Design Document – YADA Diet Manager

DASS Assignment 3

Team:
Siddharth Mago (2023101088)
Atharv Sanjaykumar Bhatt (2023101089)

Date: 07/04/2025

Product Overview

YADA (Yet Another Diet Assistant) is a command-line diet management application designed to help users track their food consumption and manage their caloric intake goals. This prototype addresses the problem of overeating by providing a comprehensive system for recording food consumption, calculating personalized calorie targets, and tracking progress over time.

Core Features

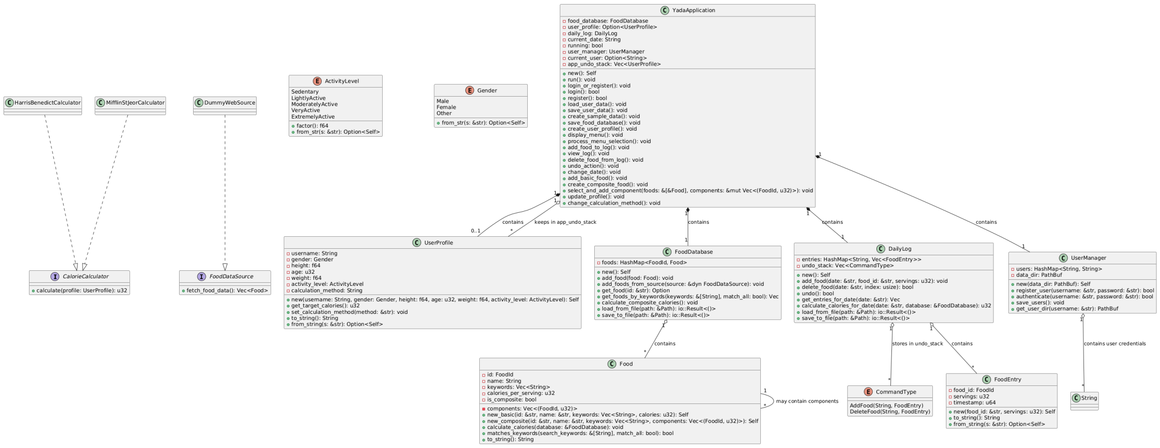
- **Multi-user support** with secure login/registration system
- **Comprehensive food database** with both basic and composite foods
- **Keyword-based food search** with flexible matching options
- **Daily food logging** organized by date with unlimited undo capability
- **Personalized calorie targets** calculated using multiple scientific methods:
    - Harris-Benedict Equation
    - Mifflin-St Jeor Equation
- **Profile management** including gender, age, height, weight, and activity level
- **Real-time calculation** of remaining daily calories
- **Composite food creation** for custom recipes and meal combinations
- **Persistent storage** using human-readable text files
- **Extensible architecture** for adding new food data sources or calculation methods

The application is built in RUST language with a strong focus on object-oriented design principles, including low coupling, high cohesion, and separation of concerns along with information security. It allows users to effectively monitor their dietary habits, create custom food combinations, and track their progress toward nutritional goals—all through an intuitive command-line interface.

Design Model

Class No. 1: YadaApplication	**Class State:** The YadaApplication class is the main controller for the diet management system. It maintains - • The food database with all basic and composite foods • The current user's profile with demographic and activity data • A daily log of food entries • The current date for logging • The application's running state • A user manager for authentication and registration • The currently logged-in user information • An undo stack for profile changes **Class Behaviour:** • Initializes the application and provides the main program loop • Handles user login, registration, and authentication • Loads and saves user data (profiles, logs) • Manages the user interface and menu system • Processes user selections and routes to appropriate functions • Provides methods for food logging (add, view, delete) • Implements profile management and updates • Allows changing calculation methods for calorie goals • Supports undoing actions
Class No. 2: FoodDatabase	**Class State:** The FoodDatabase class maintains - • A collection of all food items (both basic and composite) • A mapping between food IDs and food objects **Class Behaviour:** • Adds new food items to the database • Retrieves food items by ID or by keyword search • Calculates calories for composite foods • Loads and saves the food database to/from files • Provides access to food information for calorie calculations
Class No. 3: Food	**Class State:** The Food class represents both basic and composite food items and maintains - • A unique identifier • The food name • A list of search keywords • Calories per serving • A flag indicating whether it's a composite food • For composite foods, a list of component foods and their serving quantities **Class Behaviour:** • Creates basic food items with fixed calorie values • Creates composite food items composed of other foods • Calculates calories for composite foods based on their components • Matches search keywords to enable food search functionality • Converts food data to string format for storage

