Manas Vegi - Project Portfolio

Project: iFridge

Overview

iFridge is a desktop grocery management application to encourage home cooks to manage their food waste. The user interacts with it using a CLI, and it has a GUI created with JavaFX. It is written in Java, and has about 20kLoC.

Summary of Contributions

- **Major Enhancement**: add the ability to merge bought items into the grocery list.
  - **What it does**: This feature represents moving all bought items into the fridge (grocery list).
  - **Justification**: This feature is crucial to the product and user experience; it is a single command solution to merge all the bought shopping items into the grocery list without having to manually add them with multiple `glist add` commands.
  - **Highlights**:
    1. The feature is able to account for the fact that the fridge (grocery list) may already have items that share the name and expiry date as the bought items.
       - Amount of grocery item is just updated if bought item and grocery item have same name and expiry.
       - Bought item is added to the grocery list otherwise.
    2. The feature is able to account for the fact that users do not have to buy exactly as much as they stated in the shopping list.
       - If the user buys more than or equal to the amount stated in the shopping list, the shopping item is removed.
       - Else, the shopping item’s amount is just reduced by the amount bought and bought tag of the shopping item is removed (no bought items left to move to fridge).
    3. The feature is able to account for different units when adding the quantity of the bought item to the quantity of the grocery item.
       - Grocery item’s amount is given preference, i.e., the bought item’s amount’s unit is converted to match the unit of the grocery item’s amount.
- **Minor Enhancement**: add the ability to mark shopping items as urgent.
  - **What it does**: This feature adds a red-colored tag 'Urgent!' and moves it to the top of the shopping list.
  - **Justification**: This feature significantly adds to the user experience. All prioritized items are shown at the top of the shopping list to grab the user’s attention.

    - **Highlights**:
        1. Since a user would not consider an item urgent once they buy their requirements, an item loses its urgent tag once it is completely bought.
        2. Shopping list is sorted by urgent status first and then alphabetical order.

- **Major Enhancement**: display bought status of the amount of shopping item bought along with the shopping item in the UI and move around shopping items in the list according to the amount of each item bought.
    - **What it does**: This feature deals with user's flexibility of buying any amount of items. This feature also deals with edited amounts of shopping items. The following are the changes that will take place according to a shopping item's bought status:
        - If an item is completely bought, the shopping item will get a tag `Fully Bought! (<amount bought>)` and is **moved to the bottom of the list**.
        - If an item is partially bought, the shopping item will get a tag `Partially Bought. (<amount bought>)`
    - **Justification**: This feature makes the user experience much more convenient. Since the fully bought items are moved to the bottom of the list and urgent to top, the user will save time by looking at urgent and not bought items at the top of the list.
    - **Highlights**
        - For each shopping item the bought list is checked to deduce and display the bought status and bought amount of the shopping item.
        - Fully bought items are moved to the bottom of the list.
        - If an urgent item is completely bought, it loses its urgent tag, gets a `Fully Bought` tag and is moved to the bottom of the shopping list.
- **Major Enhancement**: edit shopping items
    - **What it does**: This feature allows the user to edit a shopping item's name and amount. This feature is able to deal with several special cases involving units, bought status and urgent status.
    - **Justification**: This feature significantly improves the user experience by dealing with editing wrongly entered items in the ideal way.
    - **Highlights**
        - If the name of a shopping item that is partially/fully bought is changed, the names of all the corresponding bought items are changed in order to maintain consistency for the user.
        - If a fully bought shopping item's amount is edited to an amount greater than the bought amount, the item's tag changes to 'Partially Bought.' and is accordingly moved up the shopping list.
          Similarly, if a partially bought shopping item's amount is edited to an amount smaller than or equal to bought amount, the item will be marked as 'Fully Bought!' and lose 'Urgent!' tag if it has one and moves to bottom of the list.
          This ensures that the user is not afraid of minor mistakes or changing requirements causing inconsistencies in iFridge.

