FitHelper - Developer Guide

By: `AY1920S2-CS2103-T09-4` Since: `Feb 2020` Licence: `MIT`

1. Setting up

Refer to the guide here.

2. Design

2.1. Architecture

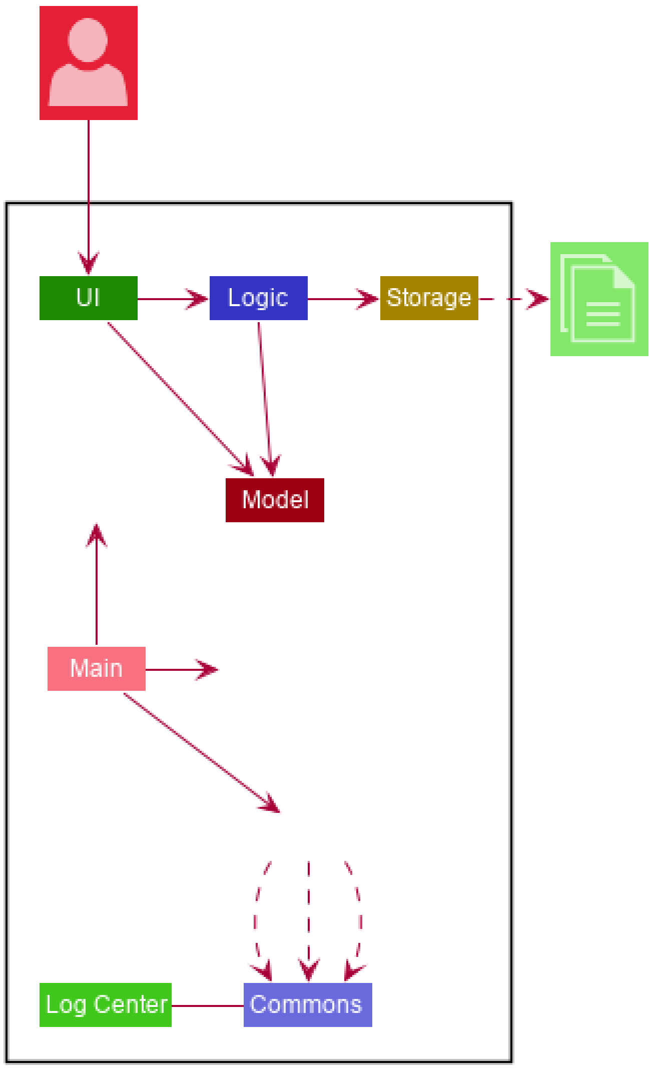


Figure 1. Architecture Diagram

The ***Architecture Diagram*** given above explains the high-level design of the App. Given below is a quick overview of each component.

TIP The `.puml` files used to create diagrams in this document can be found in the diagrams folder. Refer to the Using PlantUML guide to learn how to create and edit diagrams.

`Main` has two classes called `Main` and `MainApp`. It is responsible for,

- At app launch: Initializes the components in the correct sequence, and connects them up with each other.
- At shut down: Shuts down the components and invokes cleanup method where necessary.

`Commons` represents a collection of classes used by multiple other components. The following class plays an important role at the architecture level:

- `LogsCenter` : Used by many classes to write log messages to the App’s log file.

The rest of the App consists of four components.

- `UI`: The UI of the App.
- `Logic`: The command executor.
- `Model`: Holds the data of the App in-memory.
- `Storage`: Reads data from, and writes data to, the hard disk.

Each of the four components

- Defines its *API* in an `interface` with the same name as the Component.
- Exposes its functionality using a `{Component Name}Manager` class.

For example, the `Logic` component (see the class diagram given below) defines it’s API in the `Logic.java` interface and exposes its functionality using the `LogicManager.java` class.

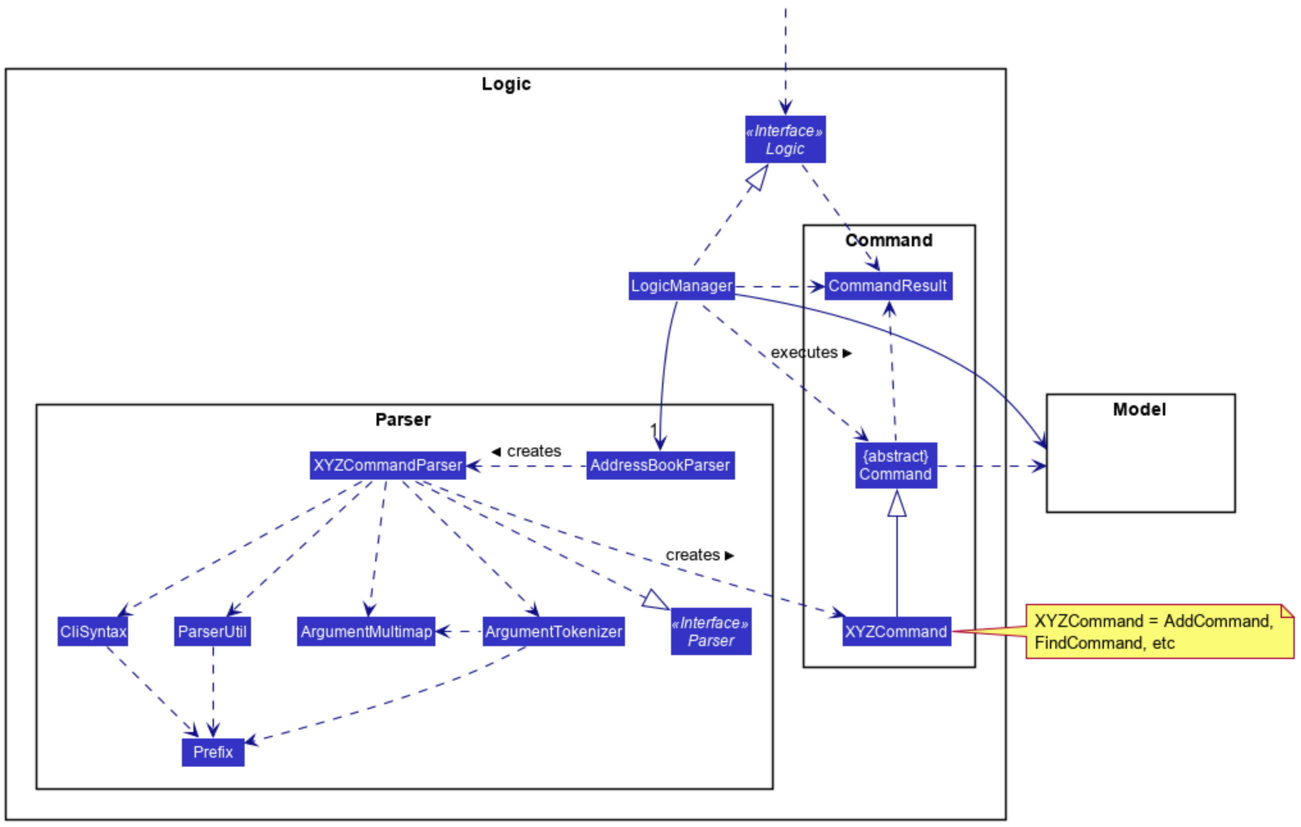


Figure 2. Class Diagram of the Logic Component

How the architecture components interact with each other

The *Sequence Diagram* below shows how the components interact with each other for the scenario where the user issues the command `delete 1`.

