Design Documentation

TITLE: YADA (Yet Another Diet App)

DATE: 07-05-2025

Team Members:

- 2023101046 - Varun Alokam
- 2023101104 - Akash Sigullapalli

Overview:

YADA is a Java-based diet-tracking application that empowers users to monitor their daily food intake, build custom food items, and stay on target with personalized nutritional goals. Its modular design separates data storage, user profiles, logging, and undoable commands, ensuring both flexibility and reliability.

Key Features

1. User Profile & Goals

i) Store daily profiles (gender, age, height, weight, activity level) per date

ii) Choose between Harris-Benedict or Mifflin-St Jeor formulas for calorie targets

iii) View and update past profiles; calculate personalized daily calorie needs

2. Comprehensive Food Database

i) **Basic Foods**: Define simple items with unique IDs, keywords, calorie values, and optional extra info (e.g., “protein=10:g”).

ii) **Composite Foods**: Create recipes by combining existing foods; auto-aggregate calories and nutritional metadata.

iii) Persistent file storage with human-readable, sorted formats; keyword-based search.

3. Daily Logging

i) Log servings by food ID for any date (defaults to today).

ii) Automatically merge duplicate entries on the same day by summing servings.

iii) View, update or delete log entries per date; only committed (saved) logs are shown.

4. Undoable Commands

i) Add, update, and delete log entries via the Command pattern.

ii) Maintain separate stacks for unsaved vs. saved operations, allowing undo of the last saved action.

5. Data Persistence & CLI

i) All foods, logs, and profiles are loaded from and saved to text files on exit or on demand.

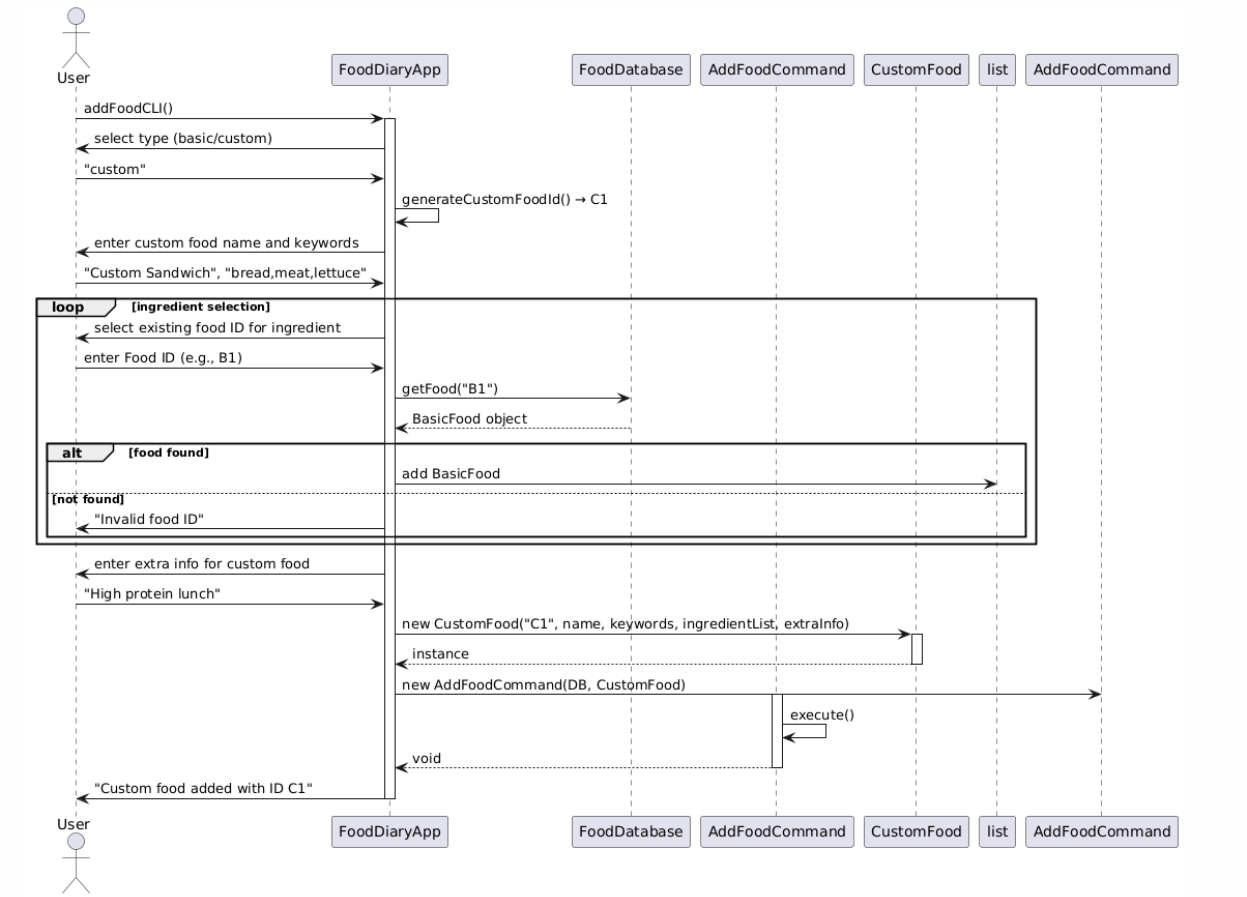
ii) Interactive command-line interface with menus for every operation, plus a Help screen.

