Zhang Xiaoyu - Project Portfolio

PROJECT: SugarMummy

Overview

SugarMummy is a desktop application used to help type-II diabetics develop healthier lifestyle. The user interacts with it using a Command Line Interface, and its GUI is created with JavaFx. It is written in Java and has about 24 kLOC.

Summary of contributions

- **Major Feature**: added **recmf command**
  - What it does: Recommends a list of medically suggested foods to type II diabetics.
  - Justification: This feature is crucial to diabetic patients since their health states are closely related to food consumption.
  - Highlights: This feature benefits the extension of more accurate and personalized food recommendations since the food model with nutrition values that can be calculated and analyzed. Besides, it is also useful for future commands that relate to diet records and analysis.
- **Minor enhancement**:
  - added a recmfmix command as a concise version of recmf command
  - added addfood command that allows the user to expand the database
  - added resetf command that allows the user to clear newly added foods
- **Code contributed**: [View RepoSense]
- **Other contributions**:
  - Project management:
    - Managed releases `v1.3` - `v1.5rc` (3 releases) on GitHub
  - Enhancements to existing features:
    - Constructed a generic Storage class for storage reading and writing: #65
  - Documentation:
    - Designed the background for the application. (Pull requests #22)
    - Updated application logo. (#105)
    - Updated `Readme` and application screenshots. (Pull requests #22)
  - Community:
    - PRs reviewed (with non-trivial review comments):
    - Contributed to forum discussions

        - Reported bugs and suggestions for other teams in the class
        - Some parts of the history feature I added was adopted by several other class mates
    - Tools:

Contributions to the User Guide

Food Recommendation

General Note

1. If a command requires no user-input arguments, all the additional inputs after this command string will be ignored and the command will be executed as usual.
2. If a command requires any parameters, the input order is flexible. Duplicate parameters are also allowed, **but only the last occurrence will be considered**.

Recommending food: `recmf`

Recommends medically suggested foods for type II diabetes patients. The user can specify `Flags` and `Food Names` as two kinds of filters, as well as one type of sorting order:

- **Flags:** specifies the wanted food type in the form of following flags:

-nsv: non-starchy vegetable, such as *Broccoli*	**-sv**: starchy vegetable, such as *Potato*
-f: fruit, such as *Cherry*	**-p**: protein, such as *Lean Lamb*
-s: snack, such *Fig Roll*	**-m**: meal, such as *Spanish Omelet*

Note:
1. Flags are case-insensitive and duplicates are allowed, but they will be recognized only when placed before any prefix.
2. If no flag is specified, it is equivalent to specifying all flags. Namely, foods of all types will be shown.

- **Food Names:**: matches foods that contain **one of** given food names in the form of `fn/[FOOD_NAME]⋯`

Note:
1. Food names are case-insensitive while matching is full-word matching. For example, "chicken" does not match "ch".
2. If no food name specified after `fn/`, it is equivalent to matching any food.

- **Sorting Order**: determines the presentation order of food cards in **one** of the two forms: `+sort/SORT_ORDER_TYPE` and `-sort/SORT_ORDER_TYPE`
    - `+` indicates in ascending order and `-` descending

- SORT_ORDER_TYPE is required and can be one of the following six:
  fn: food name; ft: food type; ca: calorie; gi: glycemic index; su: sugar; fa: fat

> **NOTE:**
> 1. +sort/SOT and -sort/SOT cannot be both present even though they may have different SOT(SORT_ORDER_TYPE).
> 2. Specially, for ft, the ascending order is predefined as: nsv, sv, f, p, s, m.

Format: `recmf [-FLAG]… [fn/FOOD_NAME…][±sort/SORT_ORDER_TYPE]`

Examples: {recmf -p -f} `{recmf fn/chicken rice}` {recmf -p -m -f fn/chicken} `{recmf -p -nsv +sort/gi}`

Recommending food combination: `recmfmix`

Recommends one food from each type. A summary card of all nutrition values will be appended at the end.

> **Note:**
> 1. All extra inputs after this command string will be ignored.
> 2. Food types with no corresponding food data will not be shown. If there is no food data at all, the summary card will not be shown as well.
> 3. If any of the summary data is decimals, it will be formatted into two decimal places.
> 4. Please note the GI (glycemic index) value is the average instead of sum. For more information about GI, please refer to this link.