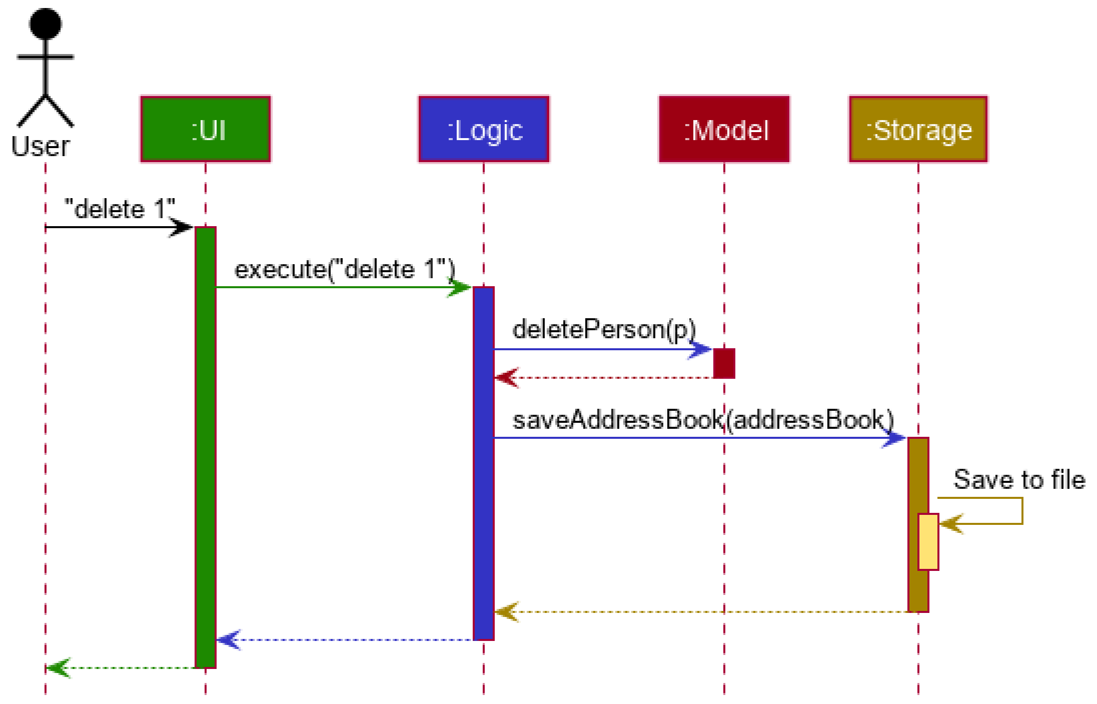


Figure 3. Component interactions for `delete 1` command

The sections below give more details of each component.

2.2. UI component

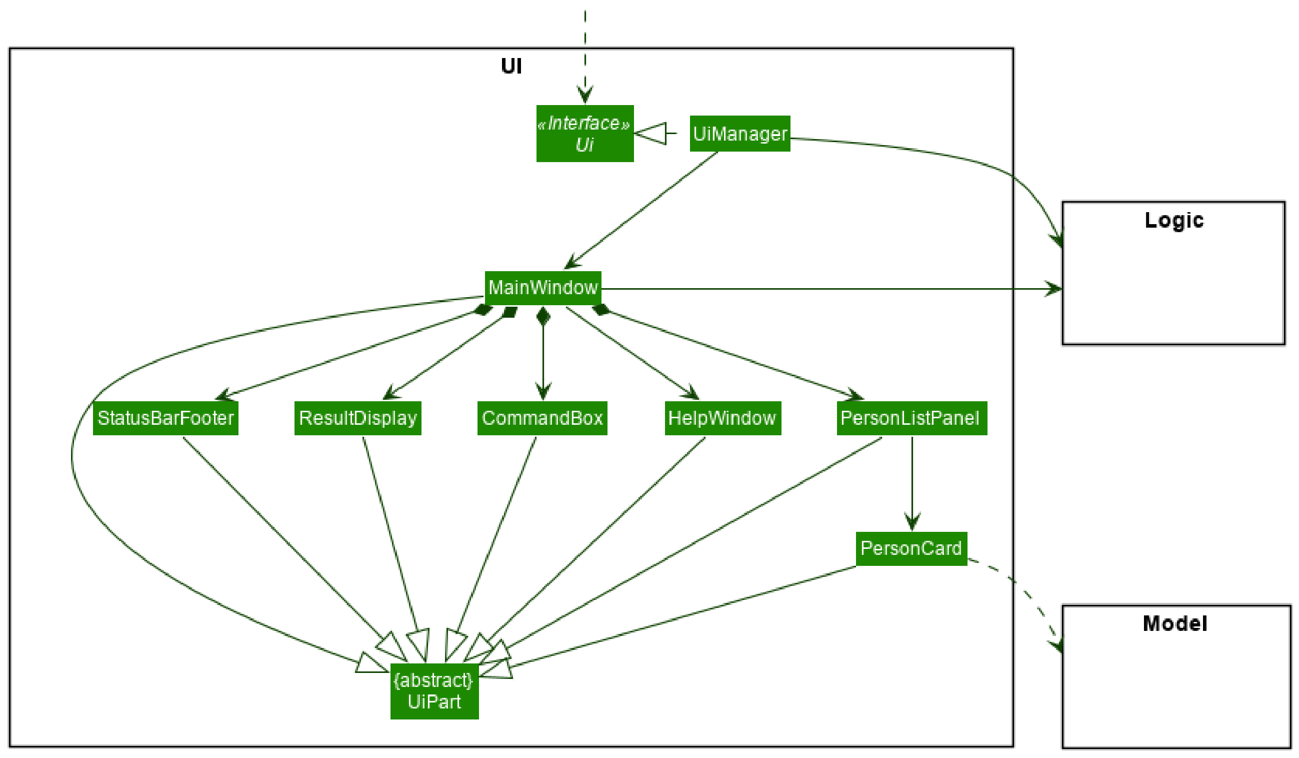


Figure 4. Structure of the UI Component

API : `Ui.java`

The UI consists of a `MainWindow` that is made up of parts e.g.`CommandBox`, `ResultDisplay`, `PersonListPanel`, `StatusBarFooter` etc. All these, including the `MainWindow`, inherit from the abstract `UiPart` class.

The `UI` component uses JavaFx UI framework. The layout of these UI parts are defined in matching `.fxml` files that are in the `src/main/resources/view` folder. For example, the layout of the `MainWindow` is specified in `MainWindow.fxml`

The `UI` component,

- Executes user commands using the `Logic` component.
- Listens for changes to `Model` data so that the UI can be updated with the modified data.

