Product Design

YADA (Yet Another Diet Assistant)

30/3/2025 Amey Bangera - 2023115019, Joshita Dhanireddy - 2023101095

Overview

YADA is a command-line diet management system that helps users track their food intake, calculate daily calorie goals, and manage their diet profile. The system is built using C++ and follows object-oriented design principles for modularity and extensibility.

Features

1. Comprehensive Food Management System
Basic Food Tracking - Stores essential nutritional data for single-ingredient foods
Composite Food Creation - Allows building complex foods from existing components
Dynamic Calorie Calculation - Automatically sums calories from all components while maintaining serving size relationships
Food Searching - Allows searching for foods with keywords matching (all or atleast one)

2. Daily Consumption Tracking
Date-Based Logging System - Stores entries by date (multiple entries per food allowed)
Log Manipulation - Add food with serving size, Remove specific entries (from any date), View daily log summary (on program start and any changes)
Nutritional Analysis - Real-time calculations (with 2 calculation methods - easily extendable to others) with total daily calories and comparison against target (for any date)

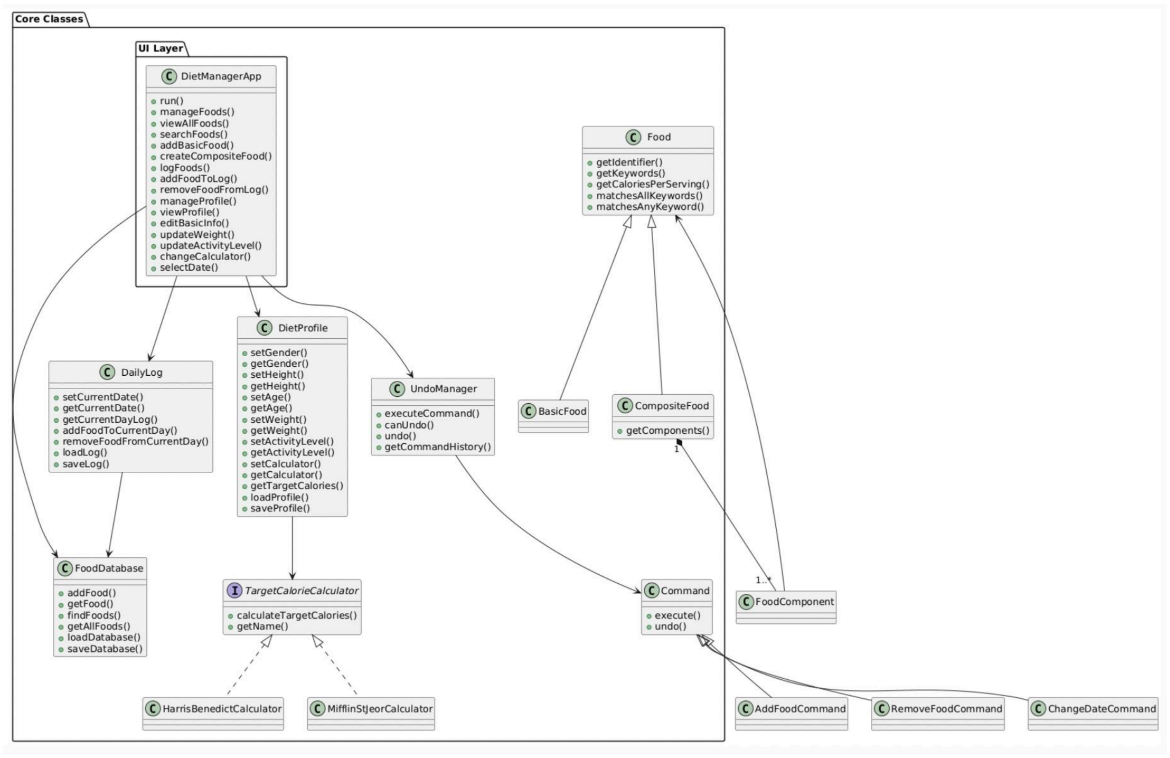
3. User Profile Management
Comprehensive profile data - Includes Gender, Height, Age, Weight, Activity level (5 tier scale). This data is updatable.
Calorie Calculation Engine - Multiple supported algorithms such as the Harris-Benedict Equation and the Mifflin-St Jeor Equation. User can choose which calculation engine to use.