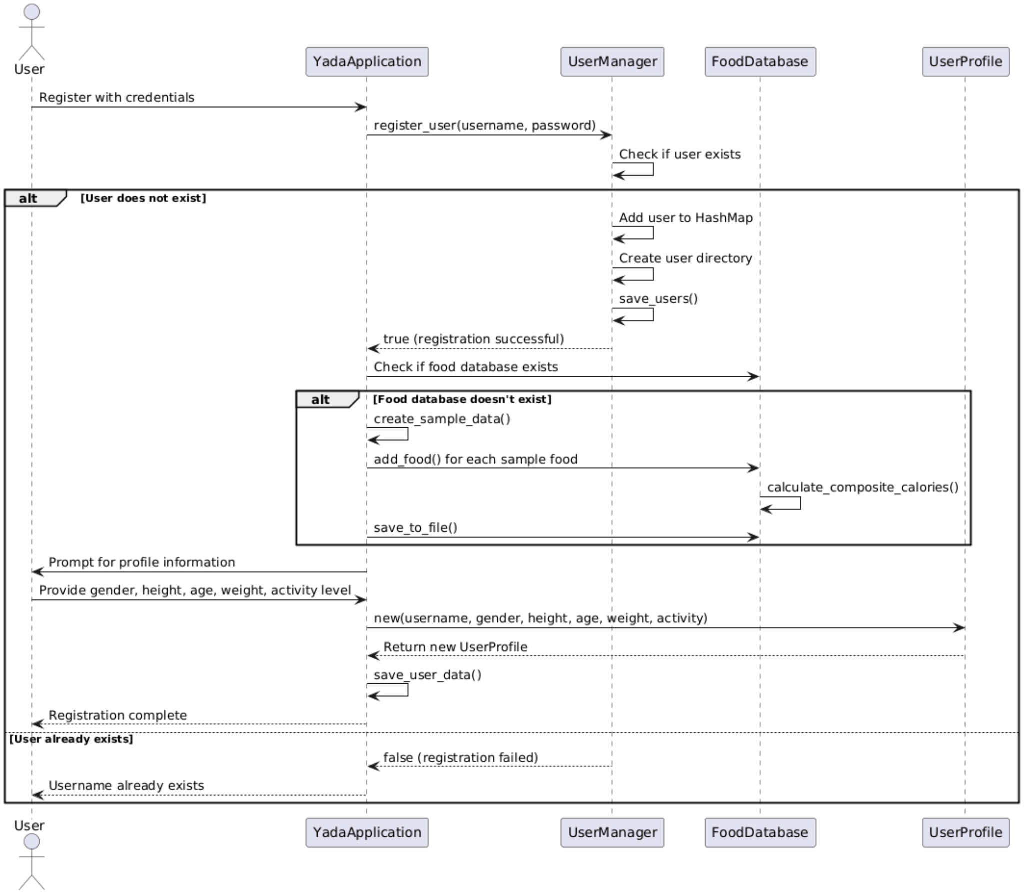
Class No. 4: DailyLog	**Class State:** The DailyLog class maintains - • Food consumption entries organized by date • An undo stack to track operations for reversal **Class Behaviour:** • Adds food entries to specific dates • Deletes food entries from the log • Supports undoing operations (add/delete) • Retrieves entries for a specific date • Calculates total calories consumed on a given date • Loads and saves the log data to/from files
Class No. 5: UserProfile	**Class State:** The UserProfile class maintains - • User identification information • Physical characteristics (gender, height, weight, age) • Activity level information • The selected calorie calculation method **Class Behaviour:** • Creates user profiles with demographic and activity data • Calculates target calorie intake based on user data • Allows changing the calculation method • Converts profile data to/from string format for storage
Class No. 6: UserManager	**Class State:** The UserManager class maintains: • User credentials (username/password pairs) • The data directory path for user-specific files **Class Behaviour:** • Registers new users • Authenticates existing users • Saves user credentials • Manages user-specific directories
Class No. 7: CalorieCalculator (interface)	**Class State:** • As an interface, it doesn't maintain state **Class Behaviour:** • Defines a contract for calorie calculation algorithms • Enables multiple calculation methods to be plugged in
Class No. 8: FoodDataSource (interface)	**Class State:** • As an interface, it doesn't maintain state **Class Behaviour:** • Defines a contract for retrieving food data from external sources • Allows for extensibility to support multiple data sources


