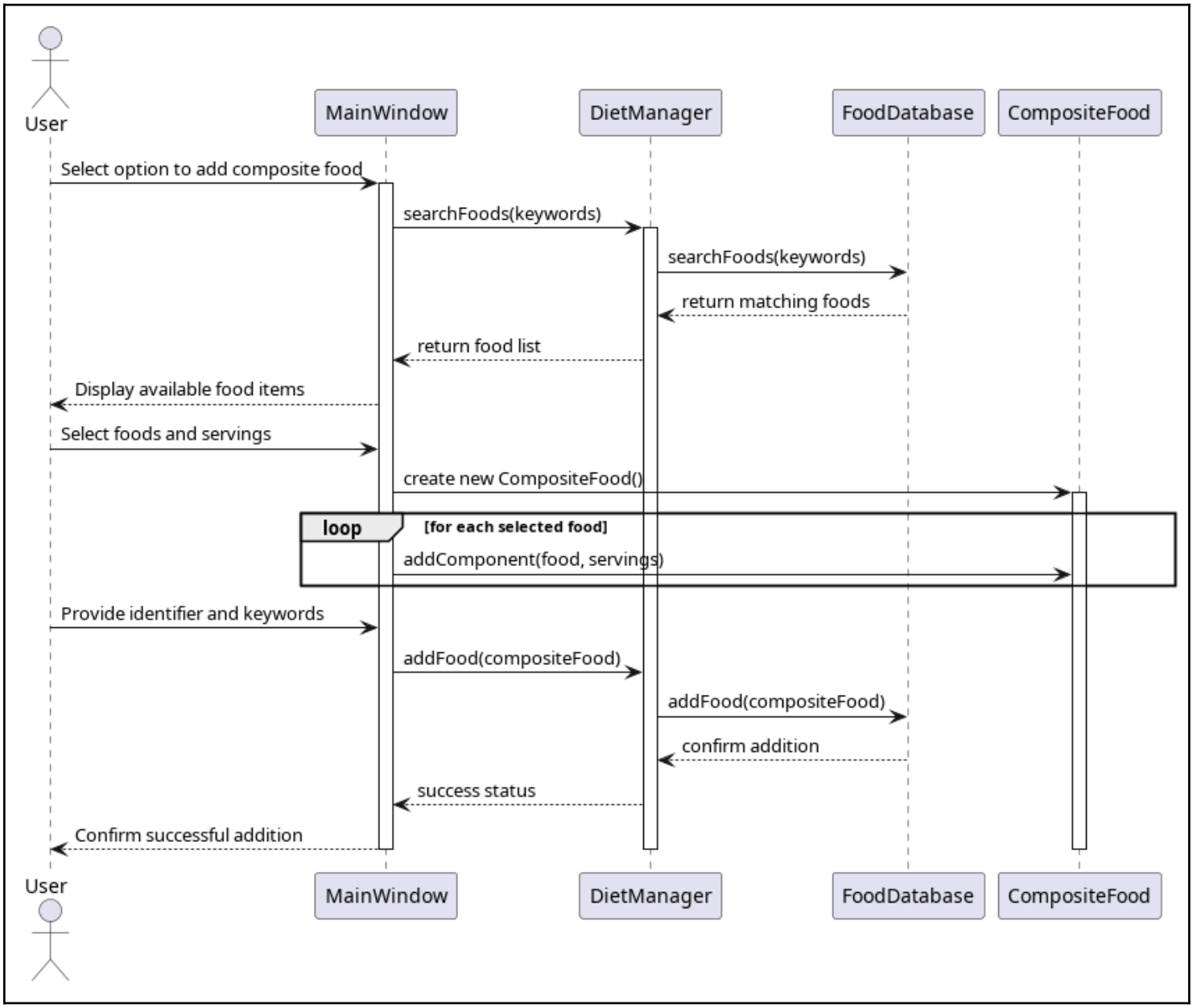
While this design is not yet implemented in the current prototype, the system is structured to accommodate it in the future. At present, users can manually input new foods by entering identifiers, keywords, and calories via the UI. The adapter framework ensures that when online food sources are introduced, their addition will be smooth and maintainable.