- **Code contributed**: The samples of my functional and test code can be found here
- **Other contributions**:
  - Project Management
    - Created and managed issues for the Shopping related tasks of iFridge
    - Managed and resolved issues from bug reports
  - Enhancements to existing features:
    - Added and modified equality checks in classes to assist in easier testing
    - Wrote additional tests (with and without Stubs) for many existing features to increase coverage
  - Documentation:
    - Updated README to include marketing jargon for iFridge: #1
  - Community:
    - Reported bugs and suggestions to other teams in the class (examples: #2, #3, #4)

Contributions to User Guide

Below are some notable sections I contributed to our User Guide, which showcase my ability to write documentation that is beneficial to our target end-users. Please refer to section 3.7 for all shopping list features.

Shopping List Management

Add new item to shopping list: `slist add`

Adds a new item to the shopping list.
Format: `slist add n/FOOD_ITEM a/AMOUNT`

- Cannot add item with duplicate name.
- Cannot add item with unit that does not match with some other pre-existing item with same name.

Examples:

- `slist add n/apple a/2units`
- `slist add n/milk a/1L`
- `slist add n/banana a/3units`

Edit item in shopping list: `slist edit`

Edits the name of a specified item in a shopping list.

Format: `slist edit INDEX [n/FOODNAME] [a/AMOUNT]`

> - Edits the food item at the specified `INDEX`. The index refers to the index number shown in the shopping list. The index **must be a positive integer** 1, 2, 3, …
> - At least one of the optional fields must be provided.
> - Existing values will be updated to the input values.
> - Units in the amount field must match that of the item specified

Examples:

- `slist edit 3 n/pomegranate` Edits the name of the third food item in the shopping list to `pomegranate`
- `slist edit 2 a/2kg` Edits the amount of the second food item in the shopping list to 2 litres.

Delete item in shopping list: `slist delete`

Delete specified item from shopping list.
Format: `slist delete INDEX`

> - Deletes the food item at the specified `INDEX`. The index refers to the index number shown in the shopping list. The index **must be a positive integer** 1, 2, 3, …

Examples:

- `slist delete 1` Deletes the first food item in the shopping list.

Show shopping list: `slist list`

Lists out all items in the shopping list with bought items first Format: `slist list`

Examples:

- `slist list` Shows all entries in the shopping list.

Mark item as 'bought’: `slist bought`

Marks the specified item as bought and assigns the expiry date and amount to that item

Format: `slist bought INDEX e/EXPIRYDATE a/AMOUNT`

- Marks the shopping item at the specified `INDEX` as bought. The index refers to the index number shown in the shopping list. The index **must be a positive integer** 1, 2, 3, …
- Both `EXPIRYDATE` and `AMOUNT` of items bought must be specified. `AMOUNT` must match with that of the item in the shopping list.
- There is flexibility in the amount of items that are bought. You can indicate more or less quantity of items bought than the number indicated in the shopping list.
- When an item is completely bought, it is moved to the bottom of the list with a 'Fully Bought' tag.
- Once an item is completely bought, it loses its 'urgent!' status if it has one.
- If an item is partially bought, it gets a 'Partially Bought' tag.

Examples:

- `slist bought 1 e/03/12/2019 a/3units` Marks the item at index 1 as bought and sets its expiry date as 3rd December, 2019 and amount as 3 units.
- `slist bought 2 e/03/11/2019 a/1kg` Marks the item at index 2 as bought and sets its expiry date as 3rd November, 2019 and amount as 1 kilograms.

Add bought items: `slist mergebought`

This command represents a user moving all the bought items into the fridge. It adds all items marked as ‘bought’ to the grocery list. Format: `slist mergebought`

- Adds all the items marked as bought in shopping list into the grocery list.
- If the item with same expiry date is already present in the shopping list, just add the amount to the amount already present in the grocery list.
- For each bought item, if the amount bought is less than the amount of that item in the shopping list, the amount of that item in the shopping list is now the remaining amount to be bought.
  - Eg: Consider a shopping item 'NAME: Orange, AMOUNT: 2units' in the shopping list.
  - Then, when marking the item as bought, you indicate the amount of 'Orange' that is bought is 1unit.
  - If now you call `slist mergebought`, shopping list will still have 'NAME: Orange, AMOUNT: 1units', i.e., the amount in shopping list is subtracted to reflect the remaining amount to be bought.

Important Note: This command cannot be undone/redone.

Contributions to Developer’s Guide

Below are some notable sections that I contributed to the Developer's Guide that showcase my ability to write technical documentation and the technical depth, completeness, and difficulty of my contributions to the project.

