YADA (Yet Another Diet Assistant)

9 April 2025

Dev Kanani - 2023101053

Kavya Bhalodi - 2023101031

OVERVIEW

YADA (Yet Another Diet Assistant) is a simple yet powerful diet management system built to help users take control of their food habits by tracking what they eat, managing daily logs, and calculating calorie needs based on their personal profile. It supports adding both basic and composite foods, lets users log and review their intake by date, and calculates target calorie intake using multiple switchable formulas. With a command-line interface and text-based storage, YADA keeps things straightforward while still offering a strong design that's ready to grow in future versions.

KEY COMPONENTS

Food Management

- **Food Types**:
    - **BasicFood**: Represents individual food items with nutritional information (e.g., calories, proteins, fats, etc.).
    - **CompositeFood**: Represents a combination of multiple food items, useful for logging meals.

- **Storage**:
    - Data is persisted in two JSON files: basic_foods.json and composite_foods.json.

- **Operations**:
    - Users can **add**, **search**, and **view** food details.

Daily Log

- **Functionality**:
    - Users can **log daily food intake** by specifying food ID and number of servings.
    - The system computes the **total calories consumed** for the day.
    - It compares this value with the user's **target calorie intake** for effective tracking.
- **Undo Feature**:
    - Implemented using the **Command Pattern**, allowing users to **undo** adding or removing food entries.

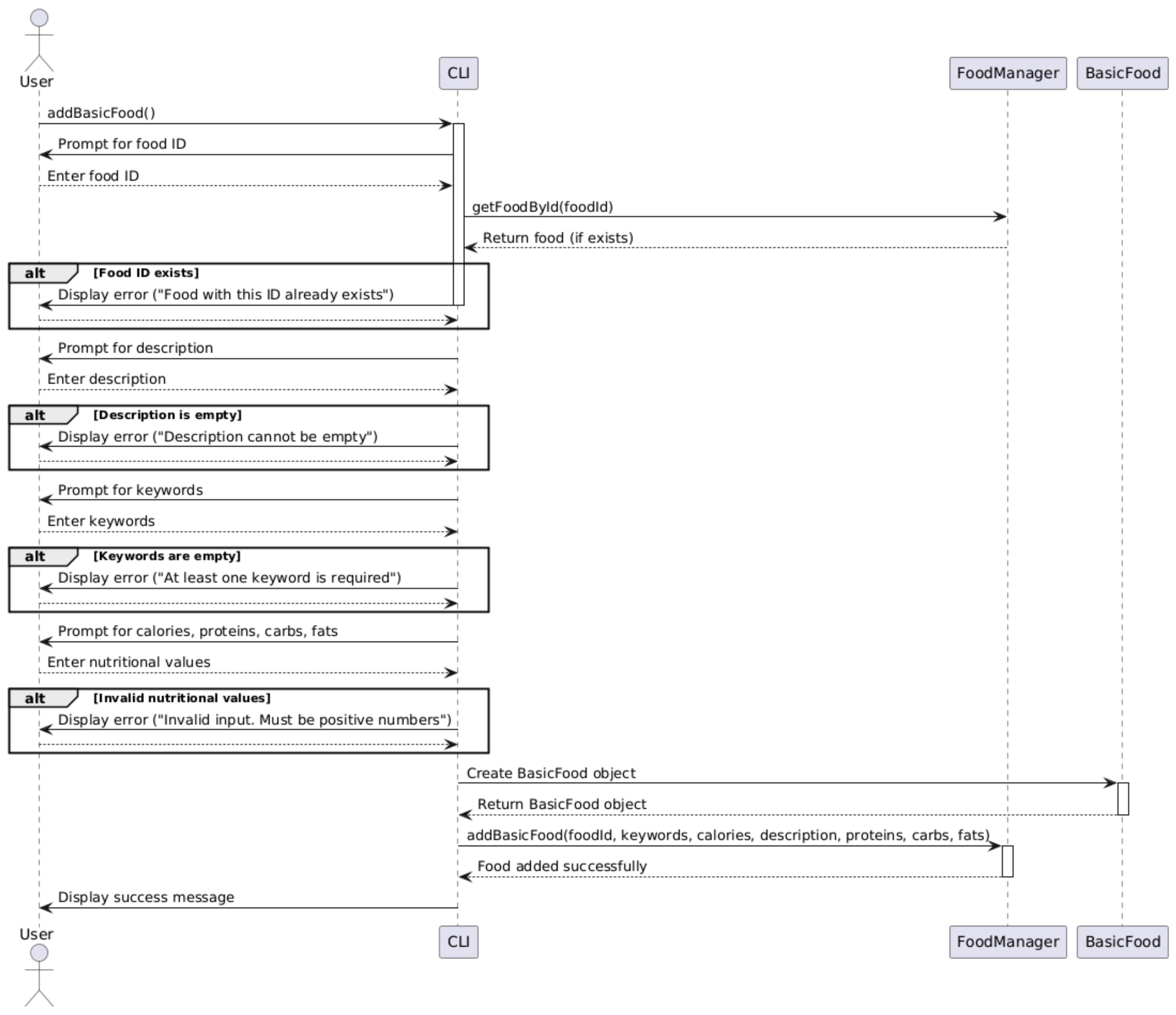
User Profile

- **Details Maintained**:
    - Gender, height, weight, age, and activity level.