2.3. Logic component

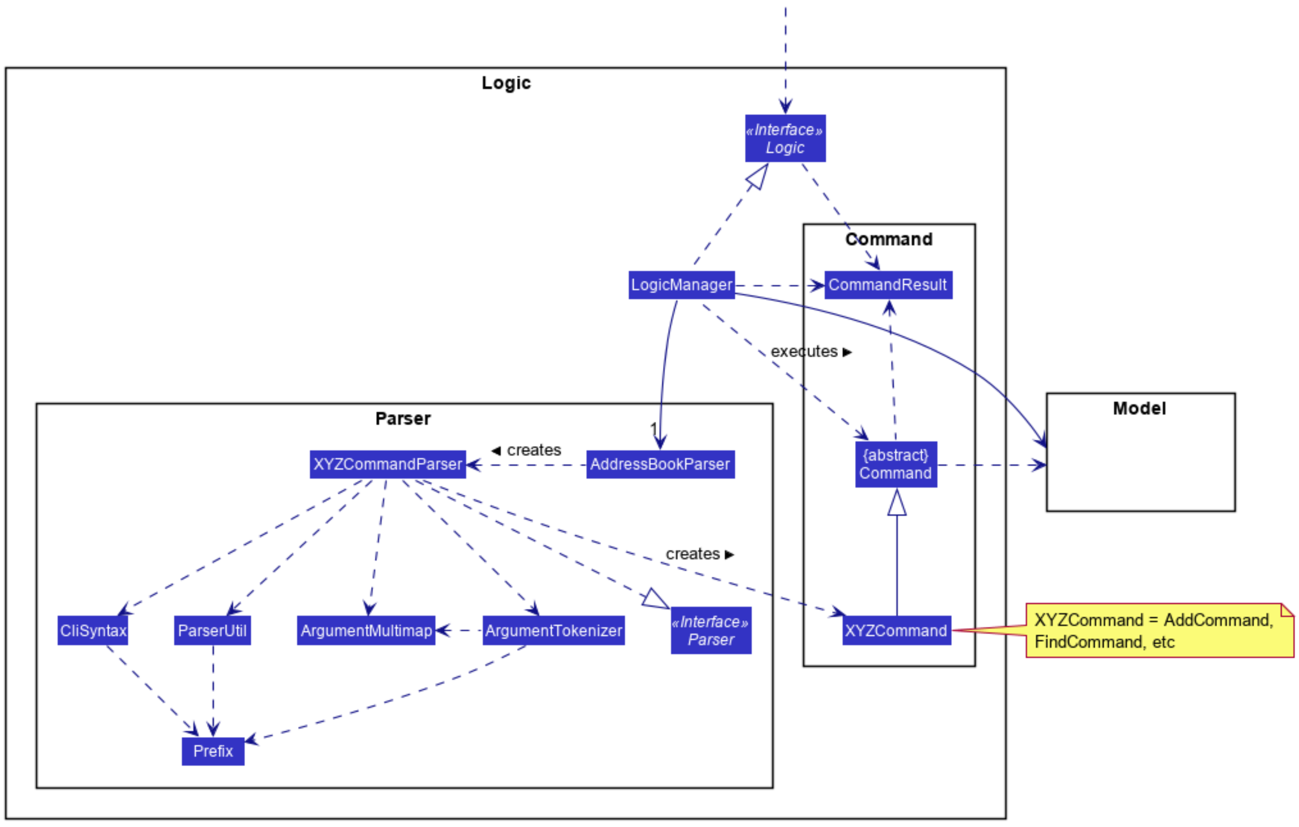


Figure 5. Structure of the Logic Component

API : `Logic.java`

1. `Logic` uses the `FitHelperParser` class to parse the user command.
2. This results in a `Command` object which is executed by the `LogicManager`.
3. The command execution can affect the `Model` (e.g. adding a person).
4. The result of the command execution is encapsulated as a `CommandResult` object which is passed back to the `Ui`.
5. In addition, the `CommandResult` object can also instruct the `Ui` to perform certain actions, such as displaying help to the user.

Given below is the Sequence Diagram for interactions within the `Logic` component for the `execute("delete 1")` API call.