MergeBought Feature

Implementations

The user is able to merge all the bought shopping items into the grocery list. This can be done with the use of the `slist mergebought' command.

The current implementation to merge bought items is as follows:

- For each boughtItem in boughtList:
    - For each groceryItem in GroceryList
        - If there is an existing groceryItem in GroceryList with same name and expiry date as the boughtItem, the quantity of that groceryItem will be updated according to the amount bought.
        - If there is no existing groceryItem with same Name and ExpiryDate, the boughtItem will be added to the GroceryList.
    - For each shoppingItem in ShoppingList find the bought shoppingItem with same name as the boughtItem:
        - If the boughtItem has greater Amount than its corresponding shoppingItem, the shoppingItem is deleted from the ShoppingList.
        - If the boughtItem has lesser Amount than its corresponding shoppingItem, the shoppingItem's quantity is just reduced.

The Merge Bought mechanism is facilitated by `ShoppingList`, `GroceryList`, and `ShoppingComparator`. The `ShoppingList` is an observable list of `ShoppingItem` and `GroceryList` is an observable list of `GroceryItem`.

Merging needs to use the BoughtList to modify the ShoppingList and GroceryList. Hence, the command retrieves the 3 lists.

When a `GroceryItem` 's amount is be added, the GroceryItem at that index is overwritten by a new GroceryItem with the updated amount. If a new boughtItem (`Groceryitem`) is to be added to the `GroceryList`, it is appended to the `GroceryList`

Similarly, when a `ShoppingItem` 's amount is reduced, the ShoppingItem at that index is overwritten by a new ShoppingItem with the updated amount. If a `ShoppingItem` is to be deleted from the `ShoppingList`, it is deleted and the `ShoppingList` is sorted using a `ShoppingComparator`.

The creation of new objects to replace the existing ones is necessary since all the objects are immutable.

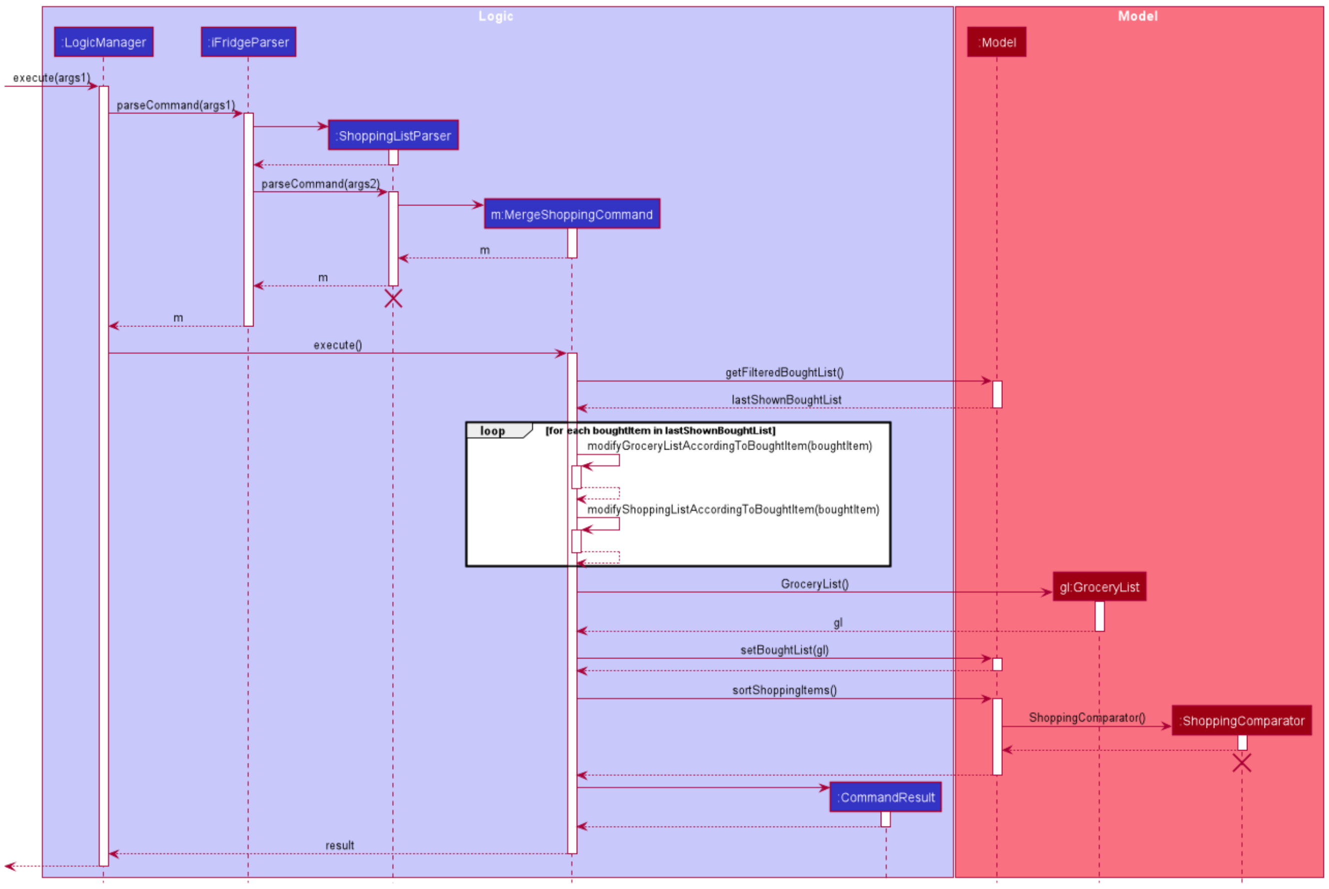
- Model#getFilteredBoughtList — Gets `ObservableList` with the elements of the `Model` 's boughtList

