Design Document

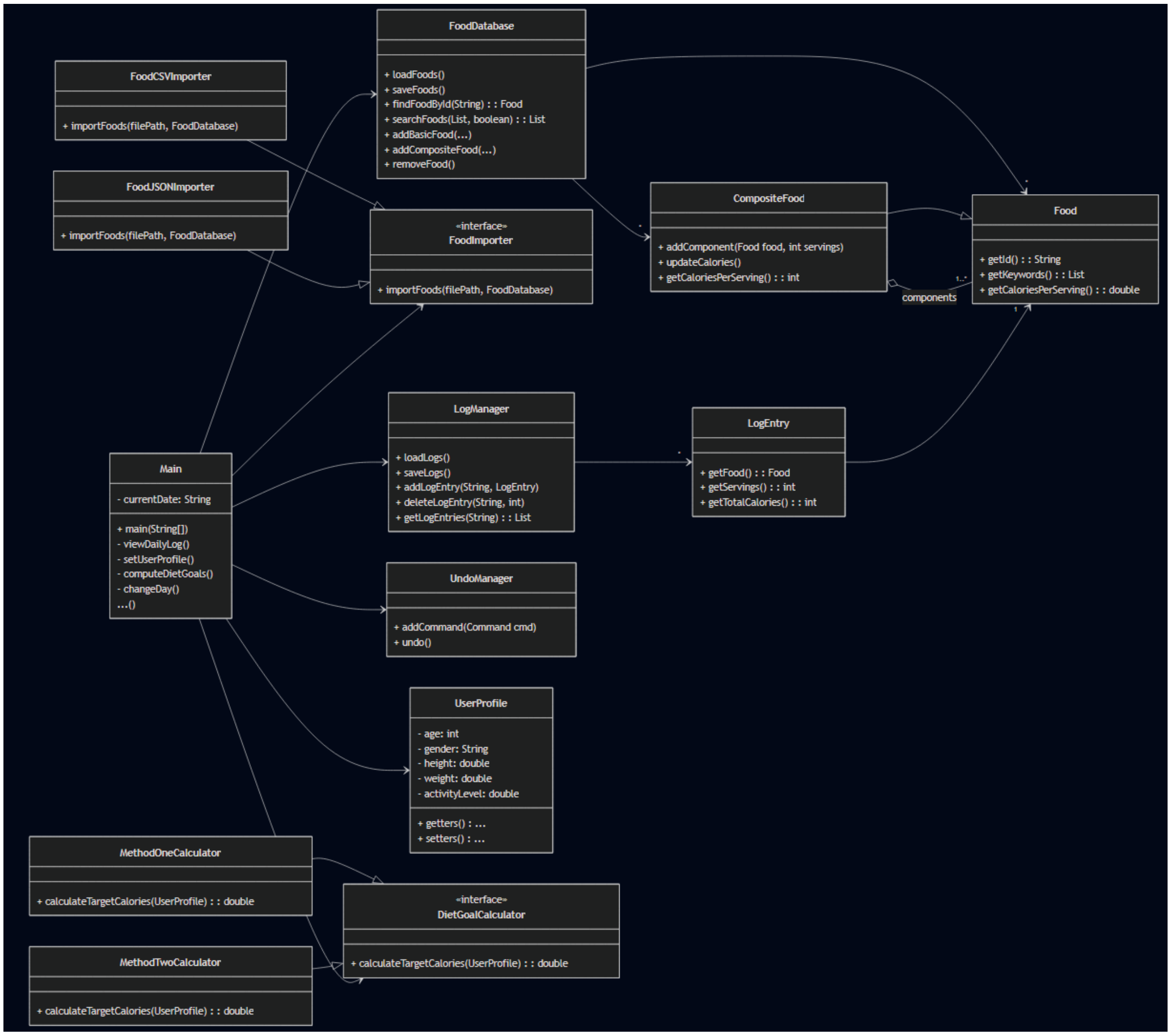
Title: YADA (Yet Another Diet Assistant)
Date: April 7, 2025
Team Members: Krishak Aneja (2023101106), Saiyam Jain (2023101135)

Overview

YADA is a console-based *Java* application that allows users to log their daily food consumption and track calories. The application supports importing food data from CSV and JSON files, managing both basic and composite food entries, logging daily intake with support for undo operations and recording user details for target calorie intake calculation. Key features include:

- Food database supporting basic and composite foods
- Logging system with undo functionality
- Keyword-based food search with ALL/ANY logic
- Personalized calorie targets using Harris-Benedict or Mifflin-St Jeor Equation
- Importers for CSV and JSON formats

UML Class Diagram



Sequence Diagrams

1. **Logging a Food Entry**

- Actor: User
- Objects: Main, InputHelper, FoodDatabase, LogManager, UndoManager
- Description: User provides date and food ID, specifies servings, and the system logs the entry and stores an undo command.

