# YADA (Yet Another Diet Assistant) - Design Document

**Date: April 8 2025**

**Team Members:**

- Indu Anuga

- Vagdevi Reddy

## Overview

YADA is a diet management system designed to help users track their food consumption, calculate daily calorie goals, and maintain nutritional logs. The system features:

1. Food Database Management:
   - Track basic foods with identifiers, keywords, and calories per serving.
   - Create composite foods from combinations of basic and other composite foods.
   - Search functionality with keyword matching options.

2. Daily Logging:
   - Record food consumption by date.
   - View and edit logs for any date.
   - Calculate total calories consumed.

3. Diet Goal Profiling:
   - Store user profile information (age, weight, height, gender, activity level).
   - Multiple calorie calculation methods (Harris-Benedict and Mifflin-St Jeor).
   - Compare consumed calories vs target intake.

4. Advanced Features:
   - Undo/redo functionality for all operations.
   - Persistent storage of all data.
   - Daily profile updates with carry-over defaults.

## UML Class diagram:

The UML class diagram illustrates the core structure of the system, highlighting key **classes**, **interfaces**, and their relationships. It includes the following design principles:

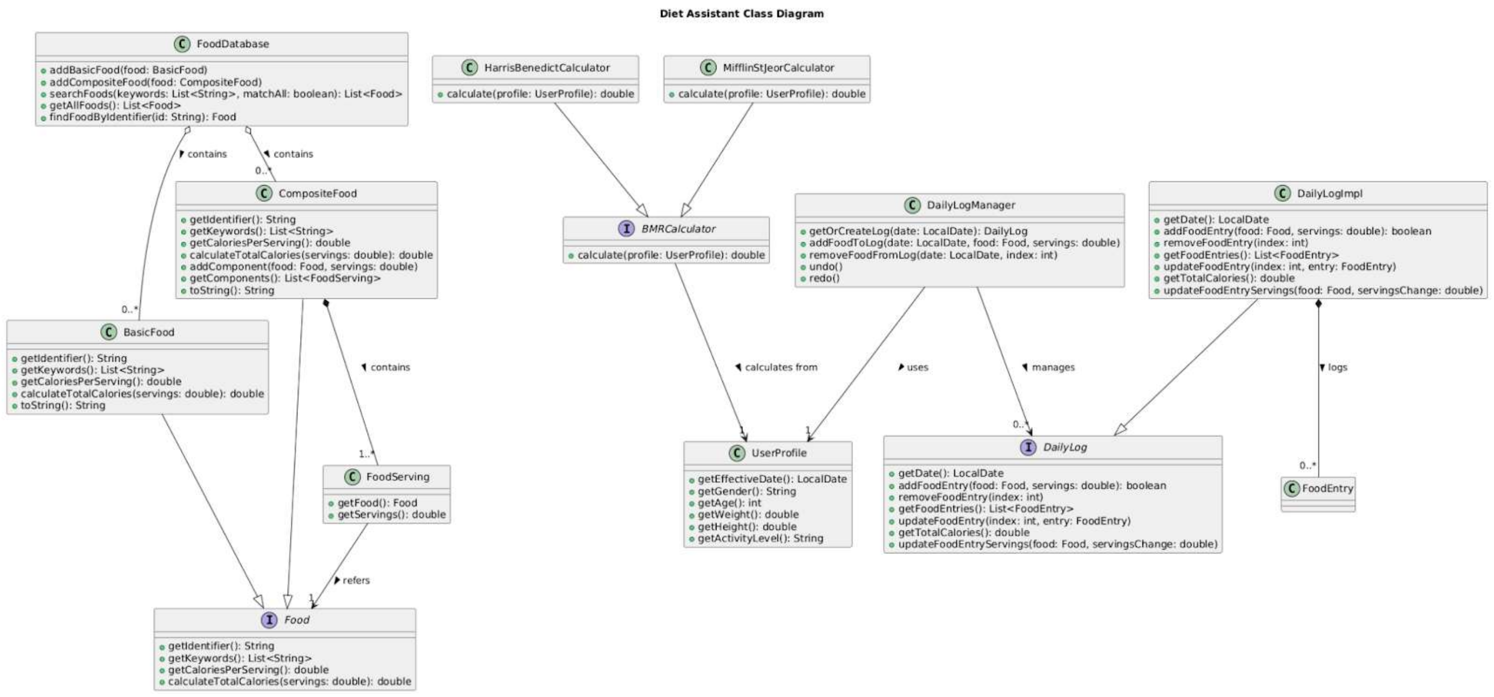
- **Interfaces** such as Food, BMRCalculator, and DailyLog define the abstract behavior expected from food items, basal metabolic rate calculators, and daily log operations.
- **Inheritance (Generalization)** is used where:
    - BasicFood and CompositeFood both implement the Food interface.
    - HarrisBenedictCalculator and MifflinStJeorCalculator implement BMRCalculator.
    - DailyLogImpl implements the DailyLog interface.