UML class diagram:

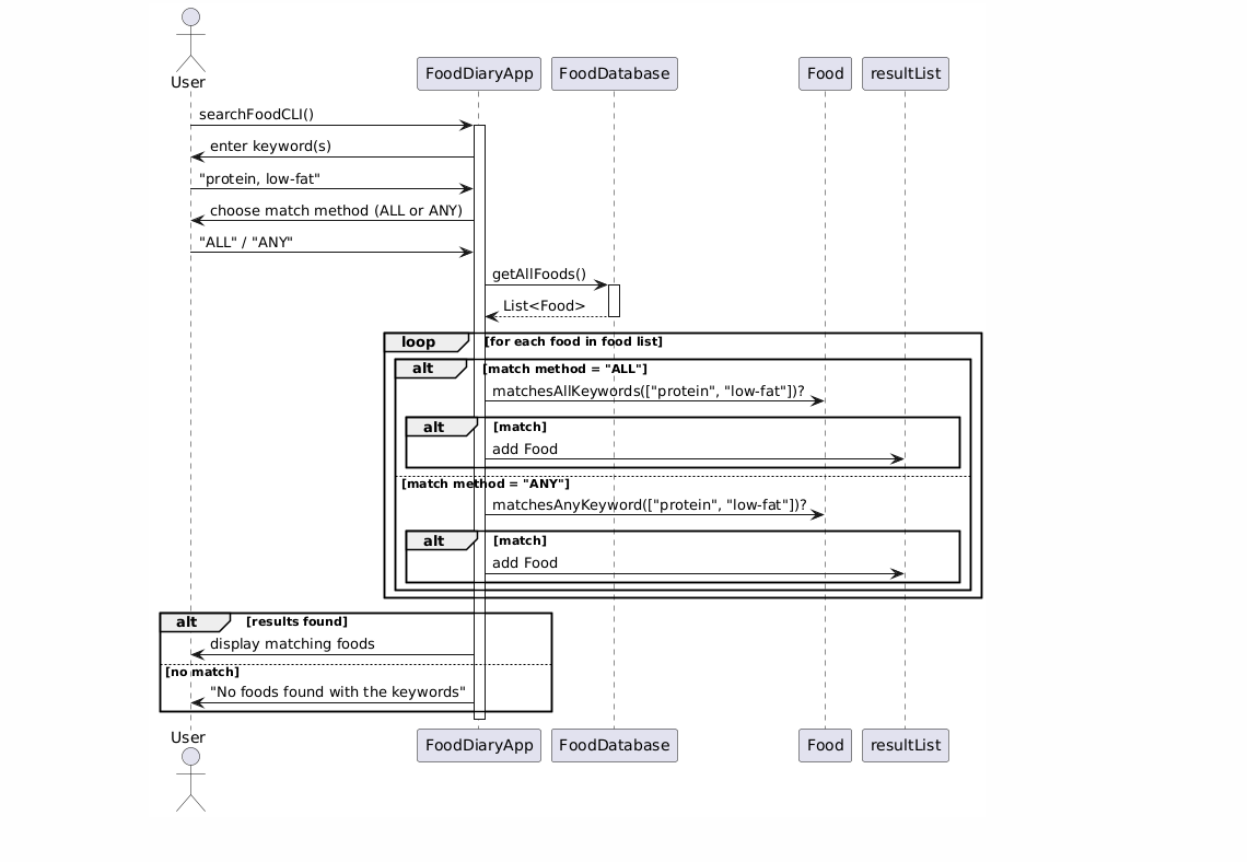
UML Class Diadram link

Sequence diagrams

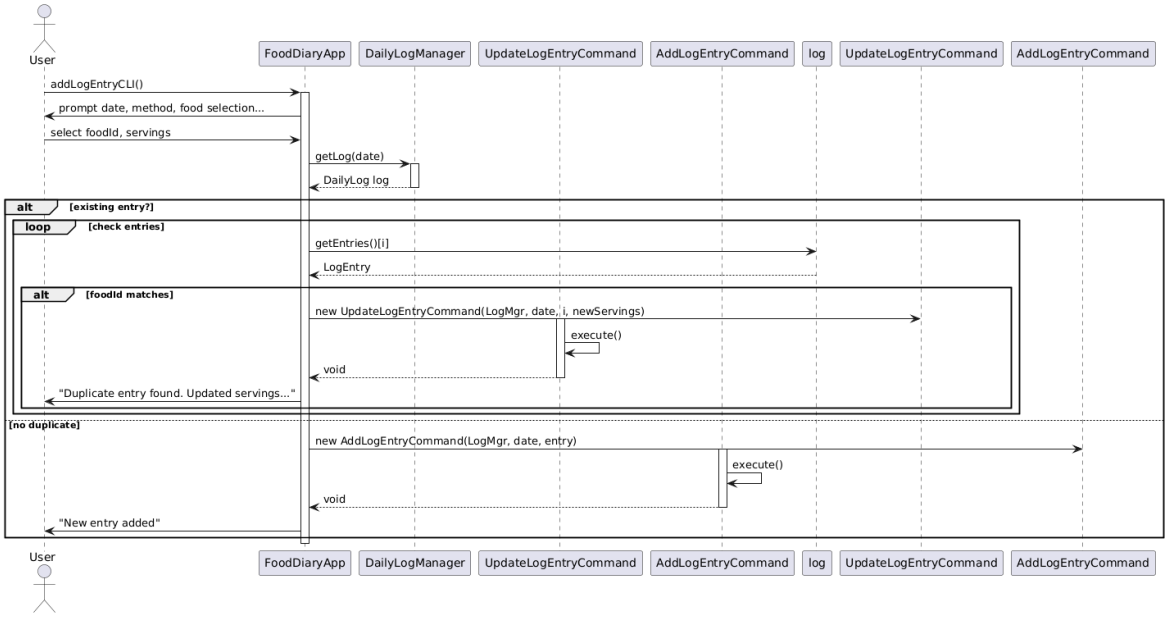
1. Add complex food item



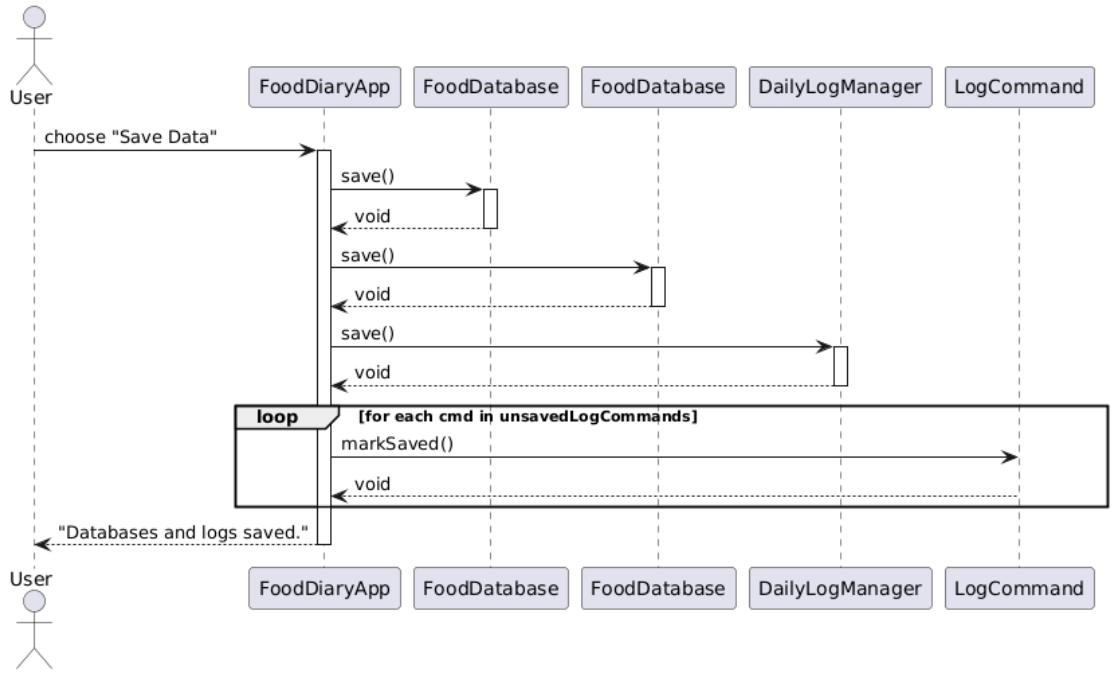
2. Search and Filter Food by Keywords



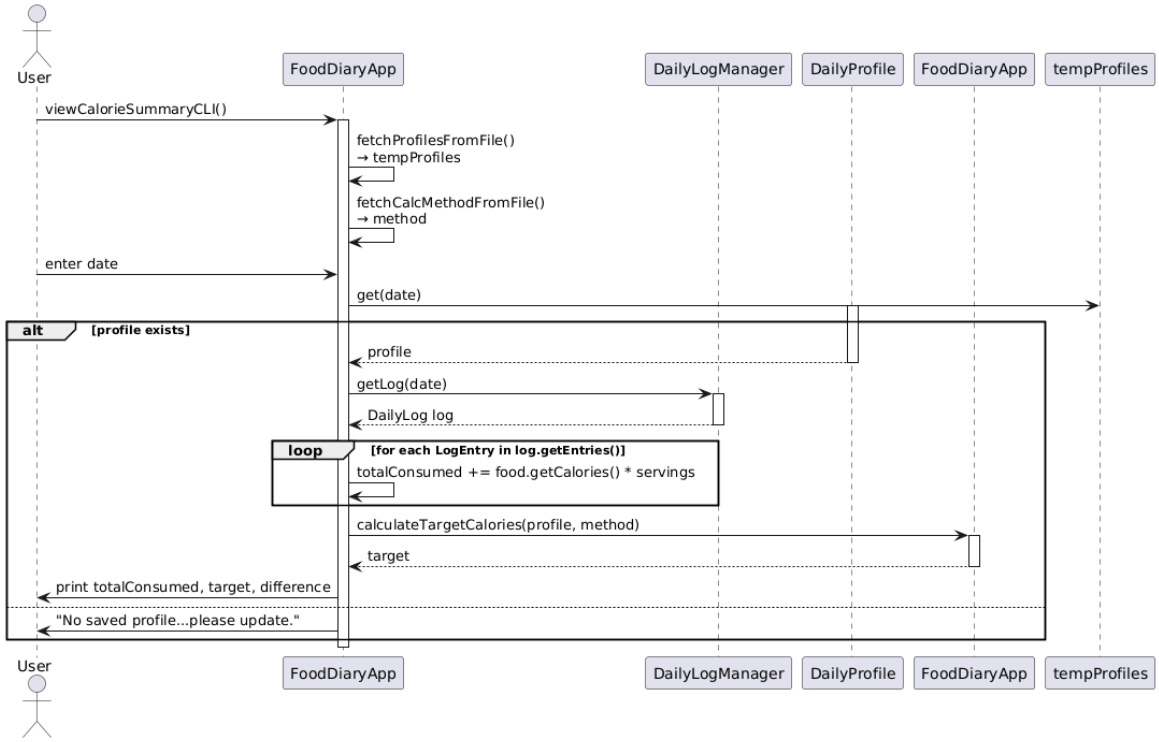
3. Add a Log Entry



4. Save All Data



5. View Calorie Summary



Design Analysis of Food Diary Application

Low Coupling

- **Package Organization:** Distinct packages (model, database, command, log, app) minimize cross-component dependencies.
- **Command Pattern Implementation:** Command interface and concrete command classes decouple operation invokers from executors.
- **Abstraction Layers:** Abstract base classes and database interfaces reduce direct dependencies between concrete implementations.
- **Centralized ID Management:** Components request IDs without knowledge of generation details, reducing system-wide dependencies.