(stored in a `GroceryList`)

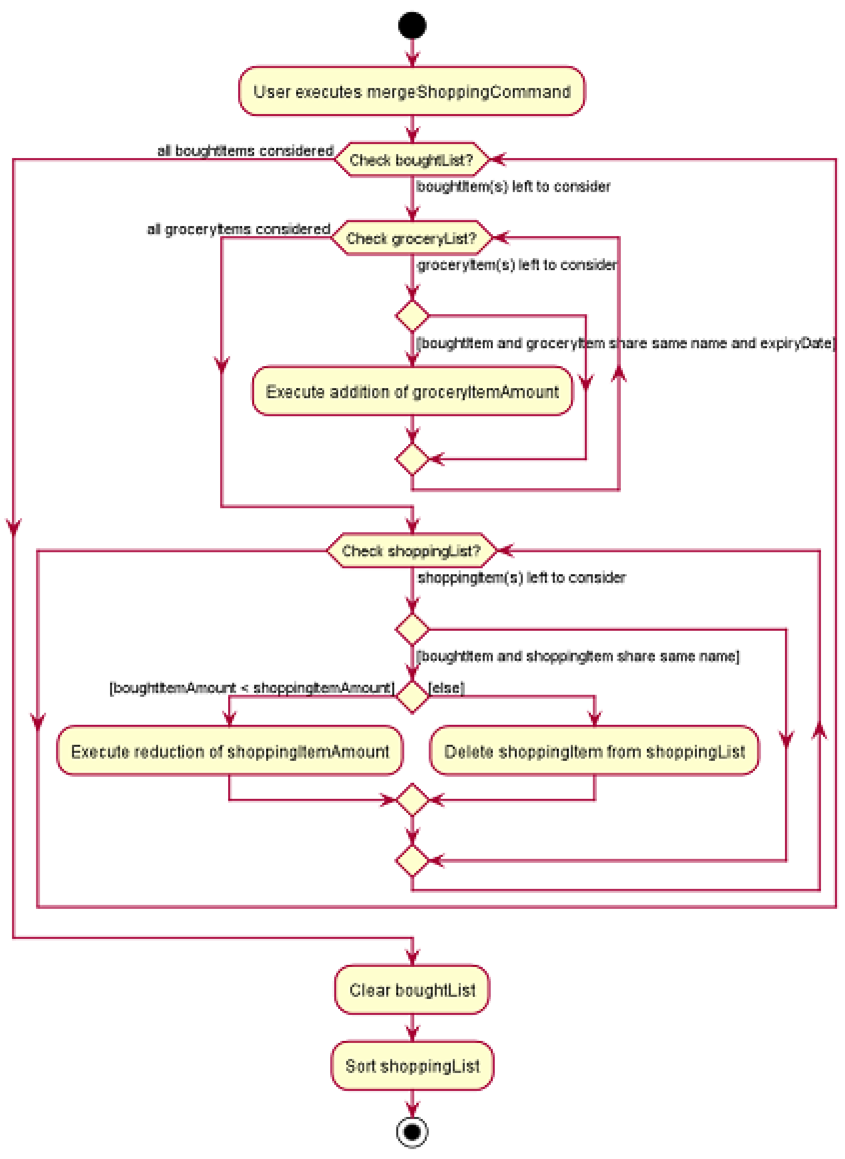
- Model#getFilteredGroceryList — Gets `ObservableList` with the elements of the `Model` 's `GroceryList`
- Model#getFilteredShoppingList — Gets `ObservableList` with the elements of the `Model` 's `ShoppingList`
- Model#setBoughtList — Sets the boughtList as the specified `GroceryList`
- Model#sortShoppingItems — Sorts the `ShoppingList` by urgent status first, and then by alphabetical order.
- MergeShoppingCommand#modifyGroceryListAccordingToBoughtItem — Updates `GroceryList` according to a boughtItem
- MergeShoppingCommand#modifyShoppingListAccordingToBoughtItem — Updates `ShoppingList` according to a boughtItem

