Wang Yuchen - Project Portfolio

PROJECT: FitHelper

Overview

FitHelper is a desktop diet-and-exercise-recording application made for Users who want to keep fit. It enables users to record their basic profile data, weight records, daily food intake and sports. Rather than just keeping the raw data, **FitHelper** also provided useful analysis and other customized services, such as calendar view and reminders.

The application is mainly written in **Java** and built by a considerable **19k Lines of Code**. The codebase is well-maintained by reasonable amount of tests. A detailed and comprehensive set of guides are also provided for both application developers and users.

Summary of contributions

- **Major enhancement 1**: added the ability to **undo/redo previous commands**
  - What it does: allows the user to undo all previous commands one at a time. Preceding undo commands can be reversed by using the redo command.
  - Justification: This feature improves the product significantly because a user can make mistakes in commands and the app should provide a convenient way to rectify them.
  - Highlights: This enhancement required an in-depth analysis of design alternatives. The implementation too was challenging as it required changes to existing models.
  - Credits: The proposed "undo/redo" features in AddressBook3 Developer Guide has guided the design of this feature in this project.
- **Major enhancement 2**: added the feature of **Today Page**
  - What it does: allows the user to view food/sports entries on today, visualize the distribution of calories in their diet, and obtain feedback for their performance.
  - Justification: This feature improves the product significantly because a user usually need to overview the today schedule and adjust the daily arrangement.
  - Highlights: This enhancement affects existing commands and commands to be added in future. It required an in-depth analysis of design alternatives. The implementation too was challenging as it required changes to existing commands.
- **Major enhancement 3**: **GUI** main structure [#145][#147]
  - Justification: This enhancement builds the FitHelper GUI from scratch, creates the main structure of pages, enables page switches and logic-model-GUI connection.
  - Justification: This feature improves the product significantly because a user usually need to overview the today schedule and adjust the daily arrangement.

    - Highlights: This enhancement affects existing commands and commands to be added in future. It required an in-depth analysis of design alternatives. The implementation too was challenging as it required changes to existing commands.

- **Code contributed**: [RepoSense Link]
- **Other contributions**:
    - Project Refactor
        - Initialize new model for FitHelper [#85]
        - Refactor and re-implement basic commands: `add`, `edit`, `delete`, `clear`, `exit`, `list` [#86]
        - Enhanced Commands: `reminder`, `find` [#100][#177][#191]
    - GUI
        - DashBoard - entry master list [#174][#177]
        - Today Page - daily schedule and performance assessment [#201][#236]
        - Diary Page - add/edit/append/delete diaries [#209][#211]
    - Test
        - Wrote additional tests for existing features to increase coverage to 29% [#319][#320][#321]
    - Documentation
        - UG Draft1 [#90][#91][#95]
        - DG Use Cases [#97][#99]
    - Project management:
        - Participated in releases `v1.3` - `v1.4` (2 releases) on GitHub by creating and updating jar files
    - Managed issues on GitHub and assigned issues to corresponding Milestones and team membersManaged issues on GitHub and assigned issues to corresponding Milestones and team members.
    - Community:
        - Reported bugs and suggestions for other teams (example: PED)
        - Reviewed User Guide and Developer Guide for other teams in class (example:https://github.com/nus-cs2103-AY1920S2/addressbook-level3/pull/3[Inclass Peer Review])
    - Tools:
        - Set up Travis to perform Continuous Integration (CI) for the group fork
        - Integrated a third party Tool (Coverall) to the project (#42)

Contributions to the User Guide

*Sections I contributed to the User Guide include **4.2 Keep Food and Sports Entries**: `add`, `list`, `reminder`, `edit`, `find`, `delete`; **4.3 Keep Diaries**, **4.5 Undo**, **4.6 Redo**, **4.8 Daily Summary**, **4.9 Weekly Summary**, **4.12 Exit**.*
They showcase my ability to write documentation targeting end-users.