- **Composition** is used to represent strong ownership:
    - CompositeFood contains multiple FoodServing instances (1..* cardinality).
    - DailyLogImpl composes multiple FoodEntry objects (0..*).

- **Aggregation** shows shared ownership:
    - FoodDatabase aggregates multiple BasicFood and CompositeFood items (0..*).

- **Association** links between collaborating objects:
    - DailyLogManager manages many DailyLog instances and is associated with a single UserProfile.
    - FoodServing references a single Food item.
    - BMRCalculator is associated with one UserProfile for performing calculations.

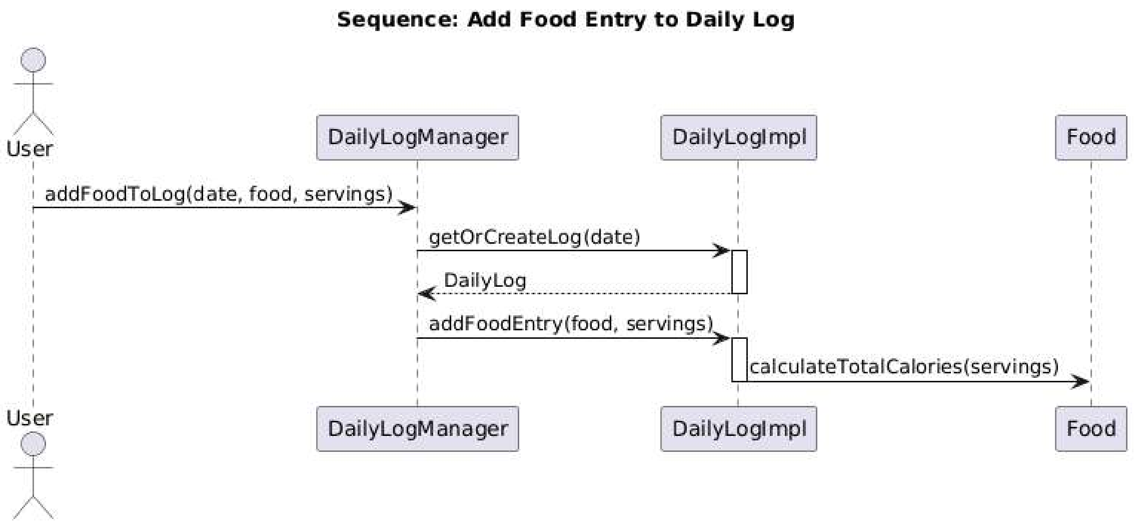
Only **important public methods** are displayed to keep the diagram focused and implementation-agnostic. Attributes and internal state are intentionally omitted to avoid clutter and over-specification.


