Liao Meng - Project Portfolio

PROJECT: FitHelper

Overview

FitHelper is a desktop diet-and-exercise-recording application. It is built based on AddressBook (Level 3), a desktop address book application used for teaching Software Engineering principles. The user interacts with it using a CLI, and it has a GUI created with JavaFX. It is written in Java, and has about 18 kLoC.

Summary of contributions

- **Major enhancement 1**: added **the ability to sort entry and reminder list**
  - What it does: allows the application to sort and display the entries in food and/or sports entry list based on time/name/calorie value in ascending or descending order. The user can specify sort order and criterion. Note that the entries in the reminder list will also be sorted in the same manner.
  - Justification: This feature allows the user to view entries in a particular order. It also helps the user to view entry that he/she may deem more important first (for example entry with a higher calorie value, as it has a larger impact on the user's fitness).
  - Highlights:
- **Major enhancement 2**: added **the ability to search for calorie data of common food/sports**
  - What it does: allows the user to search for calorie intake/consumption of food/sports by keywords.
  - Justification: This feature makes it easier for the user to estimate how much calorie he/she will gain/burn by eating/exercising, so that he/she can create food/sports entry without the need to search online for relevant information.
  - Highlights: The search will
  - Credits: The calorie data are collected from the following websites: https://www.calories.info/ and https://www.nutristrategy.com/caloriesburned.htm
- **Minor enhancement**: added and updated the Help Page, which shows the list of valid user commands and their usage, and the link to the user guide.
- **Code contributed**: [Functional and Test code]
- **Other contributions**:
  - Project management:
    - Managed releases `v1.3` - `v1.5rc` (3 releases) on GitHub

- Enhancements to existing features:
    - Updated the GUI color scheme (Pull requests #33, #34)
    - Wrote additional tests for existing features to increase coverage from 88% to 92% (Pull requests #36, #38)
- Documentation:
    - Did cosmetic tweaks to existing contents of the User Guide: #14
- Community:
    - PRs reviewed (with non-trivial review comments): #12, #32, #19, #42
    - Contributed to forum discussions (examples: 1, 2, 3, 4)
    - Reported bugs and suggestions for other teams in the class (examples: 1, 2, 3)
    - Some parts of the history feature I added was adopted by several other class mates (1, 2)
- Tools:
    - Integrated a third party library (Natty) to the project (#42)
    - Integrated a new Github plugin (CircleCI) to the team repo

{you can add/remove categories in the list above}

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.

Deleting an entry : `delete`

Deletes the entry at the specified `INDEX` from the fitness log book.
Format: `delete x/TYPE i/INDEX`

- To obtain accurate indices, the user should first switch to the home page by calling `home` or `list` and refer to the displayed list in `food entry history` and `sport entry history` fields.
- Additionally, the user can first called a `find` and then refer to the resulting list's indices to `delete` an entry. However, right after a `find`, the user can only refer to entries that are currently displayed.
  i.e. If there are 5 food entries in total, and after `find` 2 food entries are displayed. If the user does not switch back to the home page, he can only refer to the displayed 2 entries (with indices 1 and 2).
- The index **must be a positive integer** 1, 2, 3, …

Examples:

- `delete x/sports index/2` Deletes the 2nd sports entry in the sports list from dashboard.

Encrypte data files `[coming in v2.0]`

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.

Undo/Redo feature

Implementation

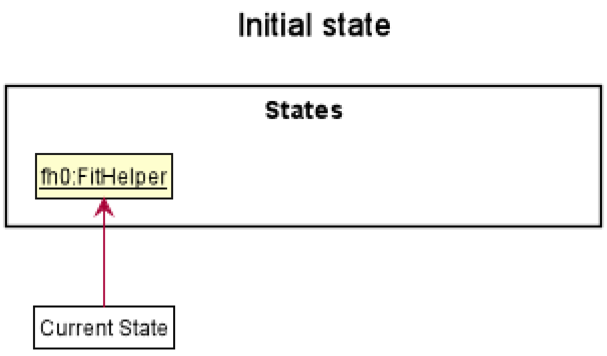
The undo/redo mechanism is facilitated by `VersionedFitHelper`. It extends `FitHelper` with an undo/redo history, stored internally as an `fitHelperStateList` and `currentStatePointer`. Additionally, it implements the following operations:

- `VersionedFitHelper#commit()` — Saves the current FitHelper state in its history.
- `VersionedFitHelper#undo()` — Restores the previous FitHelper state from its history.
- `VersionedFitHelper#redo()` — Restores a previously undone FitHelper state from its history.

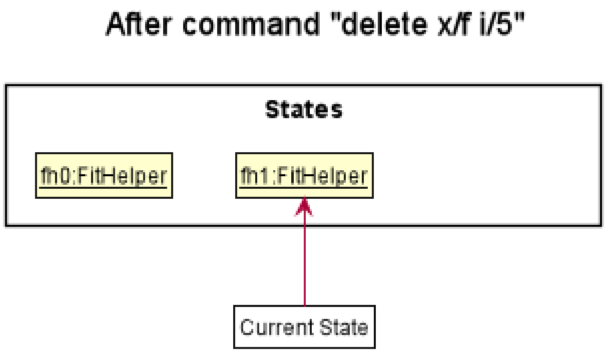
These operations are exposed in the `Model` interface as `Model#commit()`, `Model#undo()` and `Model#redo()` respectively.

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

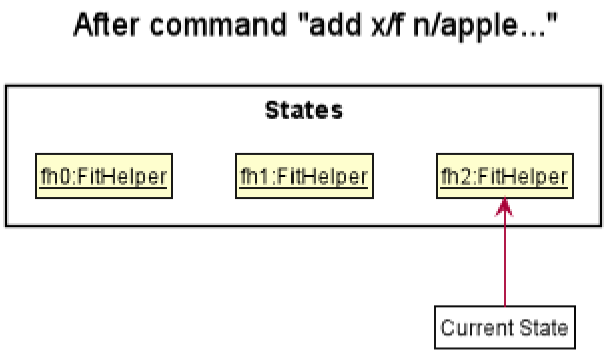
Step 1. The user launches the application for the first time. The `VersionedFitHelper` will be initialized with the initial FitHelper state, and the `currentStatePointer` pointing to that single FitHelper state.



Step 2. The user executes `delete x/f i/5` command to delete the 5th food entry in the FitHelper. The `delete` command calls `Model#commit()`, causing the modified state of the FitHelper after the `delete x/f i/5` command executes to be saved in the `fitHelperStateList`, and the `currentStatePointer` is shifted to the newly inserted FitHelper state.

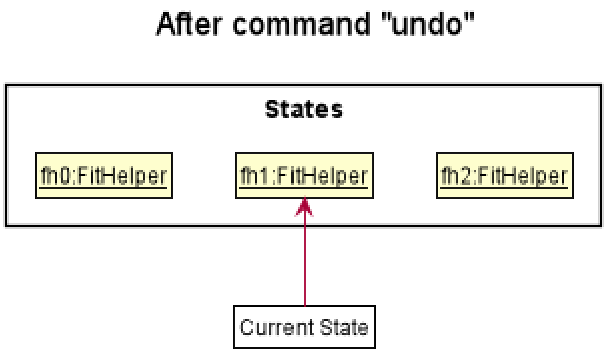


Step 3. The user executes `add x/f n/apple …` to add a new food entry. The `add` command also calls `Model#commit()`, causing another modified FitHelper state to be saved into the `fitHelperStateList`.



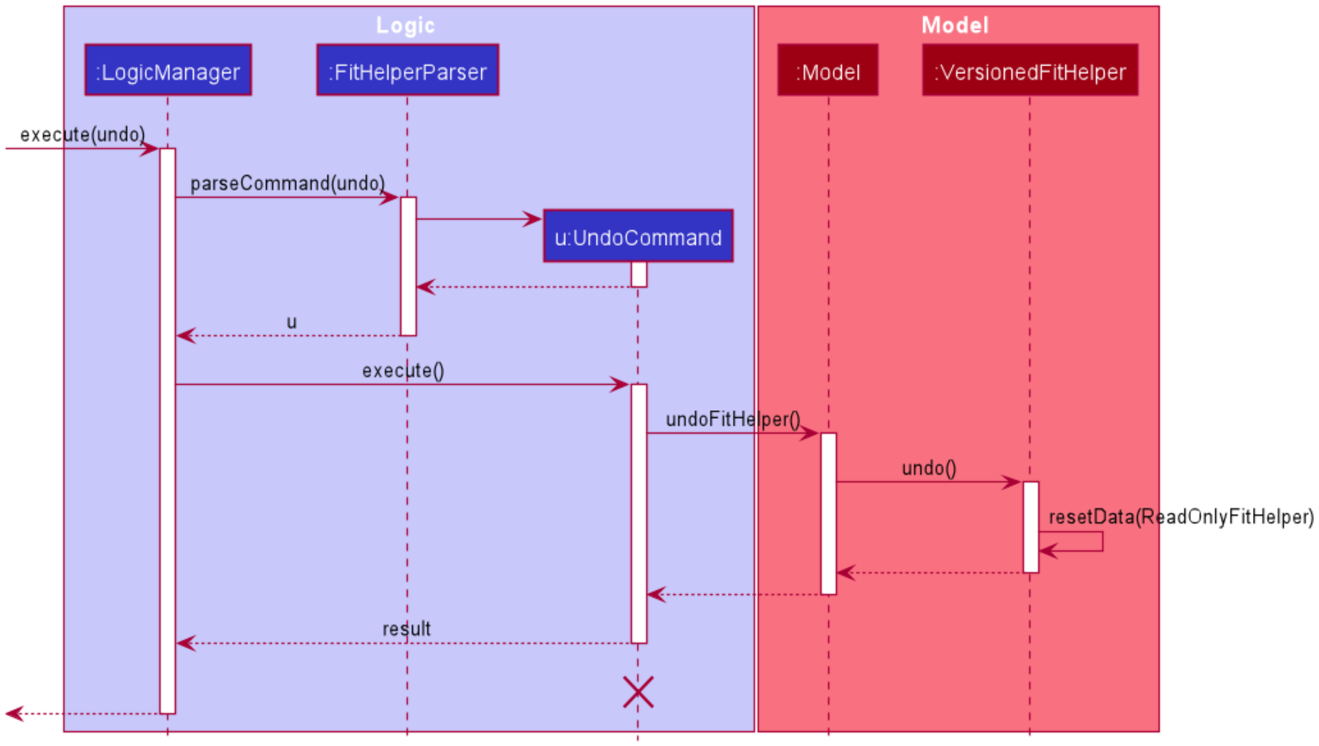
NOTE If a command fails its execution, it will not call `Model#commit()`, so the FitHelper state will not be saved into the `fitHelperStateList`.