Format: `recmfmix`

Example: `recmfmix`

Adding new food items : `addfood`

Adds a new food item for future recommendations. The following six fields are required:

- food name: `fn/FOOD_NAME`
  Food name should only contain alphabets, numbers, and whitespace. It should be less than 30 characters for display quality and readability.
- food type: `ft/FOOD_TYPE`
  Food types should be exactly one of the following: nsv(non-starchy vegetable), sv(starchy vegetable), f(fruit), p(protein), s(snack), or m(meal).
- calorie (cal): `ca/CALORIE`
  Calorie should be less than 700(cal) per serving.
- gi: `gi/GI`
  Glycemic Index should be less than 70 per serving.
- sugar (g): `su/SUGAR`
  Sugar should be less than 25(g) per serving.
- fat (g): `fa/FAT`

Fat should be less than 35(g) per serving.

> **Note:**
> 1. No duplicate food names are allowed.
> 2. All nutrition values should be non-negative numbers and contain no more than four decimals. The decimal point must be between numbers. For example, ".5" and "1." will not be accepted.
> 3. Ideally, the input values are normalized as per serving for more practical value comparisons and calculations.

Format: `addfood fn/FOOD_NAME ft/FOOD_TYPE ca/CALORIE gi/GI su/SUGAR fa/FAT`

Example: `addfood fn/Cucumber ft/nsv ca/15 gi/15 su/1.7 fa/0`

Deleting an existing food: `deletef`

Deletes a food that matches the specified food name(s).

> **Note:** FOOD_NAME matching is case-insensitive, but is strict matching for every single character, including white space. It is also full matching, such as "Rice with Chicken" does not match with "Chicken".

Format: `deletef fn/FOOD_NAME`

Example: `deletef fn/Mushroom`

Resetting food data: `resetf`

Clears all modifications, adding and deleting, on the food list. The food data will be reset to sample food data.

Format: `resetf`

Example: `resetf`

Contributions to the Developer Guide

Food Recommendation Feature

The food recommendation mechanism is based on the manipulation of `UniqueFoodList`, with the implementation of operations:

- **Showing food recommendations as cards** filtered by `Flags` and / or `FoodNames`.
- **Sorting the food list** based on comparing food fields specified in `SortOrderType`.
- **Showing combined recommendations** from each food type with an additional *summary*.
- **Adding foods** and **Deleting foods**
- **Resetting food database** which clears modifications from the user on food list

These operations are respectively exposed in the `Model` interface as `updateFilteredFoodList`, `sortFoodList`, `getMixedFoodList`, `addFood`, `deleteFood`, `setFoods`.

Overview of Data Structures

The main data structures used to support food recommendation are listed as follows.

1. Food Model

It encapsulates `FoodName`, `FoodType`, and four `NutritionValues`. These fields are visualized in `FoodCards`, which collectively compose the `FoodFlowPane`. The six fields are `Comparable` to support `sortFoodList` function. Additionally, `NutritionValues` are used by `FoodCalculator` to obtain summary statistics.

API: `Food.java`

2. UniqueFoodList

It holds the collection of foods, and it exposes necessary methods in `ModelManager`. Internally, it holds an `ObservableList` available for modifications, such as adding foods. It also implements `getMixedFoodList` method for `recmfmix` command via randomly selecting foods from its `internalUnmodifiableList`.

