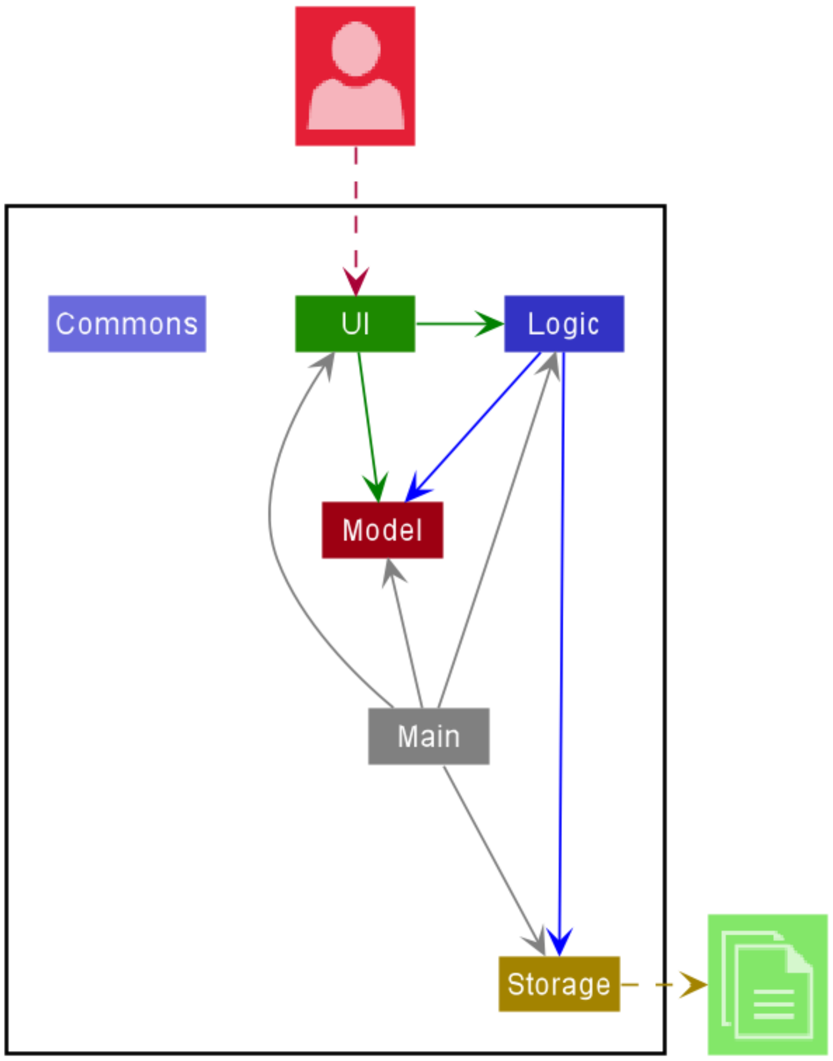
- Table of Contents {:toc}

Setting up, getting started

Refer to the guide *Setting up and getting started*.

Design

Architecture



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

:bulb: **Tip:** The `.puml` files used to create diagrams in this document can be found in the diagrams folder. Refer to the *PlantUML Tutorial* at se-edu/guides 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 methods where necessary.

`Commons` represents a collection of classes used by multiple other components.

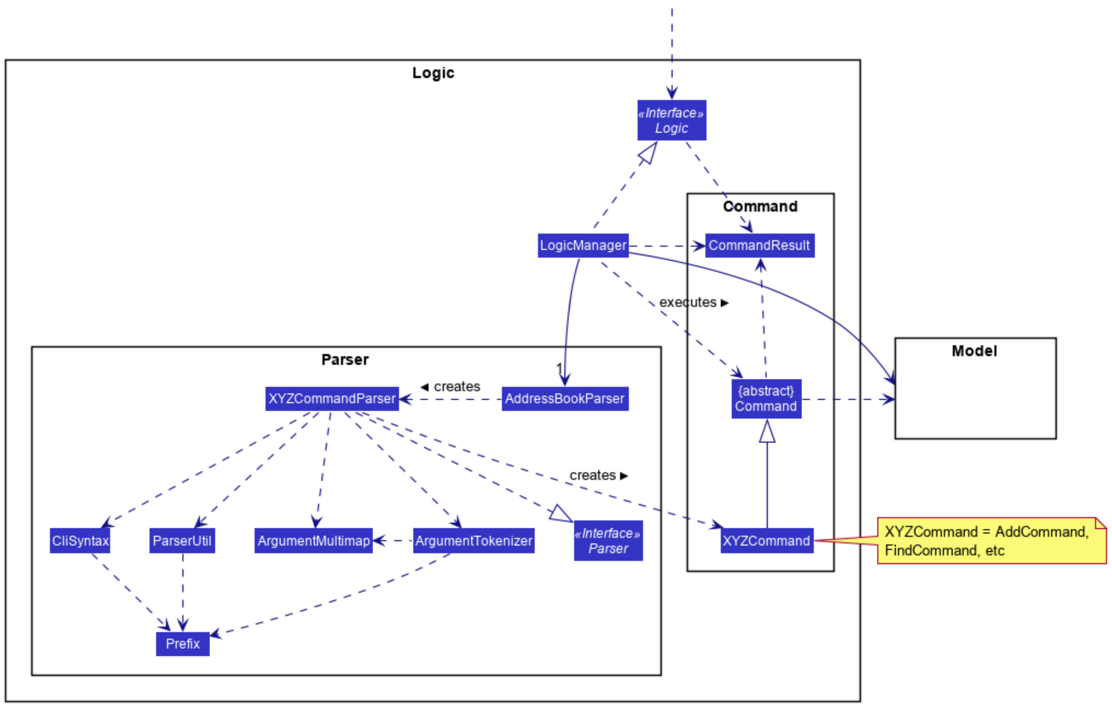
The rest of the App consists of four components.