[NOTE]

Due to page limit, only selected sections are included below. Please refer to User Guide for more sections.

Editing an entry : `edit`

Edits an existing entry in the fitness log book with the specified values.
Format: `edit x/TYPE i/INDEX [s/STATUS] [n/NAME] [t/TIME] [l/LOCATION] [c/CALORIE][dr/DURATION]…`

- At least one of the optional fields must be provided.
- If the user is currently in other pages, a successful execution of `edit` will switch the page to the dashboard.
- 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 `find` command and then refer to the resulting list's indices to `edit` 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). He can `edit` any entries only when he turns to home page after the `find`.
- The index **must be a positive integer** 1, 2, 3, …

Examples: (after switching to Home Page) * `edit x/s i/1 t/2020-04-05-16:00 l/PGP gym`
Edits the time and email location of the 1st entry to be `Friday 4pm` and `PGP gym` respectively.

Mark an entry as done

Users can mark an entry as done, either a meal or sports, where the calories intake and consumption will be taken in to consideration. Format: `edit x/TYPE i/INDEX s/Done`

Mark an entry as undone

Similar to the previous command, marking an entry as undone edits the `s/` field and modify it as `Undone`. Format: `edit x/TYPE i/INDEX s/Undone`

Find a diary: `findDiary`

- `findDiary [d/DATE] [k/ONE OR MORE KEYWORDS]`

finds diaries either on the specified DATE or contains the specified KEYWORDS.
Both DATE and KEYWORDS fields are optional. If neither appear, FitHelper will display all diaries.
If there is no diary under the specified DATE, this command will be discarded with the reminder of DIARY_NOT_FOUND.

- The field DATE has higher priority than KEYWORDS in the search. i.e. If the DATE field is non-empty, regardless of the presence and the content of the KEYWORDS field, the diary under that date will be displayed. If the specified DATE contains no previous diary logs, no diary will be listed.
- If the field DATE is left empty, only the KEYWORDS field is considered in the search, similar to the case of `find` command for food/sport entries.
- The keyword searching rules are the same as in Section 4.2.5 Locating entries by name : **find**.

Examples:

- `findDiary k/running`
  Displays diaries with their content containing the keyword `running`, ignoring the letter capitalization.
- `find d/2020-03-31 k/cake`
  Returns the diary on the date of `2020-03-31` regardless of the KEYWORDS field.

Undo : undo

`undo` revokes the last undoable command.
This command back-roll FitHelper to the previous status before the last undoable command was executed.

NOTE	Undoable commands include: `add`, `edit`, `delete`, `clear`, `addDiary`, `appendDiary`, `editDiary`, `deleteDiary`, `clearDiary`. Other commands are not affected by `undo` command. The same applies for `redo`.
NOTE	After executing `undo` or `redo`, FitHelper switches to `Home Page` (`DashBoard`).

Examples:

- `undo` (after `addDiary d/2020-03-31 dc/I am happy.`)
  This `undo` commands remove the added diary log from FitHelper.

Today Page: today

Today page serves to be a summary for the daily arrangements.
It shows the daily schedule for the user. Users can see the entries for the day, a recommended lunch place, and their performances. They can also see their diary for the day as well as the rewarding point.

Format: `today`

- **Daily Food/Sports Entries**
  The lists of food and sports entries on "today" are displayed in two list view, with indices in chronological order specific for today.
- **Plan Counter**
  The 4 counters keep track of the number of done/undone food/sports plans on today.
- **Calorie Report**
  Calorie report contains the data of daily calorie intake/consumption from done food/sports entries correspondingly. The food calorie pie chart consists of all food entries on today, regardless of the status. The labels are the corresponding indices of the food entries in `Today's Food` list. From the pie chart, the user can view the component of calorie intake of each food entry, so he/she can adjust the diet plan.

WARNING	In cases where some food entries contribute to the great majority of the total food intake, the pie chart only displays labels for food entries that contain relatively high calorie values.