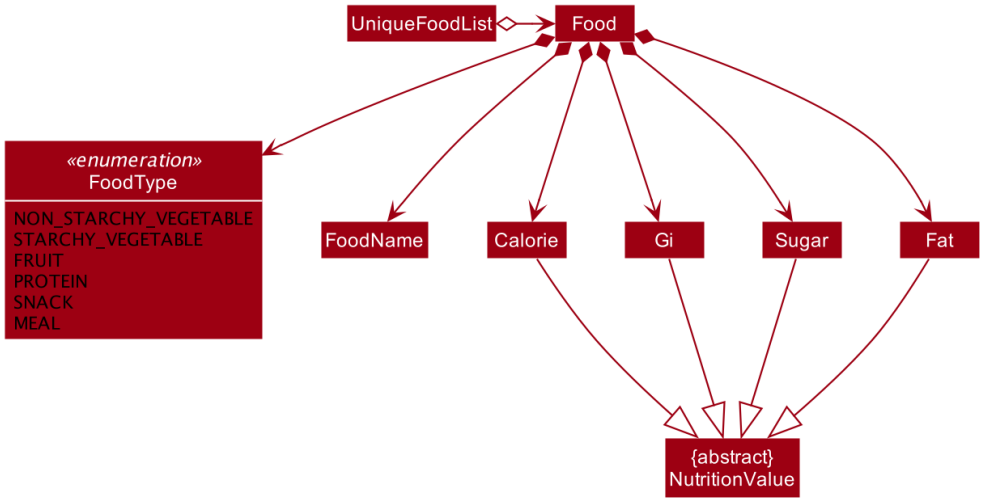
API: `UniqueFoodList.java`

3. Predicates

Both implement `Predicate` and hold desired conditions as collections, such as `List` and `Set`. They literate through the whole list to select foods that matches any of given conditions. However, if the conditions are empty, the `test` result is always set to be true.

API: `FoodNameContainsKeywordsPredicate.java` `FoodTypeIsWantedPredicate.java`

The following class diagram summaries these main components interact with each other.



Implementation of *recmf* and *recmfmix* command

recmf command

`RecmFoodCommandParser` parses user inputs to standard parameters for the customised presentation of food recommendations, detailing in the following three ways:

1. Specifying flags

Flags are similar to the usage of flags / options in Unix commands. In SugarMummy, they are used to specify food types that are intended to be shown. The existence of a certain flag depends on available `FoodTypes`. Flags will be eventually translated to FoodTypeIsWantedPredicate and applied on `UniqueFoodList`.

Data Structure

A `HashSet` is used to hold specified `FoodTypes` that are translated from user-input flags.

NOTE	If no flag is specified, `RecmFoodCommandParser#getWantedFoodTypes(flagsStr)` will return an empty `HashSet`.

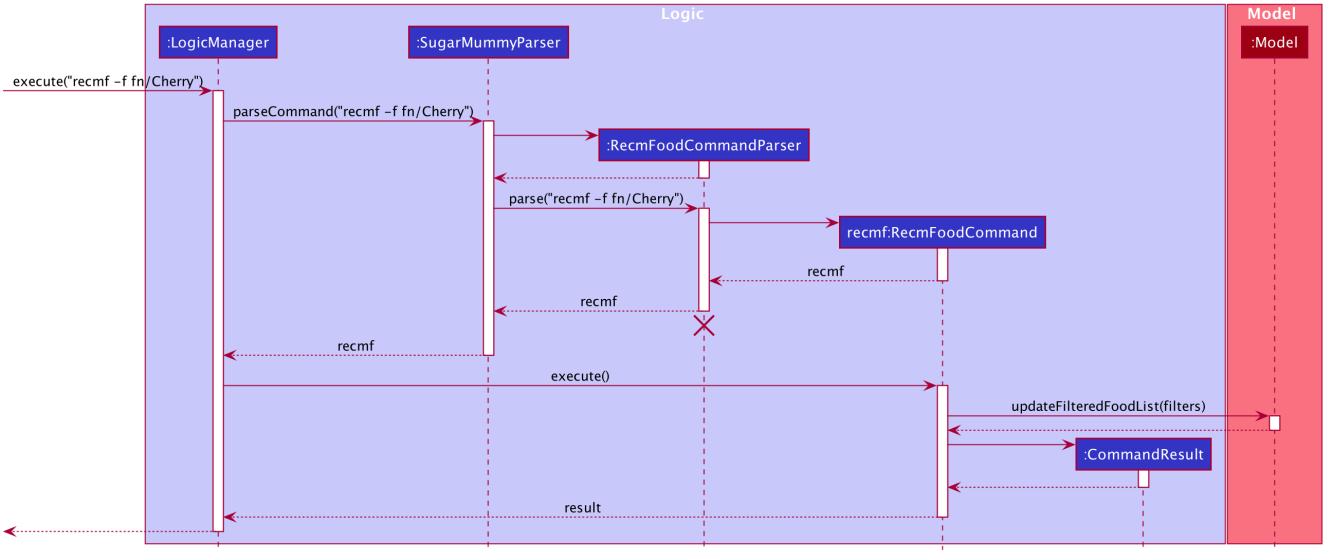
API: Flag.java

2. Filtering food names

It is similar to and simpler than the implementation of specifying flags. A `List` of strings as food names will be supplied to FoodNameContainsKeywordsPredicate.