4. Data Persistence
File-Based Storage - Three dedicated data files (human-readable) : *foods.txt* - Food database, *dailylog.txt* - Consumption records, *profile.txt* - User information. These files are automatically loaded on initial startup and exit.

5. Undo Functionality
Any update that has been saved can be undone. Undo does not persist across sessions (i.e. you cannot exit the program, re-run and try to undo an action you've performed previously)

These features are organised in the form of an interactive console interface with a hierarchical menu system.

Design Model (without state information)



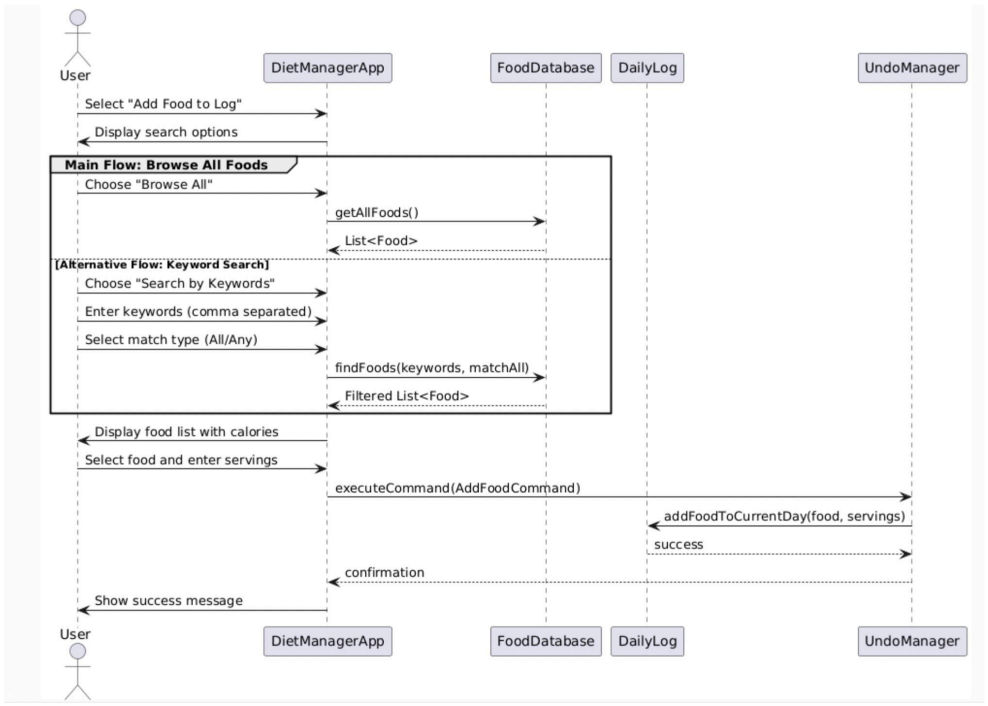
Food	- getCaloriesPerServing(): Returns calories per serving (pure virtual) - matchesAllKeywords(): Checks if food matches all given keywords - matchesAnyKeyword(): Checks if food matches any given keyword - getIdentifier(): Returns food identifier - getKeywords(): Returns food keywords
BasicFood	
CompositeFood	- getComponents(): Returns components
FoodDatabase	- addFood(): Adds food to database - getFood(): Retrieves food by ID - findFoods(): Finds foods matching keywords - getAllFoods(): Returns all foods - loadDatabase(): Loads from file - saveDatabase(): Saves to file
TargetCalorieCalculator	- calculateTargetCalories(): Calculates target calories - getName(): Returns calculator name
HarrisBenedictCalculator	
MifflinStJeorCalculator	