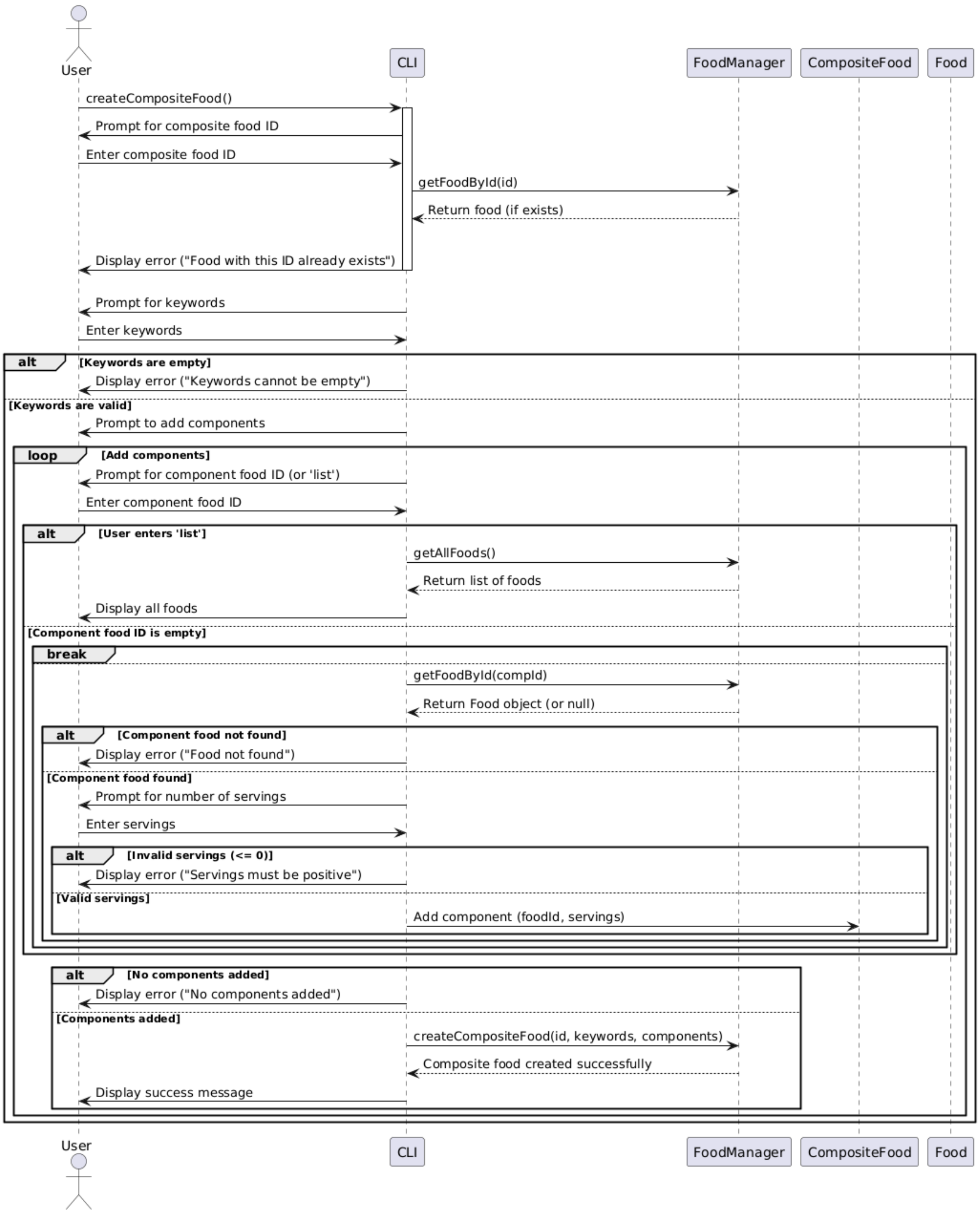
- **Calorie Requirement Calculation**:
    - Supports two methods:
        - **Harris-Benedict Equation**
        - **Mifflin-St Jeor Equation**

