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

Given below are sections I contributed to the User Guide. They showcase my ability to write documentation targeting end-users.

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

1. 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

Given below are sections I contributed to the Developer Guide. They showcase my ability to write technical documentation and the technical depth of my contributions to the project.

Food Recommendation Feature

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

- **Showing filtered food cards** — Shows recommended foods to the users, which are filtered by`Flags` and / or `FoodNames`.
- **Sorting the food list** — Sorts the recommendation order based on comparing food fields specified in `SortOrderType`.
- **Showing concise recommendations** — Recommends one food from each food type with an additional *summary* food.
- **Adding new a food** — Adds a new food to the food database and for future recommendations.
- **Resetting food database** — Clears user-added foods.

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

Overview of Data Structures

The main data structures used to support food recommendation feature are **Food Model**, **UniqueFoodList**, and **Predicates**

1. Food Model

- **Food Model**: holds data for a certain food
- Predicates: indicates the filters that the user wants to apply on partial presentation of the foods
- Unique Food List: holds the collection of all foods

API: `Food.java`

2. UniqueFoodList

The `UniqueFoodList` is the main model that contains all the foods and interacts with logic and UI. It exposes an unmodifiable `ObservableList<Food>` that associates the UI display of food recommendations.

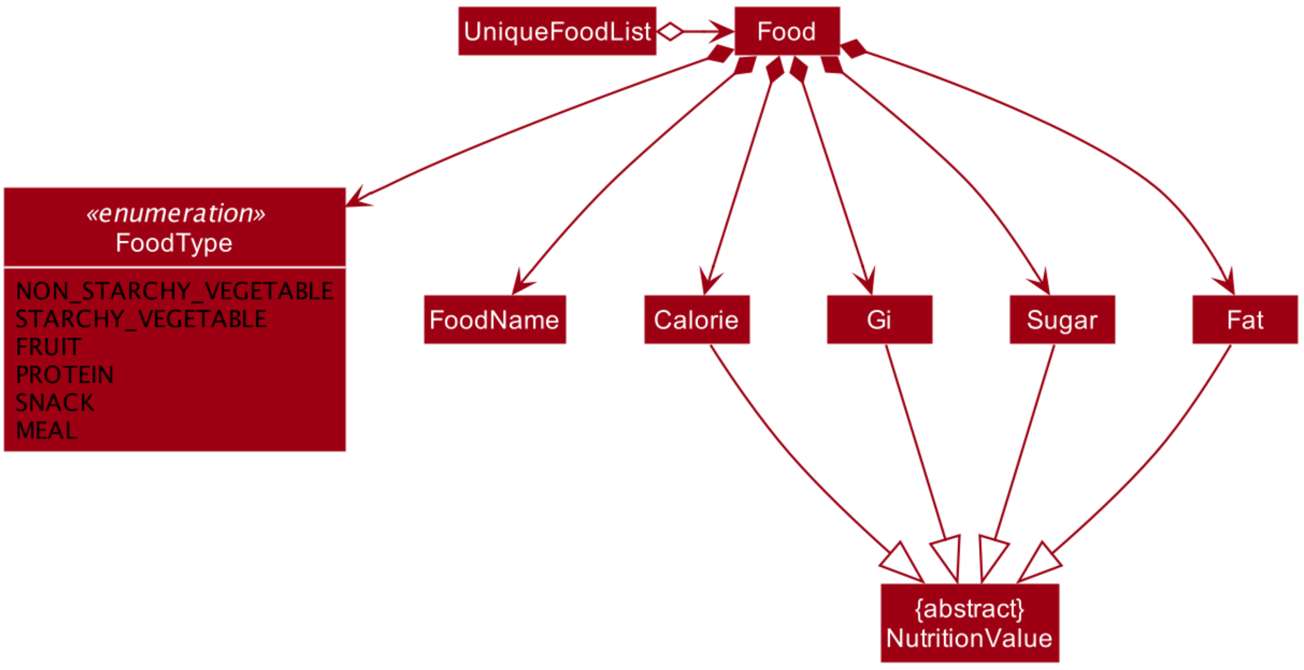
API: `UniqueFoodList.java`

3. Predicates

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

The following class diagram shows the main association and interactions among the main components. Other essential parts are summaries as follows:

- UI: `FoodFlowPanel` holds an `ObservableList` of `Food`, each visulized in the form of `FoodCard`.
- Storage: Reading data from and writing data to is handled by `JsonFoodListStorage`, inherited methods from `JsonGeneralStorage`.