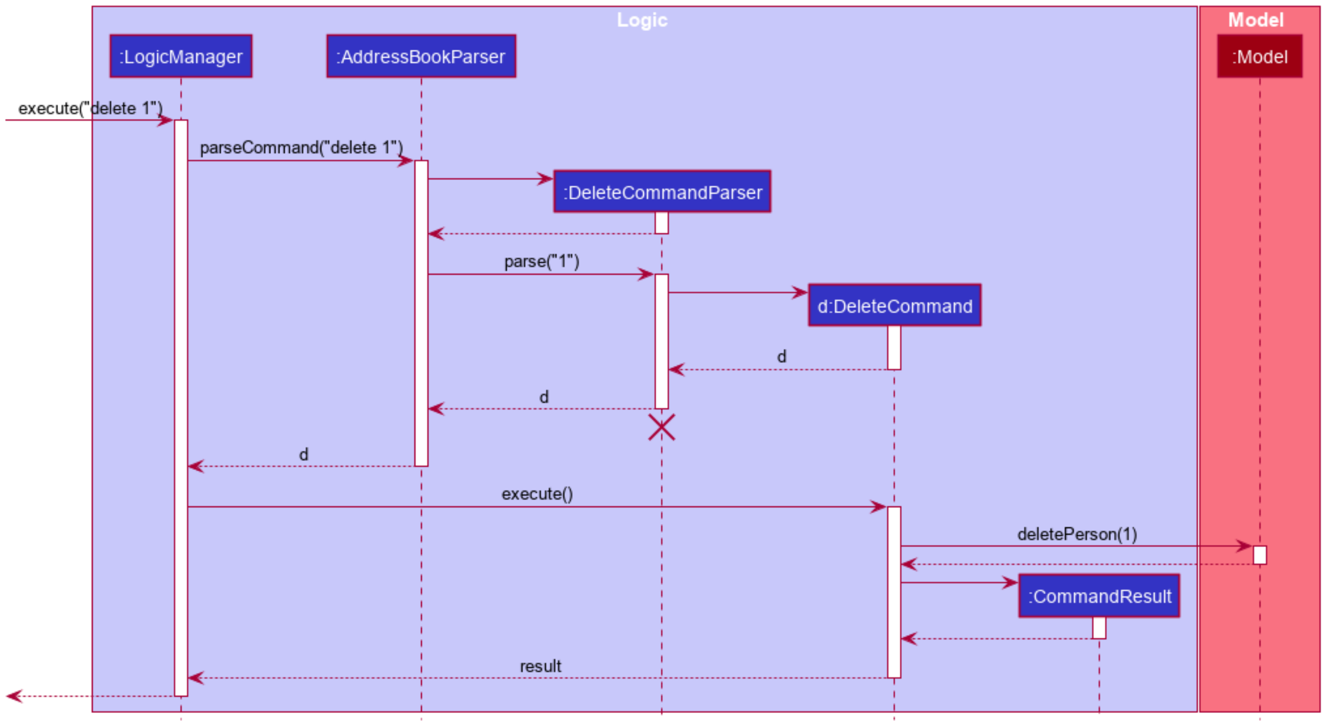


Figure 6. Interactions Inside the Logic Component for the `delete 1` *Command*

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

2.4. Model component

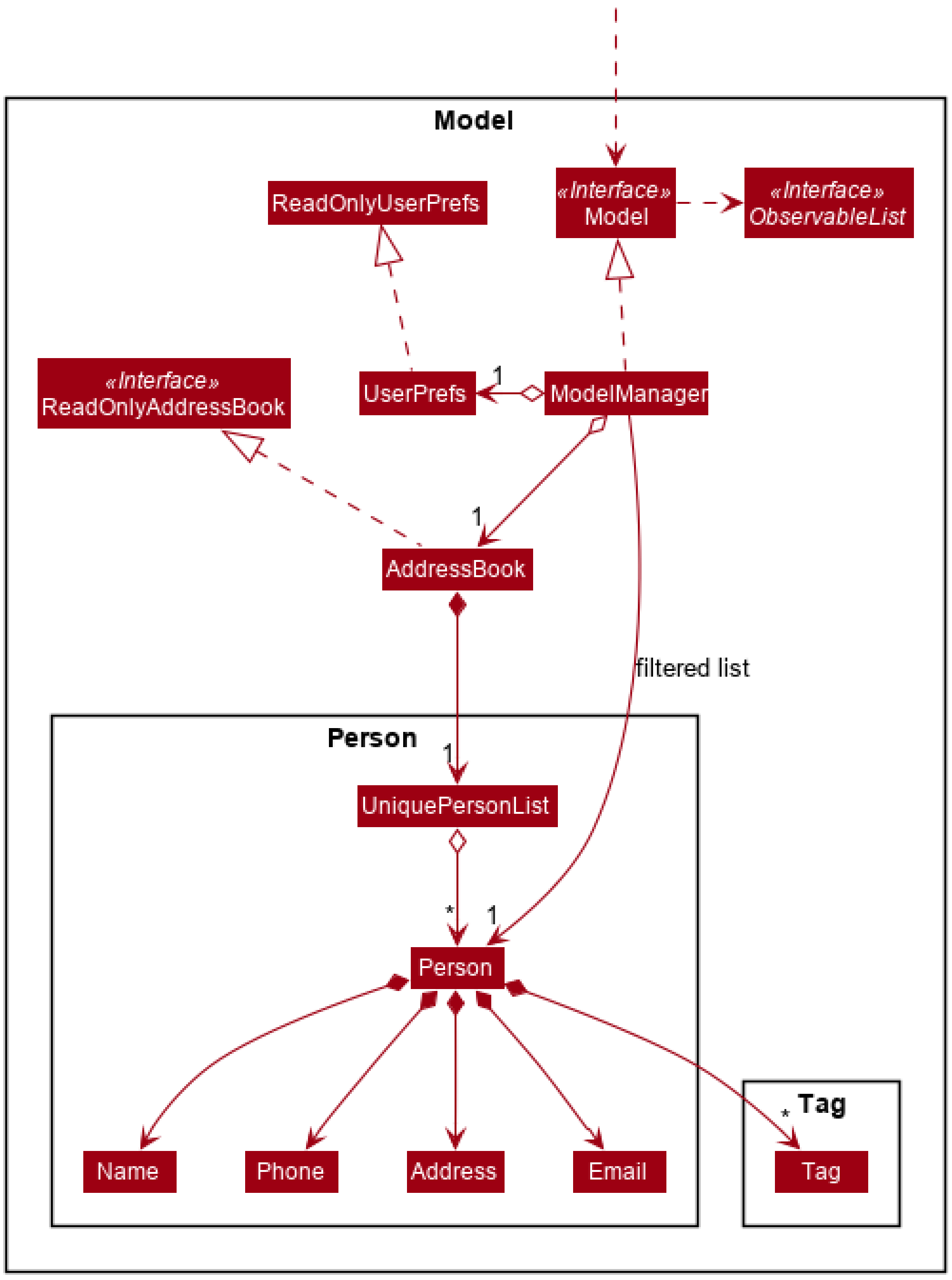


Figure 7. Structure of the Model Component

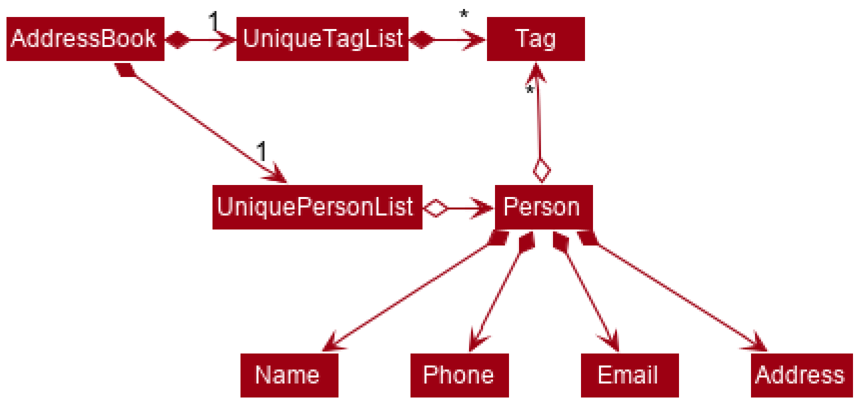
API : `Model.java`

The `Model`,

- stores a `UserPref` object that represents the user’s preferences.
- stores the Address Book data.
- exposes an unmodifiable `ObservableList<Person>` that can be 'observed' e.g. the UI can be bound to this list so that the UI automatically updates when the data in the list change.
- does not depend on any of the other three components.

NOTE

As a more OOP model, we can store a `Tag` list in `Address Book`, which `Person` can reference. This would allow `Address Book` to only require one `Tag` object per unique `Tag`, instead of each `Person` needing their own `Tag` object. An example of how such a model may look like is given below.



2.5. Storage component

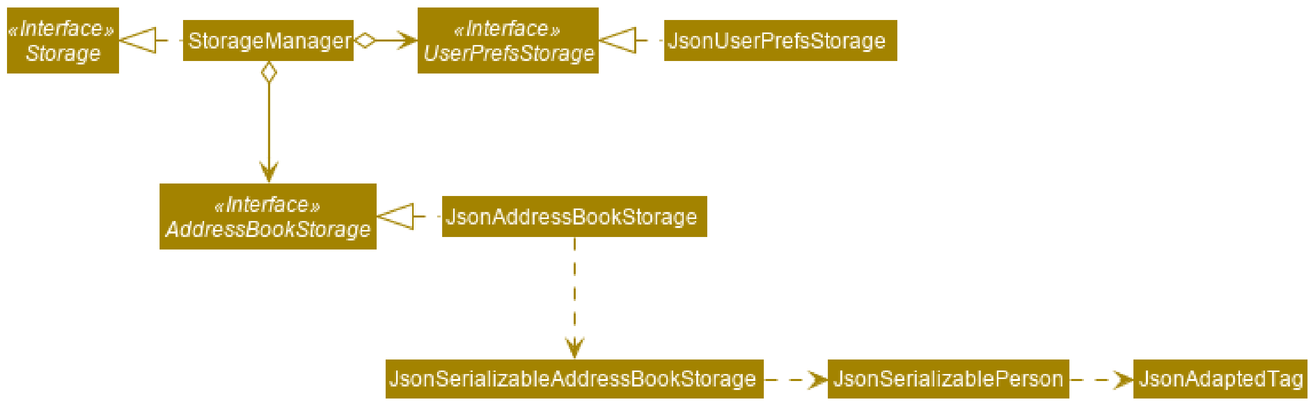


Figure 8. Structure of the Storage Component

API : `Storage.java`

The `Storage` component,

- can save `UserPref` objects in json format and read it back.
- can save the Address Book data in json format and read it back.