UML Class Diagram

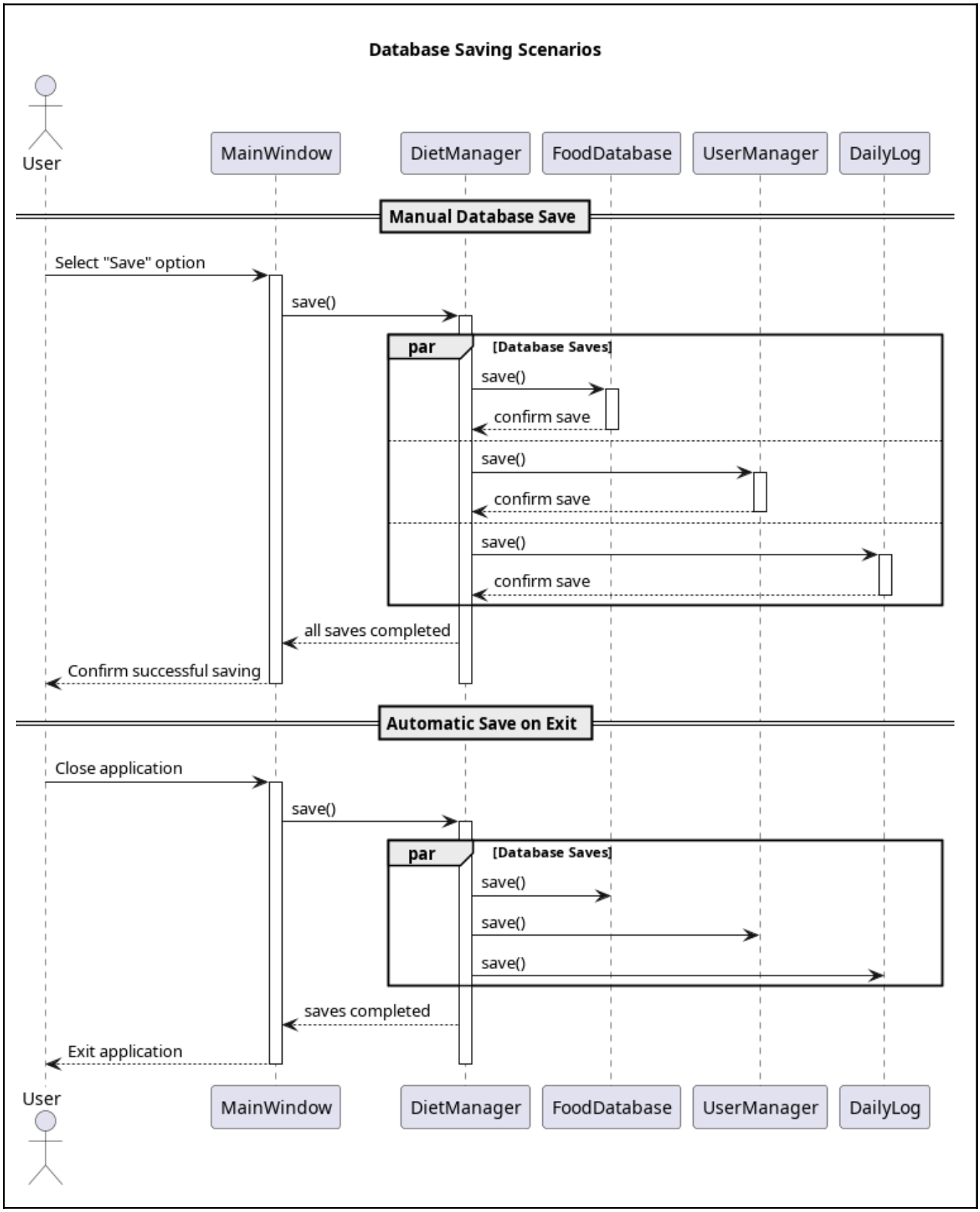


Sequence Diagrams