- `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,

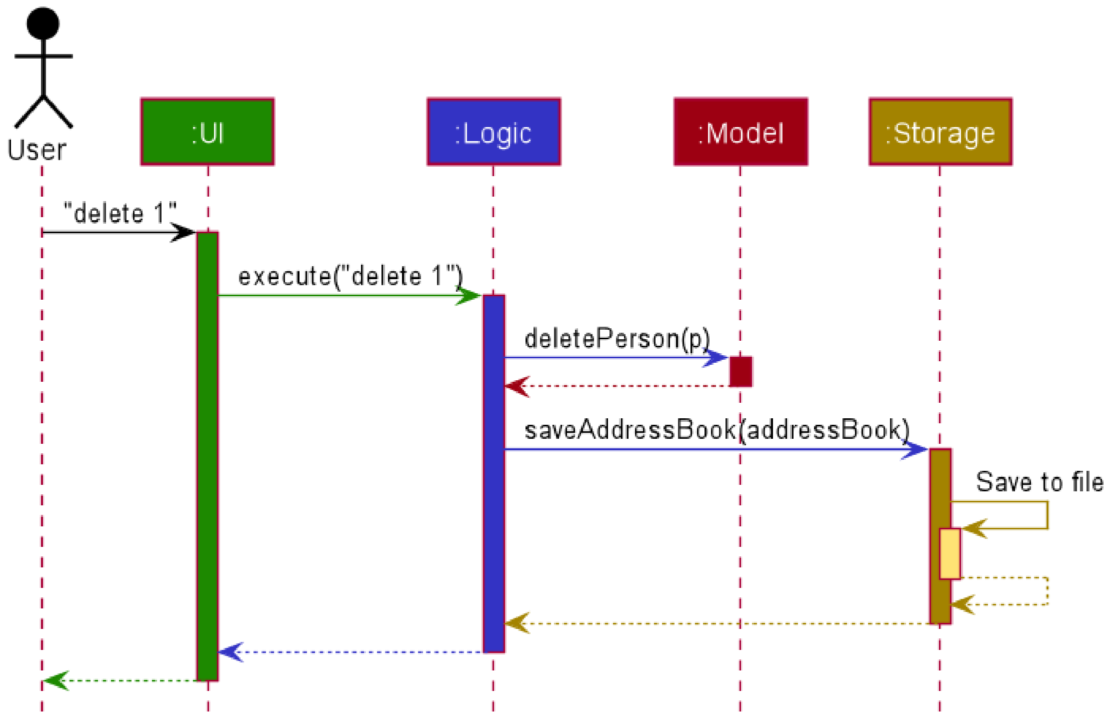
- defines its *API* in an `interface` with the same name as the Component.
- exposes its functionality using a concrete `{Component Name}Manager` class (which implements the corresponding API `interface` mentioned in the previous point.

