Dass Assignment 3

YADA - Yet Another Diet Assistant

Design Document

Date: 7 April 2025

Saloni Goyal – 2023115001

Shravani K - 2023115009

Overview

Yet Another Diet Assistant (YADA), the Daily Diet Assistant is a Java-based desktop application designed to support users in tracking their daily dietary intake and managing nutritional habits over time. Developed using the Java Swing framework for the graphical user interface and grounded in object-oriented design principles, the system provides a robust and modular platform for food logging, calorie tracking, and personal health monitoring.

The application enables users to log individual and composite food items, view summarized nutritional information, and review logs for any recorded date. It integrates persistent file-based storage to ensure continuity of data between sessions and offers an intuitive interface that encourages consistent user engagement.

The principal features of the system include:

Food Entry with Serving Quantification
Users can add food items from a predefined database, specify serving quantities, and have nutritional values automatically computed based on the food's definition. Both basic and composite foods are supported, enabling flexible dietary recording.

Search Functionality Using Keyword Matching
The system incorporates a dynamic search mechanism that allows users to locate food items either by name or through associated keywords. The matching behavior can be configured to match either any or all provided keywords, and is implemented using the Strategy design pattern to allow for extensible search behavior.

Persistent Storage and Log Management
All food log entries are stored in a plain-text file (log.txt) in a structured format that is both machine-readable and human-readable.

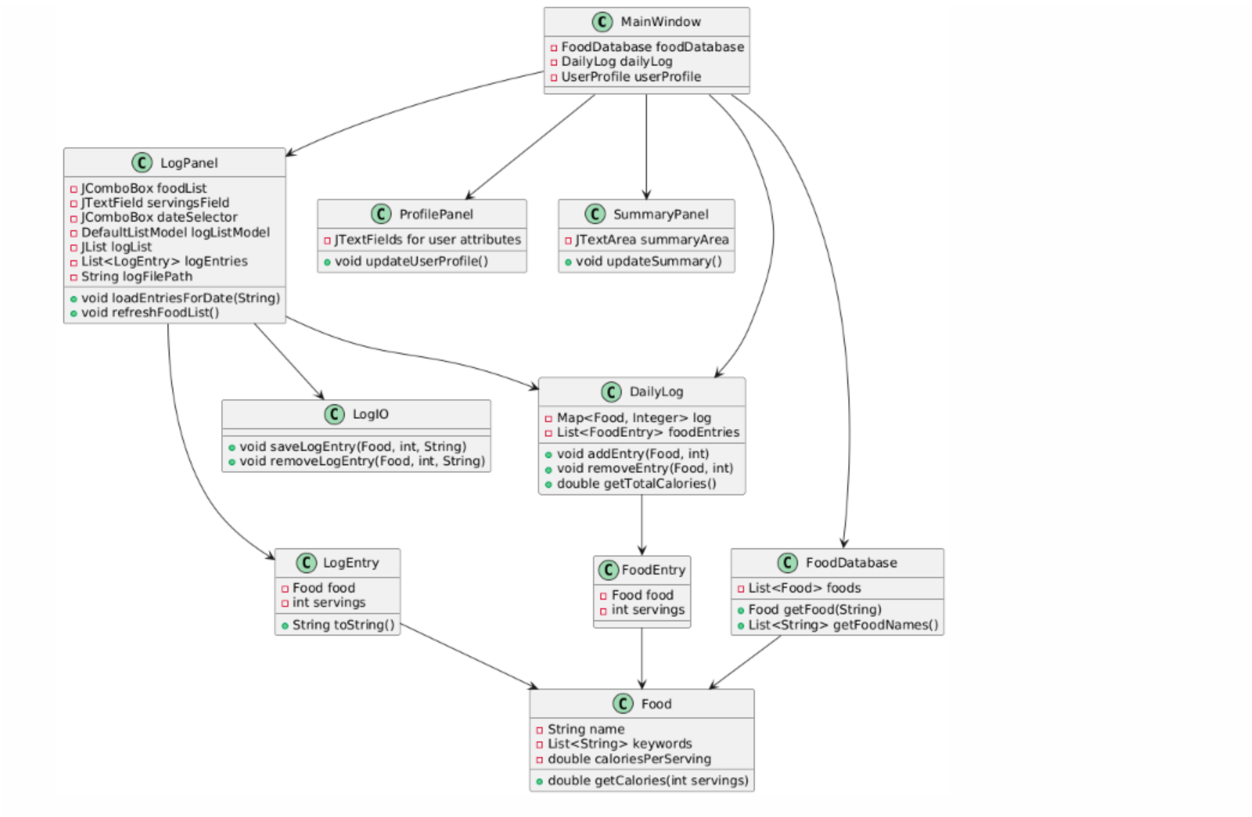
User Profile and Caloric Summary
The application maintains a user profile containing personal information such as age, weight, height, and caloric goals. The system computes a daily summary of caloric intake and compares it against the user’s goal, providing a concise overview of dietary progress.