2.6. Common classes

Classes used by multiple components are in the `seedu.addressbook.commons` package.

3. Implementation

This section describes some noteworthy details on how certain features are implemented.

3.1. [Proposed] Undo/Redo feature

3.1.1. Proposed Implementation

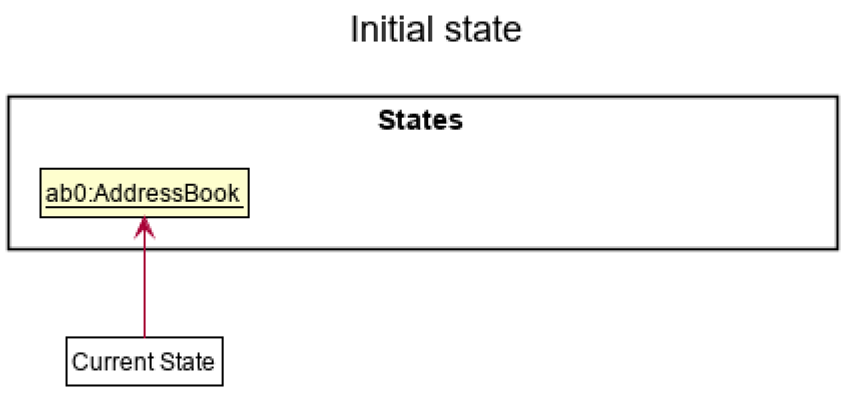
The undo/redo mechanism is facilitated by `VersionedAddressBook`. It extends `AddressBook` with an undo/redo history, stored internally as an `addressBookStateList` and `currentStatePointer`. Additionally, it implements the following operations:

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

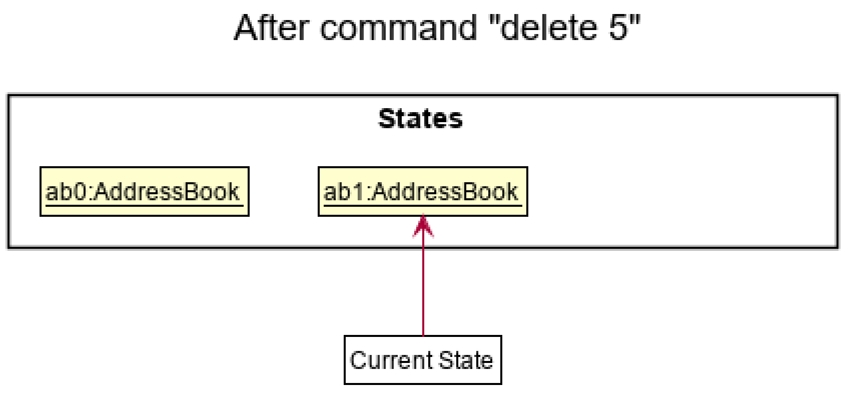
These operations are exposed in the `Model` interface as `Model#commitAddressBook()`, `Model#undoAddressBook()` and `Model#redoAddressBook()` respectively.

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

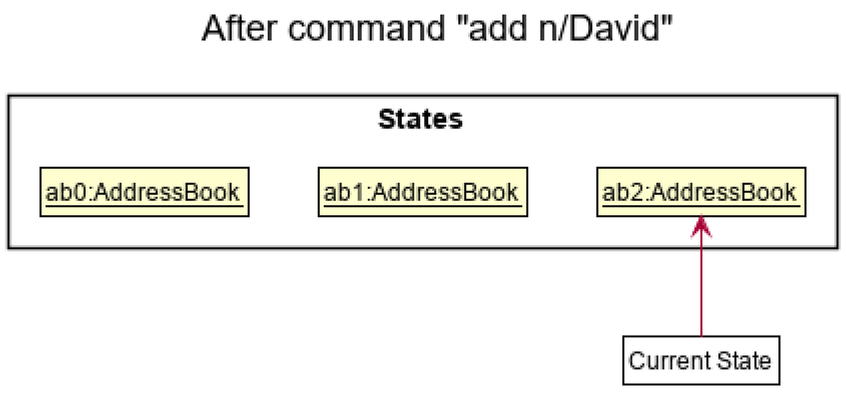
Step 1. The user launches the application for the first time. The `VersionedAddressBook` will be initialized with the initial address book state, and the `currentStatePointer` pointing to that single address book state.



Step 2. The user executes `delete 5` command to delete the 5th person in the address book. The `delete` command calls `Model#commitAddressBook()`, causing the modified state of the address book after the `delete 5` command executes to be saved in the `addressBookStateList`, and the `currentStatePointer` is shifted to the newly inserted address book state.

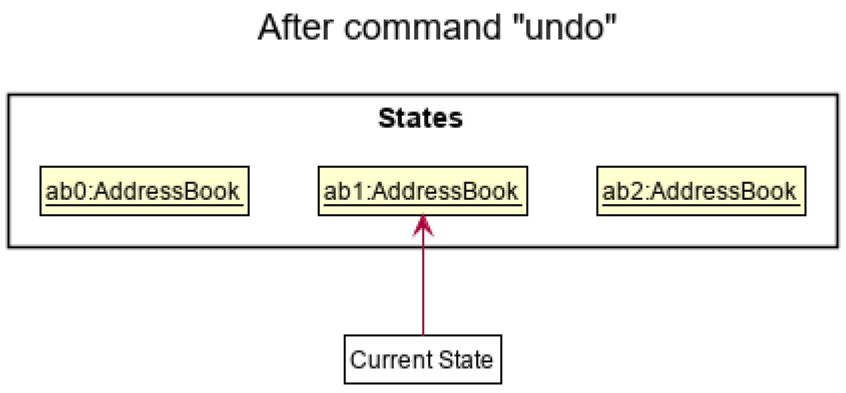


Step 3. The user executes `add n/David …` to add a new person. The `add` command also calls `Model#commitAddressBook()`, causing another modified address book state to be saved into the `addressBookStateList`.