Implementation of *recmf* and *recmfmix* command

recmf command

The `recmf` *command visualizes medically suggested foods as food cards for diabetics that are contained in* `UniqueFoodList`.

The customised presentation of food recommendations is implemented via 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 food types in `FoodType`. The flags in the user input will eventually be translated to `FoodTypeIsWantedPredicate` and applied on `UniqueFoodList`.

Data Structure

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

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

API: Flag.java

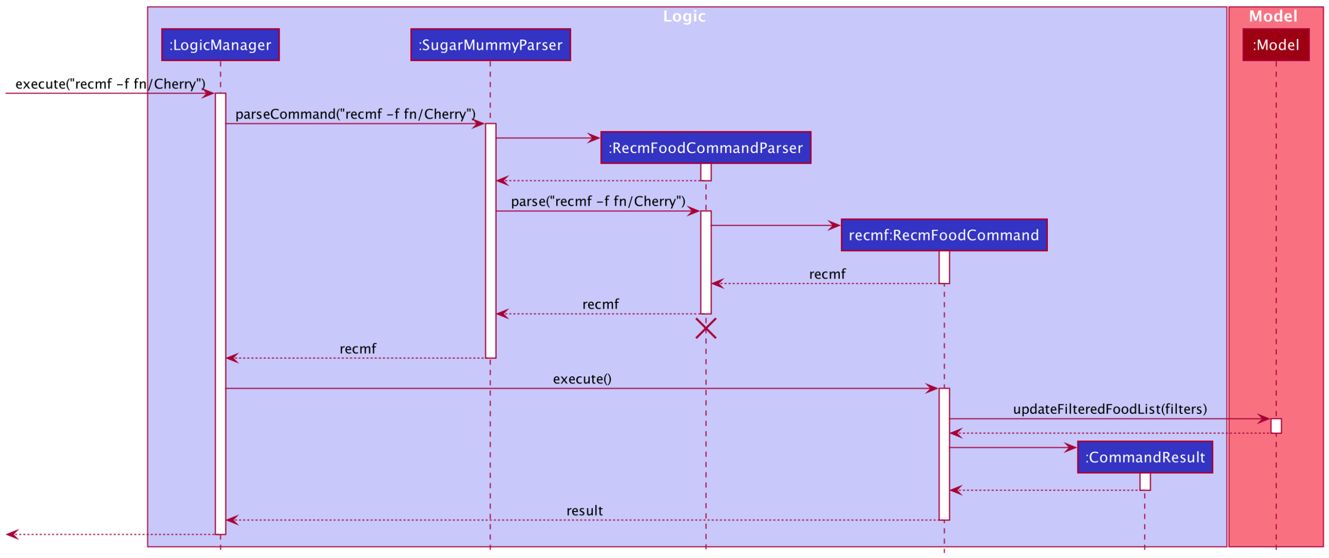
2. Filtering food names

A `List` of strings as food names will be supplied to `FoodNameContainsKeywordsPredicate`. Similar to the situation of no specified flags, an empty `ArrayList` will be returned by `RecmFoodCommandParser#getWantedFoodKeywords(namesStr)`.

The implementation is simple, details about `FoodNameContainsKeywordsPredicate` and `RecmFoodCommandParser` can be referred at Overview section.

Diagram

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



3. Sorting

Sorting the food recommendations is via supplying a `FoodComparator` to `UniqueFoodList#sortFoods`. To maintain or change the ordering of food recommendations, `model#sortFoodList` method must be added to `execute` method of relevant commands.

NOTE The default sort order is currently set to be by food type, specified in public variable `FoodComparator#DEFAULT_SORT_ORDER_STRING`. It is used when there is no specified sort type from the user and for the `addfood` command.

Data Structure

- A `Comparator` is wrapped by `FoodComparator` to handle the main logic, such as reversing the `FoodComparator` via `Comparator#reversed()`.
- A inner enum class SortOrderType` is used to specify all the available food fields for comparison and sorting. (See Food model for its field details.)

NOTE Instantiating `FoodComparator` by other classes is done by supplying `String` that matches one of the enum value of its inner enum class `SortOrderType`. Instantiate FoodComparator directly from `Comparator` is for internal usage of getting reversed `FoodComparator`.

API: `FoodComparator.java`

Diagram

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

recmfmix **command**

Compared to `recmf` command with customized options, `recmfmix` *is a simpler command that concisely recommends one food from each type with a summary food card at the end.*

General: Randomly selecting foods is implemented by `UniqueFoodList#getMixedFoodList()` that generates a separate and temporary `ObservableList` from the existing food data of `UniqueFoodList`. This list of mixed foods will be accessible by the `Model` and will be further used by the `Logic` to fill the content of `FoodFlowPanel`.

