



YADA Diet Manager

05.04.2025

—

Sherley Sonali
2023111022

Akshitha Gayatri
2023101122

Product Description

YADA (Yet Another Diet Assistant)) is a diet management system designed to help users track their daily food intake, manage nutritional goals, and maintain a healthy lifestyle. The application provides:

- A **food database** for storing and managing basic and composite foods along with nutritional information.
- A **daily log** to record consumed foods.
- A **user profile** system that calculates daily calorie targets based on personal metrics.

Built in **Java** with a **JavaFX GUI**, YADA allows users to:

- Search, add, edit and search foods.
- Create composite foods from basic ingredients.
- Log daily consumption and view nutritional summaries.
- Adjust profile settings to recalculate dietary goals.

Key Features

1. Food Database Management

- **Basic Foods:** Store foods with details like name, keywords to search, calories, and macronutrients (protein, carbs, fats).
- **Composite Foods:** Define meals by combining existing foods (e.g., a "Peanut Butter Sandwich" made of bread and peanut butter).
- **Search & Filter:** Find foods by keywords (match *all* or *any* terms).
- **Persistence:** Saves/loads food data to/from text files (**foods.txt**).

2. Daily Food Logging

- **Add/Remove Entries:** Log foods with serving sizes for any date.
- **Undo Support:** Revert accidental deletions or changes.
- **Daily Summaries:** View total calories consumed vs. target.
- **Persistence:** Logs are saved to **daily_logs.txt**.

3. User Profile & Calorie Tracking

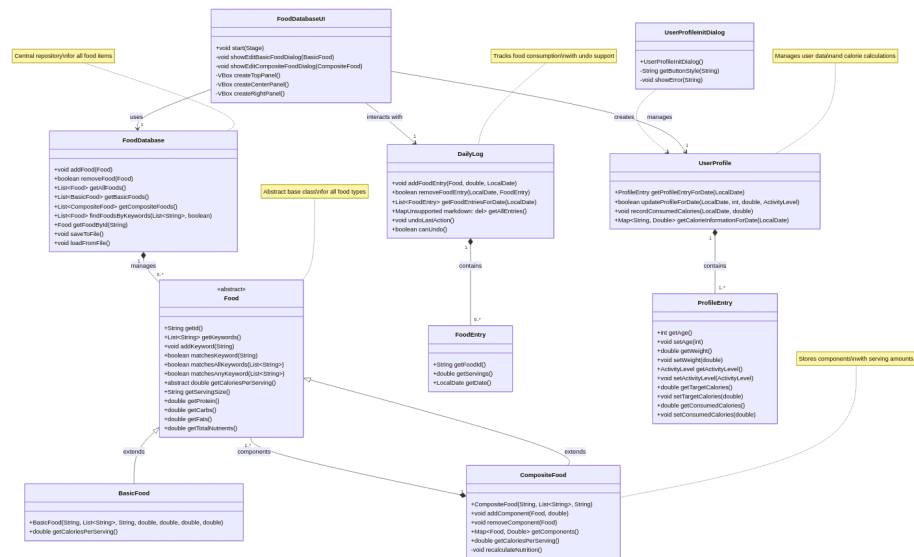
- **Profile Setup:** Collects gender, height, age, weight, and activity level.
- **Calorie Calculations:**

- Uses **Harris-Benedict** and **Mifflin-St Jeor** equations.
- Adjusts for activity level (Sedentary → Extra Active).
- **Daily Tracking:**
 - Compares consumed vs. target calories.
 - Supports historical data (entries persist across sessions).