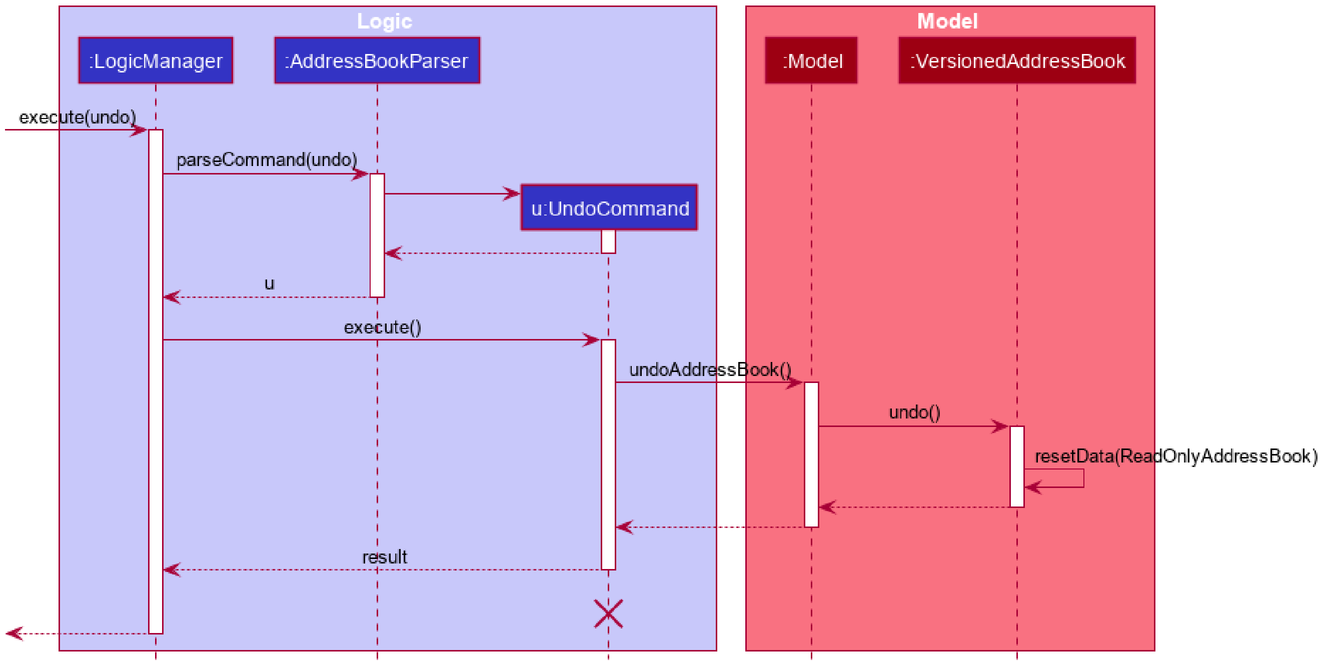
NOTE If a command fails its execution, it will not call `Model#commitAddressBook()`, so the address book state will not be saved into the `addressBookStateList`.

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



NOTE If the `currentStatePointer` is at index 0, pointing to the initial address book state, then there are no previous address book states to restore. The `undo` command uses `Model#canUndoAddressBook()` 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:

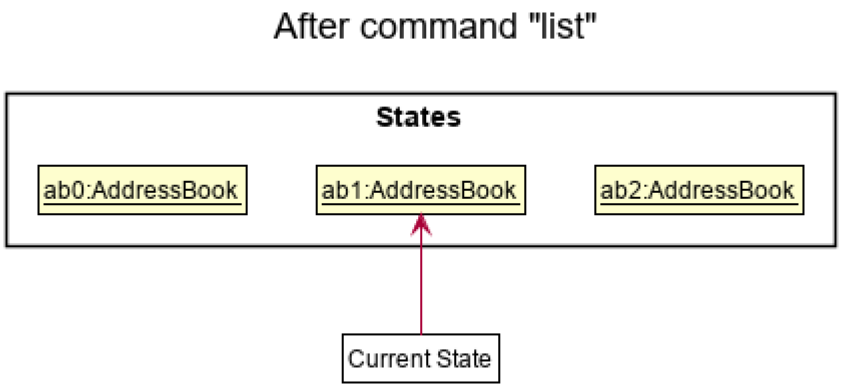


NOTE The lifeline for `UndoCommand` should end at the destroy marker (X) but due to a limitation of PlantUML, the lifeline reaches the end of diagram.

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

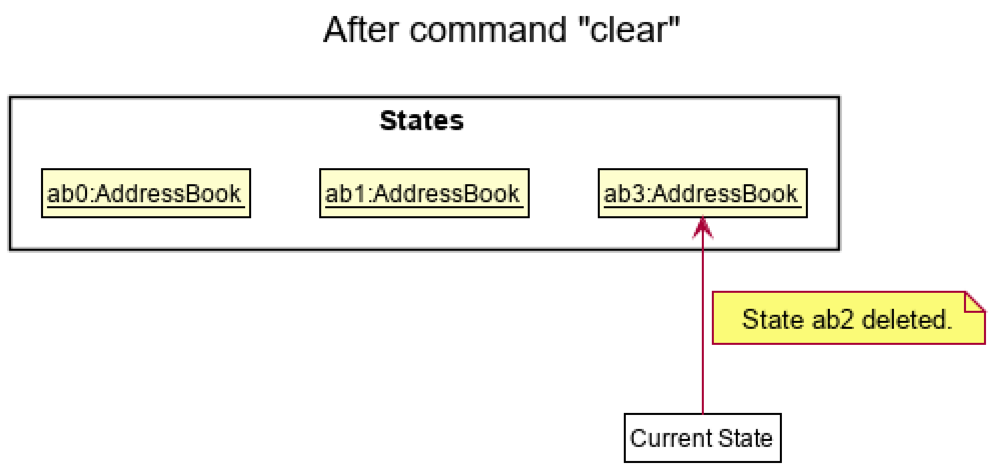
NOTE If the `currentStatePointer` is at index `addressBookStateList.size() - 1`, pointing to the latest address book state, then there are no undone address book states to restore. The `redo` command uses `Model#canRedoAddressBook()` to check if this is the case. If so, it will return an error to the 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 address book, such as `list`, will usually not call `Model#commitAddressBook()`, `Model#undoAddressBook()` or `Model#redoAddressBook()`. Thus, the `addressBookStateList` remains unchanged.

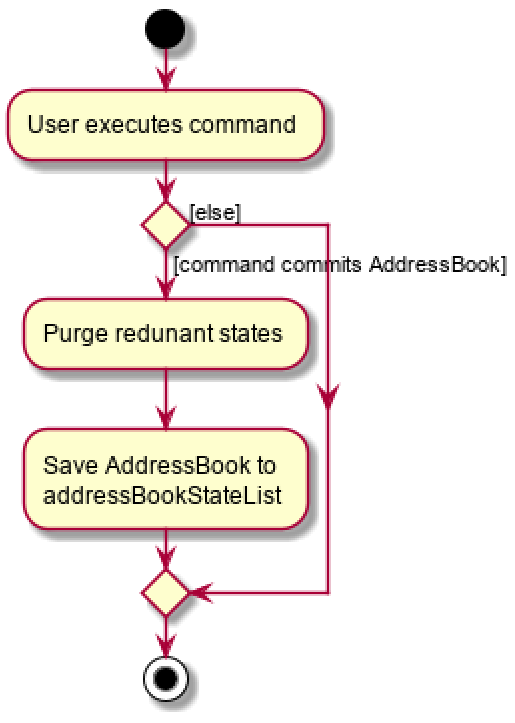


Step 6. The user executes `clear`, which calls `Model#commitAddressBook()`. Since the `currentStatePointer` is not pointing at the end of the `addressBookStateList`, all address book states after the `currentStatePointer` will be purged. We designed it this way because it no longer makes

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:


User executes command
[else]
[command commits AddressBook]
Purge redunant states
Save AddressBook to addressBookStateList

3.1.2. Design Considerations

Aspect: How undo & redo executes

- **Alternative 1 (current choice):** Saves the entire address book.
    - Pros: Easy to implement.
    - Cons: May have performance issues in terms of memory usage.
- **Alternative 2:** Individual command knows how to undo/redo by itself.
    - Pros: Will use less memory (e.g. for `delete`, just save the person being deleted).
    - Cons: We must ensure that the implementation of each individual command are correct.