SEQUENCE DIAGRAMS

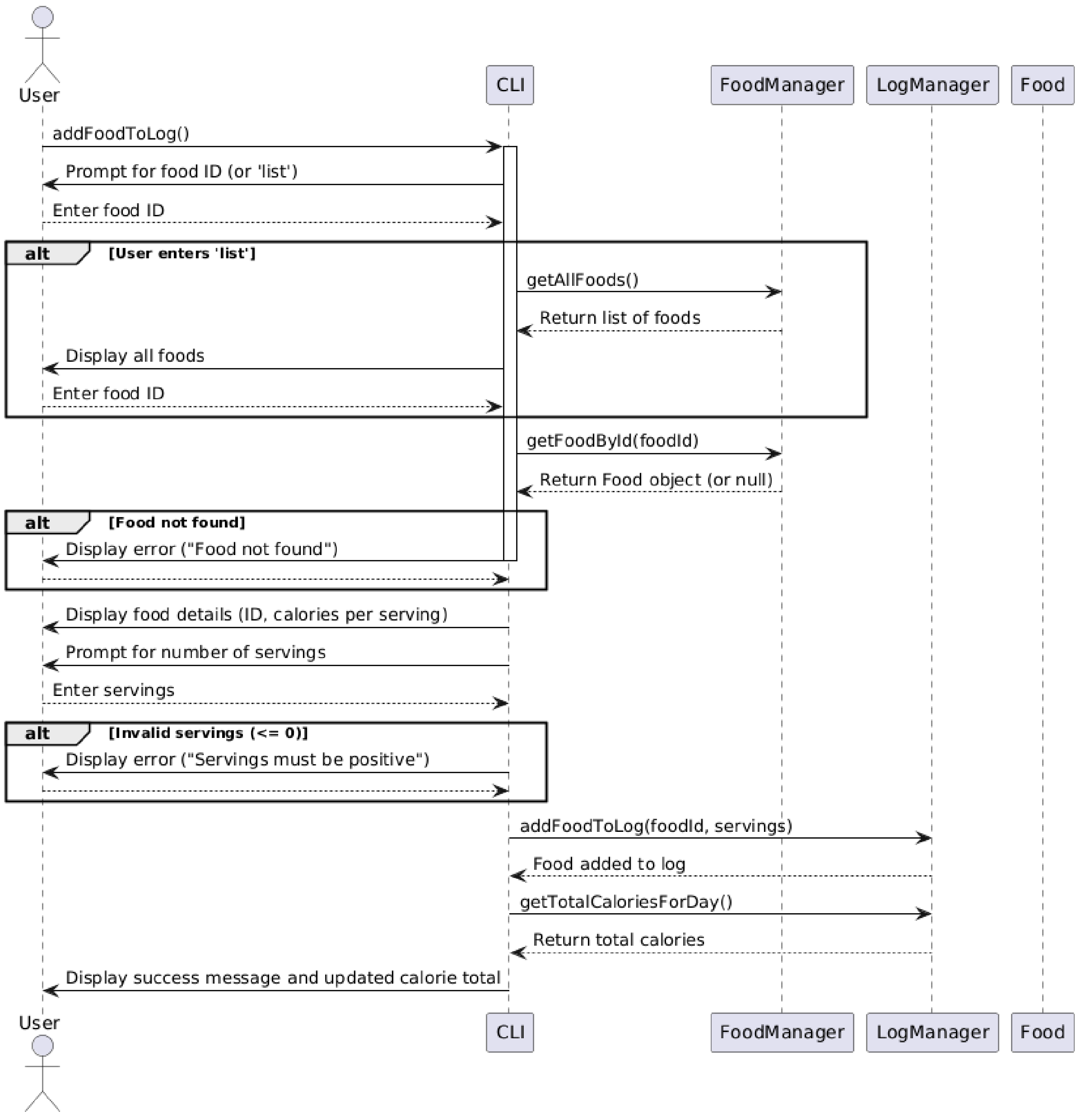
1. Add Basic Food


User
CLI
FoodManager
BasicFood
addBasicFood()
Prompt for food ID
Enter food ID
getFoodById(foodId)
Return food (if exists)
alt [Food ID exists]
Display error ("Food with this ID already exists")
Prompt for description
Enter description
alt [Description is empty]
Display error ("Description cannot be empty")
Prompt for keywords
Enter keywords
alt [Keywords are empty]
Display error ("At least one keyword is required")
Prompt for calories, proteins, carbs, fats
Enter nutritional values
alt [Invalid nutritional values]
Display error ("Invalid input. Must be positive numbers")
Create BasicFood object
Return BasicFood object
addBasicFood(foodId, keywords, calories, description, proteins, carbs, fats)
Food added successfully
Display success message
User
CLI
FoodManager
BasicFood