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



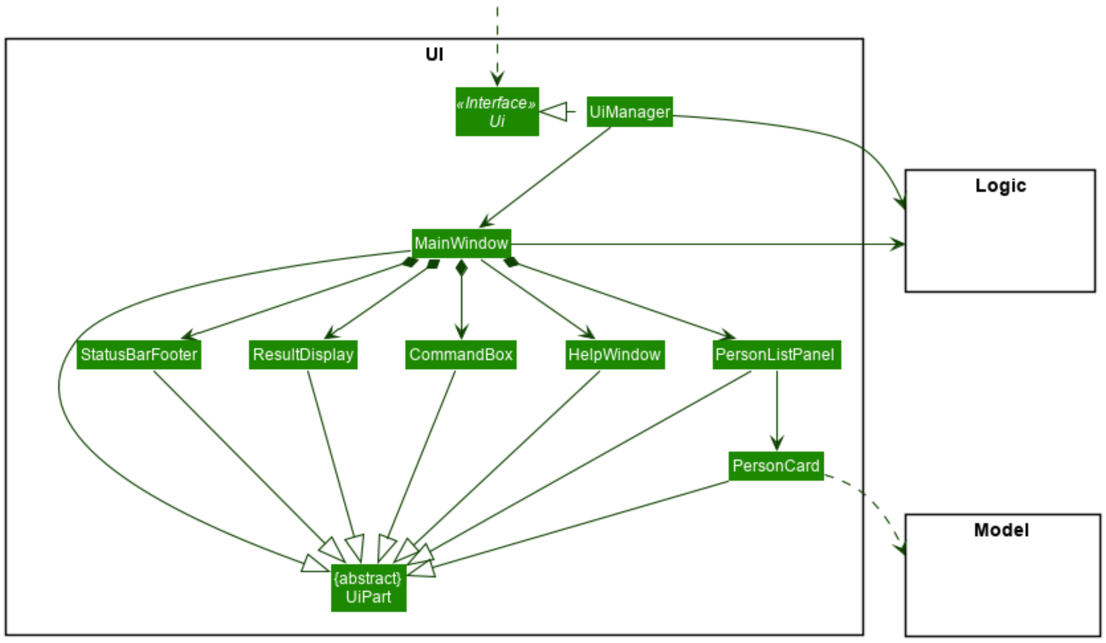
How the architecture components interact with each other

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



The sections below give more details of each component.

UI component



API : `Ui.java`

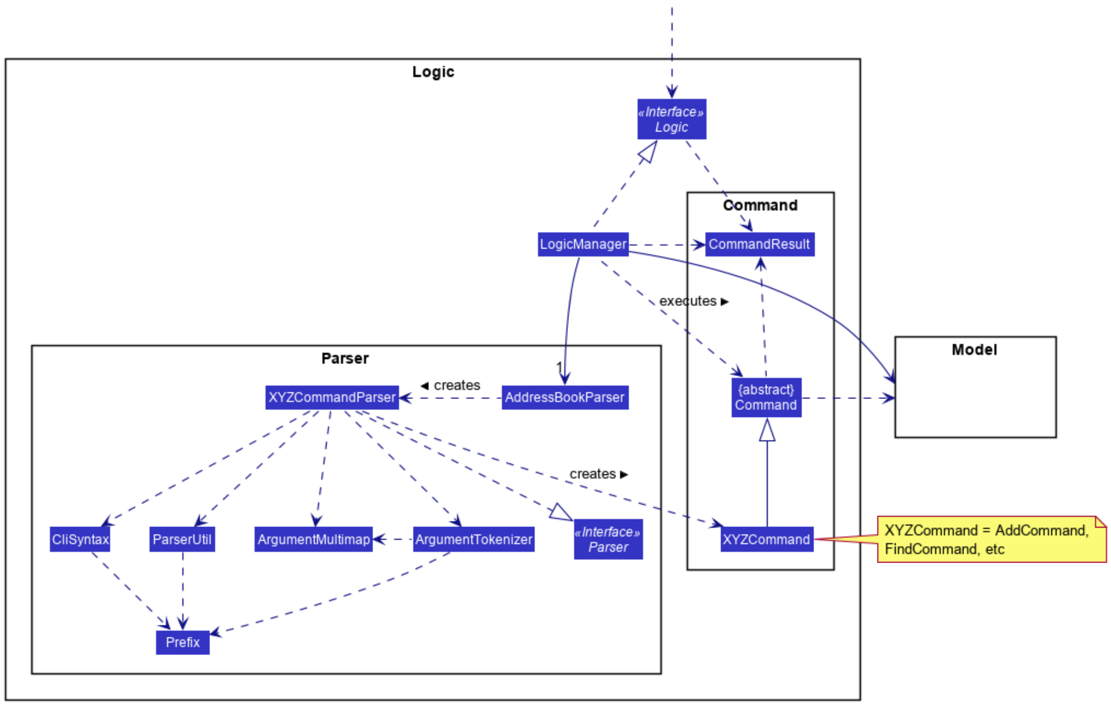
The UI consists of a `MainWindow` that is made up of parts e.g. `CommandBox` , `ResultDisplay` , `PersonListPanel` , `StatusBarFooter` etc. All these, including the `MainWindow` , inherit from the abstract `UiPart` class.

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.