2. **Searching for Food by Keywords**

- Actor: User
- Objects: Main, InputHelper, FoodDatabase
- Description: User inputs comma-separated keywords and a boolean indicating ALL/ANY match, system filters food entries accordingly.

3. **Importing Food Data from File**

- Actor: User
- Objects: Main, InputHelper, FoodImporter, FoodCSVImporter/FoodJSONImporter, FoodDatabase
- Description: User specifies file path and type; appropriate importer reads and adds food items to the database.

4. **Undoing a Log Entry**

- Actor: User
- Objects: Main, UndoManager, LogManager
- Description: User triggers an undo, which removes the last log entry added.

5. **Calculating Target Calories**

- Actor: System
- Objects: UserProfile, CalorieCalculator
- Description: Based on user profile (age, weight, height, gender, activity level), BMR is calculated using the Harris-Benedict Equation (Method one, which is the default)

Balance of Design Principles

- **Low Coupling** involving Interfaces such as `FoodImporter` and `DietGoalCalculator` decouple file-specific and target-calculation logic from core functionality.
- **Separation of Concerns**: Clear modular separation between importers, database management, and user interaction logic.
- **Cohesion** is maintained by assigning related responsibilities to specific classes (e.g., `LogManager` for log operations, `FoodDatabase` for food database manipulation, `UndoManager` for handling undo functionality). The `Main` class is responsible for user interaction via the command-line interface (CLI).
- **Information Hiding** is implemented through encapsulated (private) data members and public interfaces.
- **Law of Demeter** is respected by minimizing method chaining and limiting knowledge of internal structures. The `Main` class has access to instances of each of the core functional components, each of which acts as a liaison to its own data members and behaviors.

Design Principles Reflection

Strengths:

1. **Clear Separation of Responsibilities through a Central Orchestrator**
   One of the strongest aspects of our design is the role of the `Main` class as a central orchestrator. It is solely responsible for handling user interaction through the command-line interface (CLI), presenting the menu, capturing user input, and providing output. All core logic and data manipulation responsibilities are delegated to dedicated functional components such as `FoodDatabase`, `LogManager`, and `UndoManager`.

This separation aligns well with several good design principles — it supports **Separation of Concerns**, keeps **coupling low**, and maintains **high cohesion** within individual classes. By managing instances of functional units directly and interacting only through their public interfaces, it also adheres to the **Law of Demeter**, avoiding unnecessary dependency chains.

2. **Extensible Functional Components for Future Enhancements**
   The design provides dedicated abstractions for major functionalities. For example:

- New **data sources** for basic food information can be integrated seamlessly by implementing the `FoodImporter` interface — achieving **low coupling** and addressing extensibility in data ingestion (Key Design Point 2). Capability to handle Idiosyncratic variations in data sources is implemented by accounting for alternate headings/labels.
- The method used to compute **target calories** is centralized via the `DietGoalCalculator` interface and can be easily modified or replaced (e.g., adding more methods besides Harris-Benedict or Mifflin-St Jeor) without affecting other parts of the system. This enables future extensibility of **core logic** while minimizing ripple effects (Key Design Point 1).

Weaknesses:

1. **UI Complexity in Console-Based Interaction**
   Since the application uses a **command-line interface (CLI)** rather than a graphical user interface (GUI), there are inherent limitations in user experience. Long or deeply nested data outputs — such as the detailed breakdown of composite foods — can quickly clutter the console, making interaction less intuitive and harder to follow. The lack of structured visual elements or pagination makes it difficult to navigate or edit previously entered data. While the CLI provides simplicity and portability, it sacrifices user-friendliness in favor of lower overhead.
2. **Inefficient Search and Lookup Mechanisms**
   Currently, the system **relies on repeated linear iteration over the entire food database**

for operations like keyword search and food ID lookup. While acceptable for small datasets, this can lead to performance bottlenecks as the number of food items grows. There is no indexing, caching, or use of efficient data structures like hash maps or inverted indices to optimize retrieval, which limits scalability.