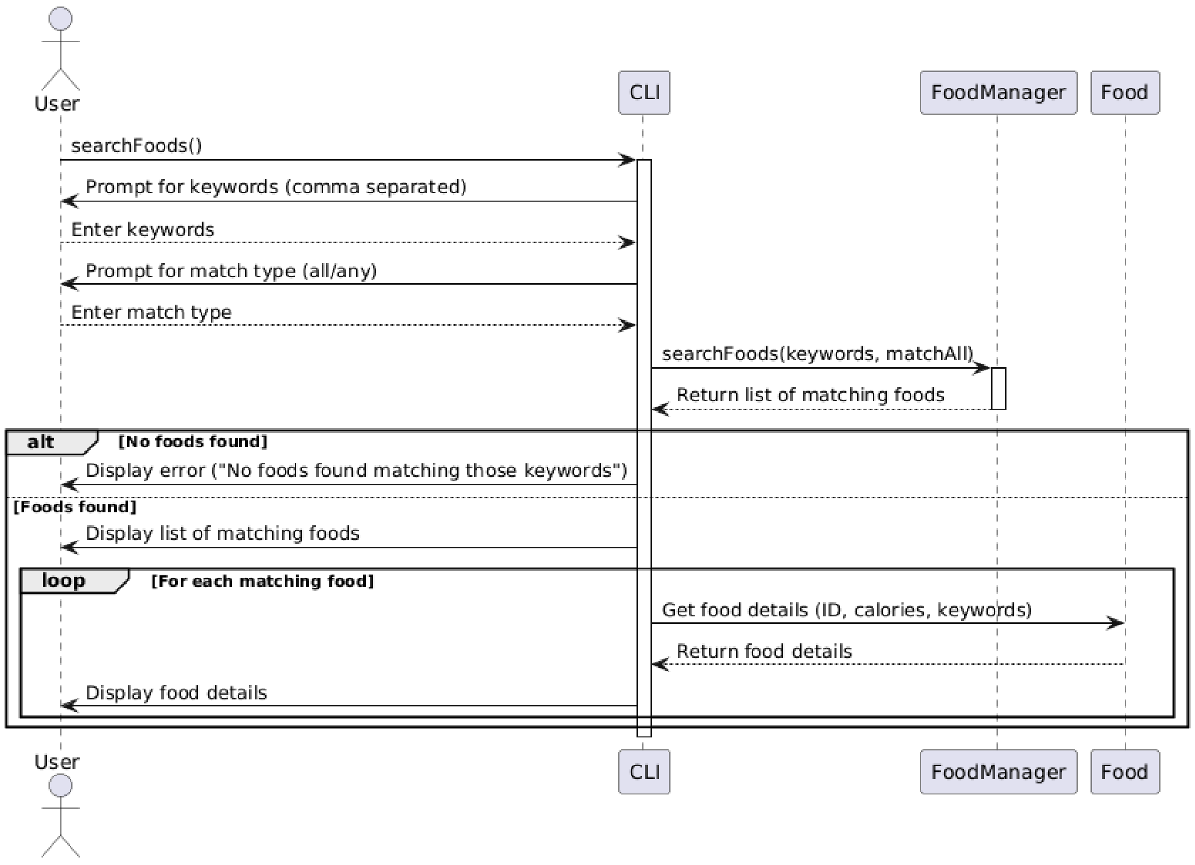
2. Add Composite Food


User
CLI
FoodManager
CompositeFood
Food
createCompositeFood()
Prompt for composite food ID
Enter composite food ID
getFoodById(id)
Return food (if exists)
Display error ("Food with this ID already exists")
Prompt for keywords
Enter keywords
alt [Keywords are empty]
Display error ("Keywords cannot be empty")
[Keywords are valid]
Prompt to add components
loop [Add components]
Prompt for component food ID (or 'list')
Enter component food ID
alt [User enters 'list']
getAllFoods()
Return list of foods
Display all foods
[Component food ID is empty]
break
getFoodById(compId)
Return Food object (or null)
alt [Component food not found]
Display error ("Food not found")
[Component food found]
Prompt for number of servings
Enter servings
alt [Invalid servings (<= 0)]
Display error ("Servings must be positive")
[Valid servings]
Add component (foodId, servings)
alt [No components added]
Display error ("No components added")
[Components added]
createCompositeFood(id, keywords, components)
Composite food created successfully
Display success message