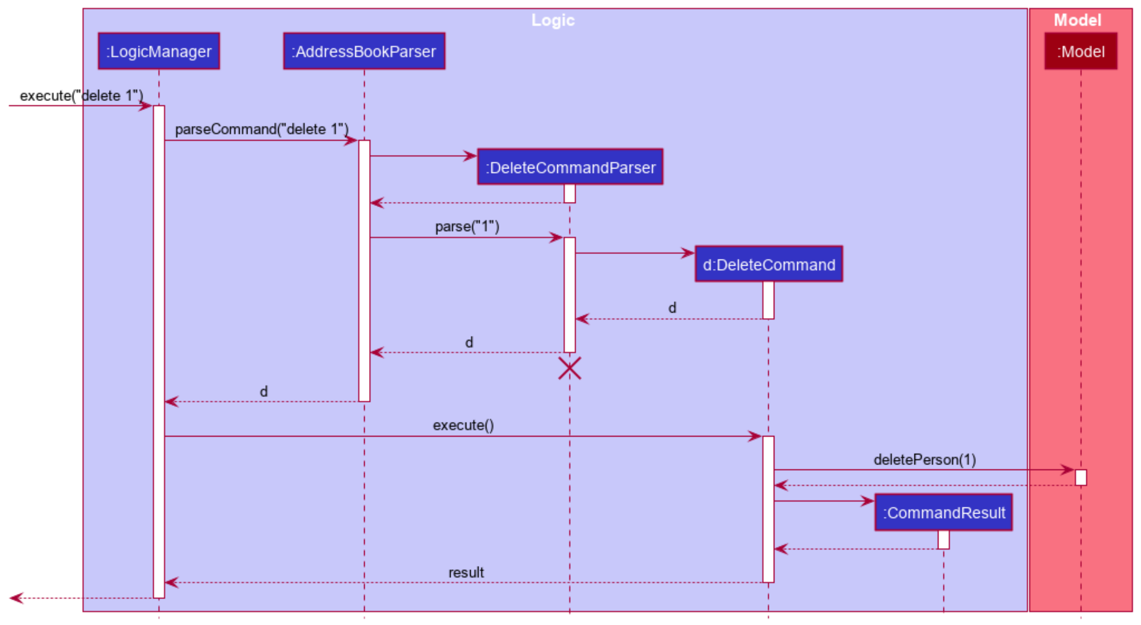
Logic component



API : [`Logic.java`]()

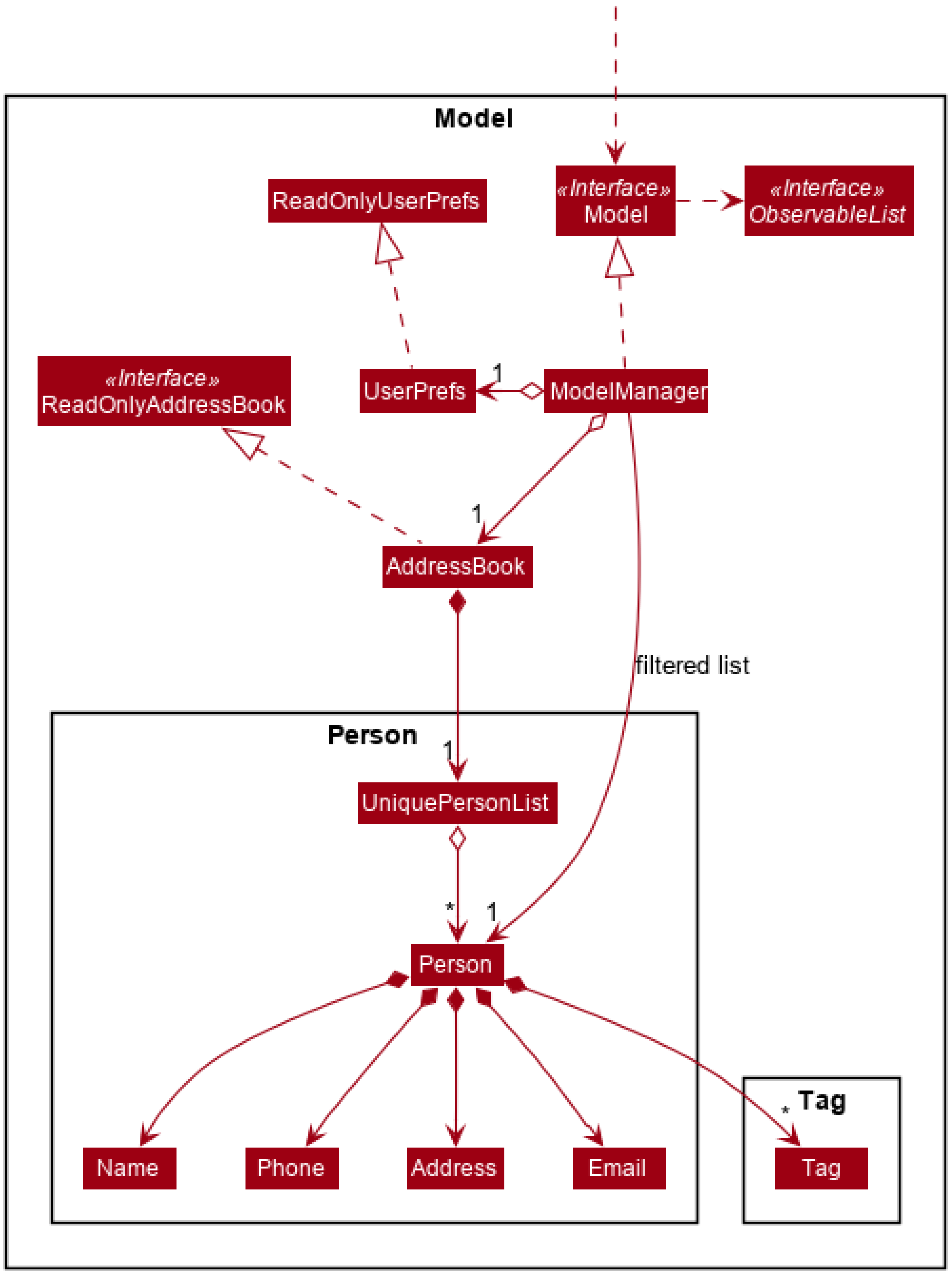
1. `Logic` uses the `AddressBookParser` class to parse the user command.
2. This results in a `Command` object which is executed by the `LogicManager` .
3. The command execution can affect the `Model` (e.g. adding a person).
4. The result of the command execution is encapsulated as a `CommandResult` 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.