Diet Assistant Class Diagram
FoodDatabase
addBasicFood(food: BasicFood)
addCompositeFood(food: CompositeFood)
searchFoods(keywords: List<String>, matchAll: boolean): List<Food>
getAllFoods(): List<Food>
findFoodByIdentifier(id: String): Food
contains
contains
0..*
0..*
CompositeFood
getIdentifier(): String
getKeywords(): List<String>
getCaloriesPerServing(): double
calculateTotalCalories(servings: double): double
addComponent(food: Food, servings: double)
getComponents(): List<FoodServing>
toString(): String
BasicFood
getIdentifier(): String
getKeywords(): List<String>
getCaloriesPerServing(): double
calculateTotalCalories(servings: double): double
toString(): String
contains
1..*
FoodServing
getFood(): Food
getServings(): double
refers
1
Food
getIdentifier(): String
getKeywords(): List<String>
getCaloriesPerServing(): double
calculateTotalCalories(servings: double): double
HarrisBenedictCalculator
calculate(profile: UserProfile): double
MifflinStJeorCalculator
calculate(profile: UserProfile): double
BMRCalculator
calculate(profile: UserProfile): double
DailyLogManager
getOrCreateLog(date: LocalDate): DailyLog
addFoodToLog(date: LocalDate, food: Food, servings: double)
removeFoodFromLog(date: LocalDate, index: int)
undo()
redo()
DailyLogImpl
getDate(): LocalDate
addFoodEntry(food: Food, servings: double): boolean
removeFoodEntry(index: int)
getFoodEntries(): List<FoodEntry>
updateFoodEntry(index: int, entry: FoodEntry)
getTotalCalories(): double
updateFoodEntryServings(food: Food, servingsChange: double)
calculates from
uses
manages
logs
1
1
0..*
0..*
UserProfile
getEffectiveDate(): LocalDate
getGender(): String
getAge(): int
getWeight(): double
getHeight(): double
getActivityLevel(): String
DailyLog
getDate(): LocalDate
addFoodEntry(food: Food, servings: double): boolean
removeFoodEntry(index: int)
getFoodEntries(): List<FoodEntry>
updateFoodEntry(index: int, entry: FoodEntry)
getTotalCalories(): double
updateFoodEntryServings(food: Food, servingsChange: double)
FoodEntry

## Sequence Diagrams:

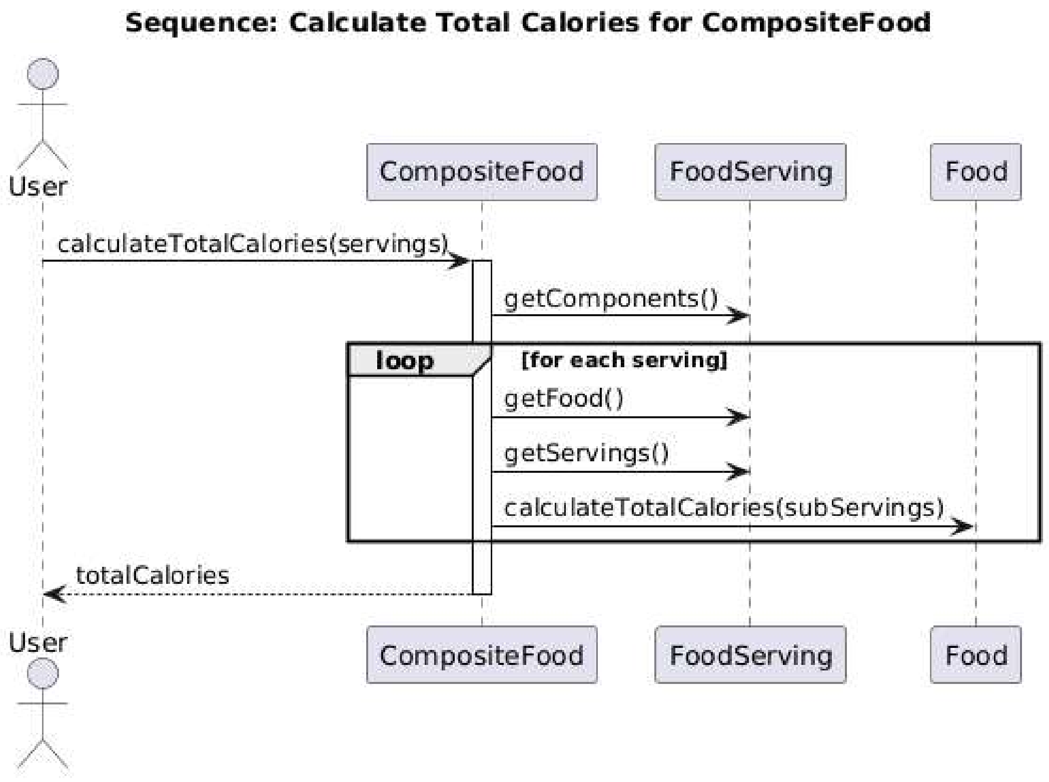
The sequence diagrams represent dynamic behavior across key interactions within the system. These interactions are consistent with the implemented logic and provide insight into how the objects collaborate at runtime.