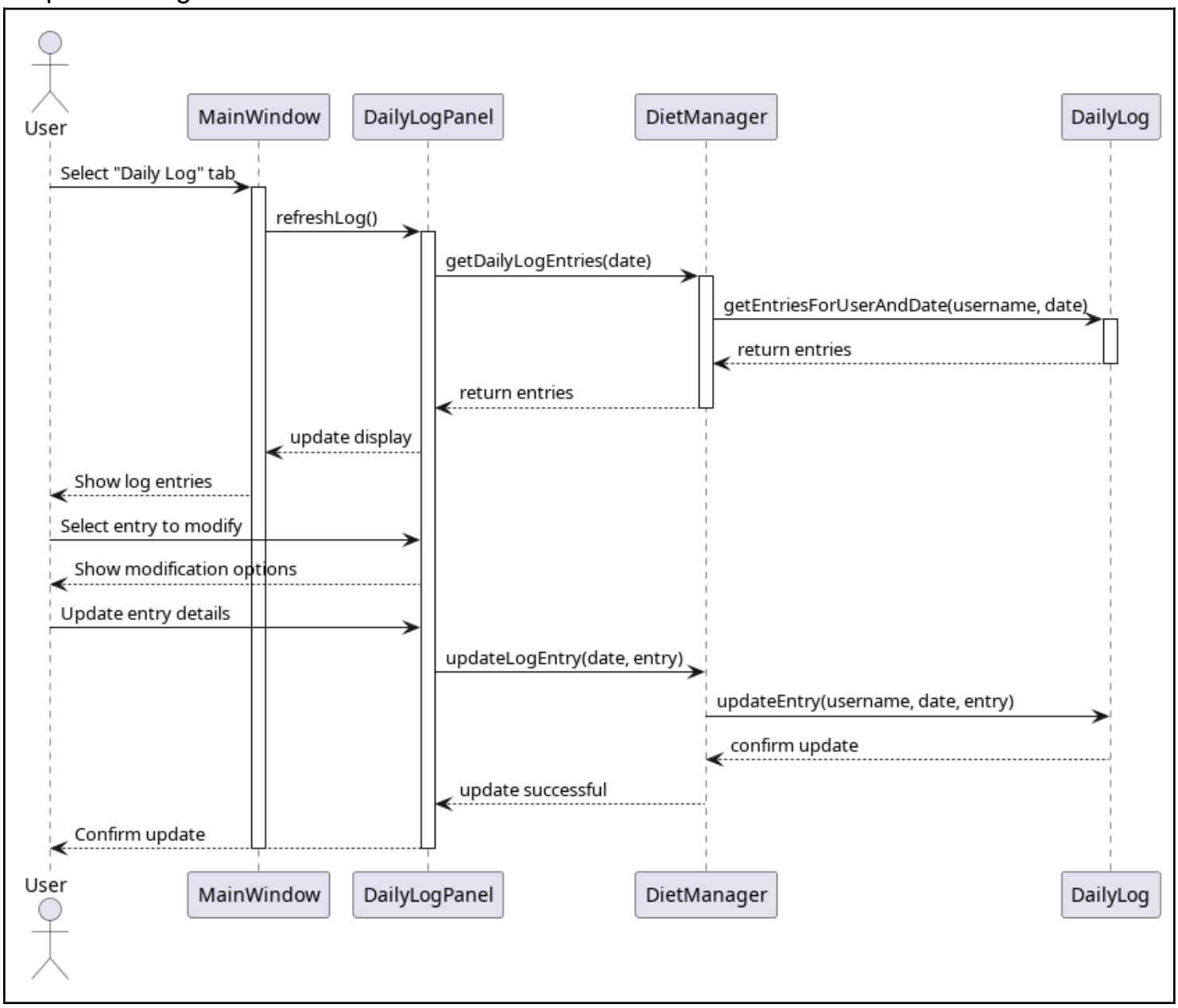
Sequence Diagram 1: Based on use case 2