:information_source: **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.

Model component



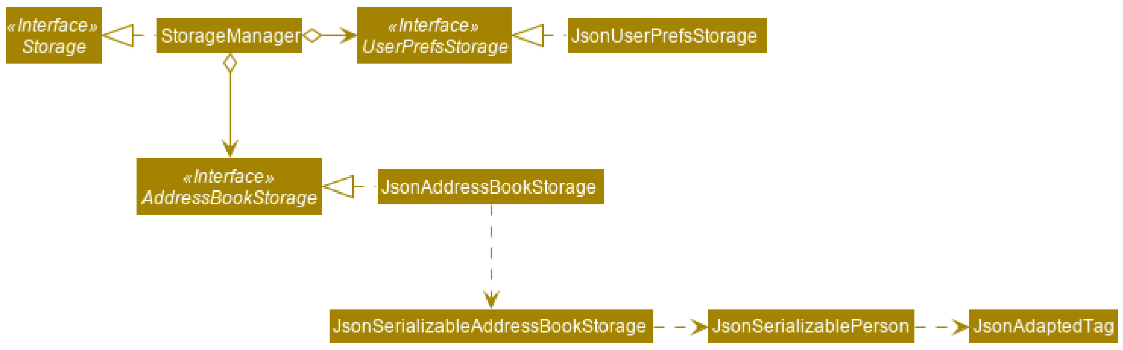
API : [`Model.java`]

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.

:information_source: **Note:** An alternative (arguably, a more OOP) model is given below. It has a `Tag` list in the `AddressBook`, which `Person` references. This allows `AddressBook` to only require one `Tag` object per unique `Tag`, instead of each `Person` needing their own `Tag` object.
![BetterModelClassDiagram](images/BetterModelClassDiagram.png)

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.

Common classes

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

Implementation

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

[Proposed] Undo/redo feature

Proposed Implementation

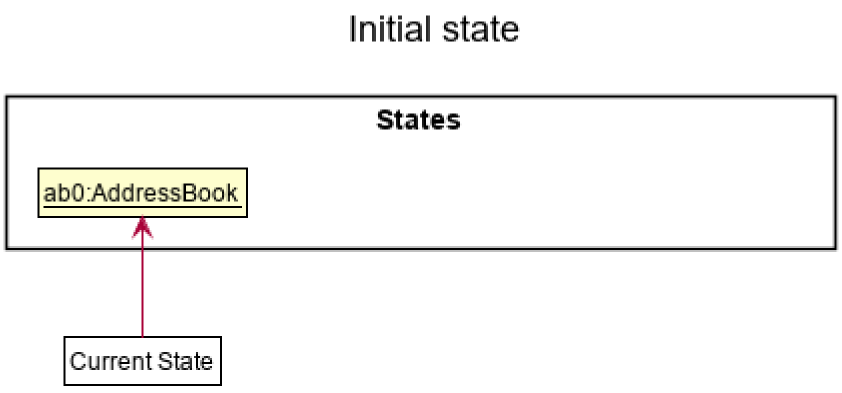
The proposed undo/redo mechanism is facilitated by `VersionedAddressBook`. It extends `AddressBook` with an undo/redo history, stored internally as an `addressBookStateList` and `currentStatePointer`. Additionally, it implements the following operations:

- `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.