4. GUI Features

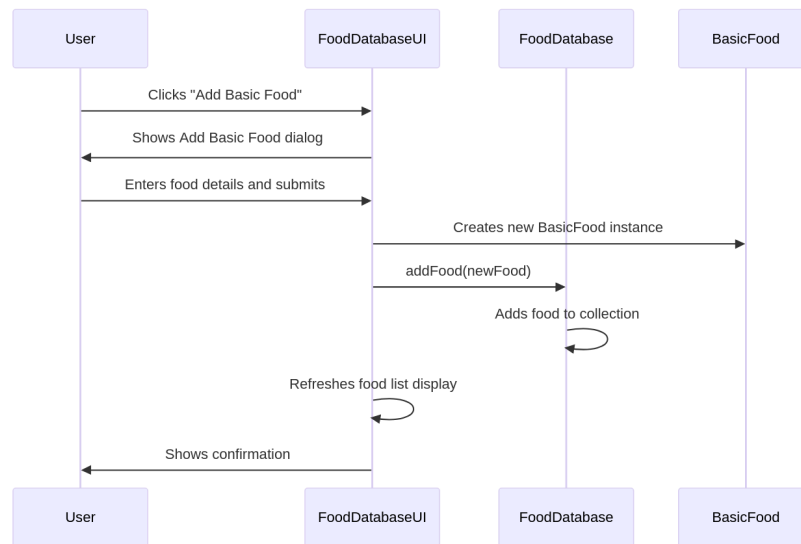
- **Food Table:** Displays all foods with filtering.
- **Food Details Panel:** Shows nutritional info for selected items.
- **Profile Dashboard:** Updates calorie stats in real time.
- **Date Picker:** Review logs for any past/future date.

