SWEN 383 - Software Design Principles and Practices

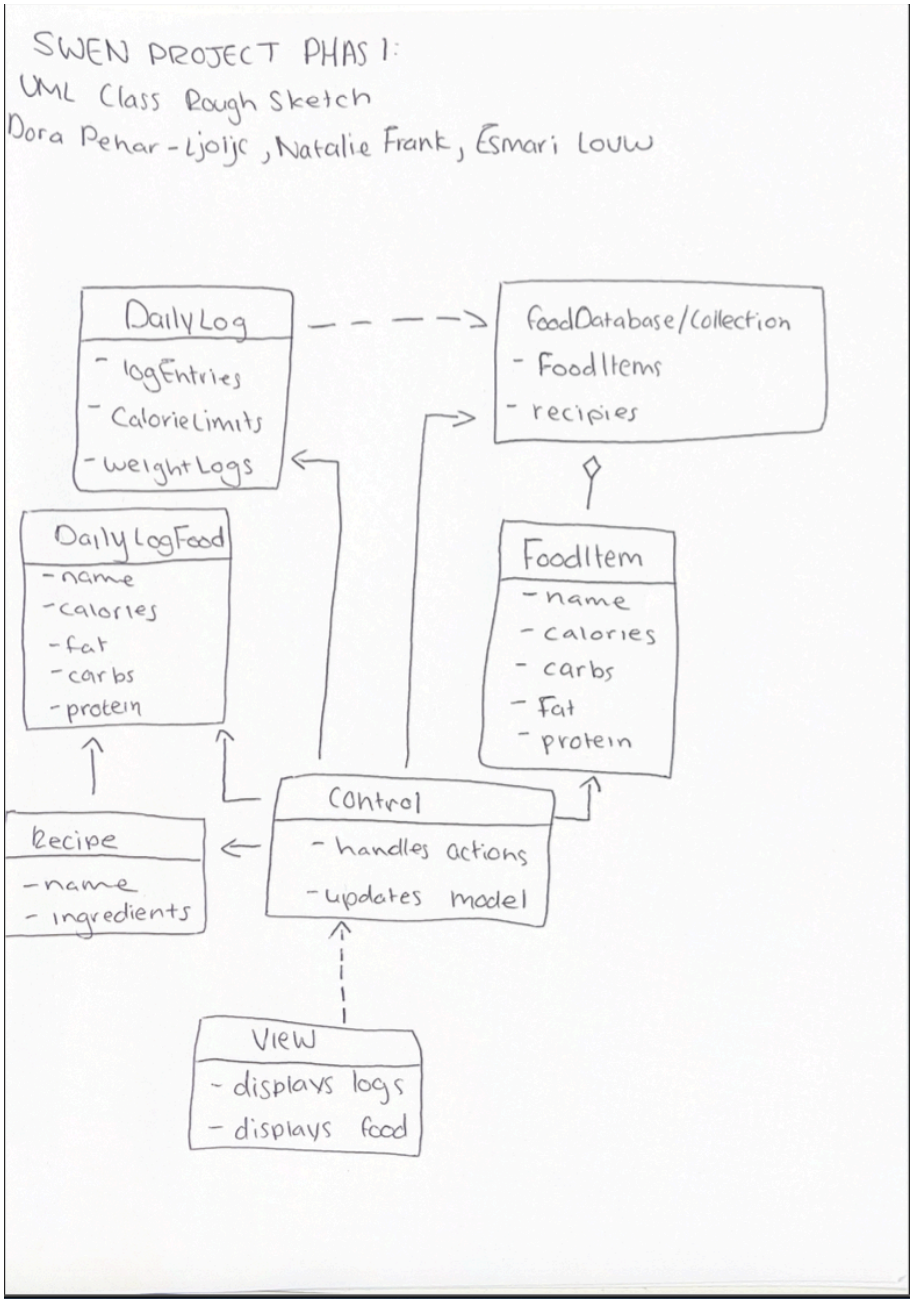
DESIGN SKETCH PHASES I & II

Version 2.0

Date: 30/04/2025
Team Identification: Group 4
Team Members:

- Esmari Louw
- Dora Pehar-Ljoljic
- Natalie Frank

Rough Design Class Sketch:



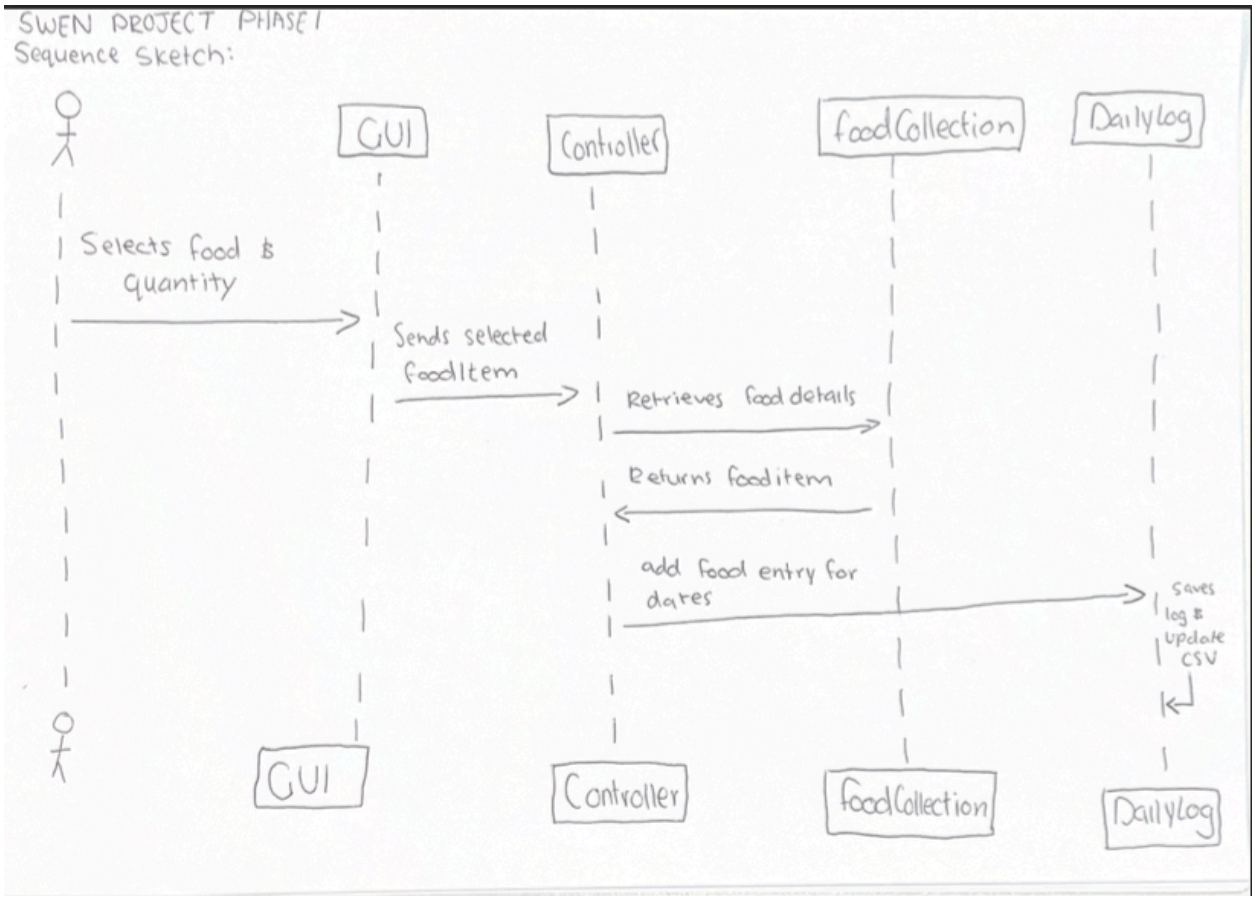
Key Classes:

- **DailyLog :** Handles food logging for each day, manages the log entries, calorie limits and the weight records
- **DailyLogFood :** Represents the individual food item with nutrition details
- **Recipe:** Special type of food composed of multiple ingredients/components
- **FoodCollection:** Manages the storage and retrieval of available food items.
- **Controller:** handles the applications logic and interaction between UI and data
- **Views:** Represents the graphical user interface to display logs and user inputs.