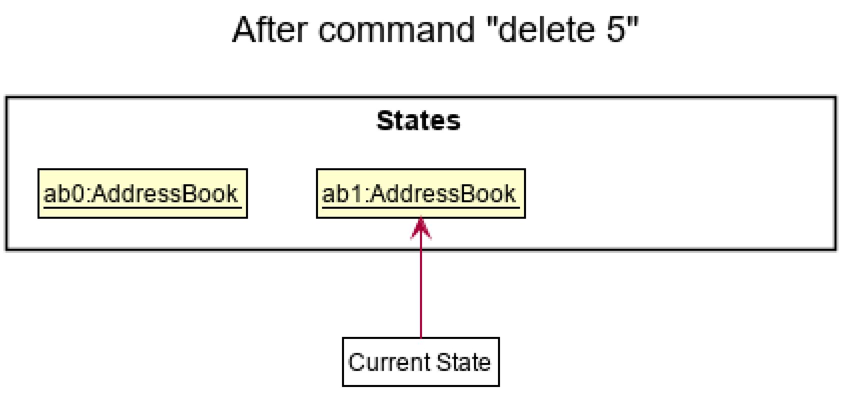
These operations are exposed in the `Model` interface as `Model#commitAddressBook()`, `Model#undoAddressBook()` and `Model#redoAddressBook()` respectively.

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

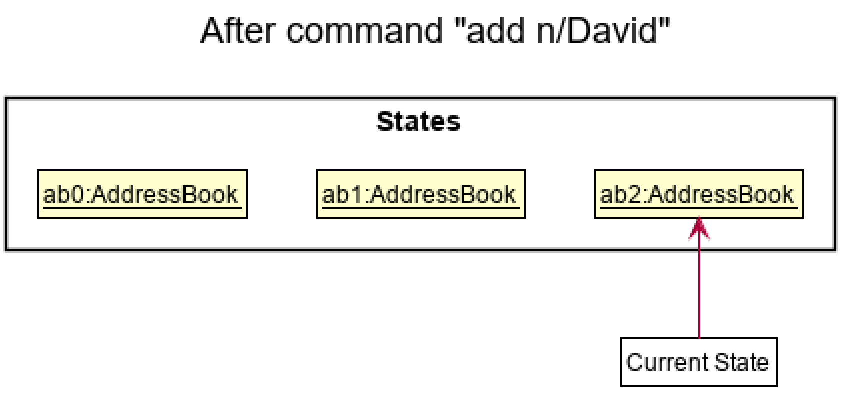
Step 1. The user launches the application for the first time. The `VersionedAddressBook` will be initialized with the initial address book state, and the `currentStatePointer` pointing to that single address book state.



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.

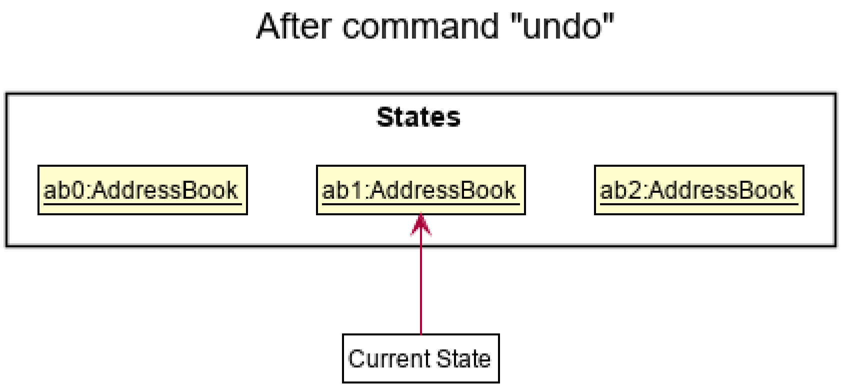


Step 3. The user executes `add n/David …` to add a new person. The `add` command also calls `Model#commitAddressBook()`, causing another modified address book state to be saved into the `addressBookStateList`.



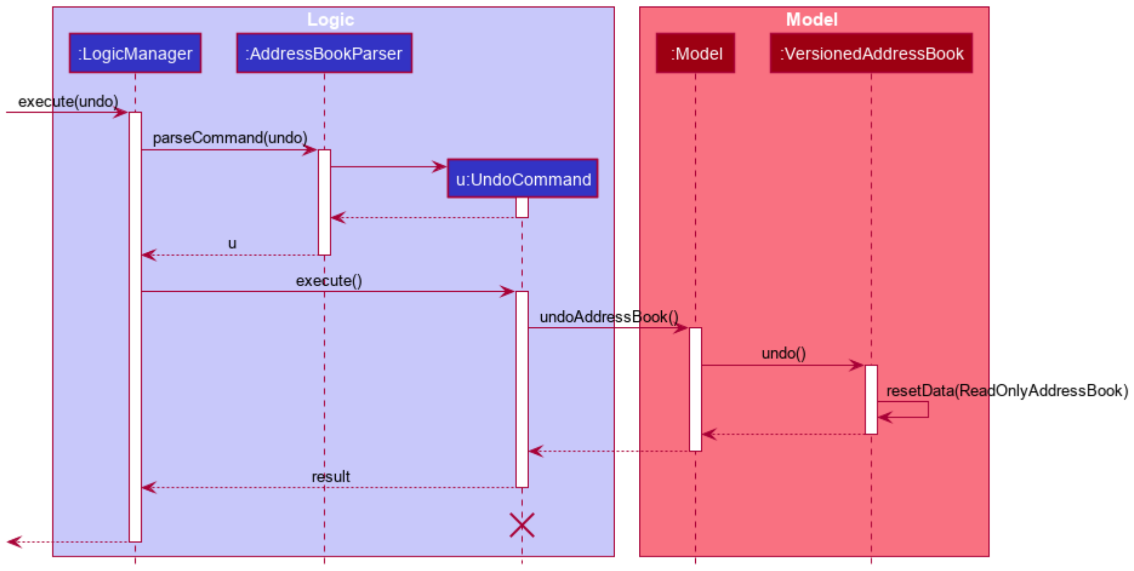
:information_source: **Note:** If a command fails its execution, it will not call `Model#commitAddressBook()`, so the address book state will not be saved into the `addressBookStateList`.