High Cohesion

- **Single-Purpose Classes:** Each class has well-defined responsibilities - food models manage nutritional data, log entries track consumption, and commands encapsulate operations.

- **Specialized Repositories:** Database classes focus exclusively on their respective domains (food items, daily logs, profiles).
- **Cohesive Hierarchies:** Model classes like Food and its subclasses maintain clear inherited relationships with related functionality grouped appropriately.

Separation of Concerns

- **Model-Persistence Separation:** Data structures are cleanly separated from storage mechanisms, allowing independent evolution.
- **Command Encapsulation:** Business operations are isolated in command objects, separating execution logic from invocation points.
- **Calculation Strategies:** Calorie calculation methods (Harris-Benedict, Mifflin-St Jeor) are implemented as separate strategies.
- **Domain Isolation:** Food tracking, profile management, and log operations exist as distinct subsystems with minimal overlap.

Information Hiding

- **Encapsulated Implementation:** Classes expose controlled interfaces while hiding internal data structures and algorithms.
- **Composition Transparency:** Composite Food hides the complexity of managing ingredient relationships and aggregated calculations.
- **Database Abstraction:** Storage mechanisms hide file formats and retrieval logic behind consistent interfaces.

Law of Demeter

- **Command Interactions:** Commands generally limit method calls to directly passed parameters and owned properties.
- **Controlled Access Paths:** Most components interact only with their immediate collaborators rather than reaching through object chains.

- **Interface Boundaries:** Systems interact through well-defined interfaces rather than accessing implementation details.

Single Responsibility Principle

- **Focused Domain Classes:** Model classes like Food, LogEntry, and DailyProfile each represent one type of domain entity.
- **Operation Encapsulation:** Each command class handles exactly one type of system operation.
- **Specialized Data Access:** Database classes focus exclusively on persistence concerns for their respective domains.

Design Patterns

- **Command Pattern:** Encapsulates operations as objects with execute/undo capabilities, supporting history tracking.
- **Composite Pattern:** Enables uniform treatment of individual foods and multi-ingredient recipes through consistent interfaces.
- **Repository Pattern:** Provides collection-like access to domain objects while hiding storage details.
- **Strategy Pattern:** Allows interchangeable calorie calculation algorithms without affecting client code.

Two Strongest Aspects of the Design

1. **Command Pattern Implementation**: The application has a well-structured command system that cleanly encapsulates operations while providing undo functionality. Commands are properly separated by type, enabling flexible operation management and making the system extensible.
2. **Food Model Hierarchy**: The inheritance structure for food items (abstract Food with BasicFood and CompositeFood implementations) demonstrates good object-oriented design. The Composite pattern allows complex food items to be

built from simpler ones while maintaining a consistent interface.

Two Weakest Aspects of the Design

1. **Error Handling and Validation**: The app's error handling is basic and doesn't cover issues like file corruption or network failures. Input validation is superficial without domain-specific checks, such as checking for negative calorie values, valid height/weight ranges, and proper date formats. It needs stronger consistency checks across related data.
2. **Lack of UI Separation**: The user interface logic is embedded directly in the main application class rather than being separated into dedicated UI components. This makes it difficult to modify the interface independently of business logic or to support multiple interface types.

Key Design Points

1. **Command-Based Operation Model**: All data modifications occur through commands that support undo, providing transactional integrity.
2. **Composite Food Structure**: The application can represent both simple foods and complex recipes built from other foods.
3. **Persistent Storage with Text Files**: Simple but effective text-based storage formats that maintain human readability.
4. **Profile-Based Calorie Calculations**: The system supports personalized calorie targets based on user profiles.
5. **ID Management System**: Centralized ID generation with recycling of deleted IDs for efficiency.
6. **Daily Log Organization**: Food consumption is tracked by date with per-serving calorie calculations.