YadaApplication
food_database: FoodDatabase
user_profile: Option<UserProfile>
daily_log: DailyLog
current_date: String
running: bool
user_manager: UserManager
current_user: Option<String>
app_undo_stack: Vec<UserProfile>
new(): Self
run(): void
login_or_register(): void
login(): bool
register(): bool
load_user_data(): void
save_user_data(): void
create_sample_data(): void
save_food_database(): void
create_user_profile(): void
display_menu(): void
process_menu_selection(): void
add_food_to_log(): void
view_log(): void
delete_food_from_log(): void
undo_action(): void
change_date(): void
add_basic_food(): void
create_composite_food(): void
select_and_add_component(foods: &[&Food], components: &mut Vec<(FoodId, u32)>): void
update_profile(): void
change_calculation_method(): void
HarrisBenedictCalculator
MifflinStJeorCalculator
DummyWebSource
ActivityLevel
Sedentary
LightlyActive
ModeratelyActive
VeryActive
ExtremelyActive
factor(): f64
from_str(s: &str): Option<Self>
Gender
Male
Female
Other
from_str(s: &str): Option<Self>
CalorieCalculator
calculate(profile: UserProfile): u32
FoodDataSource
fetch_food_data(): Vec<Food>
contains
keeps in app_undo_stack
0..1
*
1
1
UserProfile
username: String
gender: Gender
height: f64
age: u32
weight: f64
activity_level: ActivityLevel
calculation_method: String
new(username: String, gender: Gender, height: f64, age: u32, weight: f64, activity_level: ActivityLevel): Self
get_target_calories(): u32
set_calculation_method(method: &str): void
to_string(): String
from_string(s: &str): Option<Self>
contains
1
1
FoodDatabase
foods: HashMap<FoodId, Food>
new(): Self
add_food(food: Food): void
add_foods_from_source(source: &dyn FoodDataSource): void
get_food(id: &str): Option
get_foods_by_keywords(keywords: &[String], match_all: bool): Vec
calculate_composite_calories(): void
load_from_file(path: &Path): io::Result<()>
save_to_file(path: &Path): io::Result<()>
contains
1
1
DailyLog
entries: HashMap<String, Vec<FoodEntry>>
undo_stack: Vec<CommandType>
new(): Self
add_food(date: &str, food_id: &str, servings: u32): void
delete_food(date: &str, index: usize): bool
undo(): bool
get_entries_for_date(date: &str): Vec
calculate_calories_for_date(date: &str, database: &FoodDatabase): u32
load_from_file(path: &Path): io::Result<()>
save_to_file(path: &Path): io::Result<()>
contains
1
1
UserManager
users: HashMap<String, String>
data_dir: PathBuf
new(data_dir: PathBuf): Self
register_user(username: &str, password: &str): bool
authenticate(username: &str, password: &str): bool
save_users(): void
get_user_dir(username: &str): PathBuf
contains
1
*
Food
id: FoodId
name: String
keywords: Vec<String>
calories_per_serving: u32
is_composite: bool
components: Vec<(FoodId, u32)>
new_basic(id: &str, name: &str, keywords: Vec<String>, calories: u32): Self
new_composite(id: &str, name: &str, keywords: Vec<String>, components: Vec<(FoodId, u32)>): Self
calculate_calories(database: &FoodDatabase): void
matches_keywords(search_keywords: &[String], match_all: bool): bool
to_string(): String
1
*
may contain components
stores in undo_stack
1
*
CommandType
AddFood(String, FoodEntry)
DeleteFood(String, FoodEntry)
contains
1
*
FoodEntry
food_id: FoodId
servings: u32
timestamp: u64
new(food_id: &str, servings: u32): Self
to_string(): String
from_string(s: &str): Option<Self>
contains user credentials
1
*
String