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.



:information_source: **Note:** If the `currentStatePointer` is at index 0, pointing to the initial AddressBook state, then there are no previous AddressBook 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:

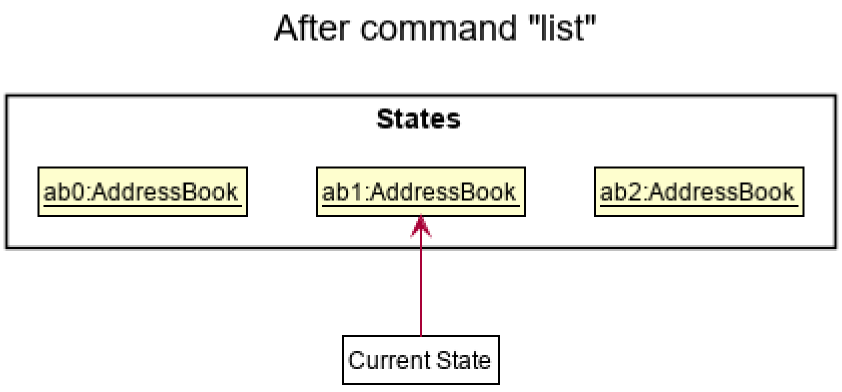


:information_source: **Note:** The lifeline for `UndoCommand` should end at the destroy marker (X) but due to a limitation of PlantUML, the lifeline reaches the end of diagram.

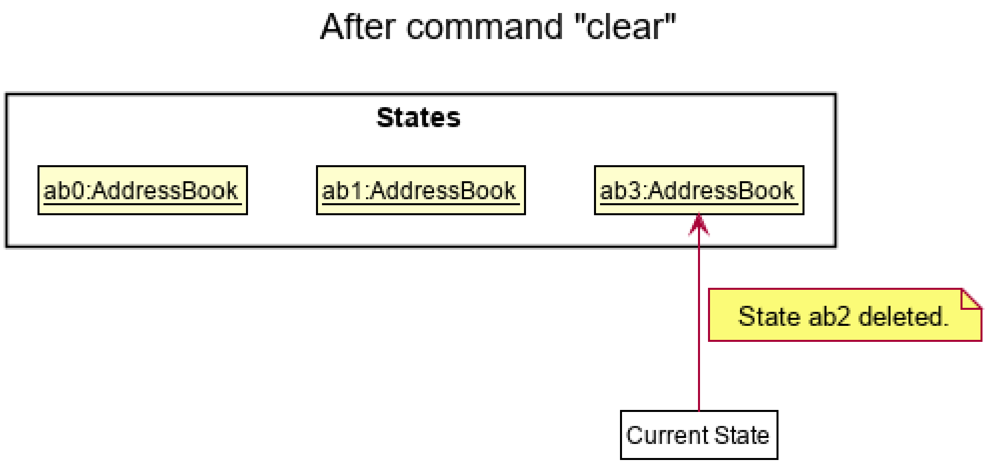
The `redo` command does the opposite — it calls `Model#redoAddressBook()`, which shifts the `currentStatePointer` once to the right, pointing to the previously undone state, and restores the address book to that state.

:information_source: **Note:** If the `currentStatePointer` is at index `addressBookStateList.size() - 1`, pointing to the latest address book state, then there are no undone AddressBook 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.