- **Task Completion**
  The user's daily task completion is shown in percentage (round to integer).
- **FitHelper Feedback**
  Based on the user's intake food calorie and sport task completion, FitHelper provides suggestions and reminders in the `FitHelper Feedback` area.

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.

[NOTE]

Due to page limit, the activity diagram of UndoRedo feature is not shown. Refer to Developer Guide for the diagram

Today Feature

Implementation

FitHelper's entries have a `Time` attribute including a `Date` and a specific `Time` in the format of `yyyy-mm-dd HH:mm`. Today feature allows the user to view entries with the `Date` of today, i.e. shows only entries in today. It fetches the `todayFoodEntries` and `todaySportsEntries` stored in FitHelper storage. Because the display of these two lists have the same logic, they are illustrated as `todayEntries` in this section, as a whole.

- In `FitHelper`, the `UniqueEntryList<Entry> todayEntries` contains all entries on today. The list is updated whenever changes are made to the general UniqueEntryList<Entry> entries which

contains entries of all dates.

- A `FilteredList<Entry> filteredTodayEntries` is stored in the `ModelManager`. `filteredTodayEntries` in the ModelManager is initialized with this `UniqueEntryList<Entry>` by converting it to an `ObservableList<Entry>`.
- `Today Page` takes in `todayEntries` as a parameter when it is initialized. The list is always displayed on the GUI page as a ListView.
- When the `today` command is executed, FitHelper switches to `Today Page` where the entries on today can be seen.
- Other features implemented in `Today Page`, like `daily calorie calculation`, `daily task completion`, and `dialy performance assessment`, all depend on the data carried by the passed in list.

An example usage scenario and how the `today` mechanism behaves at each step is shown below.

Step 1. The user launches the application for the first time. `UniqueEntryList` will be initialized with a list of default entries in FitHelper, which contains a few entries with various dates. `UniqueTodayEntryList` will be initialized concurrently by filtering out entries on today.

Step 2. `MainWindow` fetches `ObservableList<Entry> todayEntries`. `Today Page` is initialized in `MainWindow` with the `ObservableList<Entry> todayEntries` passed from the model.

Step 3. The user inputs `today` to view all today entries. `UI` passes the input to `Logic`.

Step 4. Logic passes the user input to `FitHelperParser`. `FitHelperParser` identifies that this is a `TodayCommand` through the word "today". It then creates a `TodayCommandParser` to parse the it into a `TodayCommand` and return.

Step 5. `Logic` gets the `TodayCommand` and execute it. This execution then returns a `CommandResult` to `UI`, containing the success message and a specified displayed page of `Today Page`.

Step 5. `UI` displays the response in the `CommandResult`. `UI` also switches FitHelper to `Today Page`, where the continuously updated `todayEntryList` is displayed, since `UI` is constantly listening for the change in `Model`.

The Sequence Diagram below shows how the components interact with each other for the above mentioned scenario.