These operations are exposed in the `Model` interface as `Model#getFilteredBoughtList`

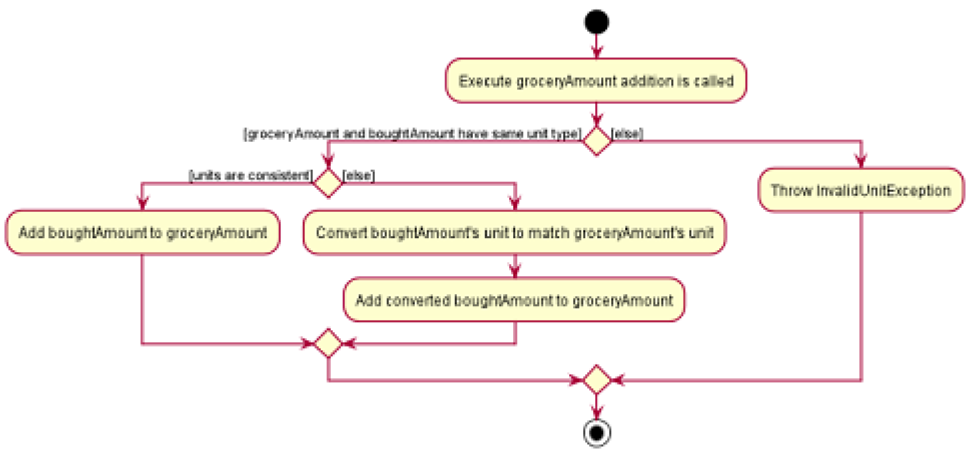
The sequence diagrams for interactions between Logic and Model components when a user executes `slist mergebought` command is shown below.



The following activity diagram summarises how `slist mergebought` is carried out.


User executes mergeShoppingCommand
all boughtItems considered
Check boughtList?
boughtItem(s) left to consider
all groceryItems considered
Check groceryList?
groceryItem(s) left to consider
[boughtItem and groceryItem share same name and expiryDate]
Execute addition of groceryItemAmount
Check shoppingList?
shoppingItem(s) left to consider
[boughtItem and shoppingItem share same name]
[boughtItemAmount < shoppingItemAmount]
[else]
Execute reduction of shoppingItemAmount
Delete shoppingItem from shoppingList
Clear boughtList
Sort shoppingList

The mergebought command supports unit conversion. The boughtAmount's unit will be converted to match the groceryAmount. The following activity diagram delineates how the unit conversion takes place.


Execute groceryAmount addition is called
[groceryAmount and boughtAmount have same unit type]
[else]
[units are consistent]
[else]
Throw InvalidUnitException
Add boughtAmount to groceryAmount
Convert boughtAmount's unit to match groceryAmount's unit
Add converted boughtAmount to groceryAmount

The conversions of units are done by `Amount` class. Unit type is necessary in the implementation to allow for keeping track of different unit groups. For example, `kg`, `g`, `lbs`, and `oz` are all categorised under the unit type `Weight`.