Architectural Objectives
The design emphasizes low coupling, high cohesion, and a clear separation of concerns across GUI, model, and logic layers. It adheres to object-oriented design principles,

including the use of abstraction, encapsulation, and polymorphism. The application architecture is modular, extensible, and maintainable, supporting future enhancements such as integration with online nutritional databases or mobile platforms.

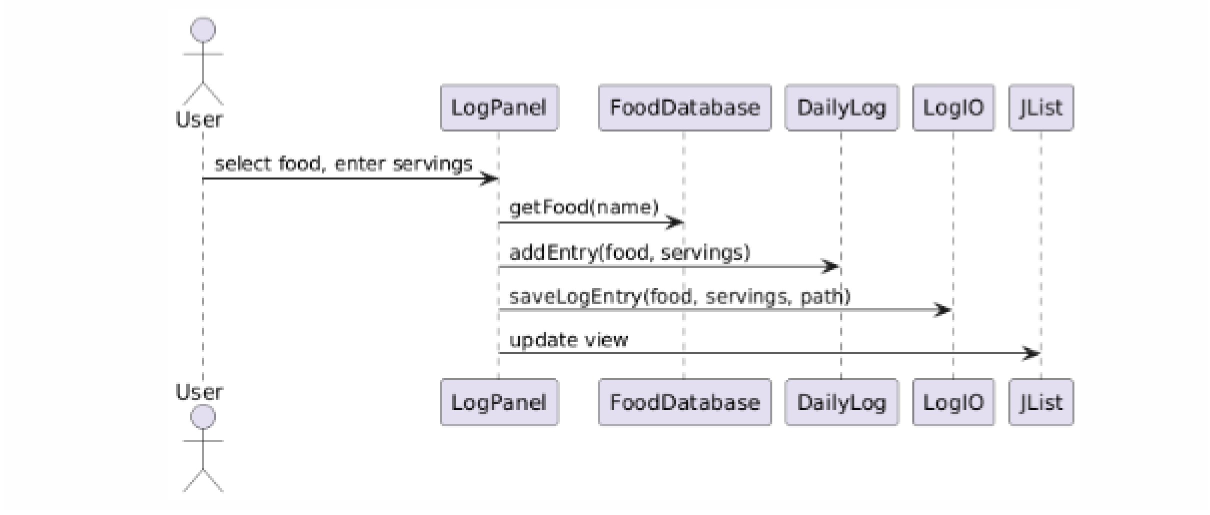
In summary, the Daily Diet Assistant offers a structured, extensible, and user-centric platform for dietary tracking. It serves as both a functional personal health tool and a demonstrative application of software engineering principles in a real-world use case.