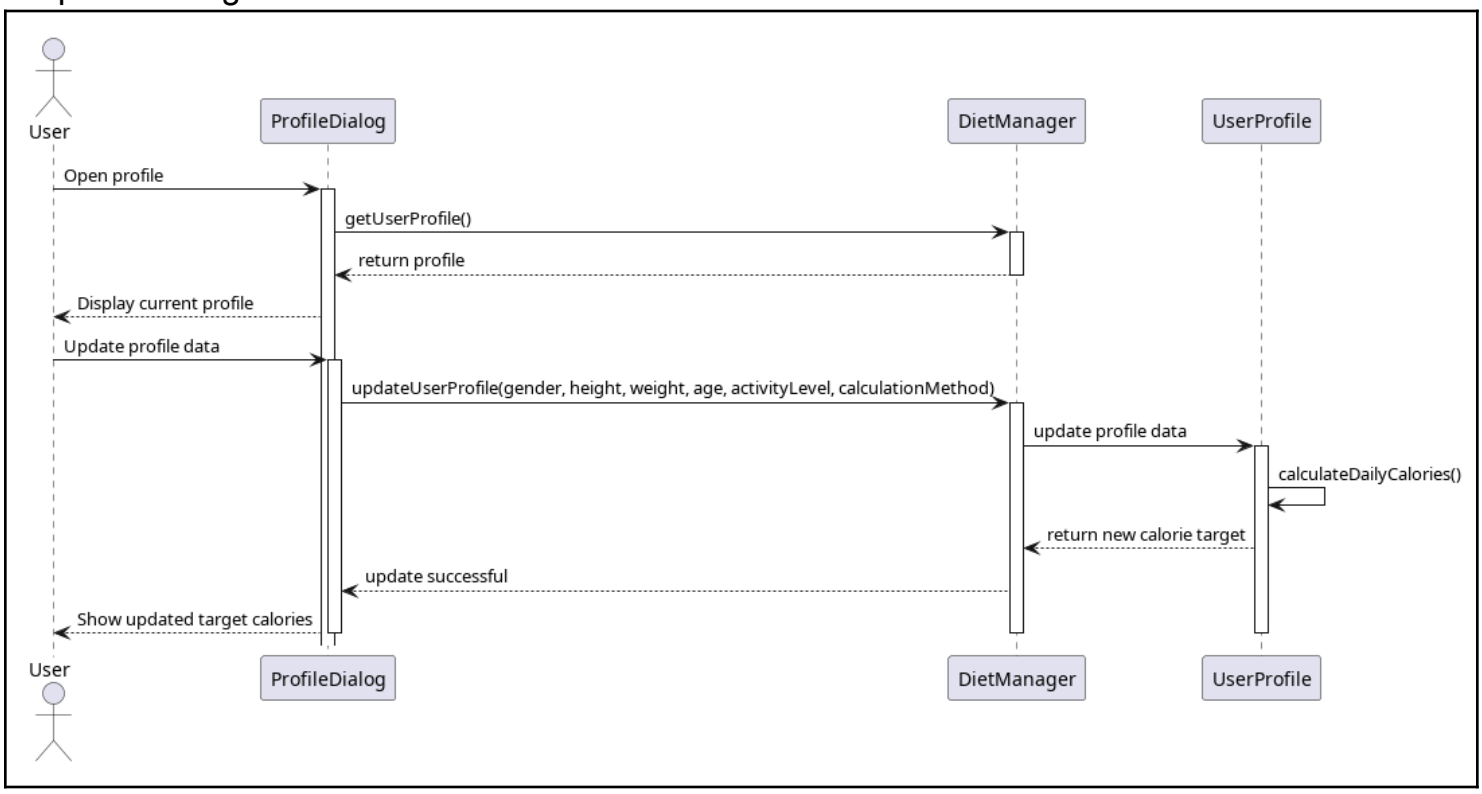
Sequence Diagram 2: Based on use case 3 and 4