Diagram

The following sequence diagrams shows the how recmf command with food name as the filter works.



3. Sorting

It is implemented via supplying a `FoodComparator` to `UniqueFoodList#sortFoods`. Afterwards, to maintain or change the ordering, `model#sortFoodList()` method is added into `execute` method of relevant commands.

Data Structure

- A `Comparator` is wrapped by `FoodComparator` to handle the main logic, such as reversing the `FoodComparator` via `Comparator#reversed()`.
- An inner enum class `SortOrderType` holds all the comparable food fields for sorting.
  - The default sort order is currently set to be by food type, which can be modified via `FoodComparator#DEFAULT_SORT_ORDER_STRING`.

NOTE	A private `FoodComparator` constructor directly taking in `Comparator` is for internal usage of getting reversed `FoodComparator`. Outside instantiation is done by supplying `SortOrderType` strings.

API: `FoodComparator.java`

Diagram

The following object diagram summaries the components in the food recommendation mechanism.

recmfmix command

`UniqueFoodList#getMixedFoodList()` generates a temporary `ObservableList` from the existing food data. This list will eventually be supplied to `FoodFlowPane` via `Model` and then `Logic`.

Food Summary Card: It is essentially treated as `Food` with *Summary* as food name and *meal* as food type. The total / average nutrition values are calculated by `FoodCalculator`.

NOTE	This command has to override the `Command#isToCreateNewPane()` to return a `true` value, since it should refresh the display pane each time by randomly getting new foods, rather than getting the existing display pane from `typeToPaneMap`.

Diagram

The following sequence diagram shows how recmfmix operation works. It also indicates the general flow between `recmf` commands with `Model`, `Logic`, and `Ui`.

API: `FoodCalculator.java`

Implementation of other supplementary commands

The following three commands are designed for user to modify the food database.

addfood and *deletef* commands

`AddFoodCommandParser` and `DeleteFoodCommandParser` are used for parsing these two commands respectively. Parameter validation is done by `RecmFoodParserUtil`.

API: `RecmFoodParserUtil.java`

resetf command

It is implemented by setting the internal list of `UniqueFoodList` to be the pre-loaded food data in `SampleFoodDataUtil`.

Example Usage Scenario and Summary

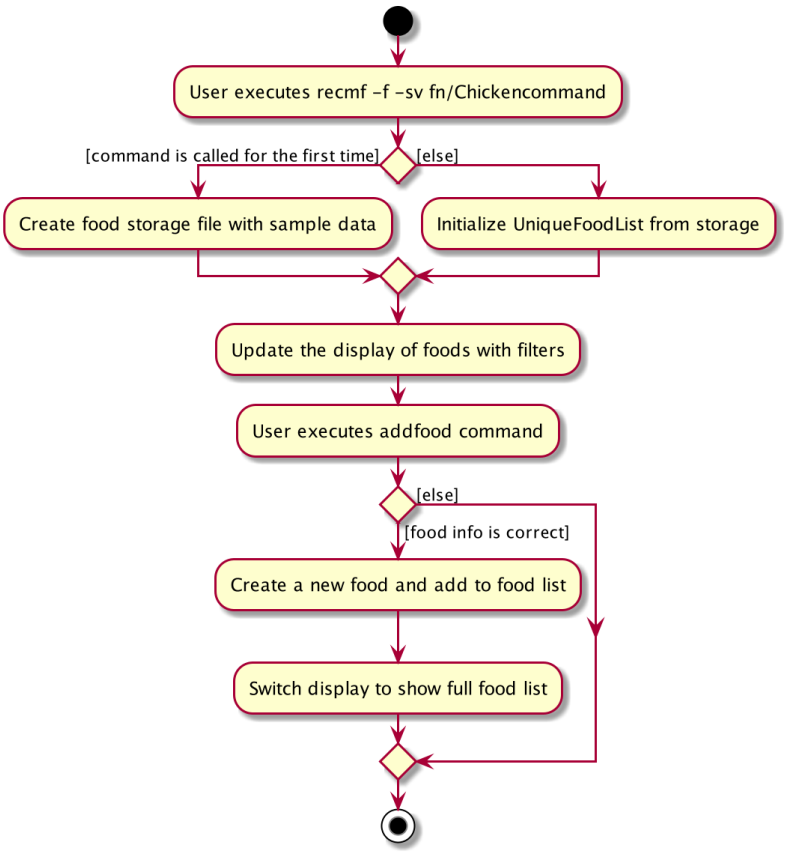
Given below is an example usage scenario and how the food recommendation mechanism behaves at each step.