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 work.

API: `FoodCalculator.java`

Implementation of other supplementary commands

The following two commands are designed to help expand and clean up database of foods.

addfood **command**

`addfood` _command adds a new food with all specified fields into the food list.

It is implemented by using `AddFoodCommandParser`, which relies on `RecmFoodParserUtil` to check the validation of input values.

API: `RecmFoodParserUtil.java`

resetf **command**

`resetf` _command clears(deletes) all newly added foods from the user.

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

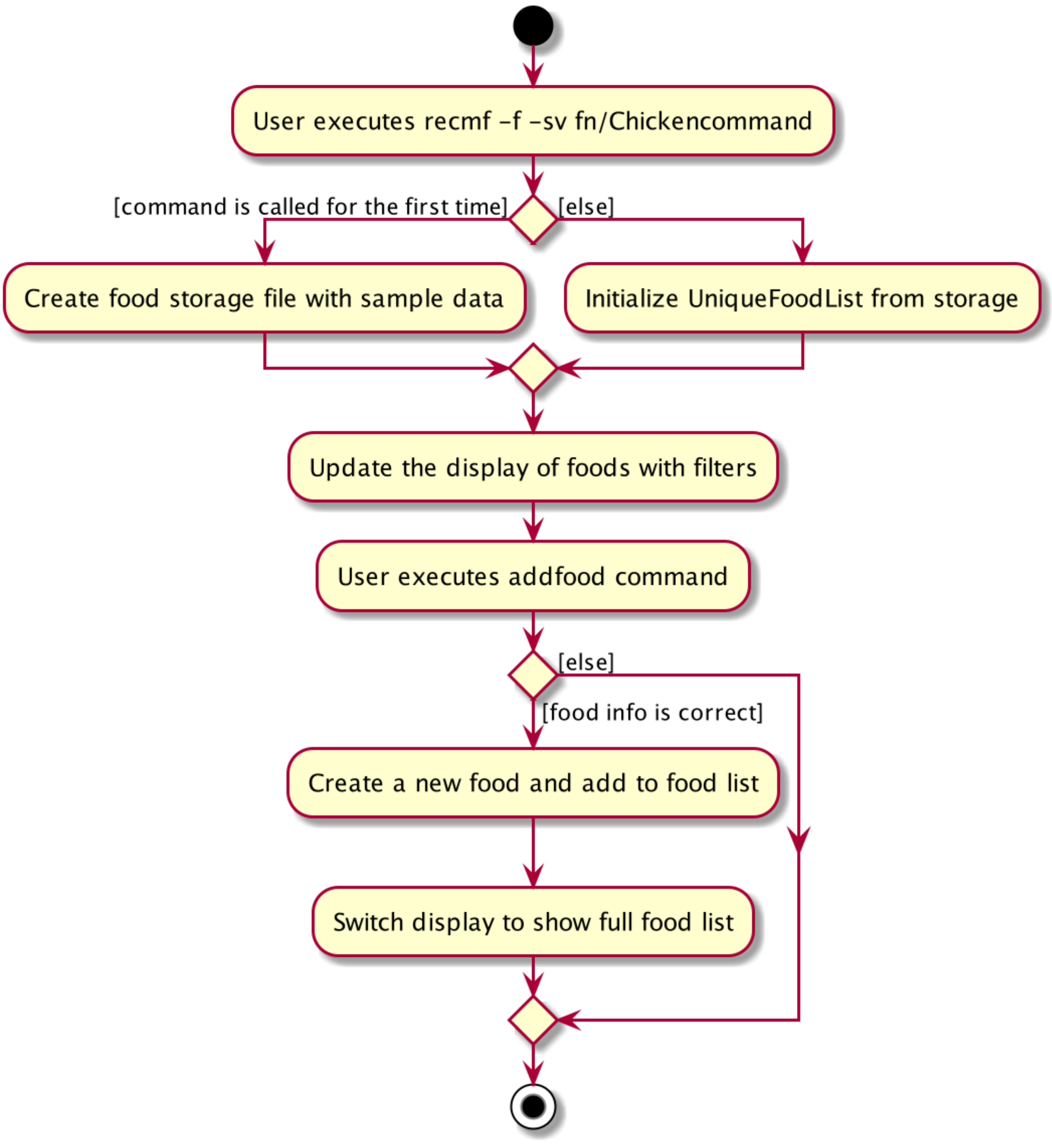
Example Usage Scenario and Summary

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` command for the first time.
   a. The `foodlist.json` storage file will be created and written with the pre-loaded food data from `SampleFoodDataUtil`.
   b. The `UniqueFoodList` will indirectly (via `Model`) supply a list of foods to `FoodFlowPane` to display.
2. The user executes `recmf -f -sv fn/Chicken` command.
   a. `RecmFoodParser` will parse the flag string as *fruit* and *starchy vegetable* for `FoodTypeIsWantedPredicate` and the food name string as *Chicken* for `FoodNameContainsKeywordPredicate`. Meanwhile, `FoodComparator` will be of default sorting order
   b. They will be supplied to instantiate a `RecmFoodCommand` to update the display of food recommendations.
3. The user executes `recmfmix` command.
   a. The `UniqueFoodList#getMixedFoodList()` method will return a list of randomly selected foods from each food type.
   b. A summary food with calculated value by `FoodCalculator` will be appended at the end.
4. The user feels the database is insufficient and wants to add a new food by executing `addfood fn/Cucumber ft/nsv ca/15 gi/15 fa/0 su/1.7`.
   a. The display will switch to show the full list containing the newly added food in the default sorting order.
   b. The `foodlist.json` storage file will be immediately updated with this new food.
5. The user executes `resetf` command.
   a. `UniqueFoodList` will reset its internal list to hold the sample data from from `SampleFoodDataUtil`.

Diagram

The following activity diagram summarizes the above steps.


User executes recmf -f -sv fn/Chickencommand
[command is called for the first time]
[else]
Create food storage file with sample data
Initialize UniqueFoodList from storage
Update the display of foods with filters
User executes addfood command
[else]
[food info is correct]
Create a new food and add to food list
Switch display to show full food list

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: Filtering, sorting, and adding new foods need to enumerating through the whole list.
- **Alternative 2:** Use a `Map` that categorizes foods based on their food types

    - Pros: For the `Flag` filtering, it can simply get the wanted types from the `Map`. 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. Thus, additionally structures needs to be defined to accept `Predicate` as filters.

Aspect: The presentation of food recommendations (UI)