UML Class Diagram

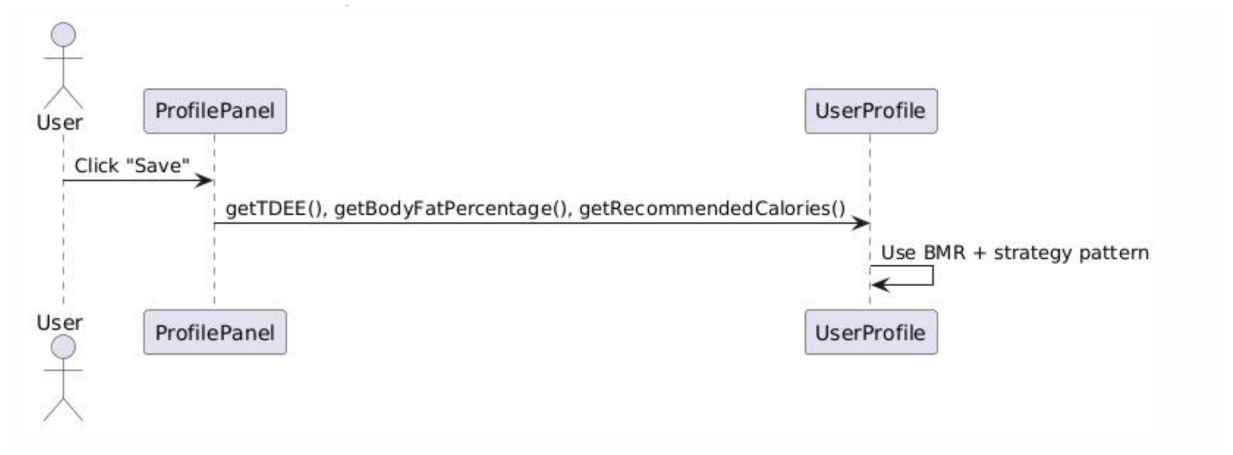


Sequence Diagrams

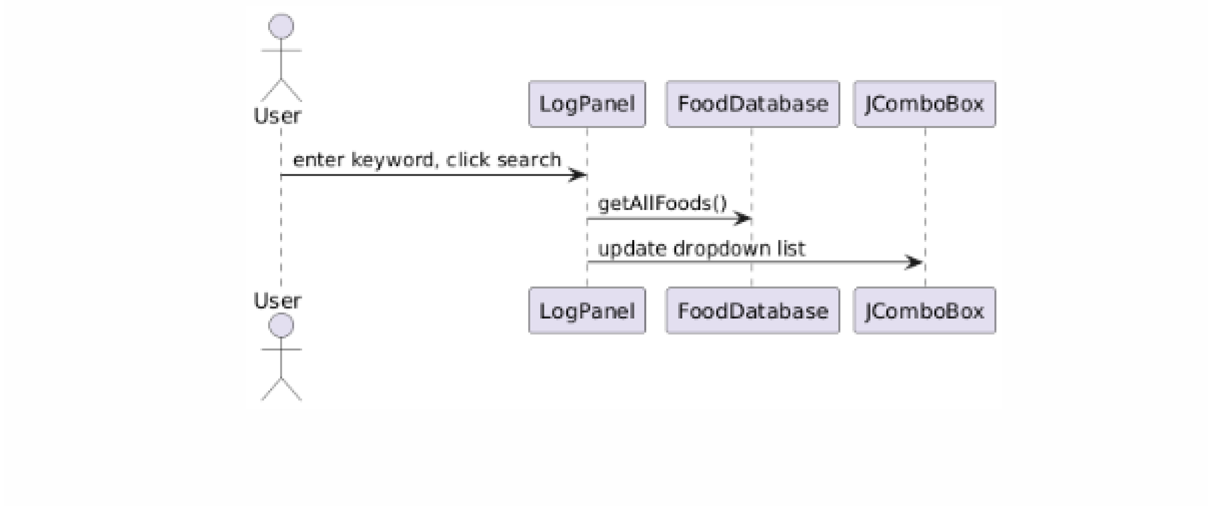
1. Add Food Entry



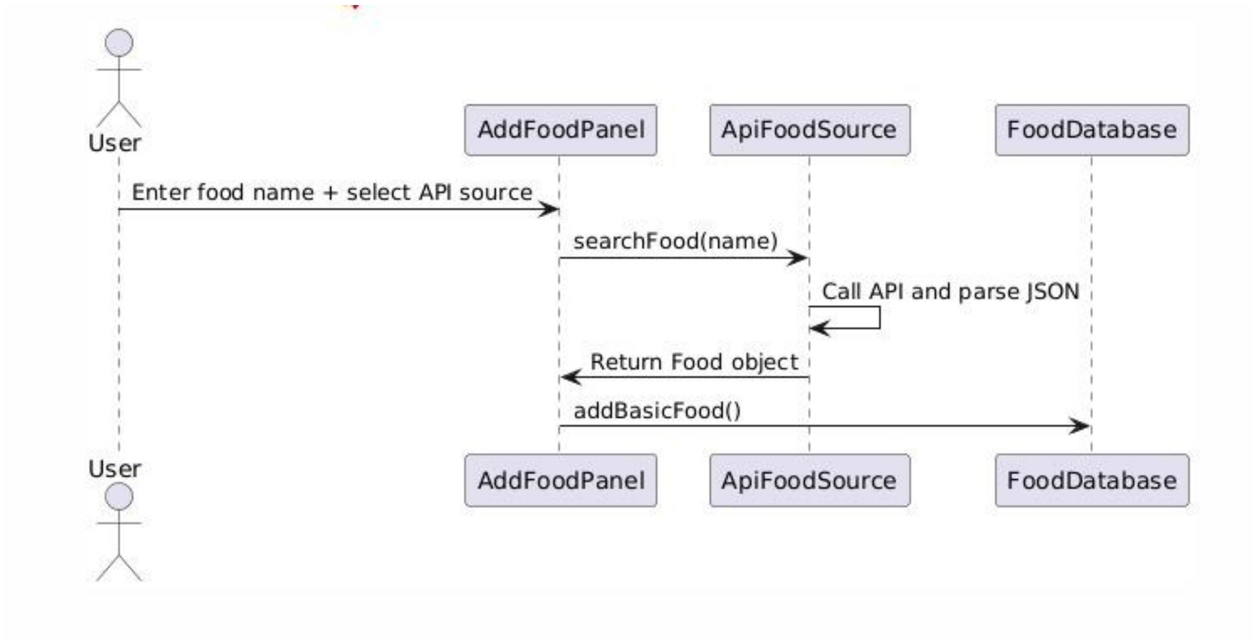
2. System Calculates Recommended Calories


User
ProfilePanel
UserProfile
Click "Save"
getTDEE(), getBodyFatPercentage(), getRecommendedCalories()
Use BMR + strategy pattern
User
ProfilePanel
UserProfile

