SWEN 383 - Software Design Principles and Patterns

Wellness Manager - Version 1.0

Overview

** Disclaimer: This academic exercise and any views expressed or interpreted within its content are in no way intended to suggest, recommend or require life-style or behavior changes, or otherwise offend the reader, users or implementers of the system.*

A Wellness Consortium has received funding to contract the development of software that will enable people to monitor and improve their daily lives.

The program you will create will provide a collection of basic foods, which can be combined to create recipes in the system. Users can also add their own basic foods or recipes to the program by providing the necessary information. For example, this information could be the name of the food, calories, grams of fat, grams of carbohydrates, and grams of protein in the food. Users can also use recipes as an ingredient in other recipes they add to the system (sub-recipes).

Each day, the user can add a log of the food they consumed during the day, including the number of servings of each food. Based on this information, the program can display totals of nutrients for each day. It could also display graphs showing the distribution of things like fat, carbs, and protein for the day. For example, if a user only logged a piece of bread for a day and bread had 5 grams of carbs and 5 grams of protein, the graph for that day would show 0% fat, 50% carbohydrates, and 50% protein.

Users can also attach other information to their log entries, like their weight or their desired intake of calories. The program should indicate to the user the status of their calorie goal (whether or not their calorie intake for the day meets their desired intake). This helps users to track their weight over time and understand whether they are under or over their goal.

The program must also save the user's data. This includes the collection of foods and recipes as well as the information on daily consumption, calorie targets, and recorded weights.

Requirements

Food Collection

1. The program shall store a food file named **exactly** ***foods.csv*** with a collection of the foods in the system in **CSV** (**C**omma **S**eparated **V**alues) format.
2. Each line in the file shall contain information on a single basic food or a single recipe.
3. For a basic food, the CSV line shall be formatted as follows:

b,*name*,*calories*,*fat*,*carb*,*protein*

where **b** is the literal letter *b*, *name* is the food name, *calories* is the number of calories in one serving of the food, and *fat*, *carb*, and *protein* are the number of grams of fat, carbohydrates, and protein in one serving of the food. The name may not contain a comma and the numeric fields must all be floating point values.

Example – Hot dog:

```
b,Hot Dog,147.0,13.6,1.1,5.1
```

Example – Hot dog bun:

```
b,Hot Dog Bun,120.0,1.9,21.3,4.1
```

4. For a recipe, the CSV line shall be formatted as follows:

   r,*name*,*f1name*,*f1count*,*f2name*, *f2count*,...,*fNname*,*fNcount*

   where **r** is the literal letter *r*, *name* is the recipe name, and the list of *name*/*count* pairs gives the names of other foods (either basic or sub-recipes) and the number of servings of the food in the recipe being defined. Names may not contain commas, and servings must be given as a floating point number.

   NOTE: Food names in the list must refer to basic foods and recipes previously defined in the file; no forward references are allowed.

   Example – Hot dog in bun with mustard:

   ```
   r,Hot Dog-Bun-Mustard,Hot Dog,1.0,Hot Dog Bun,1.0,Mustard,1.5
   ```

5. Food names, whether basic or recipe, shall be unique across all foods.
6. The file shall be loaded into an appropriate internal data structure when the program executes, and saved to the CSV on exit or when the user explicitly requests the information be saved.
7. The saved file shall not contain any duplicate information (each basic food or recipe occurs exactly once).
8. The saved file shall conform to the *no forward reference* constraint.

Daily Log

1. The program shall maintain a CSV format log file named **exactly** ***log.csv*** of the food intake, weight, and desired calorie limit on a daily basis.

2. Weight is recorded on a line with the following format:

   yyyy,*mm*,*dd*,**w**,*weight*

   where *yyyy*, *mm*, *dd* specifies the year, month, and day as integers; **w** is the literal letter *w*, and *weight* is the recorded weight in pounds as a floating point number.

   Example:

```
2021,09,30,w,185.5
```

3. If the user has entered a weight for the given date, that value should be used in the system. If the user has not input a weight for the given date, the weight that was entered on the most recent weight entry should be used. If a weight was never entered by the user, the system should default to 150.0 pounds.

4. The calorie limit is recorded on a line with the following format:

   *yyyy,mm,dd,***c**,*calories*

   where *yyyy*, *mm*, *dd* specifies the year, month, and day as integers; **c** is the literal letter *c*, and *calories* is the calorie limit as a floating point number.

   Example:

   ```
   2021,09,30,c,1800.0
   ```

5. If the user entered a calorie limit for a given date, that value should be used in the system. If the user did not enter a calorie limit for the given date, the limit that was entered on the most recent calorie limit entry should be used. If a calorie limit was never entered by the user, the system default should be 2000.0 calories.