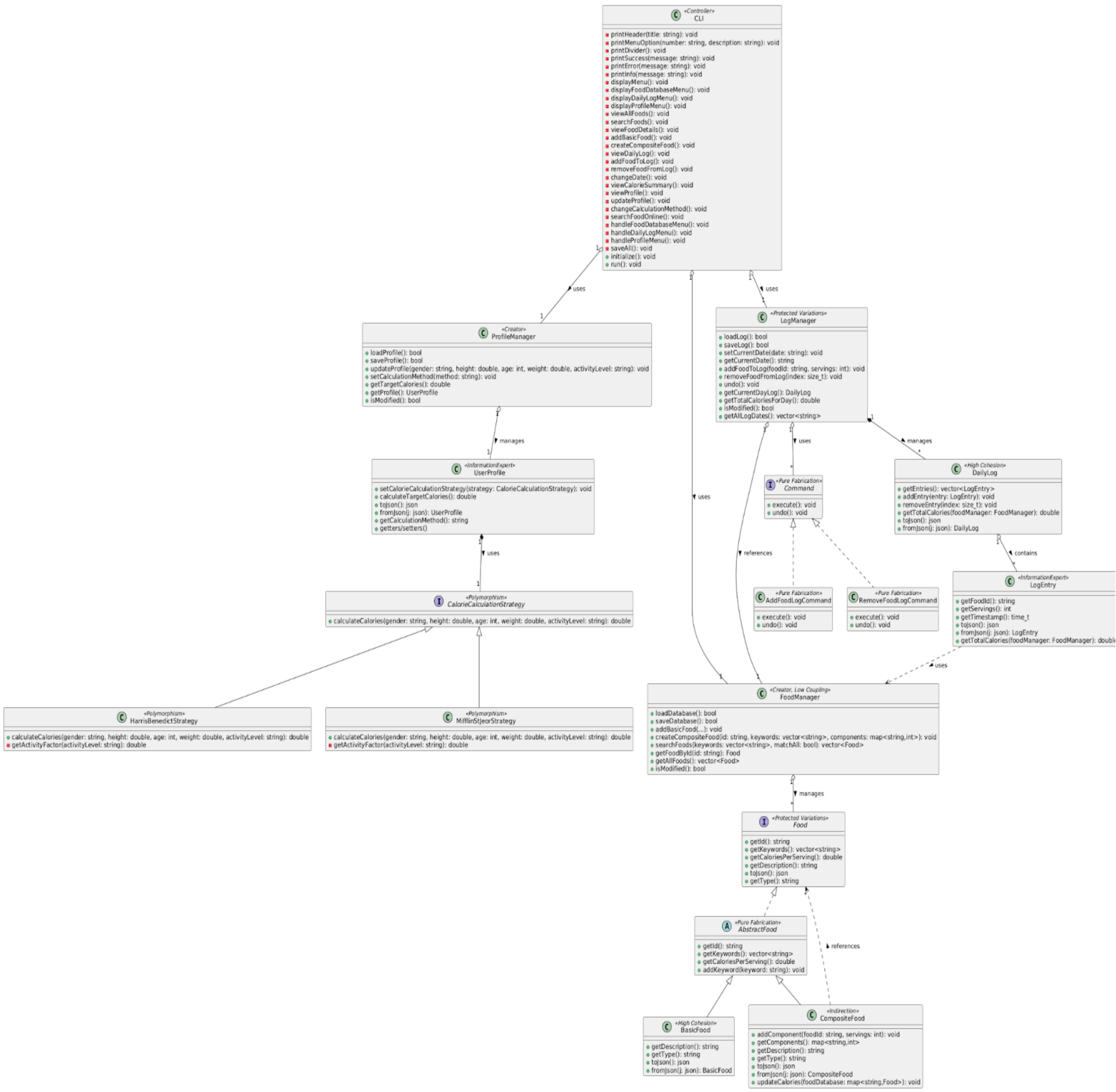
3. Add New Log


User
CLI
FoodManager
LogManager
Food
addFoodToLog()
Prompt for food ID (or 'list')
Enter food ID
alt [User enters 'list']
getAllFoods()
Return list of foods
Display all foods
Enter food ID
getFoodById(foodId)
Return Food object (or null)
alt [Food not found]
Display error ("Food not found")
Display food details (ID, calories per serving)
Prompt for number of servings
Enter servings
alt [Invalid servings (<= 0)]
Display error ("Servings must be positive")
addFoodToLog(foodId, servings)
Food added to log
getTotalCaloriesForDay()
Return total calories
Display success message and updated calorie total
User
CLI
FoodManager
LogManager
Food

4. Search


User
CLI
FoodManager
Food
searchFoods()
Prompt for keywords (comma separated)
Enter keywords
Prompt for match type (all/any)
Enter match type
searchFoods(keywords, matchAll)
Return list of matching foods
alt
[No foods found]
Display error ("No foods found matching those keywords")
[Foods found]
Display list of matching foods
loop
[For each matching food]
Get food details (ID, calories, keywords)
Return food details
Display food details
User
CLI
FoodManager
Food