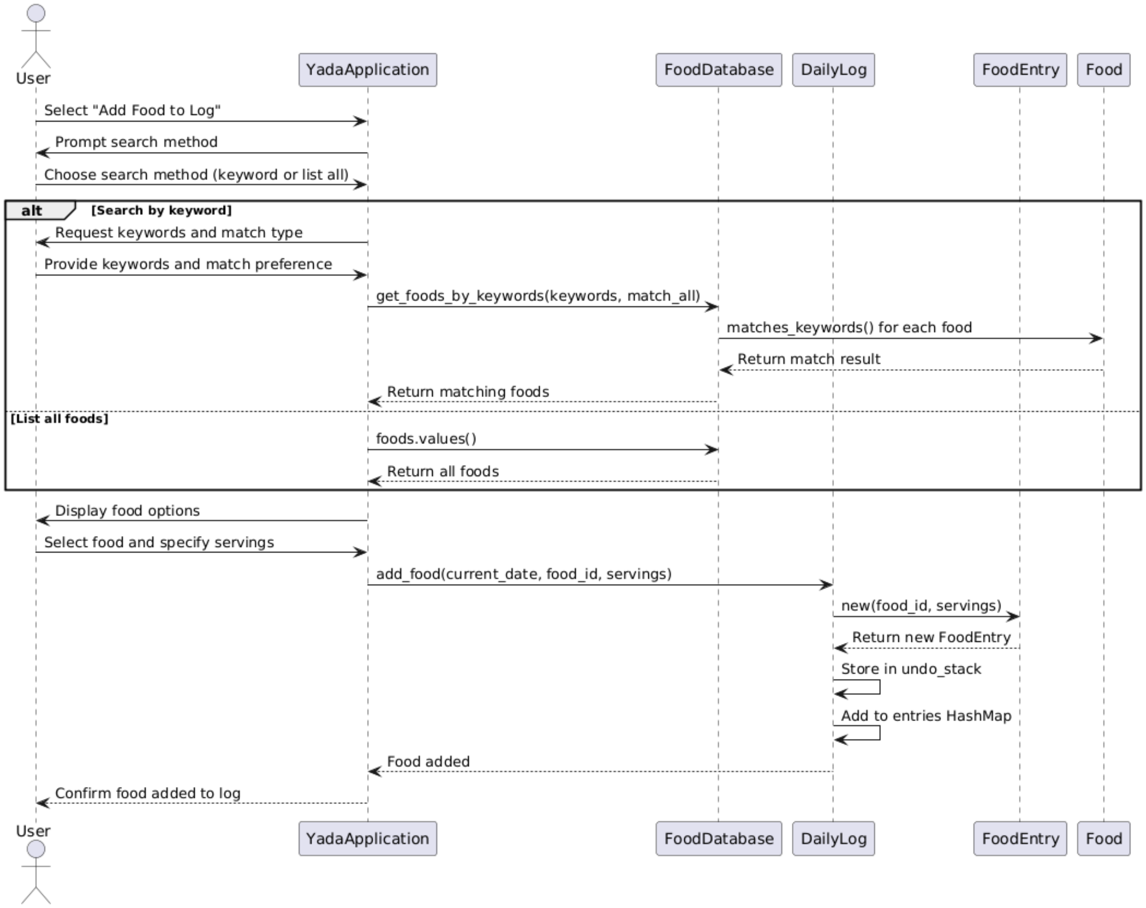
Drive Link –
https://drive.google.com/file/d/105Wlfkgz0jk7aO288LpePF-V-sasUv7e/view?usp=sharing

Sequence Diagram(s)