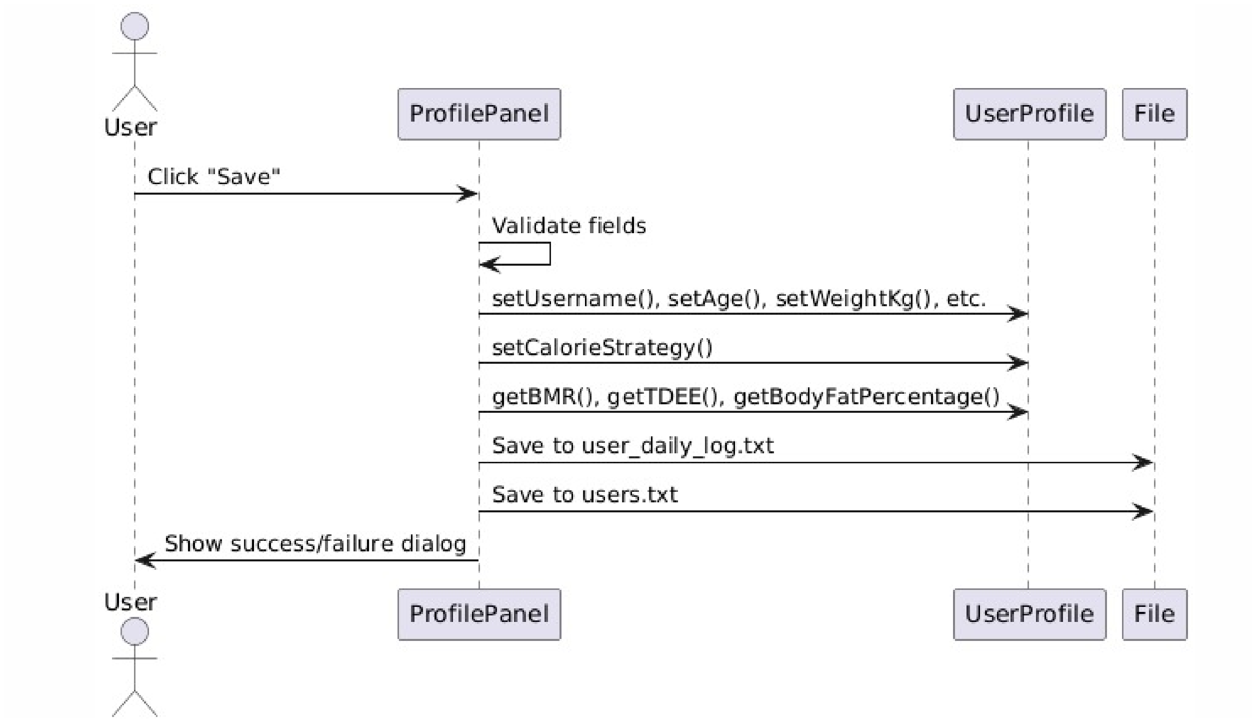
3. Search Foods by Keyword



4. User Adds Food via API Lookup



5. User Saves Profile


User
ProfilePanel
UserProfile
File
Click "Save"
Validate fields
setUsername(), setAge(), setWeightKg(), etc.
setCalorieStrategy()
getBMR(), getTDEE(), getBodyFatPercentage()
Save to user_daily_log.txt
Save to users.txt
Show success/failure dialog
User
ProfilePanel
UserProfile
File

