DASS Assignment-3
YADA Design Document
Date: 06/04/2025
Team members:

Name	Roll Number
Eshwar Sriramoju	2023101111
Likhit Bhogadi	2023101065

Overview:
This is a command-line Java application for tracking daily calorie intake, managing food databases, and monitoring nutritional goals.
Features:

1. User profile management (height, age, weight, activity level)
2. Dynamic daily calorie target calculation using multiple methods
3. Food database with basic and composite foods
4. Daily food log with timestamp tracking
5. Search functionality for foods
6. Command history with unlimited undo functionality
7. Date navigation for historical data viewing
8. Persistent storage of food database and logs

Class Diagram:

Classes:	**Interfaces:**
Food-related: Food BasicFood CompositeFood FoodEntry **CLI:** CalorieTrackerCLI **Command:** CommandInvoker AddFoodEntryCommand RemoveFoodEntryCommand **Controller:** CalorieTrackerController **DB:** JsonFoodDatabase LogManager ProfileManager **Tracking-related:** DailyLog CalorieTrackerApp **User-related:** User UserProfile **Service:** ExampleWebFoodSource FoodSourceManager HarrisBenedictCalculator LocalFoodSource MifflinStJeorCalculator	Command FoodDatabase CalorieCalculator FoodSource

Link to UML Diagram: https://tinyurl.com/2s2wwpfx

Links to Sequence Diagrams:

1. User creates composite foods from basic foods: https://tinyurl.com/yc5wxbex
2. User adds new basic food: https://tinyurl.com/48bk2tee
3. User searches for food and views information: https://tinyurl.com/5t5xysy9
4. User determines total calories consumed: https://tinyurl.com/mv3v6tbh

Design Analysis:
Balance of Design Principles in the Architecture:

Low Coupling and High Cohesion
Low Coupling
The application achieves low coupling through clearly defined interfaces and a layered architecture:

1. Interface-based Design: The system uses interfaces like FoodDatabase, CalorieCalculator, FoodSource, and Command to define contracts between components, allowing implementations to vary independently.

2. Dependency Injection: Components receive their dependencies through constructors rather than creating them internally:

```
public LogManager(String logFilePath, FoodDatabase foodDatabase) {
        this.logFilePath = logFilePath;
        this.foodDatabase = foodDatabase;
        …
}
```

3. Controller as Mediator: The CalorieTrackerController centralizes interaction between the UI and the model/service layers, preventing direct dependencies between them.

High Cohesion
Each class has a well-defined responsibility and related functionality:

1. Model Classes: Each model class (Food, BasicFood, CompositeFood, etc.) has a focused purpose representing a specific domain concept.

2. Manager Classes: LogManager handles log-related operations, ProfileManager manages user profiles, etc.

3. Service Classes: Calculators handle only calorie calculations, and food sources are solely responsible for providing food data.

Separation of Concerns
The application divides functionality into distinct layers:

1. Model Layer: Contains domain entities (Food, User, DailyLog)
2. Service Layer: Provides domain-specific services (CalorieCalculator, FoodSource)
3. Data Layer: Manages persistence (FoodDatabase, LogManager, ProfileManager)
4. Command Layer: Encapsulates user actions and enables undo functionality
5. Controller Layer: Coordinates between UI and business logic
6. UI Layer: Handles user interaction via the CLI

This layering ensures that changes to one concern (e.g., UI or persistence) don't impact other areas of the application.

Information Hiding
The application employs information hiding through encapsulation and abstraction:

1. Private Fields with Accessors: Most classes encapsulate their state with private fields and provide controlled access through public methods.

2. Abstract Base Classes: Food serves as an abstract base class that hides implementation details of its concrete subclasses (BasicFood and CompositeFood).

3. Interface Abstractions: Interfaces like FoodDatabase hide the implementation details of concrete classes like JsonFoodDatabase.

4. Immutable Views: Many methods return defensive copies of collections rather than direct references:

```
public List<FoodEntry> getEntries() {
        return new ArrayList<>(entries);
}
```

Law of Demeter
The application generally follows the Law of Demeter by limiting object communication to immediate neighbors:

1. Controller as Intermediary: The UI interacts with the model through the controller rather than directly.

2. Command Pattern: Commands encapsulate actions, hiding the details of how they interact with the system.

3. Method Chaining Limitation: The code generally avoids chains of method calls that navigate through multiple objects.

Design Strengths
Design Patterns
The application effectively uses several design patterns:

1. Command Pattern: Actions are encapsulated in command objects, enabling undo functionality.
2. Composite Pattern: CompositeFood can contain other food objects, creating a hierarchy.
3. Strategy Pattern: Different calorie calculation algorithms are interchangeable.
4. Factory Methods: Used to create different types of objects.
5. Flyweight Pattern: Food objects are shared to reduce memory usage.

Extensibility
The design facilitates extension without modification:

1. New Food Sources: Adding a new food data source requires implementing the FoodSource interface.
2. New Calculation Methods: Adding a new calorie calculation algorithm requires implementing CalorieCalculator.
3. New Commands: Additional user actions can be added by implementing the Command interface.

Design Weaknesses
1. User Class Immutability: The User class requires creating a new instance to change some properties (height, age) but not others (weight, activity level), leading to inconsistent update mechanisms.

2. Service Locator: The controller acts somewhat as a service locator, which can hide dependencies.

3. Command Structure: The commands have knowledge of both the model and data layer, potentially creating tighter coupling than necessary.