Design Patterns Used:

- **Adapter pattern** : to integrate FoodItem and Recipe under a unified DailyLogFood interface
- **Composite Pattern:** used in FoodCollection to allow treating individual FoodItems and Recipes uniformly
- **MVC Pattern:** separates concerns between Model (DailyLog & FoodCollection), View (GUI) and Controller (Control class).
- **Factory pattern:** used in GUI for creating UI components dynamically.

Rough Sequence Diagram:



Reading the Food Data:

- The system reads a food collection FoodItem and Recipe
- Recipes can contain other food items or recipes.
- FoodCollection loads and Stores the data
    - Objectives involved = Control -> FoodCollection -> DailyLogFood / Recipe

Adding Food to Daily Log:

- The user selects a food item from collection
- The selected item is added to the DailyLog for the current date.
- The system updates and saves the log.
    - Objects involved = View -> Control -> DailyLog -> DailyLogFood

Computing total Calories:

- User requests total calories computation for given date
- The system retrieves logged food entries and calculates the total calories
- If calorie limit is set, the system compares it to the logged intake
    - Objects involved = View -> Control -> DailyLog -> DailyLogFood

Class Descriptions:

- **DailyLog:** Stores food entries per date, tracks calorie limits and weight.
- **DailyLogFood:** Represents a single food item with calories, fat, protein and carbs.
- **Recipe**: Inherits from DailyLogFood, containing multiple DailyLogFood entries
- **FoodCollection:** Manages food items and recipes, allowing access to stored data.
- **Controller:** Manages logic and connects UI with the data model.
- **View:** Displays information and interacts with the user.

Our Design Rationale:

Advantages:

- Modular, keeping flexibility for future (phase II) modifications.
- The Adapter Pattern unifies the recipes and food items.
- The MVC pattern makes it easy to modify UI without affecting core logic in the program.
- The Composite Pattern allows uniform treatment of individual foods and recipes.
- The factory Pattern allows for scalable GUI component creation.

Disadvantages:

- Initial complexity of handling the recipes as nested structures.
- Large project = more complex structures of files.
- MVC introduced more components to manage, requiring proper coordination.
- Requires efficient handling of the I/O operations.

Next Steps:

- Refine Class Diagram (will be uploaded into folder on GIT-Repository)
- Improve Sequence diagram (will be uploaded into folder on GIT-Repository)
- Skeleton Code = uploaded through pushes in our git history
- GUI integration

Overview for Version 2

Diet Manager 2.0 is an expansion of the current Version 1.0 system for tracking activity and dietary consumption. It lets users register food intake, track calories, track activity, establish and track calorie goals, and see how macronutrients are distributed in graphs. Basic food collections, workouts, recipes (including subrecipes), and thorough daily diaries may all be accessed by users.

Changes Implemented & Rationale Behind Them

Due to some design constraints we had to change some of the design structure of our code. All the changes were done so any repetitions, redundancies and errors don't occur.

- **Removed FoodItemAdapter.java Class**

  We got rid of the explicit FoodItemAdapter.java class because we thought it would simplify our code, and remove redundancies. The adapter is more used in the DailyLog.java since the Recipe.java class contains a map of Food.java objects, and a custom adapter could be created to allow the Recipe.java to be handled as a single Food.java item in the application.

- **Removed the ViewFactory.java**

  We also got rid of this class since it wasn't really serving any major purpose but to initialize different views, and since we didn't need to initialize one window at a time, but the whole application, we decided against keeping the file. The Factory Pattern is still in the code, but not explicitly done (MainView.java creates the views for FoodView.java and LogView.java).

- **Added the Observer Pattern**

  The pattern is used to allow an object (like the MainView.java) to listen for changes in another object (like LogView.java or MainView.java) and update itself accordingly. For instance, when the selected date in the JSpinner (from MainView.java) changes, it triggers the Controller.java to handle that event.