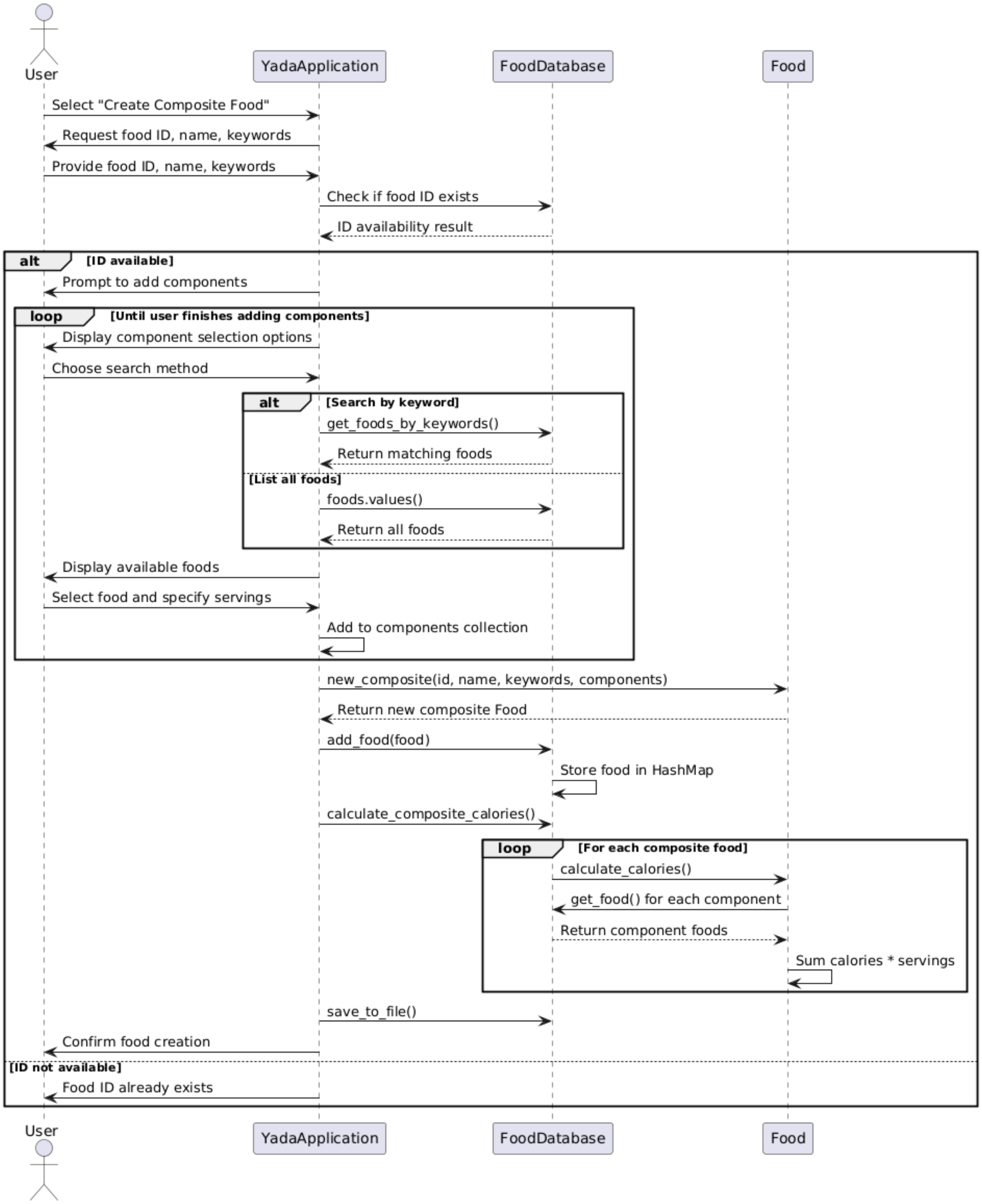
1. User Registration Sequence Diagram –


User
YadaApplication
UserManager
FoodDatabase
UserProfile
Register with credentials
register_user(username, password)
Check if user exists
alt
[User does not exist]
Add user to HashMap
Create user directory
save_users()
true (registration successful)
Check if food database exists
alt
[Food database doesn't exist]
create_sample_data()
add_food() for each sample food
calculate_composite_calories()
save_to_file()
Prompt for profile information
Provide gender, height, age, weight, activity level
new(username, gender, height, age, weight, activity)
Return new UserProfile
save_user_data()
Registration complete
[User already exists]
false (registration failed)
Username already exists
User
YadaApplication
UserManager
FoodDatabase
UserProfile