1. **Add Food Entry to Daily Log:** Describes how a user adds food to a specific day. The DailyLogManager retrieves or creates a DailyLogImpl for the date, then delegates the logging to that instance.


Sequence: Add Food Entry to Daily Log
User
DailyLogManager
DailyLogImpl
Food
addFoodToLog(date, food, servings)
getOrCreateLog(date)
DailyLog
addFoodEntry(food, servings)
calculateTotalCalories(servings)

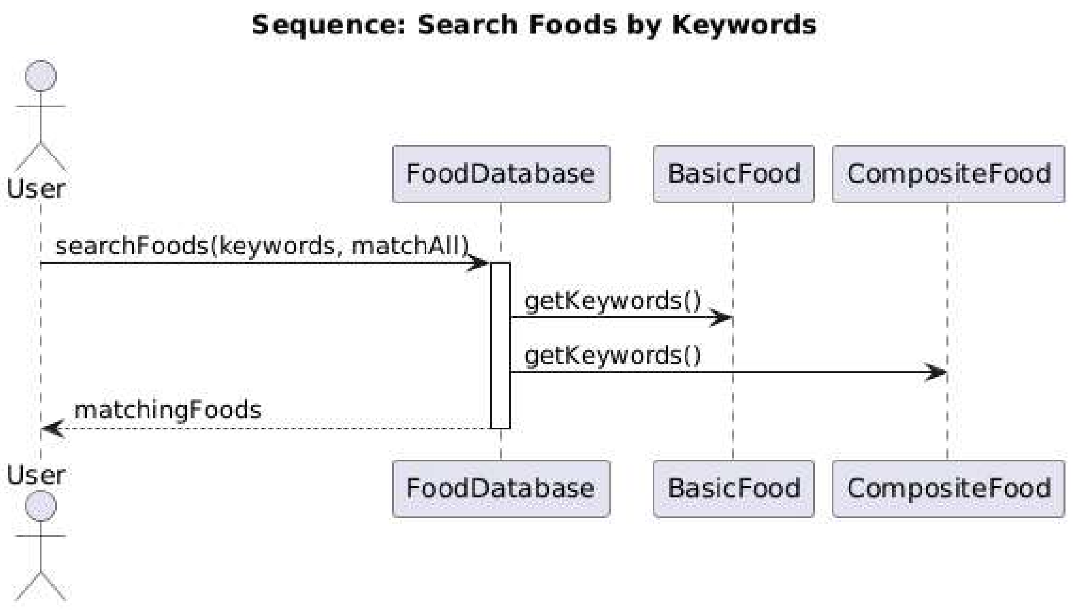
2. **Calculate Calories for a CompositeFood**

   Outlines how a CompositeFood calculates its total calories by iterating over its components (FoodServing) and summing their individual calorie values.


Sequence: Calculate Total Calories for CompositeFood
User
CompositeFood
FoodServing
Food
calculateTotalCalories(servings)
getComponents()
loop
[for each serving]
getFood()
getServings()
calculateTotalCalories(subServings)
totalCalories