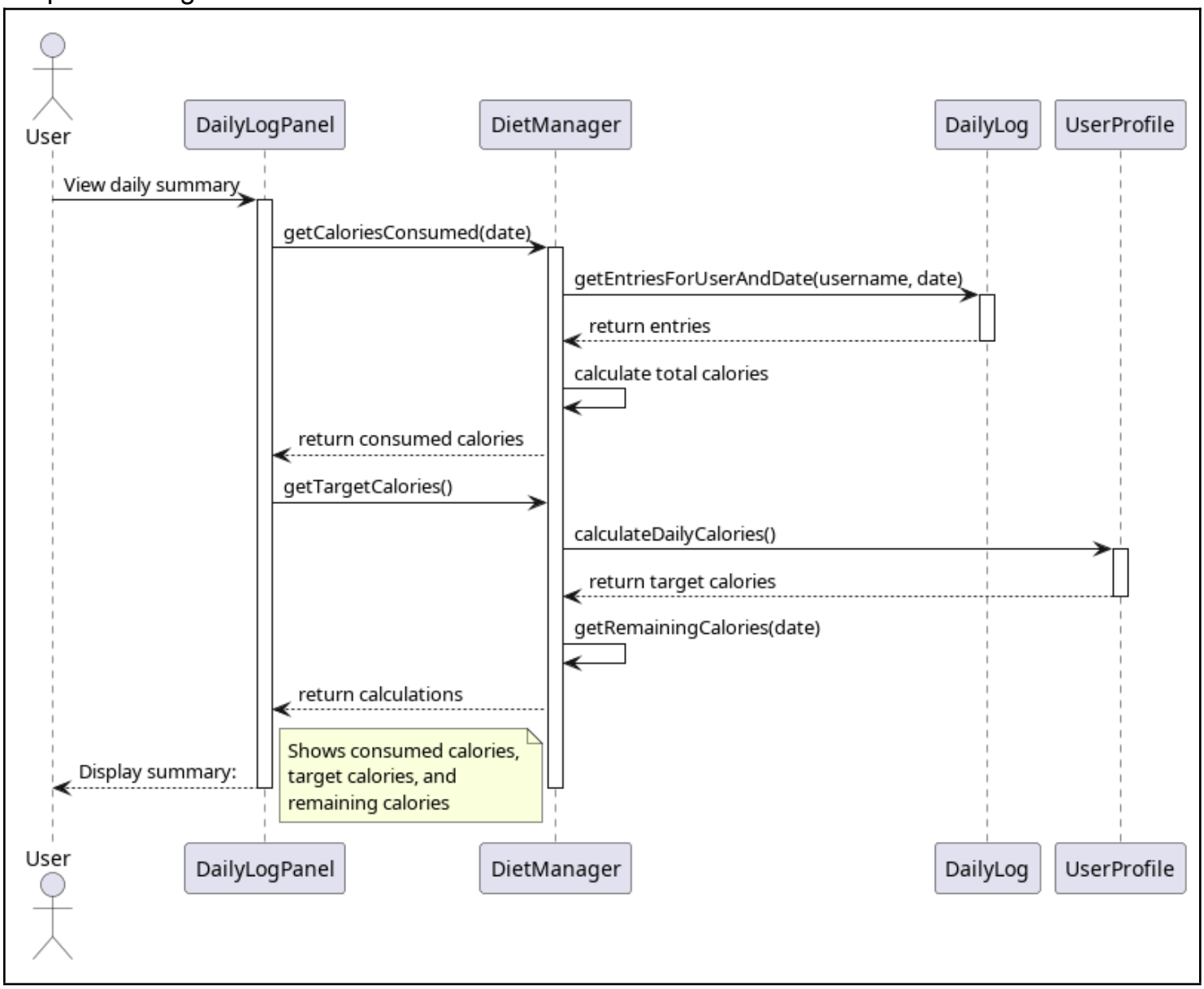
Sequence Diagram 3: Based on use case 6



Sequence Diagram 4: Based on use case 8



Sequence Diagram 5: Based on use case 9



Design Principles Narrative

The design of the DietManager system reflects a thoughtful balance among several fundamental object-oriented design principles, including **low coupling**, **high cohesion**, **separation of concerns**, **information hiding**, and adherence to the **Law of Demeter**.