Aspect: Data structure to support the undo/redo commands

- **Alternative 1 (current choice):** Use a list to store the history of address book states.
  - Pros: Easy for new Computer Science student undergraduates to understand, who are likely to be the new incoming developers of our project.
  - Cons: Logic is duplicated twice. For example, when a new command is executed, we must remember to update both `HistoryManager` and `VersionedAddressBook`.
- **Alternative 2:** Use `HistoryManager` for undo/redo
  - Pros: We do not need to maintain a separate list, and just reuse what is already in the codebase.
  - Cons: Requires dealing with commands that have already been undone: We must remember to skip these commands. Violates Single Responsibility Principle and Separation of Concerns as `HistoryManager` now needs to do two different things.

3.2. [Proposed] Data Encryption

{Explain here how the data encryption feature will be implemented}

3.3. Logging

We are using `java.util.logging` package for logging. The `LogsCenter` class is used to manage the logging levels and logging destinations.

- The logging level can be controlled using the `logLevel` setting in the configuration file (See Section 3.4, “Configuration”)
- The `Logger` for a class can be obtained using `LogsCenter.getLogger(Class)` which will log messages according to the specified logging level
- Currently log messages are output through: `Console` and to a `.log` file.

Logging Levels

- `SEVERE` : Critical problem detected which may possibly cause the termination of the application
- `WARNING` : Can continue, but with caution
- `INFO` : Information showing the noteworthy actions by the App
- `FINE` : Details that is not usually noteworthy but may be useful in debugging e.g. print the actual list instead of just its size

3.4. Configuration

Certain properties of the application can be controlled (e.g user prefs file location, logging level) through the configuration file (default: `config.json`).

3.5. Check calorie intake/consumption of some common food/sports

4. Documentation

Refer to the guide here.

5. Testing

Refer to the guide here.

6. Dev Ops

Refer to the guide here.

Appendix A: Product Scope

Target user profile:

- has a need to control weight, therefore need to record daily food intake and sports
- prefer desktop apps over other types
- can type fast
- prefers typing over mouse input
- is reasonably comfortable using CLI apps

Value proposition: achieve fitness control faster than a typical mouse/GUI driven app

Appendix B: User Stories

Priorities: High (must have) - * * *, Medium (nice to have) - * *, Low (unlikely to have) - *

Priority	As a …	I want to …	So that I can…
* * *	new user	record my basic information such as name and gender	have a more complete profile

Priority	As a …	I want to …	So that I can…
`* * *`	user who is concerned about body shape	record and update my current height and weight	have a clear view of my current body condition
`* * *`	user who wants to lose weight	set my target weight	have a clear target to work towards
`* *`	user who wants keep fit	acknowledge my weight change trend according to time	keep track of my weight change easily
`* *`	user who wants to lose weight	compare between my current weight and target weight	know the gap clearly
`* *`	user	update my basic information such as address and name if necessary	have an updated profile at any time
`* *`	user	view pending tasks and status of daily calories goals in a calendar	have cleaner display of data

{More to be added}

Appendix C: Use Cases

(For all use cases below, the **System** is the `FitHelper` and the **Actor** is the `user`, unless specified otherwise)

Use case: UC01 - Add an Entry

MSS

1. User adds an entry specifying a meal or a sport with name, time, location, and calorie.
2. FitHelper stores the entry to the specific date file.
3. FitHelper display successful record and the entry status.

   Use case ends.

Extensions

   1a. User input incomplete values.

      1a1. FitHelper shows an error message.

      Use case ends.

   1b. The input time has clashes with previous entries.

      1b1. FitHelper shows an error message.

      Use case ends.

Use case: UC02 - Edits an Entry

MSS

1. User edits an entry specifying a meal or a sport with name, time, location, and calorie.
2. FitHelper modifies the entry to the specific date file.
3. FitHelper display successful record and the entry status.

   Use case ends.

Extensions

   1a. User input repeated values that are already stored in the entry.

      1a1. FitHelper ignores the edit command.

      Use case ends.

Use case: UC03 - Deletes an Entry

MSS

1. User deletes an entry by using the`delete` command.
2. FitHelper deletes the corresponding entry in the list and in the file.
3. FitHelper display the entry status and the successfully-delete message.

   Use case ends.

Extensions

1a. The `INDEX` specified by the user does not exist.

1a1. FitHelper shows an error message.

Use case ends.

Use case: UC04 - Display the List

MSS

1. User requests to display the list of a day.
2. FitHelper displays the list corresponding to the specified date. If the user does not specify a date, FitHelper will display the list for `today`.

   Use case ends.

Extensions

1a. The date specified by the user does not contain any entries.

1a1. FitHelper shows the message reminding the user that that specific day is empty.

Use case ends.

Use case: UC05 - Show Reminders

MSS

1. User requests to display the undone tasks for a specific day.
2. FitHelper displays the undone task list corresponding to the specified date. If the user does not specify a date, FitHelper will display all the undone tasks in the coming 7 days.

   Use case ends.

Extensions

1a. The date specified by the user does not contain any entries.

1a1. FitHelper shows the message reminding the user that that specific day is empty.

Use case ends.

Use case: UC06 - Clear

MSS

1. User requests to clear the entries for a specific day.

2. FitHelper clears the entries in the daily file for the specified day. If the user does not input a `DATE`, FitHelper will delete all the entries for `today`.

   Use case ends.

Extensions

  1a. The date specified by the user does not contain any entries.

    1a1. FitHelper shows the message reminding the user that that specific day is empty.

    Use case ends.

Use case: UC07 - Undo

MSS

1. User requests to undo the previous command.
2. FitHelper undoes the previous command.

   Use case ends.

Extensions

  1a. There is no previous command.

    1a1. FitHelper shows an error message indicating there is no previous command.

    Use case ends.

  1b. The previous command is not undoable. e.g. The previous command is a `list` command where `undo` does not make sense.

    1b1. FitHelper shows an error message indicating the previous command is not undoable.

    Use case ends.

Use case: UC08 - Exit

MSS

1. User requests to exit the application.
2. FitHelper displayes the `goodbye` word and terminates the application.

   Use case ends.

Use case: UC09 - Record Profile

MSS

1. User requests to record profile and input values
2. FitHelper store user profile data.
3. FitHelper display successful record and show profile page.

   Use case ends.

Extensions

1a. User input incomplete values.

  1a1. FitHelper shows an error message.

  Use case ends.

1b. User has recorded basic data before.

  1b1. FitHelper shows an error message.