- **Alternative 1 (current choice):** Show the user a pane of cards. Different types are indicated by the different background colors of the food names.
    - Pros: Easy to implement. The usage of cheerful colors may make reading recommendations 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 some 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 card for different food types can be more targeted. For example, for most proteins, the sugar and gi of value 0 can be omitted while protein values can be added.
    - Cons: The operations targeting at the whole list, such as filtering based on food names, need to be applied repeatedly for each food list.

Aspect: Inputting New Food Data

- **Alternative 1 (current choice):** Require inputs for all fields (e.g. calorie, gi…).
    - Pros: It is easy for data manipulation. Specifically, this prevents some foods from permanently having empty fields. Additionally, this may further hinder the data usage for data analysis.
    - Cons: Some data may not be currently available while the user still to want to record a new food by simply inputting the food name.
- **Alternative 2:** Allow temporarily empty fields and use a separate list to hold such incomplete inputs.
    - Pros: This provides the user with more freedom and flexibility of entering data.
    - Cons: Every change or manipulation on food data needs to be applied on two lists. Transferring data from one list to the other may also be error-prone.

Aspect: Data Recovery after Resetting

- **Alternative 1 (current choice):** There is additional storage for holding the food database just before resetting.
    - Pros: The implementation is straightforward. The management of storage is also simple since it only needs synchronized with one `UniqueFoodList`.
    - Cons: There is no way for the user to recover data.
- **Alternative 2:** Pop up another command to confirm with the user about the resetting.

    - Pros: Prevent the situation of resetting all food data due to accidentally entering a wrong command.,
    - Cons: The recovery is still unavailable. Besides, this expands one command to two steps.
- **Alternative 3:** Save only one copy before resetting all the food data. Update that copy whenever `reset` command is executed.
    - Pros: Provide more flexibility for the user to temporarily clean up the food data. This may be useful when another user want to temporarily use the same jar file on the same PC to get food recommendations.
    - Cons: Only the latest history of food list is available.
- **Alternative 4:** Save every copies before resetting.
    - Pros: Provide more freedom to the user to manipulate the history.
    - Cons: The implementation can be complex. Besides, it may take up much more storage if the database is large.

Future Development Suggestion

- Editing and Deleting 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.
- Expanding the food database This would relieve the extra work required from the user to input unavailable food data. Ideally, the recommendation data can be connected to online database for dynamic updates while can be stored locally for offline operations.
- 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.