Low Coupling

The system minimizes dependencies between classes by defining clear, stable interfaces and interaction points. For instance, the `FoodDataAdapter` interface decouples the core application logic from external

food data sources. This means new data sources like restaurant APIs or diet databases can be integrated by implementing this interface, without requiring any changes to other parts of the system. This approach makes the system easier to maintain and extend, and reduces the risk of unintended side effects.

High Cohesion

Each component in the system is designed to handle a specific, well-defined responsibility. For example, the `UserProfile` class encapsulates all logic related to user attributes, goals, and calorie computations. The UI panels such as `DailyLogPanel` and `FoodPanel` are focused solely on rendering and managing user interactions. This high cohesion ensures that components are focused and logically organized, which improves readability and testability.

Separation of Concerns

The architecture is structured to separate different concerns across packages:

- `com.yada.model` handles the domain logic,
- `com.yada.ui` manages the user interface,
- `com.yada.user` focuses on user-related operations, and
- `com.yada.util` includes reusable utility logic, such as adapters for third-party APIs. This clear separation improves modularity, allowing developers to modify one aspect of the system (e.g., UI) without affecting others (e.g., model or backend logic).

Information Hiding

Implementation details are hidden behind public interfaces. For example, the way food data is fetched from external sources is abstracted through the `FoodDataAdapter` interface. Classes expose only what is necessary, minimizing the public API surface and reducing the risk of misuse. Internals such as how a `BasicFood` object stores its nutrients are hidden from external consumers, who interact only through well-defined methods.

Law of Demeter

The system avoids “train wreck” calls (e.g., `a.getB().getC().doSomething()`) by encouraging direct communication between neighboring objects. UI components do not dig into internal structures of the model or backend services but rely on provided interfaces. This promotes a disciplined structure where objects only interact with their immediate “friends,” improving flexibility and reducing breakage when the internal structure of a class changes.

Design Reflection

Strongest Aspects

1. **Well-Isolated Functional Layers**
   The architecture clearly delineates between core modules such as data models, UI components, user management, and utility services. This high cohesion within modules and loose coupling between them enhances maintainability and reduces the ripple effect of changes. It also supports Separation of Concerns, making each part easier to test, reason about, and independently evolve as the application grows in complexity.

2. **Plugin-Friendly Architecture via Adapter Integration**
   A standout strength of our design lies in the forward-thinking integration of the Adapter Pattern, allowing for seamless expansion to external food data providers (e.g., McDonald's). By defining a common `FoodDataAdapter` interface, we enable third-party APIs to "plug into" our system without altering any existing business logic. This not only embodies the Open/Closed Principle, but also demonstrates proactive thinking for real-world production scaling and integration flexibility.

Weakest Aspects

1. **Reactive Instead of Defensive Programming**
   The current implementation lacks robust checks for malformed or missing inputs, especially in areas such as profile file parsing and user data entry. The absence of a centralized validation mechanism means that errors are caught only when exceptions occur. Adopting a more defensive programming stance with proper validation and error messages would elevate the reliability and user-friendliness of the system.

2. **Rigid and Unscalable UI Design**
   The graphical interface, though functional, is primarily hardcoded and lacks adaptability to varying user environments or accessibility needs. Features like dynamic component resizing, keyboard navigation, or theming are not supported. For long-term maintainability, it would benefit from a migration toward a more responsive, MVC-based GUI framework or even a web-based frontend that can cater to a wider user base.