  Use case ends.

Use case: UC10 - Update Profile

MSS

1. User requests to update profile attribute values.
2. FitHelper update data accordingly.
3. FitHelper display successful update and show profile page.

   Use case ends.

Extensions

1a. User input unknown attribute.

  1a1. FitHelper shows an error message.

  Use case ends.

1b. User input wrong type for values.

  1b1. FitHelper shows an error message.

  Use case ends.

Use case: UC11 - Show Profile

MSS

1. User requests to show profile page.

2. FitHelper display profile page.

   Use case ends.

Extensions

1a. User has not record profile data yet.

 1a1. FitHelper shows an error message and reminder user to record.

 Use case ends.

Use case: UC12 - Show Weight Graph Page (for certain period)

MSS

1. User requests to show Weight Graph page (for certain period)
2. FitHelper display Weight page with information.

   Use case ends.

Extensions

1a. User input invalid time.

 1a1. FitHelper shows an error message.

 Use case ends.

Use case: UC13 - Keep a Diary

MSS

1. User adds comments as today’s diary
2. FitHelper adds the diary log to `today` file.

   Use case ends.

Extensions

1a. User input is incomplete.

 1a1. FitHelper shows an error message.

 Use case ends.

1b. User specifies a `DATE` after the `diary` keyword.

 1b1. FitHelper will take the `date` as part of the comments as `diary` is only for `today` .

Use case ends.

Use case: UC14 - View the Reward

MSS

1. User requests to view the rewarding points and current fitness level
2. FitHelper displays the users rewarding points together with the current fitness level.

   Use case ends.

Use case: UC15 - View Today Page

MSS

1. User requests to view `Today Page` with an optional `DATE` input
2. FitHelper displays `Today Page` for `today` if the `DATE` field is null and the `Today Page` for the specified date if there is a valid date.

   Use case ends.

Extensions

1a. User specifies an invalid date.

1a1. FitHelper shows shows an error message indicating the date is invalid.

1a2. FitHelper displays `Today Page` for `today`.

Use case ends.

1b. User specifies a date in the future.

1b1. FitHelper shows an error message indicating the date should be at least today or prior to today.

1b2. FitHelper shows the `Today Page` for `today`.

Use case ends.

Use case: UC16 - View Weekly Report

MSS

1. User requests to view `Weekly Report` with an optional `DATE` input
2. FitHelper displays `Weekly Report` for the current week if the `DATE` field is null and the `Weekly Report` for the week containing the specified date if there is a valid date.

   Use case ends.

Extensions

1a. User specifies an invalid date.

1a1. FitHelper shows shows an error message indicating the date is invalid.

1a2. FitHelper displays `Weekly Report` for the current week.

Use case ends.

1b. User specifies a date in the future.

1b1. FitHelper shows an error message indicating the date should be at least today or prior to today.

1b2. FitHelper shows the `Weekly Report` for the current week.

Use case ends.

Use case: UC17 - Get Reminders

MSS

1. User requests to be reminded about undone tasks.
2. FitHelper displays all the undone tasks for the coming 7 days if the `DATE` field is null and for the specified date if there is a valid date.

   Use case ends.

Extensions

1a. User specifies an invalid date.

1a1. FitHelper shows shows an error message indicating the date is invalid.

Use case ends.

Use case: UC18 - Find Entries Containing the Keyword

MSS

1. User requests to find entries containing a specific keyword.
2. FitHelper displays all the entries containing the keyword.

   Use case ends.

Extensions

1a. User inputs a keyword that does not show in any entries.

1a1. FitHelper shows shows an error message indicating no entry matches the keyword.

Use case ends.

Use case: UC19 - Calendar View

MSS

1. User requests to display tasks in calendar view.
2. FitHelper displays all tasks of the current months with colored indication of whether the daily target is achieved or not.

   Use case ends.

Extensions

1a. User specifies an invalid command.

1a1. FitHelper shows shows an error message indicating the command is invalid.

Use case ends.

{More to be added}

Appendix D: Non Functional Requirements

1. Should work on any mainstream OS as long as it has Java 11 or above installed.
2. Should be able to hold up to 1000 entries without a noticeable sluggishness in performance for typical usage
3. Should be able to function normally without internet access.
4. A user with above average typing speed for regular English text (i.e. not code, not system admin commands) should be able to accomplish most of the tasks faster using commands than using the mouse.
5. A user can get response from the system within 5 seconds after command input.
6. A user can be familiar with the system commands and interface within half an hour usage.

{More to be added}

Appendix E: Glossary

Mainstream OS

Windows, Linux, Unix, OS-X

Table 1. Command Prefix

Prefix	**Meaning**	**Used in the following Command(s)**
n/	Name	addX, editX, recordprofile
t/	Time in format of **yyyy-mm-dd-24-60**	addX, editX
l/	Location	addX, editX
c/	Calories	addX, editX
s/	Status	addX, edit, editX
d/	Date in format of **yyyy-mm-dd**	list, listX, reminder, reminderX, edit, editX, deleteX, periodstart, periodend
dr/	Duration in format of **yyyy-mm-dd yyyy-mm-dd**	list, listX, reminder, reminderX, weightgraph
r/	Remark	editX
addr/	Address	recordprofile
g/	Gender	recordprofile
h/	Height	recordprofile
cw/	Current Weight	recordprofile
tw/	Target Weight	recordprofile
attr/	Attribute	update
v/	Attribute Value	update

Table 2. Possible Command Flags

Command	**Flag**	**Meaning**
Sort	-a	Sort in **ascending** order
Sort	-d	Sort in **descending** order
Sort	-t	Sort according to **time**
Sort	-c	Sort according to **calorie intake**

Appendix F: Product Survey

Product Name

Author: …

Pros:

- …
- …

Cons:

- …
- …

Appendix G: Instructions for Manual Testing

Given below are instructions to test the app manually.

NOTE These instructions only provide a starting point for testers to work on; testers are expected to do more *exploratory* testing.

G.1. Launch and Shutdown

1. Initial launch
   a. Download the jar file and copy into an empty folder
   b. Double-click the jar file
   Expected: Shows the GUI with a set of sample contacts. The window size may not be optimum.
2. Saving window preferences
   a. Resize the window to an optimum size. Move the window to a different location. Close the window.
   b. Re-launch the app by double-clicking the jar file.
   Expected: The most recent window size and location is retained.

{ more test cases … }

G.2. Deleting a person

1. Deleting a person while all persons are listed
   a. Prerequisites: List all persons using the `list` command. Multiple persons in the list.
   b. Test case: `delete 1`
   Expected: First contact is deleted from the list. Details of the deleted contact shown in the status message. Timestamp in the status bar is updated.
   c. Test case: `delete 0`
   Expected: No person is deleted. Error details shown in the status message. Status bar remains the same.
   d. Other incorrect delete commands to try: `delete`, `delete x` (where x is larger than the list size) *{give more}*
   Expected: Similar to previous.

{ more test cases … }

G.3. Saving data

1. Dealing with missing/corrupted data files
    a. *{explain how to simulate a missing/corrupted file and the expected behavior}*

{ more test cases … }