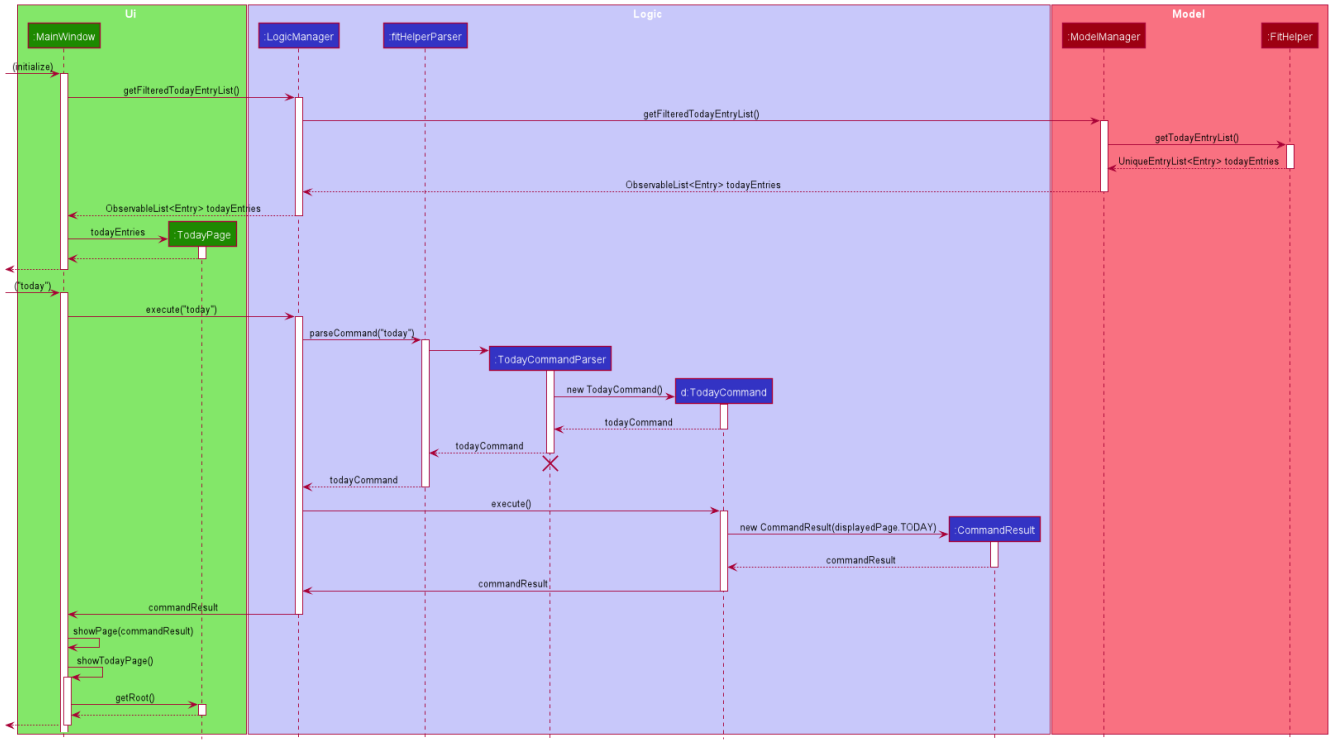


Figure 1. Sequence Diagram for Today Feature

Diary Feature

Implementation

FitHelper also allows the user to keep a diary with a `Date` in the format of `yyyy-mm-dd` and the `content` with no more than 200 characters. The diaries are represented as model `Diary` with the attributes of `DiaryId`, `DiaryDate`, and `Content`. This feature allows the user to view their diaries. It fetches the `filteredDiaryList` stored in FitHelper storage.

The diary feature is facilitated by FilteredList which wraps a ObservableList and filters using the provided Predicate. A `FilteredList<Diary> filteredDiaries` is stored in the `ModelManager`. In `FitHelper`, there is an `ObservableList<Diary> diaries` which contains all diaries, regardless of its `DiaryDate`. `filteredDiaries` in the ModelManager is initialized with this ObservableList.

Since a FilteredList needs a Predicate, which matches the elements in the source list that should be visible, the filter mechanism implements the following operation to support filtering:

- `Model#updateFilteredDiaryList(Predicate<Diary> predicate)` — Sets the value of the property Predicate in the `filteredDiaries`.
    - The predicate is declared statically in the `Model` interface, namely `PREDICATE_SHOW_ALL_DIARIES`. In particular `PREDICATE_SHOW_ALL_DIARIES` is as follows

```
Predicate<Diary> PREDICATE_SHOW_ALL_DIARIES = unused -> true;
```

- The `DiaryCommand` will call this method to change the visibility of diaries with different status by passing in the corresponding predicate.

An example usage scenario and how the diary mechanism behaves at each step is shown below.