Please refer to section 3.13.2 of DG for design considerations of mergebought feature

Shopping List Manual Testing

1. List all items in shopping list
    a. Test case: `slist list`
    Expected: The shopping list is displayed, with 3 sections (urgent, not urgent and not fully bought, fully bought). Within each of the three sections of the shopping items, items are sorted alphabetically.
2. Adding an item to the shopping list
    a. Test case: `slist add n/Grapes a/500g`
    Expected: Shopping item with name `Grapes` and amount `500g` is created and added to the shopping list. The details of the added shopping item are displayed. The new shopping item appears after urgent items and before fully bought items in the shopping list.
    b. Test case: `slist add n/Grapes a/invalidAmount`
    Expected: Error message displayed indicating invalid amount entered.
    c. Test case: `slist add n/Grapes`
    Expected: Error message displayed that indicates that the command format is wrong. The correct format for the add shopping command is also displayed.
    d. Test case: `slist add a/5g`
    Expected: Error message displayed that indicates that the command format is wrong. The correct format for the add shopping command is also displayed.
    e. Test case: `slist add`
    Expected: Error message displayed that indicates that the command format is wrong. The correct format for the add shopping command is also displayed.
3. Editing an item in the shopping list
    a. Test case: `slist edit 1 n/Oranges`
    Expected: First shopping item’s name is edited to 'Oranges'. Details of the edited shopping item are displayed. If some or all of the item was bought, the name of the corresponding bought item is also changed in storage to avoid inconsistencies.
    b. Test case: `slist edit 1 a/50g`
    Expected: First shopping item’s amount is edited to 50g. Details of the edited shopping item are displayed.
    c. Test case: `slist edit 1`
    Expected: Error message displayed indicating that at least one field to be edited must be provided.
4. Deleting an item in the shopping list
    a. Test case: `slist delete 1`
    Expected: First shopping item is deleted from the shopping list. Details of the removed shopping item are displayed.
5. Marking a shopping item as urgent

a. Test case: `slist urgent 1`
   Expected: Marks the first shopping item as urgent. Details of the shopping item marked as urgent are displayed. The shopping item gets an `Urgent!` tag and moves towards the upper section of the shopping list (with other urgent items).

b. Test case: `slist urgent`
   Expected: Error message displayed indicating wrong command format. The correct format of an urgent shopping command is also displayed.

6. Buying a shopping item (Marking a shopping item as Bought)

   a. Test case: `slist bought 1 a/50g e/24/11/2019`
      Expected: First shopping item is marked as bought. A bought item is created with the name of the shopping item and with the amount and expiry date provided in the command. It gets a `Partially Bought` tag if the amount bought is less than the amount of the shopping item, and it gets a `Fully Bought` tag if the bought amount is greater than or equal to the amount of the shopping item. In either case, the tag also displays the amount of shopping item bought. **If an item is fully bought, it moves to the bottom of the shopping list (regardless of whether it is urgent or not)**. Details of the shopping item marked as bought are also shown.

   b. Test case: `slist bought 2 a/0g e/24/11/2019`
      Expected: Error message displayed indicating that the amount specified cannot be 0 or negligible. The guidelines of how to use amount are also specified.

   c. Test case: `slist bought 1 a/50g`
      Expected: Error message displayed indicating that the command format is wrong. The correct format for a bought shopping command is also displayed.

   d. Test case: `slist bought 1 e/23/04/2019`
      Expected: Error message displayed indicating that the command format is wrong. The correct format for a bought shopping command is also displayed.

7. Merging bought items into the grocery list (moving bought items into fridge)

   a. Prerequisites
      Some of the items in the shopping list must be bought for changes in the lists to take place.

   b. Test case: `slist mergebought`
      Expected: All bought items are added to the grocery list according to the following conditions: If a bought item does not have a corresponding grocery item (same name and expiry date) in the grocery list, a new grocery item with the bought item's details is added to the grocery list. If a bought item has a corresponding grocery item in the grocery list, the grocery item's amount is added instead of creating a new grocery item.
      If a shopping item was fully bought, it is removed from the shopping list while merging. If the item is only partially bought, the quantity that is bought is subtracted from the shopping item's quantity and the bought tag will be removed.

Please refer to section 2.5 of the developer's guide to see my Design Architecture for Storage