UML CLASS DIAGRAM


«Controller»
CLI
printHeader(title: string): void
printMenuOption(number: string, description: string): void
printDivider(): void
printSuccess(message: string): void
printError(message: string): void
printInfo(message: string): void
displayMenu(): void
displayFoodDatabaseMenu(): void
displayDailyLogMenu(): void
displayProfileMenu(): void
viewAllFoods(): void
searchFoods(): void
viewFoodDetails(): void
addBasicFood(): void
createCompositeFood(): void
viewDailyLog(): void
addFoodToLog(): void
removeFoodFromLog(): void
changeDate(): void
viewCalorieSummary(): void
viewProfile(): void
updateProfile(): void
changeCalculationMethod(): void
searchFoodOnline(): void
handleFoodDatabaseMenu(): void
handleDailyLogMenu(): void
handleProfileMenu(): void
saveAll(): void
initialize(): void
run(): void
uses
uses
uses
«Creator»
ProfileManager
loadProfile(): bool
saveProfile(): bool
updateProfile(gender: string, height: double, age: int, weight: double, activityLevel: string): void
setCalculationMethod(method: string): void
getTargetCalories(): double
getProfile(): UserProfile
isModified(): bool
manages
«InformationExpert»
UserProfile
setCalorieCalculationStrategy(strategy: CalorieCalculationStrategy): void
calculateTargetCalories(): double
toJson(): json
fromJson(j: json): UserProfile
getCalculationMethod(): string
getters/setters()
uses
«Polymorphism»
CalorieCalculationStrategy
calculateCalories(gender: string, height: double, age: int, weight: double, activityLevel: string): double
«Polymorphism»
HarrisBenedictStrategy
calculateCalories(gender: string, height: double, age: int, weight: double, activityLevel: string): double
getActivityFactor(activityLevel: string): double
«Polymorphism»
MifflinStJeorStrategy
calculateCalories(gender: string, height: double, age: int, weight: double, activityLevel: string): double
getActivityFactor(activityLevel: string): double
«Protected Variations»
LogManager
loadLog(): bool
saveLog(): bool
setCurrentDate(date: string): void
getCurrentDate(): string
addFoodToLog(foodId: string, servings: int): void
removeFoodFromLog(index: size_t): void
undo(): void
getCurrentDayLog(): DailyLog
getTotalCaloriesForDay(): double
isModified(): bool
getAllLogDates(): vector<string>
uses
manages
references
«Pure Fabrication»
Command
execute(): void
undo(): void
«Pure Fabrication»
AddFoodLogCommand
execute(): void
undo(): void
«Pure Fabrication»
RemoveFoodLogCommand
execute(): void
undo(): void
«High Cohesion»
DailyLog
getEntries(): vector<LogEntry>
addEntry(entry: LogEntry): void
removeEntry(index: size_t): void
getTotalCalories(foodManager: FoodManager): double
toJson(): json
fromJson(j: json): DailyLog
contains
«InformationExpert»
LogEntry
getFoodId(): string
getServings(): int
getTimestamp(): time_t
toJson(): json
fromJson(j: json): LogEntry
getTotalCalories(foodManager: FoodManager): double
uses
«Creator, Low Coupling»
FoodManager
loadDatabase(): bool
saveDatabase(): bool
addBasicFood(...): void
createCompositeFood(id: string, keywords: vector<string>, components: map<string,int>): void
searchFoods(keywords: vector<string>, matchAll: bool): vector<Food>
getFoodById(id: string): Food
getAllFoods(): vector<Food>
isModified(): bool
manages
«Protected Variations»
Food
getId(): string
getKeywords(): vector<string>
getCaloriesPerServing(): double
getDescription(): string
toJson(): json
getType(): string
«Pure Fabrication»
AbstractFood
getId(): string
getKeywords(): vector<string>
getCaloriesPerServing(): double
addKeyword(keyword: string): void
references
«High Cohesion»
BasicFood
getDescription(): string
getType(): string
toJson(): json
fromJson(j: json): BasicFood
«Indirection»
CompositeFood
addComponent(foodId: string, servings: int): void
getComponents(): map<string,int>
getDescription(): string
getType(): string
toJson(): json
fromJson(j: json): CompositeFood
updateCalories(foodDatabase: map<string,Food>): void

NARRATIVE OF DESIGN

1) Separation of Concerns

- **food.cpp** handles food definitions and the FoodManager logic.
- **profile.cpp** encapsulates user data and calorie calculation strategies.
- **log.cpp** manages daily logs, undo/redo, and food logging commands.
- **cli.cpp** serves purely as the interface layer with clean UI presentation logic.
- **main.cpp** delegates responsibility immediately to the CLI without bloating.

2) Low Coupling

- LogManager depends on FoodManager, but only through a reference passed during construction — this keeps it decoupled and easy to mock for testing.
- CLI aggregates FoodManager, LogManager, and ProfileManager, but doesn't tangle their responsibilities. It **invokes**, not **manages**, their logic.
- LogEntry doesn’t interact directly with Food; it only stores a foodId, leaving resolution to LogManager.

  This decoupling allows you to change how food or profile data is stored or processed without rewriting CLI or log components.

3) High Cohesion

- UserProfile only handles user attributes and calorie calculation delegation.
- DailyLog is solely concerned with a day's entries and calorie totals.
- FoodManager handles all loading, saving, and searching of food data.

- The AddFoodLogCommand and RemoveFoodLogCommand classes handle one specific log action each, like adding or removing food. They follow the **command pattern**, which means each action is wrapped in its own class to make it easier to run and undo later.