Step 4. The user now decides that adding the food entry was a mistake, and decides to undo that action by executing the `undo` command. The `undo` command will call `Model#undo()`, which will shift the `currentStatePointer` once to the left, pointing it to the previous FitHelper state, and restores the FitHelper to that state.



NOTE If the `currentStatePointer` is at index 0, pointing to the initial FitHelper state, then there are no previous FitHelper states to restore. The `undo` command uses `Model#canundo()` to check if this is the case. If so, it will return an error to the user rather than attempting to perform the undo.

The following sequence diagram shows how the undo operation works:

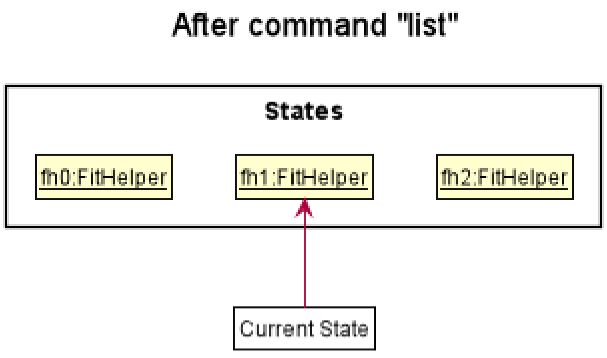


NOTE The lifeline for UndoCommand should end at the destroy marker (X) but due to a limitation of PlantUML, the lifeline reaches the end of diagram.

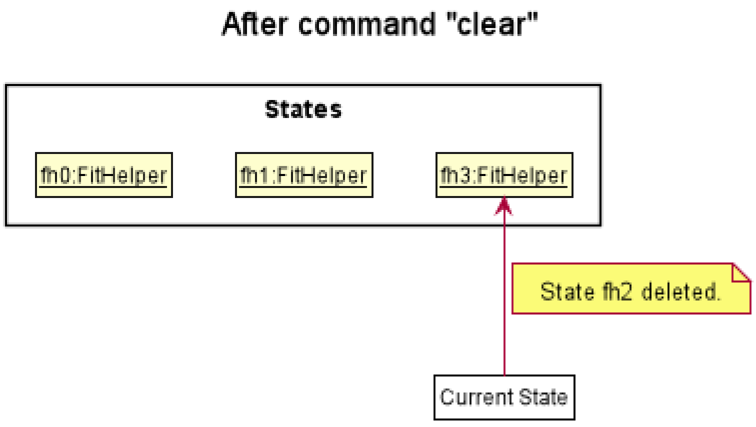
The redo command does the opposite — it calls Model#redo(), which shifts the currentStatePointer once to the right, pointing to the previously undone state, and restores the FitHelper to that state.

NOTE If the currentStatePointer is at index fitHelperStateList.size() - 1, pointing to the latest FitHelper state, then there are no undone FitHelper states to restore. The redo command uses Model#canRedo() to check if this is the case. If so, it will return an error to the user rather than attempting to perform the redo.

Step 5. The user then decides to execute the command list. Commands that do not modify the FitHelper, such as list, will usually not call Model#commit(), Model#undo() or Model#redo(). Thus, the fitHelperStateList remains unchanged.



Step 6. The user executes clear, which calls Model#commit(). Since the currentStatePointer is not pointing at the end of the fitHelperStateList, all FitHelper states after the currentStatePointer will be purged. We designed it this way because it no longer makes sense to redo the add n/David … command. This is the behavior that most modern desktop applications follow.



PROJECT: PowerPointLabs

{Optionally, you may include other projects in your portfolio.}