Design Patterns Used

- **MVC Pattern -** separates concerns between model (Daily Log & Food Collection), View (all GUI classes - FoodView, LogView, MainView), and Controller

- **Composite Pattern -** used in FoodCollection.java to allow treating individual FoodItems and Recipes uniformly

- **Adapter Pattern -** DailyLog stores both FoodItem.java and Recipe.java in the same Map<Date, List<Food>> because the log doesn't care about the concrete type but only that they implement Food

- **Observer Pattern -** the Controller manually notifies views (for example, mainView.updateNutritionGraph()), and the views update dynamically when the Model changes

- **Factory Pattern -** implicit use of a factory when new instances of components like FoodView.java, LogView.java, and SimpleNutritionGraph.java are created in MainView.java, since there is no abstract interface it is implicit

- **Facade Pattern -** MainView.java acts as a facade by providing a simplified interface to manage and coordinate multiple complex subsystems as it wraps and initializes multiple components: FoodView.java, LogView.java, SimpleNutritionGraph.java, and connects them together and hides their individual components from the outside world

- **Singleton Pattern -** implicit use for objects like Model and Controller as they are created once at the start of the application and used throughout. But since we don't have any private constructors and static methods, it is used implicitly.

Class Descriptions

- **Core MVC Classes**
  - Model - Central data holder
    - Manages all application data (foods, exercises, logs)
    - Contains subcomponents: FoodCollection, DailyLog, ExerciseManager
    - Tracks current date
  - View (Interface) - UI Contract
    - Implemented by all view classes
    - Defines basic display and controller binding requirements
  - Controller - Mediator
    - Coordinates between Model and Views
    - Handles date changes and data updates
    - Triggers view refreshes

- **Food Hierarchy (Composite Pattern)**
  - Food (Abstract) - Component
    - Base class for all food types
    - Defines common interface (getNutrition(), displayInfo())
  - FoodItem - Leaf
    - Represents basic foods
    - Stores concrete nutrition values
  - Recipe - Composite
    - Contains other Foods (including sub-recipes)
    - Calculates aggregate nutrition values

- **Data Management Classes**
  - **FoodCollection** - Food Database
    - Manages both basic foods and recipes
    - Handles CSV file I/O
    - Enforces no-forward-references rule
  - **DailyLog** - Daily Tracking
    - Records food consumption, exercises, weight
    - Calculates daily totals
    - Manages log.csv persistence
  - **ExerciseManager** - Exercise Database

        - Maintains exercise catalog
        - Handles exercise.csv persistence
        - Calculates calorie expenditure

- **View Components**
    - **FoodView** - Food Management UI
        - Lists all foods/recipes
        - Provides CRUD operations
        - Shows detailed nutrition info
    - **LogView** - Daily Tracking UI
        - Displays daily consumption
        - Shows nutrition breakdown
        - Manages exercise logging
    - **SimpleNutritionGraph** - Visualization
        - Renders macronutrient bar chart
        - Updates dynamically with data changes

- **Supporting Classes**
    - **Exercise** - Exercise Data
        - Stores name, calories/hour, duration
        - Handles calorie calculations
    - **FoodDetailsDialog** - Popup Detail View
        - Shows comprehensive nutrition info
        - Different displays for basic vs recipe foods

Our Design Rationale

Our design decisions were driven by the need to simplify the codebase, reduce redundancy, and make the application easier to extend in future phases. Here's a summary of the rationale behind key decisions:

- **Simplification of Architecture:** We removed classes like FoodItemAdapter.java and ViewFactory.java because they introduced unnecessary complexity without providing significant value. Their functionality was either redundant or could be implemented more efficiently elsewhere.

- **Improved Flexibility Using Design Patterns:** By implementing established design patterns (such as MVC, Composite, Observer, and Adapter), we ensured that our application remains organized and scalable. Each component has a clear responsibility, which reduces coupling and makes it easier to update or replace features independently.

- **Unified Data Handling:** The use of the Composite and Adapter patterns allows both FoodItem and Recipe to be handled uniformly within the system. This simplifies how we log and calculate nutrition data, and supports future extensions like sub-recipes or meal planning.

- **Dynamic User Interface:** The Observer pattern allows views to respond to model changes in real-time (e.g., date selection updates all related views automatically), which enhances the user experience and responsiveness of the application.

- **Centralized Usage Through Facade:** MainView.java acts as a central coordinator for UI components, reducing the complexity of cross-class communication and encapsulating setup logic for easier readability and maintenance.

What Our Program Does Now

Our NutritionManager application currently saves all the food (including the basic foods and recipes), calculates the calories and weight, users can now add food entries to the daily log, as well as exercises. Users can also remove food, recipes, data from logs and exercises. We still need to work on fixing the graphing of the data.