4) Information Hiding

- Member variables are private across the board (LogEntry, DailyLog, UserProfile), and exposed only via assessors like getCaloriesPerServing() or toJson().
- CompositeFood doesn't expose raw calorie math externally — instead it uses updateCalories() internally to maintain consistency.

5) Law of Demeter

"Only talk to immediate friends" — the code largely adheres:

- CLI talks to FoodManager, LogManager, and ProfileManager, but not their internal components directly.
- LogEntry never calls food methods directly, instead it requests calorie info through LogManager, preserving Demeter.

  This breaks the rule of keeping things private, just to make undo work. A better way would be to add a function like removeEntryById() to do it properly.

6) Extensibility & Flexibility

Code is extensible without modification:

- To add a new calorie formula, just extend CalorieCalculationStrategy.
- To add a new food type, subclass AbstractFood and register it in FoodManager.

WHY THE ABOVE PRINCIPLES ARE USED FOR CLASSES

1) Polymorphism

- **CalorieCalculationStrategy**
  Used as an interface so different strategies can be swapped (like Harris or Mifflin), enabling flexible behavior.

- **HarrisBenedictStrategy** & **MifflinStJeorStrategy**
  Implement the strategy interface; the app can choose either at runtime without changing how it uses them.

2) InformationExpert

- **UserProfile**
  Knows all the user's personal data, so it's the right place to calculate target calories.

- **LogEntry**
  Has all the data needed for a single log, so it can compute its own calorie total.

3) Creator

- **ProfileManager**
  Responsible for managing a user profile, so it also creates and updates it.

- **FoodManager**
  Manages food items, so it also creates new basic and composite foods.

4) Protected Variations

- **Food**
  Interface allows changing food types without affecting the rest of the system.

- **LogManager**
  Supports adding/removing log entries using commands, so logging logic can change easily.

5) Pure Fabrication

- **AbstractFood**
  Doesn't represent a real-world concept but helps reduce code duplication for food types.

- **Command, AddFoodLogCommand, RemoveFoodLogCommand**
  These classes encapsulate log operations so you can undo/redo them—logic-focused, not real-world objects.

6) Indirection

- **CompositeFood**
  Sits between the user and multiple food components; helps manage food mixtures more flexibly.

7) High Cohesion

- **BasicFood**
  Has tightly related functions and data (nutrition info) that all support its main purpose.

- **DailyLog**
  Focuses only on one thing—managing entries for a single day—making it clear and focused.

8) Low Coupling

- **FoodManager**
  Well-isolated; other classes interact with it through simple methods, not deep internals.

9) Controller

- **CLI**
  Handles user interaction and coordinates between different parts of the system.

STRONGEST ASPECTS

1) **Modular and Extensible Design:**

Your use of object-oriented design is very good for this particular design. Classes are encapsulated (Food, UserProfile, LogEntry, etc.) with clean responsibilities. The use of inheritance (e.g., BasicFood, CompositeFood) and design patterns like Strategy (for calorie calculation) and Command (for undo functionality).

2) **Separation of Concerns and Intuitive CLI:**

The system is well-structured with clear separation of responsibilities. For instance:FoodManager handles food-related operations. LogManager manages daily logs and undo functionality. ProfileManager handles user profile data and calculations. CLI focuses on user interaction and presentation. This separation improves maintainability and readability. The command-line interface is thoughtfully designed, with clear menus, color-coded feedback, and robust input validation. The CLI not only guides users well but is also easy to extend — evident in how online food search functionality was integrated.

WEAKEST ASPECTS

1) **Coupling Between CLI and Core Logic:**

The CLI directly includes and interacts with core components like food.cpp, log.cpp, and profile.cpp. This tight coupling reduces testability and violates separation of concerns. A better approach would be to use interfaces or service layers between the CLI and core logic to promote cleaner architecture. The length of the CLI class is large as of now, which can be modified in the future versions.

2) **Lack of Unit Tests:**

The codebase lacks automated unit tests for critical components like FoodManager, LogManager, and ProfileManager. This absence makes it harder to ensure the correctness of the system and increases the risk of introducing bugs during future modifications. While care has been taken for almost all while manual testing, but in real words problems may arise.

Class Documentation

Food

- **Methods:**
  - getId(): Returns the food ID.
  - getKeywords(): Returns a list of keywords associated with the food.
  - getCaloriesPerServing(): Returns the calories per serving.
  - getDescription(): Returns a description of the food.
  - toJson(): Serializes the food to JSON.
  - getType(): Returns the type of the food (e.g., "basic" or "composite").

AbstractFood

- **Attributes:**
  - id: Unique identifier for the food.
  - keywords: List of keywords associated with the food.
  - caloriesPerServing: Calories per serving.
- **Methods:**
  - addKeyword(keyword): Adds a keyword to the food.