DietProfile	- Various setters/getters for profile attributes - getTargetCalories(): Calculates target calories - loadProfile(): Loads from file - saveProfile(): Saves to file
Command	- execute(): Executes command - undo(): Undoes command
AddFoodCommand	
RemoveFoodCommand	
ChangeDateCommand	
UndoManager	- executeCommand(): Executes and stores command - undo(): Undoes last command - canUndo(): Checks if undo is possible - getCommandHistory(): Returns command history
DailyLog	- Manages daily food entries - add/remove food entries - calculate total calories - load/save from/to file - date management
DietManagerApp	- Main application class that coordinates all components - Implements user interface and menu system - Manages data loading/saving - Allows profile updation in coordination with the Profile class

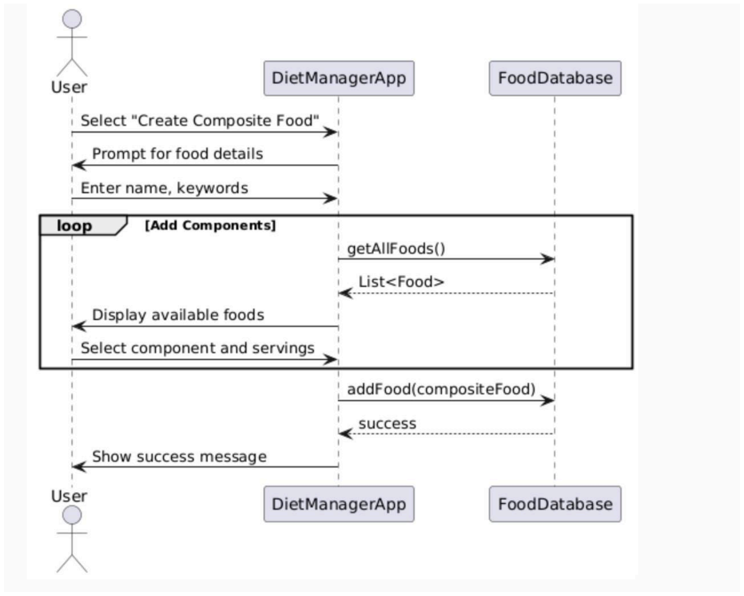
.

Sequence Diagrams

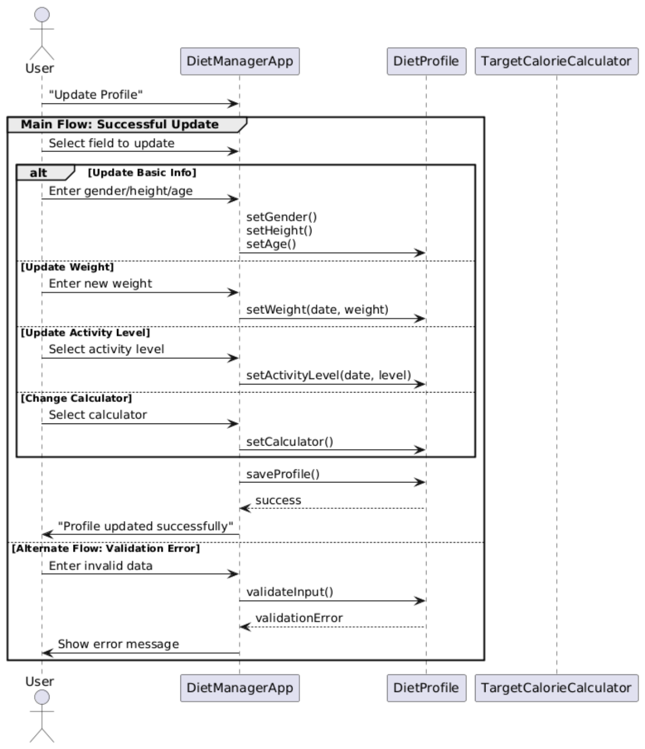
USE CASE 1 : Adding Food To Log


User
DietManagerApp
FoodDatabase
DailyLog
UndoManager
Select "Add Food to Log"
Display search options
Main Flow: Browse All Foods
Choose "Browse All"
getAllFoods()
List<Food>
[Alternative Flow: Keyword Search]
Choose "Search by Keywords"
Enter keywords (comma separated)
Select match type (All/Any)
findFoods(keywords, matchAll)
Filtered List<Food>
Display food list with calories
Select food and enter servings
executeCommand(AddFoodCommand)
addFoodToCurrentDay(food, servings)
success
confirmation
Show success message

