Ananda Kumar - Project Portfolio

PROJECT: WishBook

Overview

WishBook (WB) is a piggy bank in a digital app form, made for people who need a user-friendly solution for keeping track of their disposable income. Not only does WB keep records of savings, it also allows users to set items as goals for users to work towards, serving as a way to cultivate the habit of saving. WB is made for people who have material wants in life, but are uncertain of purchasing said wants, by helping them to reserve disposable income so that they know they can spend on their wants. The user interacts with it by using a Command Line Interface (CLI), and it has a GUI created with JavaFX. It is written in Java, and has about 15 kLoC.

WishBook is an application adapted from addressbook-level4, the original application developed by the se-edu team.

The source code of addressbook-level4 can be found here.

This project portfolio consists of the contributions I have made to WishBook over the semester.

Summary of contributions

- **Major enhancement**: Added **the ability to save towards a wish or remove an amount from a wish.**
    - What it does: Allows the user to save a specified amount of money to a particular wish or remove a specified amount from it. A wish will be fulfilled when the user saves up to 100% of a wish's price and excess amount cannot be saved towards it any longer.
    - Justification: This feature is a crucial feature in the WishBook as it cannot fulfill its primary requirement without it.The fact that two actions, namely saving and removing can be done with a single command means that there are less commands to remember.
    - Highlights: This enhancement touches multiple components and required thorough consideration of alternative design. Hence it is naturally my most time consuming contribution. Significant effort was also expended in modifying the tests to test this new feature.
- **Minor enhancement**: Added **the ability to find wishes by name, tag and/or remark with specified level of match with keywords.**
    - Justification: The flexible design of this command allows user to specify varying number of arguments to the wish which gives the user multiple ways to find a wish or a group of wishes for ease of access.
    - Highlights: It required an in-depth analysis of design alternatives. The implementation too was challenging as it required rewriting and adapting the code written for an existing command. Significant effort was also expended in modifying the tests to test this new feature.

- **Minor enhancement**: Converted email attribute in a wish to date attribute which stores the due date of a wish. The date cannot be earlier or equal to the current date.
- **Minor enhancement**: Made the wish list to be permanently sorted by due date regardless of the commands applied on them.
    - Highlights: Required modification of test data which led to modification of a significant number of tests.
- **Minor enhancement**: Made the wishes be identified only by a newly added attribute, UUID. This allows for addition of wishes with the same attribute values, something which was not allowed previously. UUID of a wish also cannot be changed upon addition of a wish.
- **Minor enhancement**: Added a remark command.
    - Credits: Code was adapted from https://github.com/se-edu/addressbook-level4/pull/599.
- **Code contributed**: [Functional code]
- **Other contributions**:
    - Project management:
        - Managed release `v1.1` on GitHub
    - Documentation:
        - Did structural changes to User Guide: #67, #136
    - Community:
        - PRs reviewed (with non-trivial review comments): #4
        - Reported bugs and suggestions for other teams in the class (examples: #71, #20, #127, #130, #5, #6, #7, #8)

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.

Locate your wish with speed and precision: `find`

Find wishes which match the given search keywords.
Format: `find [-e] [n/NAME_KEYWORD]… [t/TAG_KEYWORD]… [r/REMARK_KEYWORD]…`

- At least one keyword must be provided.
- Searching multiple keywords of the **same prefix** will return wishes whose attribute corresponding to the prefix contain **any** one of the keywords.
- Searching with keywords of **different prefixes** will return only wishes that match will **all** the keywords of the different prefixes.
- Using the exact match flag, `-e` returns wishes whose corresponding attributes contain **all** the keywords.
- The search is case insensitive. e.g. watch will match Watch.

Examples:

- `find n/wat`
  Returns any wish whose name contains the *wat*.
- `find n/wat n/balloon n/appl`
  Returns wishes whose names which contain at least any one of *wat*, *balloon* or *appl*.
- `find -e n/wat n/balloon n/appl`
  Returns only wishes whose names that contain all of *wat*, *balloon* and *appl*.
- `find n/watch t/important`
  Returns any wish whose name contains *watch*, and whose tags contains *broke wishes*.
- `find n/wat n/balloon t/important`
  Returns any wish whose name contains *wat* or *balloon*, and whose tags contains *important*.

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.