UML Class Diagram

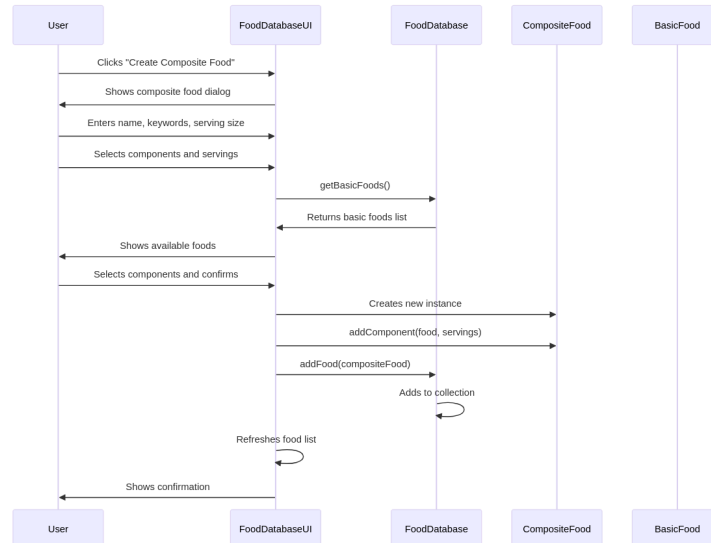


Sequence Diagrams

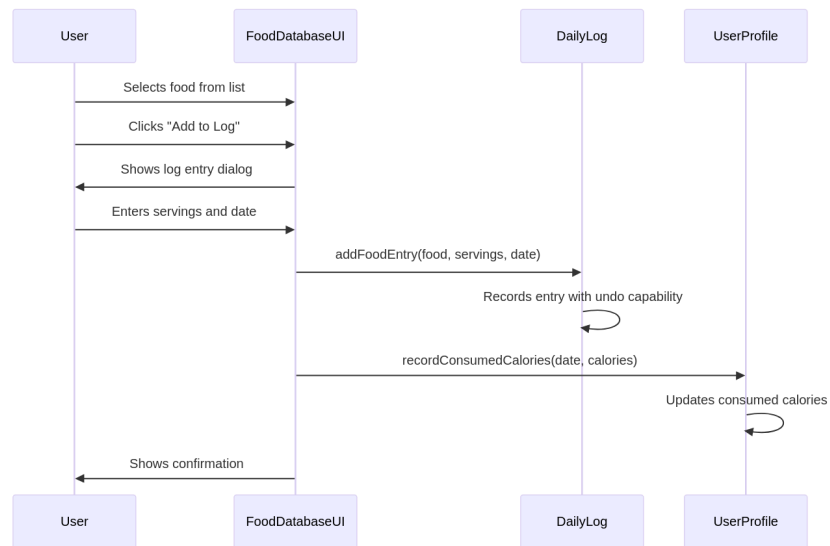
Adding Basic Food



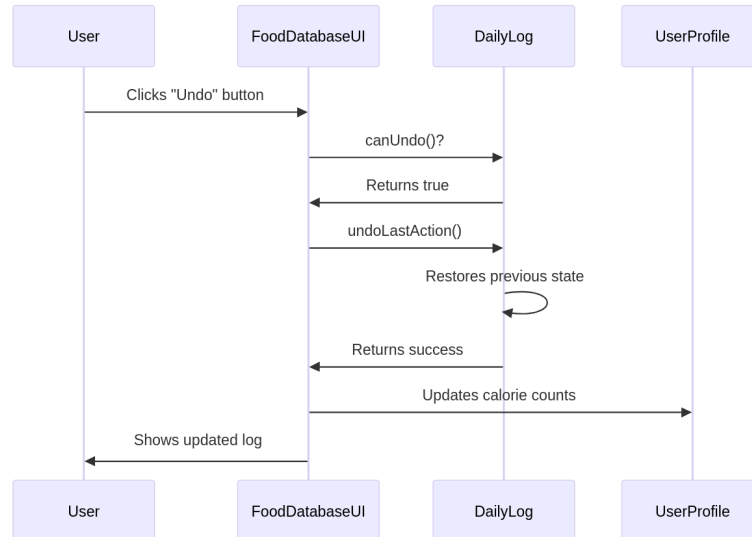
Creating a Composite Food



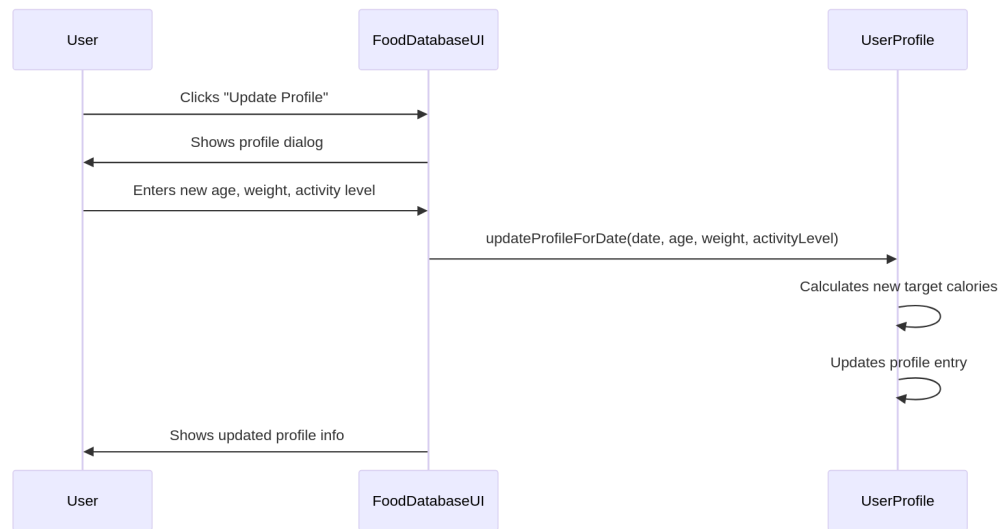
Adding Food to Daily Log



Undo Operation



Profile Update



Narrative on Design Principles

Low Coupling

The design achieves low coupling through:

- Abstract **Food** class that serves as an interface for both **BasicFood** and **CompositeFood**
- **FoodDatabase** acts as a central repository without direct dependencies on UI
- **UserProfile** and **DailyLog** are independent components that communicate through well-defined interfaces

High Cohesion

Each class has a single responsibility:

- **Food** and subclasses handle food data and calculations
- **FoodDatabase** manages food persistence
- **DailyLog** handles food logging with undo capability
- **UserProfile** manages user data and calorie calculations

Separation of Concerns

- UI logic is separated in `FoodDatabaseUI`
- Business logic is in model classes
- Persistence logic is encapsulated in respective classes

Information Hiding

- Internal data structures are private (e.g., `CompositeFood.components`)
- Access is through well-defined public methods
- Implementation details like file formats are hidden

Law of Demeter

- Objects only talk to their immediate neighbors
- UI delegates operations to model classes rather than reaching through them

Additional Design Principles Considered

Open/Closed Principle (OCP)

- Internal data structures are private (e.g., add `ProcessedFood`) requires **no changes** to existing code.
- New calorie formulas can be added via `CalorieCalculationMethod`.
- **Future proofs** the design.