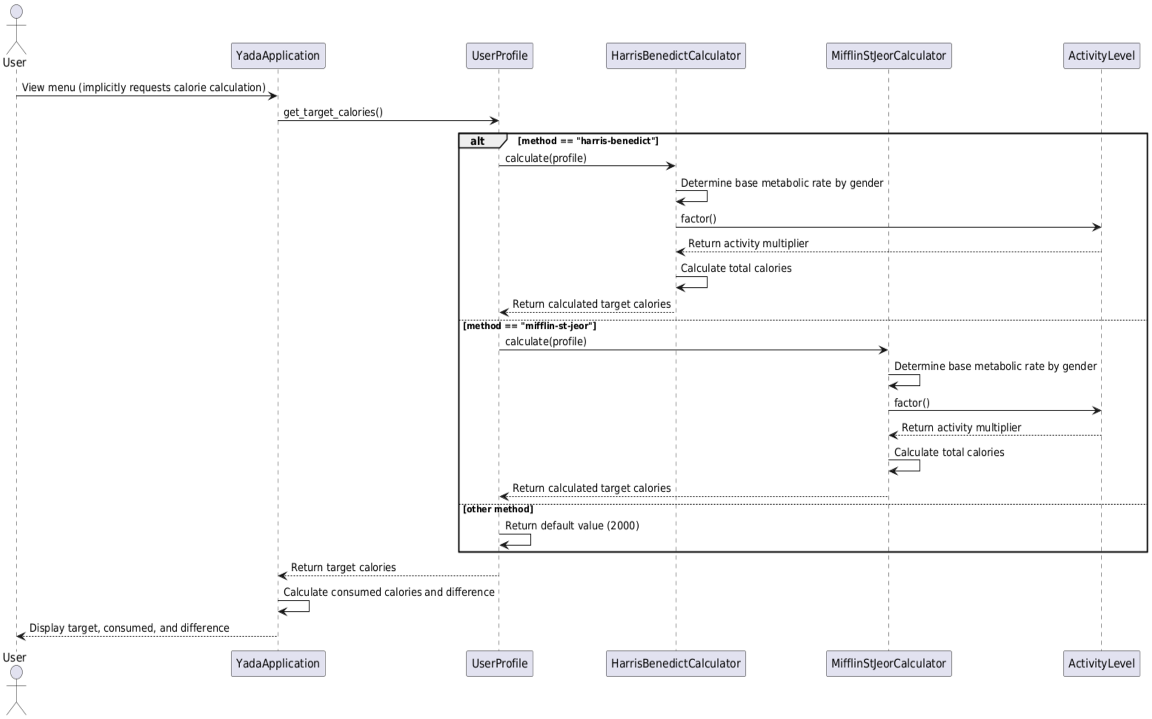
2. Add Food to daily log Sequence Diagram –



3. Create Composite food Sequence Diagram –



4. Calculate Target Calories Sequence Diagram –


User
YadaApplication
UserProfile
HarrisBenedictCalculator
MifflinStJeorCalculator
ActivityLevel
View menu (implicitly requests calorie calculation)
get_target_calories()
alt
[method == "harris-benedict"]
calculate(profile)
Determine base metabolic rate by gender
factor()
Return activity multiplier
Calculate total calories
Return calculated target calories
[method == "mifflin-st-jeor"]
calculate(profile)
Determine base metabolic rate by gender
factor()
Return activity multiplier
Calculate total calories
Return calculated target calories
[other method]
Return default value (2000)
Return target calories
Calculate consumed calories and difference
Display target, consumed, and difference
User
YadaApplication
UserProfile
HarrisBenedictCalculator
MifflinStJeorCalculator
ActivityLevel

Design Narrative

The YADA Diet Manager design reflects a thoughtful balance of competing software design principles. We've carefully considered trade-offs between coupling, cohesion, separation of concerns, information hiding, and other design principles to create a maintainable and extensible system.

Low Coupling - Our design achieves low coupling through several key mechanisms:

1. **Interface-Based Design**: We've used traits like CalorieCalculator and FoodDataSource to decouple concrete implementations from clients. The UserProfile doesn't need to know the internal details of calorie calculation algorithms, it simply invokes the chosen strategy.
2. **Data Transfer Objects**: By using a string-based ID system (FoodId) to reference foods in log entries, we've reduced dependency between the DailyLog and Food classes.
3. **Modular Subsystems**: Each major component (user management, food database, daily log) is self-contained and communicates through well-defined interfaces. For example, the UserManager handles all user-related operations without needing to know about diet planning.