Model

Wish Model

A wish is uniquely identified by its Universal Unique Identifier (UUID) which is generated randomly only once for a particular wish, upon its creation through the `AddCommand`. A wish stores the following primary attributes:

- Name
- Price
- Date
- Saved Amount
- Url
- Remark

- Tags
- UUID

NOTE It is impossible for the user to create a duplicate wish as it is impossible to modify a wish’s UUID.

Wish Priority

A wish needs to be prioritised in a specific order such that the wishes with the highest priority will be visible on the top of the list. In WishBook, the priority is determined primarily by the due date of the wish which is stored in every wish’s `Date` attribute. Ties are broken by `Name`. Further ties are broken by `UUID` as it is possible for the `Date` and `Name` of two wishes to be identical.

The sorting of the displayed results is done by the `filteredSortedWishes` list. The sorting order is specified by `WishComparator`.

Design Considerations

Aspect: Uniqueness of a Wish

- **Alternative 1(current choice):** Identify a `Wish` by a randomly generated UUID.
  - Pros: Extremely low probability of collision.
  - Pros: No extra maintenance required upon generation as every `Wish` is unique.
  - Cons: UUID does not map to any real world entity and it is used strictly for identification.
  - Cons: It is more difficult to system test the `AddCommand` with the current group of methods for system tests as UUID is randomly generated each time.
- **Alternative 2:** Identify a wish by `Name`, `Price`, `Date`, `Url`, `Tags`. Wishes with identical values for these attributes will be represented by a single `WishCard`. The `WishCard` will be augmented with a `Multiplicity` to indicate the number of identical wishes.
  - Pros: WishBook will be more compact and every attribute stored in a `Wish` maps to a real entity.
  - Cons: Additional attribute `Multiplicity` may have to be frequently edited as it is another attribute that is affected by multiple commands.
- **Alternative 3:** Identify a wish by a new attribute `CreatedTime`, which is derived from the system time when the wish is created.
  - Pros: The attribute maps to a real entity. It can be an additional information presented to the user about a wish.
  - Cons: There might be collisions in `CreatedTime` if the the system time is incorrect.

Find Wish Feature

Current Implementation

The find mechanism is supported by `FindCommandParser`. It implements `Parser` that implements the following operation:

- `FindCommandParser#parse()` — Checks the arguments for empty strings and throws a `ParseException` if empty string is found. It then splits the arguments using `ArgumentTokenizer#tokenize()` and returns an `ArgumentMultimap`. Keywords of the same prefix are then grouped using `ArgumentMultimap#getAllValues()`.

The find mechanism is also facilitated by `FindCommand`. It extends `Command` and implements the following operation:

- `FindCommand#execute()` — Executes the command by updating the current `FilteredSortedWishList` with the `WishContainsKeywordPredicate`.

The predicate `WishContainsKeywordsPredicate`, takes in three lists of the keywords for the following attributes:

- Name
- Tags
- Remark

and also the `isExactMatch` argument. The result of the predicate is determined by checking whether a `Wish` contains the given keywords at their corresponding attributes. The match threshold is dictated by the value of `isExactMatch`.

Example

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

Step 1. The user launches the application for the first time.