BasicFood

- **Attributes:**
  - description: Description of the food.
  - proteins: Protein content per serving.
  - carbs: Carbohydrate content per serving.
  - fats: Fat content per serving.
- **Methods:**
  - getDescription(): Returns a detailed description of the food.
  - getType(): Returns "basic".
  - toJson(): Serializes the food to JSON.
  - fromJson(json): Creates a BasicFood object from JSON.

CompositeFood

- **Attributes:**
    - components: Map of food IDs to servings.
- **Methods:**
    - addComponent(foodId, servings): Adds a component to the composite food.
    - getComponents(): Returns the components of the composite food.
    - getDescription(): Returns a description of the composite food.
    - getType(): Returns "composite".
    - toJson(): Serializes the food to JSON.
    - fromJson(json): Creates a CompositeFood object from JSON.
    - updateCalories(foodDatabase): Updates the calorie count based on the components.

BasicFoodFactory

- **Methods:**
    - createBasicFood(data): Creates a BasicFood object from JSON data.

JsonBasicFoodFactory

- **Methods:**
    - createBasicFood(data): Creates a BasicFood object from JSON data.

FoodManager

Manages the food database.

- **Attributes:**
    - foodDatabase: Map of food IDs to Food objects.
    - modified: Tracks whether the database has been modified.
- **Methods:**
    - loadFromFile(filename): Loads foods from a file.
    - loadDatabase(): Loads the food database.
    - saveDatabase(): Saves the food database.
    - addBasicFood(...): Adds a new basic food.

    - createCompositeFood(...): Creates a new composite food.
    - searchFoods(keywords, matchAll): Searches for foods by keywords.
    - getFoodById(id): Retrieves a food by its ID.
    - getAllFoods(): Returns all foods in the database.
    - isModified(): Checks if the database has been modified.

LogEntry

- **Attributes:**
    - foodId: ID of the food.
    - servings: Number of servings.
- **Methods:**
    - toJson(): Serializes the log entry to JSON.
    - fromJson(json): Creates a LogEntry object from JSON.
    - getTotalCalories(foodManager): Calculates the total calories for the entry.

DailyLog

- **Attributes:**
    - entries: List of LogEntry objects.
- **Methods:**
    - addEntry(entry): Adds a log entry.
    - removeEntry(index): Removes a log entry by index.
    - getTotalCalories(foodManager): Calculates the total calories for the day.
    - toJson(): Serializes the log to JSON.
    - fromJson(json): Creates a DailyLog object from JSON.

Command

Interface for undoable commands.

- **Methods:**
    - execute(): Executes the command.

    - undo(): Undoes the command.

AddFoodLogCommand

- **Methods:**
    - execute(): Adds the food to the log.
    - undo(): Removes the food from the log.

RemoveFoodLogCommand

- **Methods:**
    - execute(): Removes the food from the log.
    - undo(): Restores the removed food.

LogManager

- **Attributes:**
    - logs: Map of dates to DailyLog objects.
    - undoStack: Stack of undoable commands.
    - currentDate: Current date.
- **Methods:**
    - loadLog(): Loads logs from a file.
    - saveLog(): Saves logs to a file.
    - addFoodToLog(foodId, servings): Adds food to the log.
    - removeFoodFromLog(index): Removes food from the log.
    - undo(): Undoes the last action.
    - getCurrentDayLog(): Returns the log for the current day.
    - getTotalCaloriesForDay(): Calculates the total calories for the current day.

CalorieCalculationStrategy

- **Methods:**
    - calculateCalories(...): Calculates target calories based on user profile.

HarrisBenedictStrategy

- **Methods:**
    - calculateCalories(...): Calculates target calories.

MifflinStJeorStrategy

- **Methods:**
    - calculateCalories(...): Calculates target calories.

UserProfile

- **Attributes:**
    - gender, height, age, weight, activityLevel: User details.
    - calorieStrategy: Strategy for calorie calculation.
- **Methods:**
    - calculateTargetCalories(): Calculates the target calories.
    - toJson(): Serializes the profile to JSON.
    - fromJson(json): Creates a UserProfile object from JSON.

ProfileManager

- **Attributes:**
    - profile: The user's profile.
    - modified: Tracks whether the profile has been modified.
- **Methods:**
    - loadProfile(): Loads the profile from a file.
    - saveProfile(): Saves the profile to a file.
    - updateProfile(...): Updates the profile details.
    - setCalculationMethod(method): Sets the calorie calculation method.
    - getTargetCalories(): Returns the target calories.

CLI

- **Methods:**
    - initialize(): Initializes the application by loading data.

- run(): Runs the main application loop.
- displayMenu(): Displays the main menu.
- handleFoodDatabaseMenu(): Handles the food database menu.
- handleDailyLogMenu(): Handles the daily log menu.
- handleProfileMenu(): Handles the profile menu.
- saveAll(): Saves all data.