1. The user launches the application and enter `recmf`.
    a. For the first time, the `foodlist.json` storage file will be created and written with the pre-loaded food data. Otherwise, data is loaded from the existing storage file.
    b. `FoodFlowPane` obtains food list information from `Logic` and displays food cards to the user.
2. The user executes `recmf -f -m +sort/gi` command.
    a. `FoodTypeIsWantedPredicate` selects foods of *fruit* and *meal* types. `FoodComparator` sorts foods in ascending order based on their GI values.
    b. `Model` updates the `filteredFoodList` with this predicate and sorts the list with this comparator.
    c. `FoodFlowPane` notices the such updates from `Listener` and refreshes the GUI.
3. The user executes `recmfmix` command.
    a. `UniqueFoodList#getMixedFoodList()` is executed, during which `FoodCalculator` calculates the nutrition values for *Summary* food.
4. The user executes `addfood fn/Cucumber ft/nsv ca/15 gi/15 su/1.7 fa/0`.
    a. The display switches to show the full list containing the newly added food.
    b. The storage file updates accordingly.
5. The user executes `resetf` command.

a. `UniqueFoodList` resets its internal list to hold the sample data from from `SampleFoodDataUtil`.

Diagram

The following activity diagram summarizes the above steps.



Design Considerations

Aspect: Data Structure of the Food Collection

- **Alternative 1 (current choice):** Use a `List` to store all the foods
  - Pros: The logic can be easily understood.
  - Cons: Operations on foods, such as filtering and adding, need to iterating through the whole list.
- **Alternative 2:** Use a `Map` that categorizes foods based on their food types
  - Pros: Impoves efficiency of filtering by flags by simply `get()`. Besides, maintaining the order after adding a new food only requires to sort foods of the same type. It can improve efficiency especially the database is large.
  - Cons: There is no `FilteredMap` class supported by JavaFX. Extra work is needed to make it accept `Predicate`.

Aspect: The presentation (UI) of food recommendations

- **Alternative 1 (current choice):** Show the user a pane of cards. Different types are indicated by the different colors.
  - Pros: Easy to implement. Cheerful colors may make reading more pleasant.

    - Cons: The size of food cards cannot be customized. If the window size is relatively small, the user may need to repeatedly scroll up and down to locate certain foods.
- **Alternative 2:** Use several horizontal `ListViews` to hold different food type.
    - Pros: The content is more organized and the user does not need to specify food types in the filter. Besides, the food cards can be customized for each `ListView`, such as omitting GI and Sugar for proteins since they are usually zero.
    - Cons: The operations targeting at the whole list need to be applied separately for each food list.

Aspect: Inputting New Food Data

- **Alternative 1 (current choice):** Require inputs for all fields (e.g. calorie, gi…).
    - It prevents some foods from permanently having empty fields, which may result in inappropriate sorting and summaries.
    - Cons: There is no way to add new foods with currently unavailable fields.
- **Alternative 2:** Use a separate list to hold foods with incomplete inputs.
    - Pros: This makes user inputs more flexible.
    - Cons: Extra work is needed to apply changes on two lists and transfer data from one list to the other.

Future Development Suggestions [Proposed Features]

- Recovering data after resetting This would be useful if the user wrongly enter `resetf` command, or another user want to temporarily use the same jar file on the same PC.
- Editing Foods This would provide more flexibility to the user to manipulate food data, instead of resetting all the food data.
- Disliking Foods This would prevent the user from repeatedly seeing the foods they dislike, cannot eat (due to religion reason), or are allergic to.
- Recording and Analyzing diets This would allow the user to have an overview of his food consumption statistics. Bases on such statistics, more specific suggestions can be proposed to to balance the user's nutrition intake.