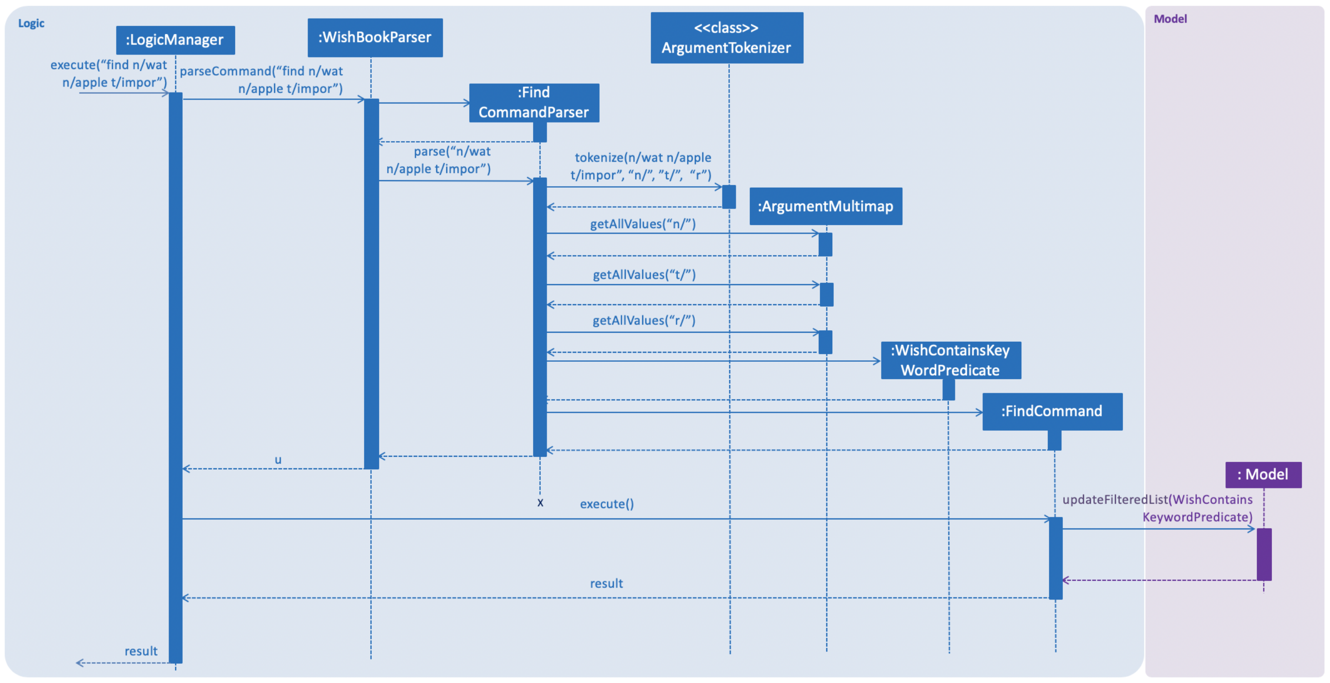
Step 2. The user executes `find n/wat n/apple t/impor` command to get all wishes whose name contains the keywords 'iphone' or 'tablet'.

Step 3. The `FindCommandParser#parse()` is called and the `WishContainsKeywordPredicate` is constructed with the arguments of the find command.

Step 4. `FindCommand#execute()` is then called.

Step 5. The entire list of wishes is filtered by the predicate `WishContainsKeywordsPredicate`.

Step 6. The filtered list of wishes is returned to the GUI.


Logic
:LogicManager
:WishBookParser
<<class>>
ArgumentTokenizer
Model
execute("find n/wat n/apple t/impor")
parseCommand("find n/wat n/apple t/impor")
:Find CommandParser
parse("n/wat n/apple t/impor")
tokenize(n/wat n/apple t/impor", "n/", "t/", "r")
:ArgumentMultimap
getAllValues("n/")
getAllValues("t/")
getAllValues("r/")
:WishContainsKey WordPredicate
:FindCommand
u
: Model
x
execute()
updateFilteredList(WishContains KeywordPredicate)
result
result

Design Considerations

Aspect: Argument format

- **Alternative 1 (Current choice):** Require the user to prepend every keyword argument with the appropriate Wish attribute prefix.
  - Pros: Easier to implement as it easier to match keyword against a Wish if the attribute to match against is known.
  - Pros: User has more control over the results returned.
  - Cons: User is required to type slightly more.
- **Alternative 2:** No prefixes are required in the arguments. Keywords can match with any one of the following chosen wish attributes: `Name`, `Tags` or `Remark`.
  - Pros: Less typing required from user.
  - Cons: Command might be slightly slower as every keyword has to be checked against all chosen attributes of the wish.
  - Cons: User has less control over the results returned.

Aspect: Default threshold for match without the exact match flag

- **Alternative 1 (Current choice):** Keywords appended to different prefixes are grouped with a logical AND and keywords appended to the same prefixes are grouped with a logical OR when being matched against a `Wish`.
  - Pros: A more intuitive way to find wishes.
  - Cons: Can be restrictive in some situations.
- **Alternative 2:** Keywords appended to different prefixes are grouped with a logical OR and keywords appended to the same prefixes are grouped with a logical OR when being matched against a `Wish`.

- Pros: Search results will be more inclusive.
- Cons: Very slim chance for such a use case.