Step 1. The user launches the application for the first time. `UniqueDiaryList` contains no default diaries before the user adds any.

Step 2. The user inputs `diary` to list all diaries. `UI` passes the input to `Logic`. Logic then uses a few `Parser` classes to extract layers of information out as seen from steps 3 to 5.

Step 3. Logic passes the user input to `FitHelperParser`. `FitHelperParser` identifies that this is a `DiaryCommand` through the word "diary". It then creates a `DiaryCommandParser` to parse the it into a `DiaryCommand` and return.

Step 4. `Logic` finally gets the `DiaryCommand` and execute it. The execution firstly calls `Model#updateFilteredDiaryList(Predicate<Diary> predicate)` to update the Predicate in `filteredDiaries` in `Model`. This execution then returns a `CommandResult` to `UI`, containing the response to the user.

Step 5. `UI` displays the response in the `CommandResult`. In addition, UI will change to display diaries after model updates `filteredDiaries`, since `UI` is constantly listening for the change in `Model`.

The Sequence Diagram below shows how the components interact with each other for the above mentioned scenario.

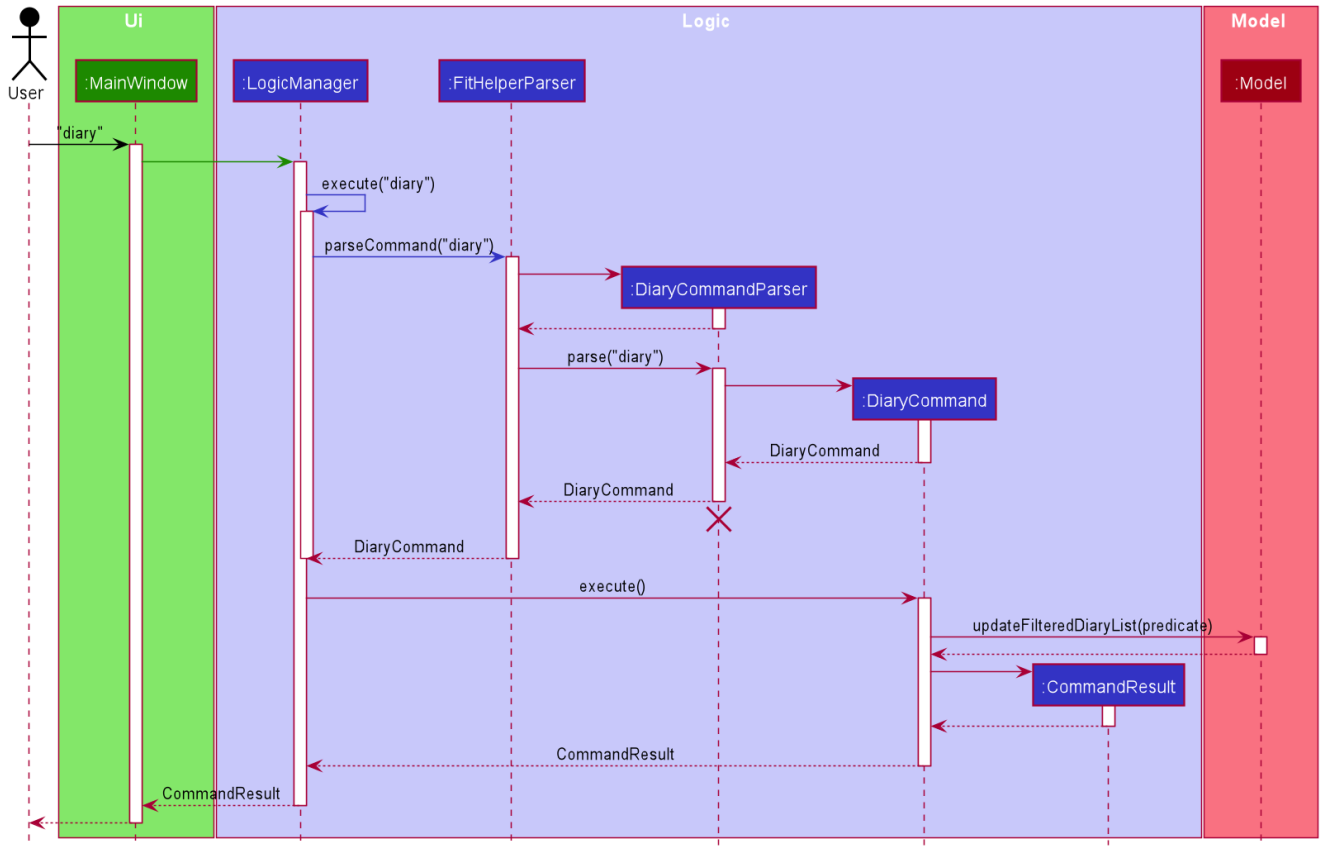