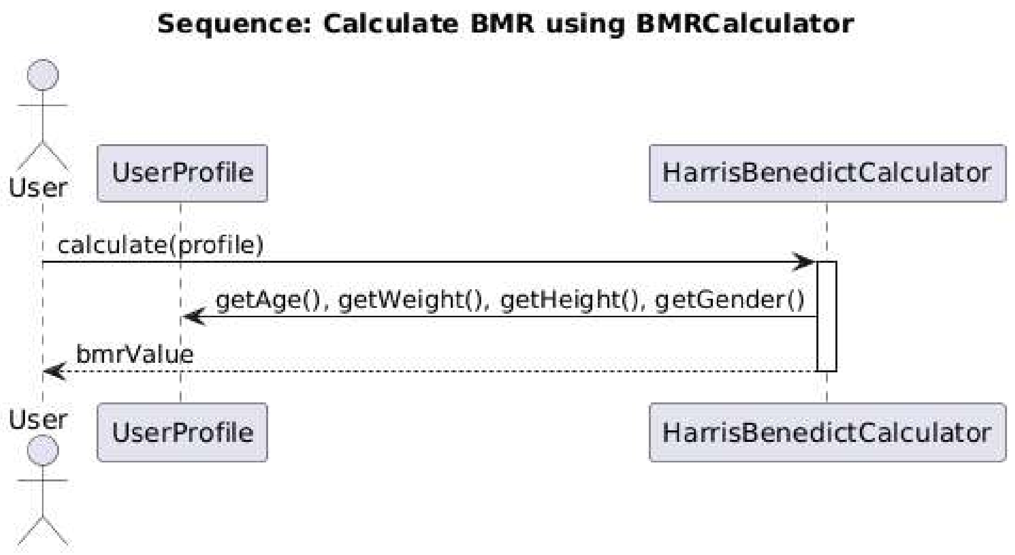
3. **Search Foods by Keywords**

   Shows the FoodDatabase filtering BasicFood and CompositeFood entries by matching user-specified keywords.



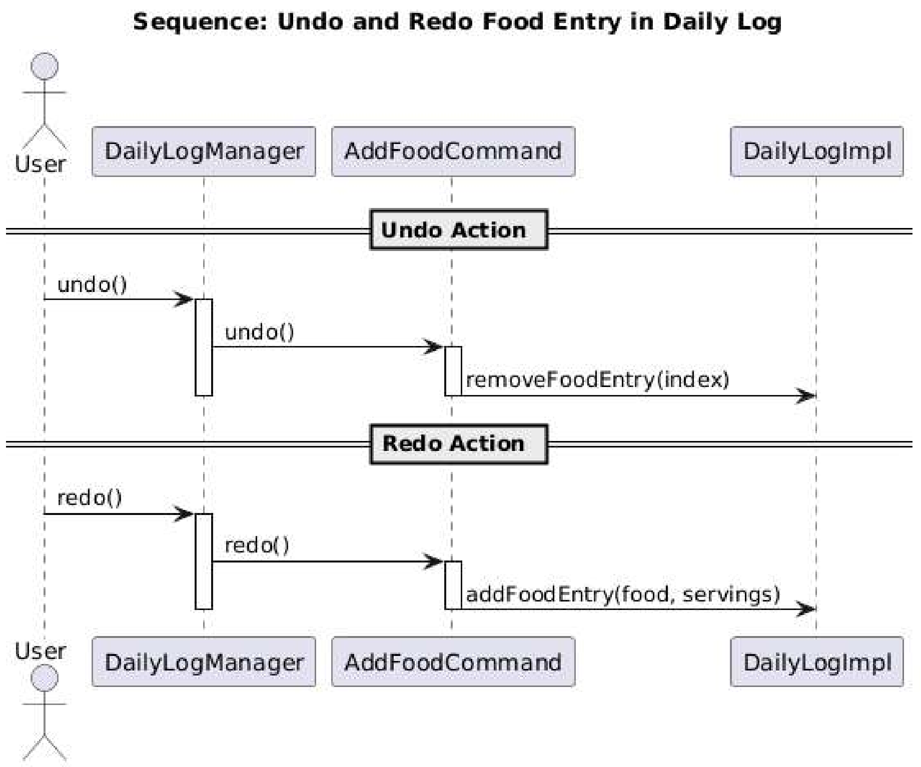
4. **Calculate BMR**

   Demonstrates how a BMRCalculator implementation (e.g., HarrisBenedictCalculator) accesses data from UserProfile to compute the basal metabolic rate.



5. **Undo/Redo Logging Actions**

   Captures the behavior of undoing or redoing log changes in DailyLogManager using the command pattern. Each Command encapsulates reversible operations on the DailyLogImpl.



## Expandability & Maintainability Considerations

System is designed with a high degree of modularity and extensibility in mind, ensuring that future enhancements or changes will not cause ripple effects throughout the program.