High Cohesion - We've ensured high cohesion by organizing functionality into logically related classes:

1. **Single Responsibility Classes**: Each class has a clear purpose - Food manages food information, DailyLog handles food consumption records, and UserProfile maintains user demographic data.
2. **Functional Cohesion**: Methods within each class are strongly related to each other. For example, all methods in the FoodDatabase class deal with storing, retrieving, and managing food items.
3. **Service-Oriented Design**: Classes like UserManager provide a cohesive set of services related to a specific domain concept (user authentication and registration).

Separation of Concerns - The design separates distinct concerns into dedicated components:

1. **Data Storage vs. Business Logic**: The persistence mechanisms (load_from_file, save_to_file) are separated from core logic.
2. **User Interface vs. Core Functionality**: The YadaApplication handles UI interactions, delegating actual business operations to specialized classes.
3. **Authentication vs. Application Logic**: User authentication is handled by a dedicated UserManager class, keeping authentication concerns separate from diet management.

Information Hiding - We've employed information hiding to encapsulate implementation details:

1. **Private Fields**: Internal state is kept private within each class, with access controlled through methods.
2. **Encapsulated File Formats**: The specific format of stored data is encapsulated within the persistence methods, allowing it to change without affecting clients.
3. **Implementation Details**: Complex calculations, like composite food calorie computation, are hidden behind simple interfaces.

The Law of Demeter - The design follows the Law of Demeter (principle of least knowledge) in several ways:

1. **Method Chaining Limitations**: Classes generally interact only with their immediate dependencies without reaching through them to access deeper objects.
2. **Parameter Passing**: Required objects are passed as parameters rather than being accessed through global variables or long navigation chains.
3. **Localized Knowledge**: Each class only knows about its immediate collaborators, not the entire object graph.

Strategic Design Patterns - We've employed strategic design patterns to address specific requirements:

1. **Strategy Pattern**: The CalorieCalculator trait with concrete implementations enables easy addition of new calculation methods without modifying existing code.
2. **Command Pattern**: The CommandType enum and undo stack implementation provide robust undo functionality.
3. **Composite Pattern**: The food system uses a composite pattern to represent both basic and composite foods uniformly.

Design Trade-offs

We've made deliberate trade-offs to balance competing principles:

1. **Simplicity vs. Full Separation**: While we could have further separated concerns with more interfaces, we chose a pragmatic approach that balances separation with simplicity.
2. **Performance vs. Flexibility**: We opted for a more flexible design at the cost of some performance overhead, especially in the food reference system.
3. **Text-Based Storage vs. Binary Efficiency**: We prioritized human-readable storage formats over more efficient binary representations to meet the requirement for text files editable in standard text editors.

Reflection of Design

Strongest Aspects

1. **Extensibility for Calorie Calculation Methods**: The implementation of the CalorieCalculator trait and concrete calculator classes exemplifies our strongest design feature. New calculation methods can be added simply by implementing the trait and registering a new strategy, without touching existing code. This directly addresses the requirement that "new ways of computing target calories must be easy to add without ripple effects."

2. **Food Composition System**: Our food system elegantly handles both basic and composite foods with a unified interface. The recursive calculation of calories for composite foods demonstrates good use of the Composite pattern. The system also efficiently stores food entries by reference rather than duplication, addressing the requirement to "reduce or eliminate duplicate copies of objects."

Weakest Aspects

1. **Command Pattern Implementation**: While our undo system works, it could be more comprehensive. The current implementation only handles undoing log actions and profile updates, but doesn't extend to all operations in the system. A more thorough approach would apply the Command pattern universally, making every significant action undoable.

2. **Data Validation**: The design lacks robust validation mechanisms for input data. While there is basic error handling for parsing values from strings, a more comprehensive validation framework would improve data integrity and error reporting. Input validation is scattered throughout the codebase rather than being handled through a unified approach, which could lead to inconsistencies.