Figure 2. Sequence Diagram for Diary Feature

Undo/Redo feature

Implementation

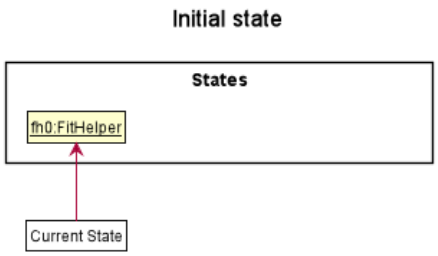
The undo/redo mechanism is facilitated by `VersionedFitHelper`. It extends `FitHelper` with an undo/redo history, stored internally as an `fitHelperStateList` and `currentStatePointer`. Additionally, it implements the following operations:

- `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.

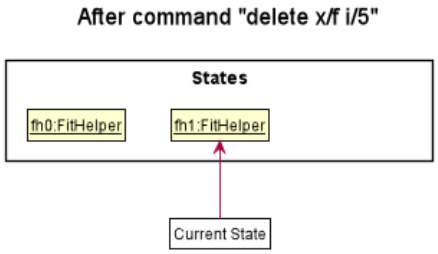
These operations are exposed in the `Model` interface as `Model#commit()`, `Model#undo()` and `Model#redo()` respectively.

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

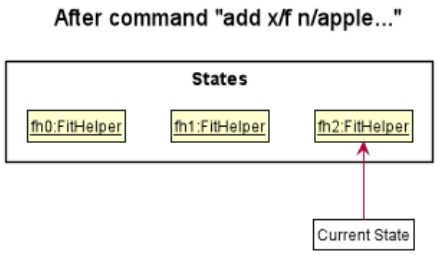
Step 1. The user launches the application for the first time. The `VersionedFitHelper` will be initialized with the initial FitHelper state, and the `currentStatePointer` pointing to that single FitHelper state.



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.

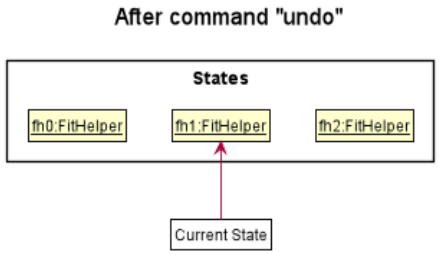


Step 3. The user executes `add x/f n/apple …` to add a new food entry. The `add` command also calls `Model#commit()`, causing another modified FitHelper state to be saved into the `fitHelperStateList`.