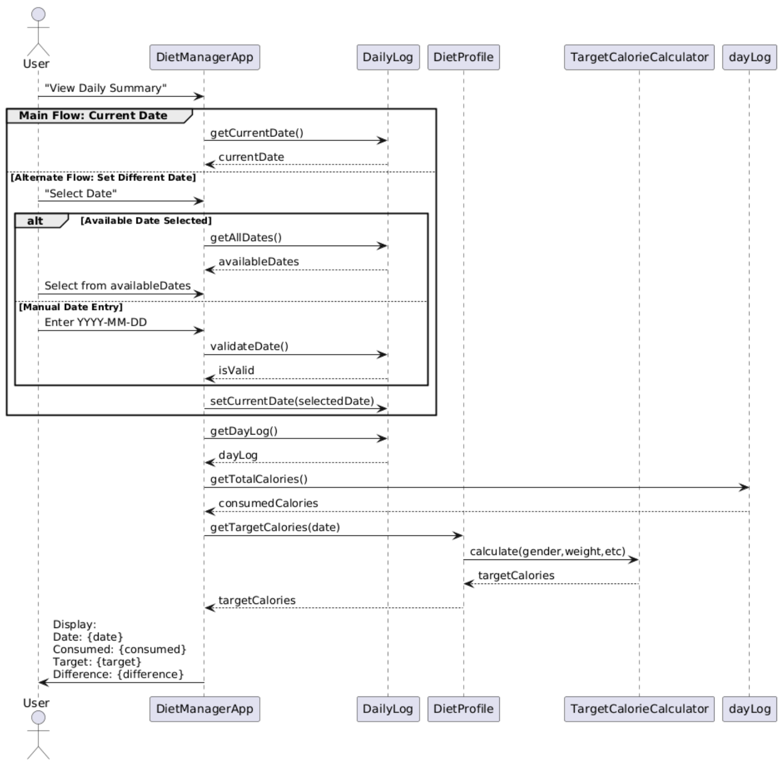
USE CASE 2 : Creating Composite Food



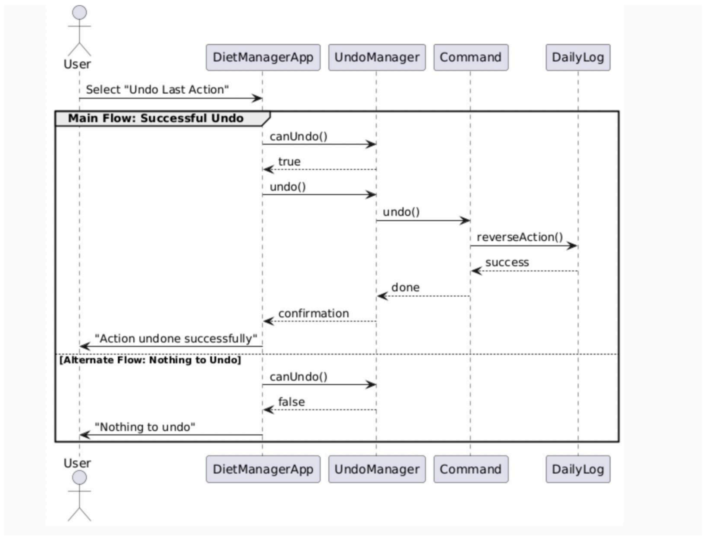
USE CASE 3 : Updating Profile



USE CASE 4 : Getting Daily Calorie Intake Summary (by selecting a Date)



USE CASE 5 : Undoing a Command



Design Narrative

Low Coupling and High Cohesion

Our system is divided into distinct components that have clear responsibilities:

- **Food hierarchy**: Handles food representation with `Food` as an abstract class and `BasicFood` and `CompositeFood` as concrete implementations
- **Calculators**: Encapsulate different algorithms for calorie calculation
- **DailyLog and DayLog**: Manage food consumption records
- **DietProfile**: Maintains user information
- **Command pattern**: Provides undo functionality
- **Observer pattern**: Enables reactive updates between components

Each class has a well-defined purpose, contributing to high cohesion. Dependencies between classes are managed through interfaces and shared pointers rather than direct references to implementations, reducing coupling.

Separation of Concerns

Our design effectively separates different concerns:

1. **Data representation**: `Food`, `DayLog`, `DietProfile` classes
2. **Persistence**: Each major component handles its own serialization
3. **Business logic**: Calculators, FoodTracker
4. **User interaction**: DietManagerApp
5. **Command execution**: Command hierarchy and UndoManager

This separation ensures that changes to one aspect of the system have minimal impact on others.

Information Hiding

Our implementation uses encapsulation effectively through:

- **Private data members**: Attributes are typically kept private
- **Public interfaces**: Classes expose functionality through well-defined methods
- **Shared pointers**: Implementation details of objects are hidden behind pointer interfaces

For example, the `Food` class hierarchy hides the details of how calories are calculated from clients who only need to call `getCaloriesPerServing()`.

Law of Demeter

Our design generally follows the Law of Demeter by limiting object interactions to immediate neighbors. For instance, the `DietManagerApp` interacts with high-level abstractions like `DailyLog`, `FoodDatabase`, and `DietProfile` without directly manipulating their internal components.

Observer Pattern Implementation

The Observer pattern is well-implemented to maintain consistency between components. When the `DailyLog` or `DietProfile` changes, observers like `FoodTracker` are notified and can update accordingly. This reduces the need for components to directly query each other for state changes.

Extensibility

New Sources of Nutritional Information

Our design shows clear separation of concerns that makes it easy to integrate new data sources:

1. Food Data Sources:
   - Our `FoodDatabase` class acts as a central repository
   - We can easily create site-specific adapters to fetch food data
2. Command Pattern Implementation:
   - We've used the Command pattern (`Command.cpp`) which makes adding new functionality straightforward
   - New commands can be created for importing from different websites without affecting existing code
3. Observer Pattern:
   - Our implementation of the Observer pattern (`Observer.cpp`) allows for efficient updates
   - New data sources can notify observers when data changes

New methods for Calorie Calculation

Our implementation uses the Strategy pattern for calorie calculations, making it simple to add new calculation methods:

1. Abstract Calculator Interface:
   - We have a base `TargetCalorieCalculator` interface
   - This defines the common method `calculateTargetCalories()` that all calculators must implement
2. Concrete Calculator Implementations:
   - `HarrisBenedictCalculator` and `MifflinStJeorCalculator` in `Calculator.cpp`
   - Each implements the calculation algorithm independently
3. Swappable Strategy:
   - In `DietProfile.cpp`, our profile stores the current calculator:
4. UI for Calculator Selection:
   - `DietManagerApp::changeCalculator()` allows users to switch between calculation methods

Size of the Log File

By exclusively using `std::shared_ptr` to manage objects, we avoid unnecessary deep copies. This approach ensures that only references are shared across components like `DietManagerApp`, `UndoManager`, and `DailyLog`, significantly reducing memory overhead and saving log file space. Instead of duplicating full object data repeatedly, we simply manage ownership and references smartly—keeping the log clean and efficient.

Reflection

Strongest Aspects of Our Design

1. **Composite Pattern for Food Representation**: The design uses the Composite pattern effectively for food modeling. This allows both simple (BasicFood) and complex (CompositeFood) foods to be treated uniformly. This is elegant and extensible, making it easy to create recipes and handle food hierarchies.
2. **Strategy Pattern for Calorie Calculation**: The `TargetCalorieCalculator` interface with concrete implementations (Harris-Benedict and Mifflin-St Jeor) demonstrates excellent use of the Strategy pattern. Adding new calculation methods requires minimal changes, fulfilling the requirement for easy extension.

Weakest Aspects of Our Design

1. **Singleton Usage for FoodDatabase**: While the Singleton pattern ensures a single instance of the food database, it introduces global state and could make testing more difficult. A dependency injection approach might have been more flexible and testable.
2. **Limited Abstraction for Data Persistence**: The persistence mechanism is somewhat hardcoded with direct file I/O in multiple classes. A more abstract persistence interface would have made it easier to swap out storage mechanisms (e.g., switching from file-based to database storage).