### Expanding Nutritional Information

Currently, the BasicFood class tracks the identifier, search keywords, and calories per serving. However, our design makes it easy to expand this to support additional nutritional data such as:

- Protein, carbohydrates, fiber
- Fat, saturated fat
- Vitamins and minerals (e.g., Vitamin A, Iron, Calcium)

This is possible because:

- Nutritional properties are stored using a flexible attribute structure or can be added via composition or encapsulation.
- Calculations (like total nutritional value in CompositeFood) iterate over components, making the logic extendable without altering the structure.

**Result**: Minimal code change needed when adding new nutritional fields.

### Supporting New Food Sources from the Web

Production systems will involve downloading food data from websites like McDonald's or USDA databases. These sites may use different data formats (e.g., JSON, XML, HTML scraping).

To accommodate this:

- We abstracted the parsing logic using the Strategy or Adapter Pattern.
- Each new website parser implements a common interface (e.g., FoodDataParser) which can be plugged in easily.

```
interface FoodDataParser {

   List<BasicFood> parse(InputStream data);

}
```

Adding a new site involves only:

1. Creating a new class implementing FoodDataParser

2. Registering it without touching the existing core

Result: No ripple effects across the core system when adding new sites.

## Design Narrative – Balancing Key Software Principles

Design reflects a careful balance of several key object-oriented principles:

### Low Coupling

- Classes interact through interfaces (e.g., Food, BMRCalculator, DailyLog) rather than concrete implementations.
- Web parsers and BMR calculators are decoupled using strategies and factory methods.

### High Cohesion

- Each class has a single, well-defined responsibility.
  - FoodDatabase handles storage/searching of food items.
  - DailyLogImpl focuses on managing entries for a specific date.

### Separation of Concerns

- Logic for food storage, calorie calculation, user profile, and logging are in separate modules.
- Undo/Redo is abstracted via the Command Pattern, isolating reversible actions.

### Information Hiding

- Internal states of objects like DailyLogImpl and CompositeFood are encapsulated, only accessible via methods.

### Law of Demeter

- Methods only talk to their immediate collaborators.
- For example, DailyLogManager does not manipulate FoodEntry directly but uses well-defined APIs in DailyLogImpl.

**Result**: The system is robust, maintainable, and easily extensible.

# Reflection on the Design – Strengths and Weaknesses

## Strongest Aspects of the Design

- **High Modularity and Extensibility**
    - Adding new types of food, new nutritional attributes, or new calorie calculators does not require changes to existing code—only new classes that implement the existing interfaces.
    - The system is structured using key object-oriented principles such as interfaces, abstraction, and composition, which allow future changes or feature additions with minimal impact on the rest of the system.

- **Clear Separation of Concerns**
    - Each component has a focused responsibility:
        - FoodDatabase manages food items.
        - DailyLogImpl manages food entries per date.
        - BMRCalculator handles personalized calorie goals.
    - This clarity helps with testing, debugging, and scaling.

- **Support for Undo/Redo**
    - The use of the Command Pattern allows robust undo/redo operations with isolated logic per command type, supporting both user experience and data integrity.

- **Flexible Parsing for Web Integration**
    - Abstracting food data parsers enables seamless support for various food data sources, making the system adaptable to real-world diet APIs and inconsistent formats.

## Weakest Aspects of the Design

1. **Initial Simplicity of Nutritional Modeling**
    - The current implementation only tracks calories. Although expandable, the current logic (especially in calorie calculations) may require careful generalization before supporting full nutrient tracking (e.g., protein, fiber).
    - Future extensions will require refactoring methods like calculateTotalCalories() into a more generic calculateTotalNutrition() or similar.

2. **Limited Validation and Error Handling**
    - The current structure assumes ideal inputs and successful operations (e.g., valid food items, correct servings). In a production environment,

robust validation, input sanitation, and error recovery will be necessary.

3. **Storage/Serialization Not Yet Abstracted**
    - Persistence (saving/loading logs or food databases) is not detailed in the design. Incorporating persistent storage while maintaining modularity may require future abstraction layers (e.g., FoodRepository, LogSerializer).