Dependency Inversion Principle (DIP)

- High-level modules (e.g., `FoodDatabaseUI`) depend on abstractions (e.g., `Food` interface), not concrete implementations.
- The `FoodDatabase` depends on the abstract `Food` class rather than specific subclasses.

Key Design Points

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

This requirement is well satisfied through:

- The `CalorieCalculationMethod` enum in `UserProfile` class
- The strategy pattern implementation in `calculateTargetCalories()` method that delegates to specific calculation methods
- Adding a new calculation method would only require:
 - Adding a new enum value
 - Implementing the calculation method (like `calculateHarrisBenedictCalories`)
 - Adding a case in the switch statement in `calculateTargetCalories()`
- No other parts of the code would need to change
 - High-level modules (e.g., `FoodDatabaseUI`) depend on abstractions (e.g., `Food` interface), not concrete implementations.
 - The `FoodDatabase` depends on the abstract `Food` class rather than specific subclasses.

2. New sources of basic food information must be easy to add without ripple effects throughout the program

This is satisfied through:

- The `FoodDataSource` interface in `FoodDatabase`
- The `importFromSource()` method that accepts any `FoodDataSource`
- To add a new source:
 - Implement the `FoodDataSource` interface
 - Call `importFromSource()` with our implementation
- The rest of the program remains unchanged
- The database loading/saving logic is isolated in `FoodDatabase`

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 handled well through:

- The `internFoodId()` method in `DailyLog` that uses a `foodIdCache` map to ensure string IDs are interned
- The `Object[]` array storage in daily logs keeps memory usage low
- The undo functionality uses snapshots rather than keeping full copies
- Food entries reference food IDs rather than containing full food objects

- The `deepCopyCurrentState()` method is used judiciously only when needed for undo

Design's Support for Additional Nutritional Data

1. Current Capabilities

- **Base Food class** already tracks **protein, carbs, fats**
- **BasicFood** stores fixed nutritional values
- **CompositeFood** auto-calculates nutrition from components
- **Database** saves/loads all existing nutritional fields

2. Easy Expansion for New Nutrients

- **Add new fields** (e.g., fiber, sodium, vitamins) directly in **Food** class
- **Update constructors** in **BasicFood** to accept new values
- **Extend `recalculateNutrition()`** in **CompositeFood** to include new nutrients
- **Modify database format** to store/load additional fields

3. Why It's Scalable

No major refactoring needed – changes are **additive**

Open/Closed Principle – extend without breaking existing logic

UI updates optional – display new nutrients only if needed

No ripple effects – core logic remains unchanged

How My Design Supports Adding New Food Data Sources Without Ripple Effects

1. My Current Implementation

- We already have a **FoodDataSource** interface in **FoodDatabase.java** that defines how to fetch food data.
- Our system **doesn't care where the data comes from**—only that it follows the interface.



2. Adding a New Website (e.g., McDonald's or USDA)

- We just **create a new adapter class** for that website (e.g., `McDonaldsFoodScraper`).
- The adapter **implements `FoodDataSource`** and handles the website's unique format.
- Then **register it once** with `importFromSource()`—no other changes needed.

3. Why This Prevents Ripple Effects

- **`Food` and `CompositeFood` classes stay the same**—they don't know where data comes from.
- **database and UI don't need updates**—they treat all food data the same way.
- **logging and user profiles work unchanged**—no modifications required


Reflection

Strongest Aspects

- **Flexible Food Hierarchy:** The abstract `Food` class with `BasicFood` and `CompositeFood` subclasses provides an elegant solution for handling both simple and complex foods while allowing easy extension.
- **Undo Functionality:** The `DailyLog` class implements a robust undo mechanism using the Command pattern (via `Runnable` commands) that can handle multiple action types without complex branching logic.

Weakest Aspects

- **UI-Model Coupling:** The `FoodDatabaseUI` has some direct dependencies on model classes that could be further abstracted through interfaces or a controller layer.

- 
- **Persistence Format:** The current file formats for food database and logs are somewhat brittle and could benefit from a more standardized approach like JSON or XML.