6. Each food item consumed on a given day is recorded on a line in the following format:

   *yyyy,mm,dd,***f**,*name,count*

   where *yyyy*, *mm*, *dd* specifies the year, month, and day as integers; **f** is the literal letter *f*, *name* is the food name (either a basic food or recipe), and *count* is the number of servings consumed:

   Example:

   ```
   2021,09,30,f,Hot Dog-Bun-Mustard,2.5
   ```

   NOTE: Different entries for the same food on a given day may not be combined. For example, if the user records two cups of coffee in the morning and one in the afternoon, then there should be two distinct log entries.

7. The log file shall be loaded into an appropriate internal data structure when the program executes, and be saved to the CSV on exit or when the user explicitly requests the information be saved.

Program Operation

1. The program shall employ a simple user interface that is sufficient to achieve the functionality in this document.
2. When started, the program shall load the food and log data into their internal data structures.
3. The program shall allow the user to select a specific date to log activities for (the default on start-up is the current date).

4. If the selected date does not already exist in the log, the information shall be initialized as follows:

   - The food intake is empty.

    - The calorie limit and weight are determined using the rules from the **Daily Log** section above.

5. The program shall allow the user to change the daily calorie limit and weight for the selected date.

6. The program shall support adding of new basic foods and recipes in a manner consistent with the specifications under **Food Collection**.

7. The program shall, by some means, provide the user access to the following information for the selected date:

    - Each food item consumed, along with the number of servings and total calories.
    - The total number of calories consumed for the day and some indication of whether this exceeds the set limit.
    - The weight for the day.
    - A breakdown of nutrition in terms of the percentage of total grams of fats, carbohydrates, and protein, each to the nearest 1%. The total must be 100%.

8. The program shall allow the user to add a food item and the number of servings to the daily log on the currently selected date.
9. The program shall allow the user to delete food items in the daily log. The user shall be able to unambiguously specify which food to delete even if several entries have the same food name.
10. All data about the selected date shall immediately reflect the results of additions, changes, and deletions to foods, weight, or calorie limit.
11. The program shall provide a means for the user to save the current system state (the food collection and the daily log). In particular, the saved food collection must be in a form consistent with the no forward references constraint in the **Food Database** section.

Design Constraints

1. The design **must incorporate the Composite pattern** in an appropriate and visible manner.
2. The design **must incorporate the MVC pattern** with an appropriate specification of which classes belong to the **M**odel, the **V**iew, and the **C**ontroller subsystems.

3. Use of other patterns in an appropriate way (in particular, for reasons other than impressing your instructor with all the patterns you can cram in) will lead to higher design grades.

Deliverables (follow this link)

Suggested Design Approach

Design is inherently a creative activity; as such, it is impossible to give a formula or algorithm that ensures success. However, there are some rules of thumb that typically result in an acceptable, even superb, object-oriented design.

1. Read the project description carefully, and extract all the ***nouns*** and ***noun phrases***. These form the basis for initial selection of the problem classes (be sure to detect synonyms and only use one of the names). In some cases you'll find that a noun actually refers to a simple attribute of a class.
2. ***Adjectives*** and ***adverbs*** often point to information that defines the state of the objects in a class. As you identify such state information, assign it to what appears to be the most appropriate class.

Don't worry if an attribute seems equally appropriate to two or more classes; just choose one at this point.

3. ***Verbs*** are signs of the methods that must be implemented. For each verb that seems appropriate, create an associated method name and associate it with a candidate class. As with state information, don't obsess at this point over whether or not you've identified the optimal class.
4. ***Sketch an initial class diagram*** (pencil and paper are fine), showing interconnections among classes that are drawn from the implications of the problem statement. This is a good point at which to try to identify an initial set of patterns appropriate to your system.

5. Now comes the hard but rewarding part - ***iterate*** until an adequate design is achieved.

    - For each of the user operations the system must support, trace a flow through the system objects to be sure the operation can be done. Sequence charts can help here, but don't get hung up on formality - paper and pencil are easier to work with (and easier to discard if things don't go well).
    - For each of the inter-class connections, ask whether the coupling could be reduced. In the extreme case, two classes could become totally uncoupled, but this is unusual. Instead, look for ways to reduce the amount of information about other classes that is implicit in the coupling. For instance, if you moved some responsibilities from one class to another is the coupling reduced?
    - Try to improve each class's cohesion, and make sure each class carries its own weight. A class that is just data members with getters and setters is rarely optimal. If you can't think of any "real" responsibilities to assign to the class, perhaps you don't need it.
    - Look for classes that address several concerns and try to separate these (either by assigning responsibilities to existing classes or inventing new ones).
    - Roughly estimate the size and complexity of each class. Strive for balance, where the responsibilities defined in the problem statement are dispersed in roughly equal portions to the classes in your design.
    - The goal is "balance", which can't be formally described. ***YKIWYSI*** - You'll know it when you see it.

Item	Percentage
Design Sketch	10%
Preliminary Design	20%
Skeleton Java Implementation **(focus on primary M,V,C classes)**	10%
Version 1.0 Team Presentation	10%
Version 1.0 Design	30%
Version 1.0 Java Implementation **(requires a CLI "front-end" only!)**	20%