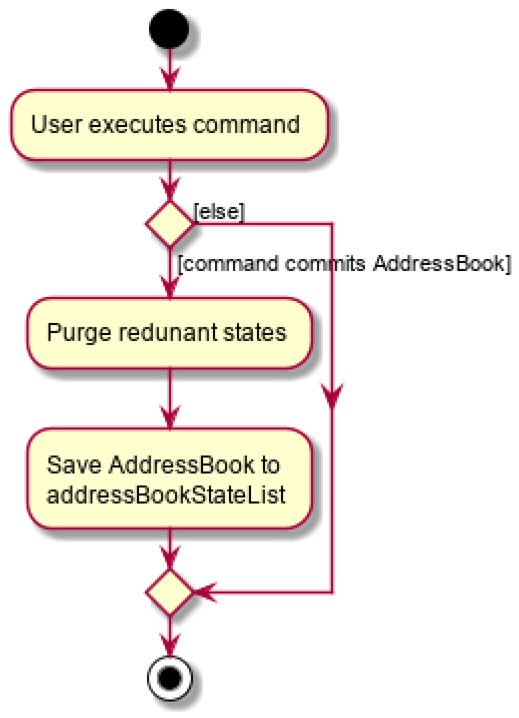
Step 5. The user then decides to execute the command `list`. Commands that do not modify the address book, such as `list`, will usually not call `Model#commitAddressBook()`, `Model#undoAddressBook()` or `Model#redoAddressBook()`. Thus, the `addressBookStateList` remains unchanged.



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. Reason: 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

Design consideration:

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.

{more aspects and alternatives to be added}

Aspect: How to archive Data

- **Current Choice:** Using Archive Command.
    - Implementation: Have a separate Archive Command to let the User specify the file location.
    - Pros: The user can specify the location where users want to save the data.
    - Cons: Error prone. The user need to enter the file location specification which may be difficult for users who are not used to command prompt.

Documentation, logging, testing, configuration, dev-ops

- Documentation guide
- Testing guide
- Logging guide
- Configuration guide
- DevOps guide

Appendix: Requirements

Product scope

Target user profile:

- has a need to manage their workouts
- prefer desktop apps over other types
- can type fast
- prefers typing to mouse interactions
- is reasonably comfortable using CLI apps

Value proposition: manage workouts faster than a typical mouse/GUI driven desktop/mobile app

User stories

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

Priority	As a …	I want to …	So that I can…
`* * *`	user	add an exercise	keep track of calories burnt through the day
`* * *`	user	have a system that tolerate invalid/incomplete command	
`* * *`	data conscious user	list down all the exercises for the day	monitor the calories burned accurately
`* * *`	user	delete an exercise in case I key in wrongly	
`* *`	user	update an exercise	
`* *`	user	save my data in a file	import the saved data into the new computer

{More to be added}

Use cases

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

Use case: add an exercise

MSS

1. User requests to add an exercise

2. Calo adds the exercise Use case ends.

Use case: Find exercises with a keyword

MSS

1. User requests to find exercises with a keyword
2. Calo shows a list of exercises which contain the keyword Use case ends.

Extensions

The list is empty
1a1. Calo shows a message indicating that no such exercise exists.
Use case ends.

Use case: Update an exercise

MSS

1. User requests to update a specific exercise in the list
2. Calo updates the exercise Use case ends.

Extensions

1a. The index is invalid
1a1. Calo shows a message indicating that no such exercise exists.
Use case ends.

Use case: Delete an exercise

MSS

1. User requests to delete a specific exercise in the list.

2. Calo deletes the exercise

   Use case ends.

Extensions

1a. The index is invalid
1a1. Calo shows a message indicating that no such exercise exists.
Use case ends.

Use case: Archive data

MSS

1. User requests to archive data to a different file location
2. Calo archives data to the specified location Use case ends.

Extensions

1a. User does not have permission to create file at specified location
1a1. Calo shows a message indicating that file cannot be created at specified file.
Use case ends.

Use case: List exercises

MSS

1. User requests to list exercises
2. Calo shows a list of exercises Use case ends.

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 exercise items without a noticeable sluggishness in performance for typical usage.
3. 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.

Glossary

- **Mainstream OS**: Windows, Linux, Unix, OS-X
- **Exercise**: an exercise record entered by the user, consisting of exercise name, description, and date (optionally calories)

Appendix: Instructions for manual testing

Given below are instructions to test the app manually.

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

Launch and shutdown

1. Initial launch
    1. Download the jar file and copy into an empty folder
    2. 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
    1. Resize the window to an optimum size. Move the window to a different location. Close the window.
    2. Re-launch the app by double-clicking the jar file.
       Expected: The most recent window size and location is retained.
3. *{ more test cases … }*

Deleting an exercise

1. Deleting an exercise while all exercises are being shown
    1. Prerequisites: List all exercises using the `list` command. Multiple exercises in the list.
    2. 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.
    3. Test case: `delete 0`
       Expected: No exercise is deleted. Error details shown in the status message. Status bar remains the same.
    4. Other incorrect delete commands to try: `delete`, `delete x`, `...` (where x is larger than the list size)
       Expected: Similar to previous.
2. *{ more test cases … }*

Saving data

1. Dealing with missing/corrupted data files
    1. *{explain how to simulate a missing/corrupted file, and the expected behavior}*
2. *{ more test cases … }*