Design Principles and Quality

1. New ways of computing target calories must be easy to add without ripple effects throughout the program.

Soln used: We used the **Strategy Pattern** to make calorie calculations flexible and modular. Each method (like BMR-based, activity-adjusted, or weight-loss focused) is in its own class, all following a common interface. The main program uses this interface, so we can add new strategies without changing existing code. This keeps the app easy to maintain and extend.

2. New ways of computing target calories must be easy to add without ripple effects throughout the Program.

Soln used: We use the **Strategy Pattern**, just like in the first case. Different calorie calculation methods are implemented as separate strategies following a common interface. This makes it easy to add or switch strategies without changing the rest of the code.

3. The log file may grow to be quite large; for this reason, using approaches that reduce or eliminate duplicate copies of objects is highly desirable (this is a very broad hint).

Soln used: We used the **Flyweight Pattern** to avoid storing the same food data multiple times. Instead of each log entry copying full food info (like "Banana"), it just references a shared object. This reduces memory usage and keeps the app efficient, even with lots of data.

YADA reflects design practices emphasizing modularity and clarity:

- The design of the Daily Diet Assistant (YADA) demonstrates adherence to fundamental software engineering principles intended to promote maintainability, reusability, and clarity across the system's architecture. Key design goals such as modularization, clarity of responsibility, and decoupling are consistently reflected in the implementation.
- Low Coupling
  Each major component (LogPanel, SummaryPanel, ProfilePanel) interacts with shared model classes (e.g., DailyLog, UserProfile) but maintains independence from the internal behavior of other components. This promotes isolation of concerns and ensures that changes in one module (e.g., the food log) do not cause unintended effects to others (e.g., the profile panel). For instance, LogPanel invokes DailyLog's interface without assuming knowledge of its internal data structures.
- High Cohesion
  Classes are designed to perform well-defined, focused tasks. LogIO exclusively handles persistence-related operations such as reading and writing to the log file. DailyLog is solely responsible for managing the list of food entries for a given day. This high internal cohesion within components improves testability and simplifies reasoning about system behavior.
- Separation of Concerns
  The system architecture follows the Model-View-Controller (MVC) paradigm in spirit. GUI components are strictly responsible for user interaction, while model classes manage domain logic and data structures. File I/O operations are encapsulated in distinct utility classes such as LogIO. This separation allows

independent evolution of user interface and backend logic and facilitates potential transitions to alternate front-end frameworks or storage layers.

- Information Hiding
  The application consistently encapsulates internal data representations. Data structures such as logEntries and foodEntries are kept private within their respective classes, and access is exposed only through controlled public interfaces. This reduces the likelihood of unintended side-effects and enables the developer to modify internal implementations without impacting external modules.
- Law of Demeter (Principle of Least Knowledge)
  The system minimizes knowledge chaining, i.e., "train wrecks" such as obj.getA().getB().getC().doSomething(). Instead, interactions are streamlined and encapsulated. For instance, DailyLog provides high-level methods such as addEntry(Food, servings), abstracting away the underlying data structures and enforcing a cleaner interaction boundary.
- These principles collectively result in a codebase that is easy to navigate, scale, and debug, providing a strong foundation for both iterative development and long-term maintainability.

Reflection

Strengths:

1. **Date-Based Log Handling**: The date-filterable log interface is intuitive and supports historical data tracking without clutter.
2. **Extensibility**: The modular architecture allows for easy addition of features like undo/redo, charts, or export tools.

Weaknesses:

1. **Undo/Redo System**: The implementation is partially complete and could be refined using a command pattern or Memento-based stack.
2. **Error Handling/UI Feedback**: Error messages are minimal and rely on JOptionPane; more contextual and inline feedback would improve UX.