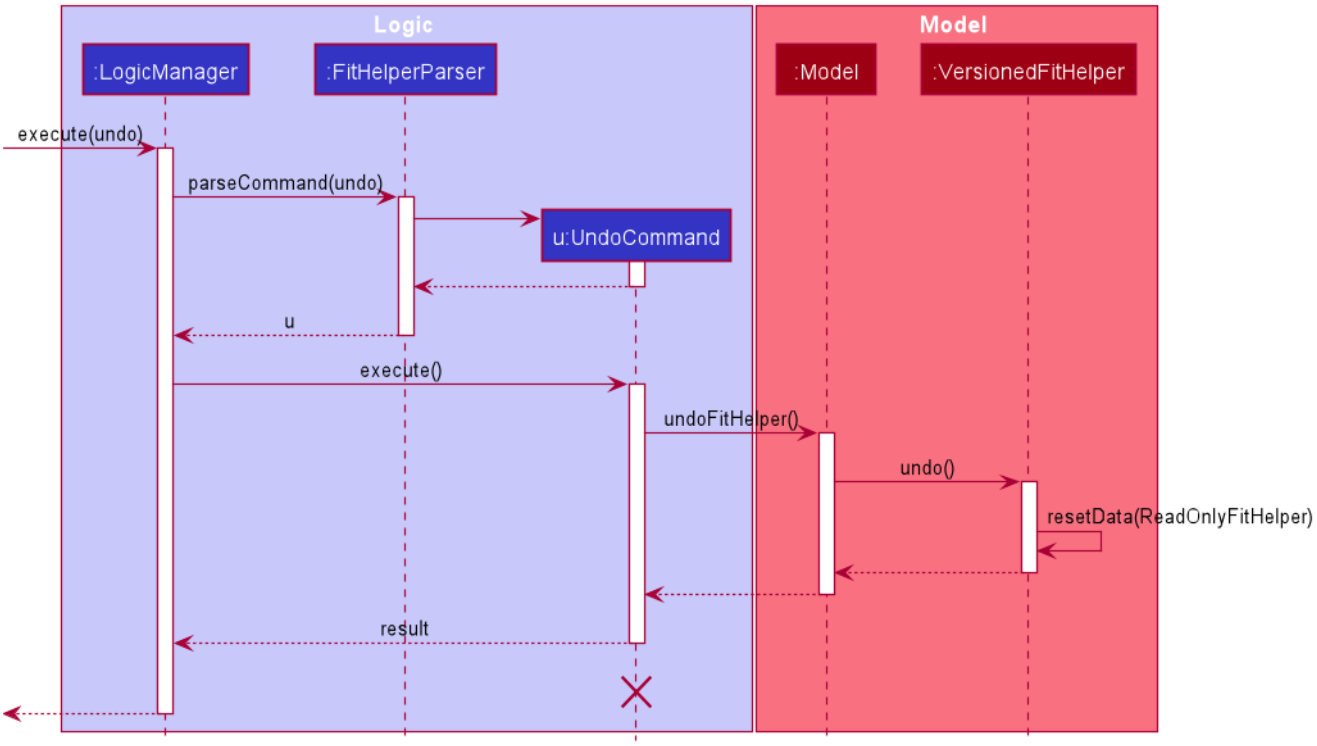
NOTE If a command fails its execution, it will not call `Model#commit()`, so the FitHelper state will not be saved into the `fitHelperStateList`.

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:



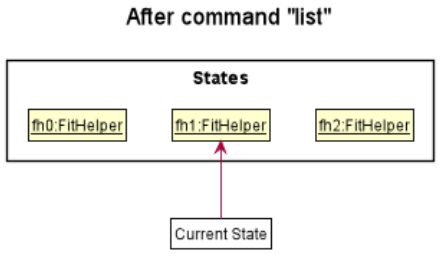
NOTE The lifeline for `UndoCommand` should end at the destroy marker (X) but due to a limitation of PlantUML, the lifeline reaches the end of diagram.

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.

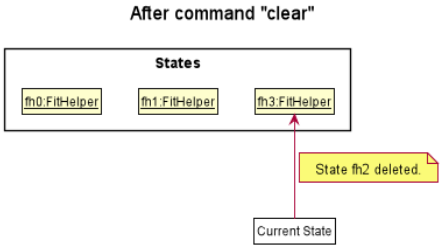
NOTE

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.



The following activity diagram summarizes what happens when a user executes a new command: