HealthSync Developer Guide

Acknowledgements

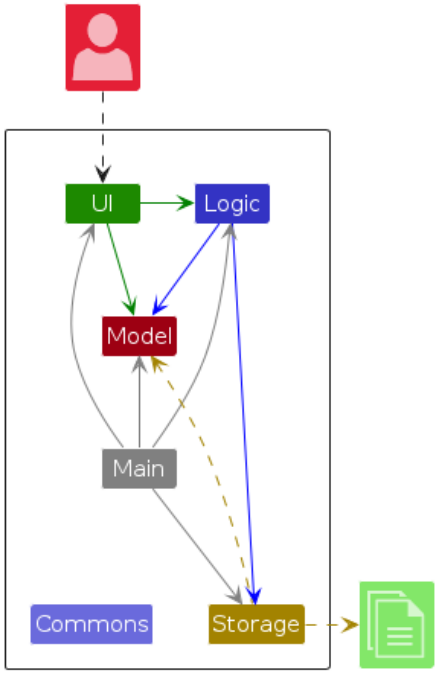
{ list here sources of all reused/adapted ideas, code, documentation, and third-party libraries -- include links to the original source as well }

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 main components and how they interact with each other.

Main components of the architecture

`Main` (consisting of classes `Main` and `MainApp`) is in charge of the app launch and shut down.

- At app launch, it initializes the other components in the correct sequence, and connects them up with each other.
- At shut down, it shuts down the other components and invokes cleanup methods where necessary.

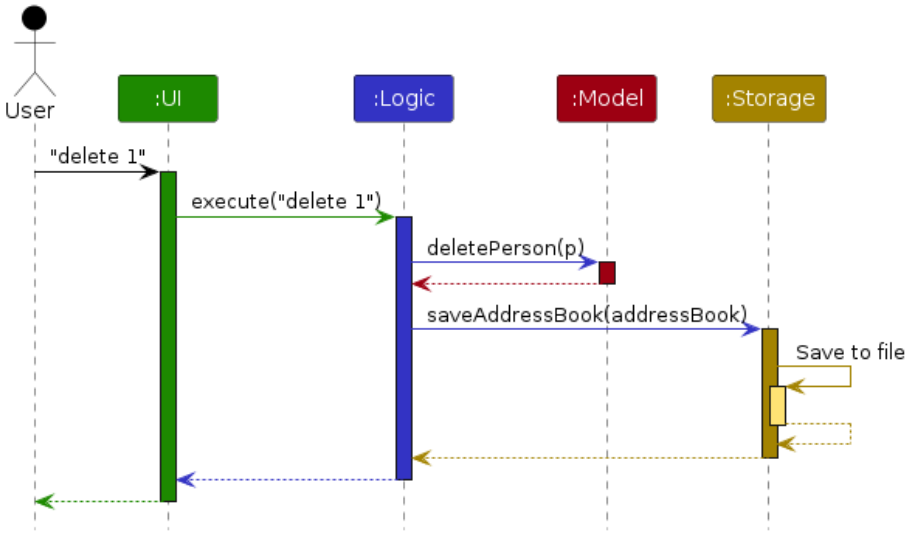
The bulk of the app's work is done by the following four components:

- `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.

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

How the architecture components interact with each other

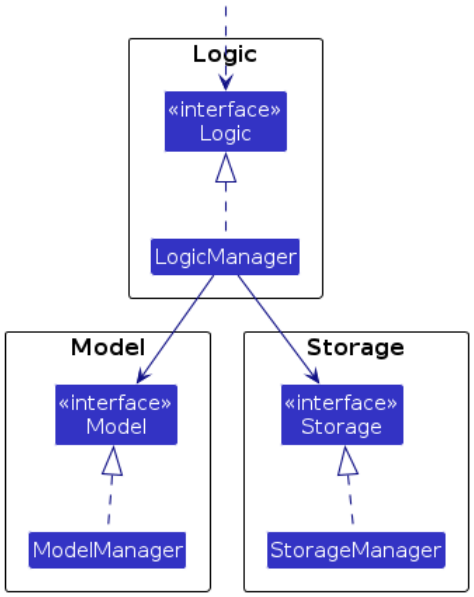
The *Sequence Diagram* below shows how the components interact with each other for the scenario where the user issues the command `delete 1`.



Each of the four main components (also shown in the diagram above),

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

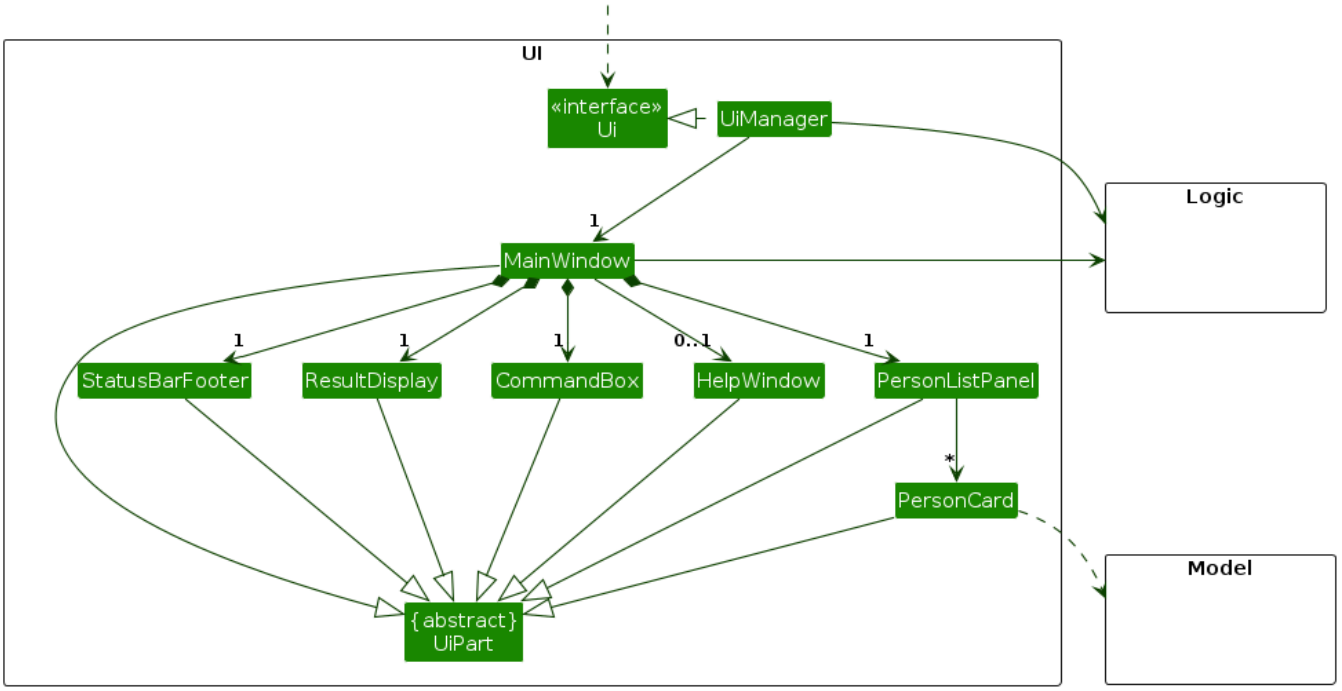
For example, the `Logic` component defines its API in the `Logic.java` interface and implements its functionality using the `LogicManager.java` class which follows the `Logic` interface. Other components interact with a given component through its interface rather than the concrete class (reason: to prevent outside component's being coupled to the implementation of a component), as illustrated in the (partial) class diagram below.



The sections below give more details of each component.

UI component

The **API** of this component is specified in `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 which captures the commonalities between classes that represent parts of the visible GUI.

The `UI` component uses the 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`

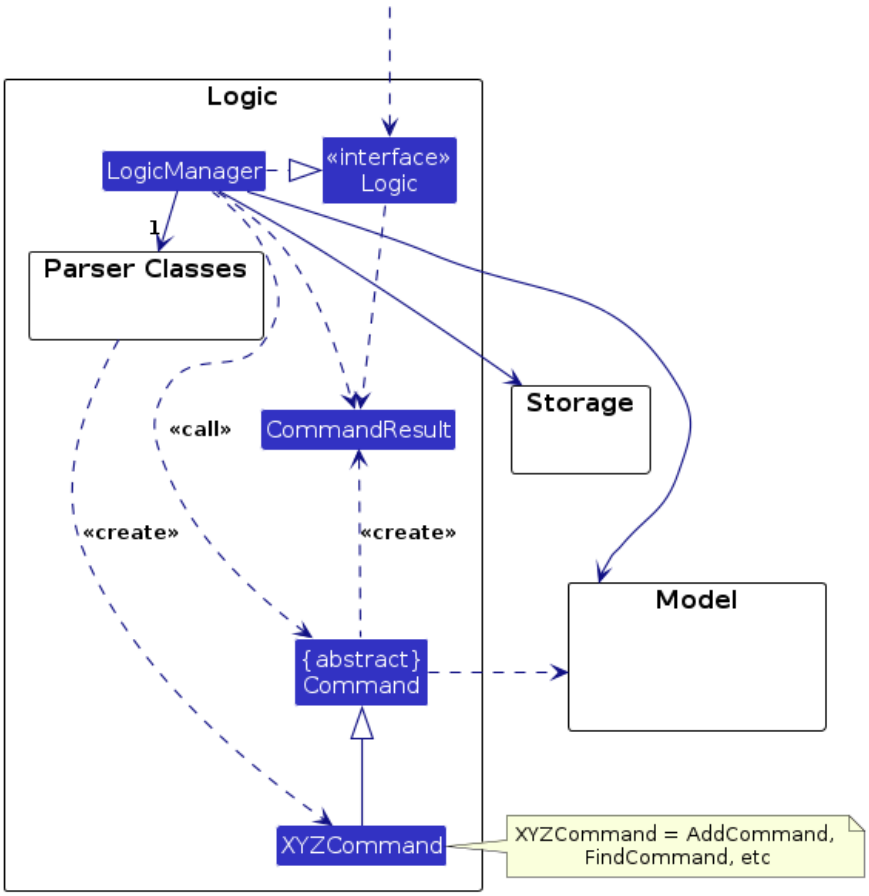
The `UI` component,

- executes user commands using the `Logic` component.
- listens for changes to `Model` data so that the UI can be updated with the modified data.
- keeps a reference to the `Logic` component, because the `UI` relies on the `Logic` to execute commands.
- depends on some classes in the `Model` component, as it displays `Person` object residing in the `Model`.

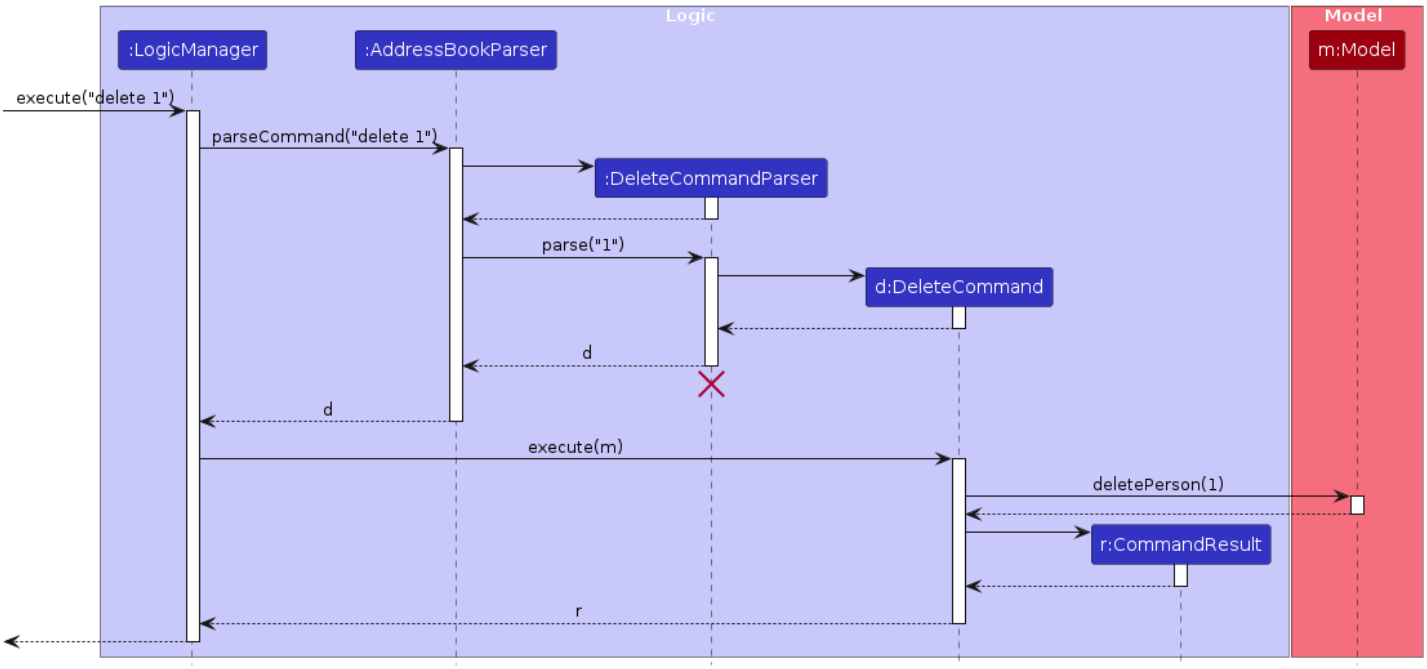
Logic component

API : `Logic.java`

Here's a (partial) class diagram of the `Logic` component:



The sequence diagram below illustrates the interactions within the `Logic` component, taking `execute("delete 1")` API call as an example.

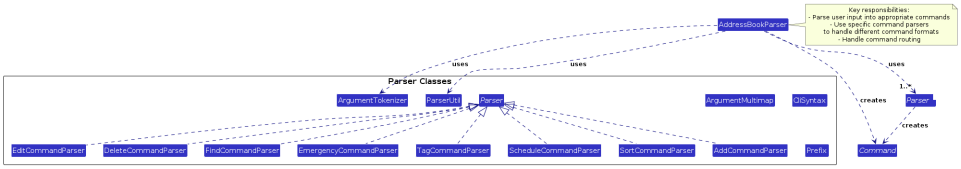


i **Note:** The lifeline for `DeleteCommandParser` should end at the destroy marker (X) but due to a limitation of PlantUML, the lifeline continues till the end of diagram.

How the `Logic` component works:

1. When `Logic` is called upon to execute a command, it is passed to an `AddressBookParser` object which in turn creates a parser that matches the command (e.g., `DeleteCommandParser`) and uses it to parse the command.
2. This results in a `Command` object (more precisely, an object of one of its subclasses e.g., `DeleteCommand`) which is executed by the `LogicManager`.
3. The command can communicate with the `Model` when it is executed (e.g. to delete a person).
   Note that although this is shown as a single step in the diagram above (for simplicity), in the code it can take several interactions (between the command object and the `Model`) to achieve.
4. The result of the command execution is encapsulated as a `CommandResult` object which is returned back from `Logic`.

Here are the other classes in `Logic` (omitted from the class diagram above) that are used for parsing a user command:

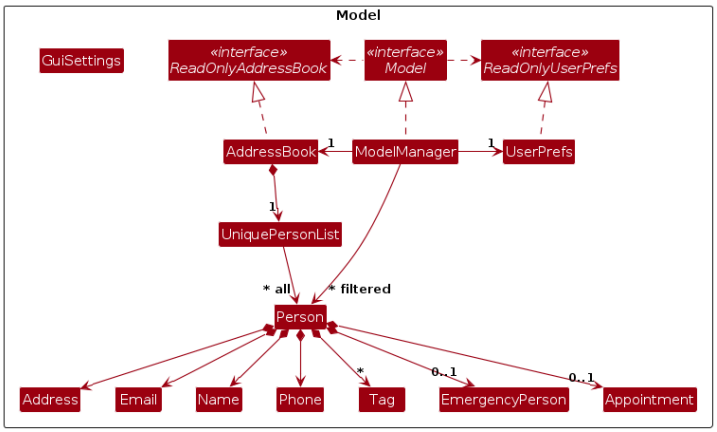


How the parsing works:

- When called upon to parse a user command, the `AddressBookParser` class creates an `XYZCommandParser` (`XYZ` is a placeholder for the specific command name e.g., `AddCommandParser`) which uses the other classes shown above to parse the user command and create a `XYZCommand` object (e.g., `AddCommand`) which the `AddressBookParser` returns back as a `Command` object.
- All `XYZCommandParser` classes (e.g., `AddCommandParser`, `DeleteCommandParser`, ...) inherit from the `Parser` interface so that they can be treated similarly where possible e.g, during testing.

Model component

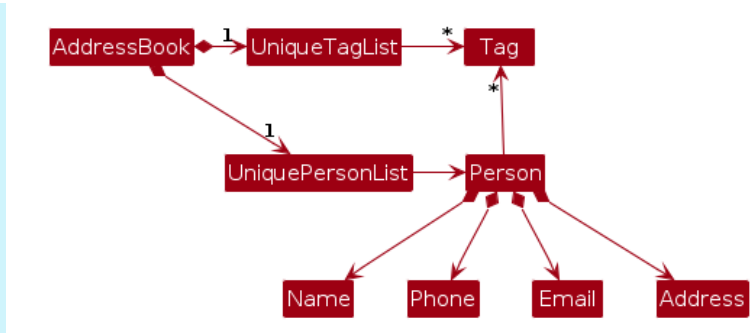
API : `Model.java`



The `Model` component,

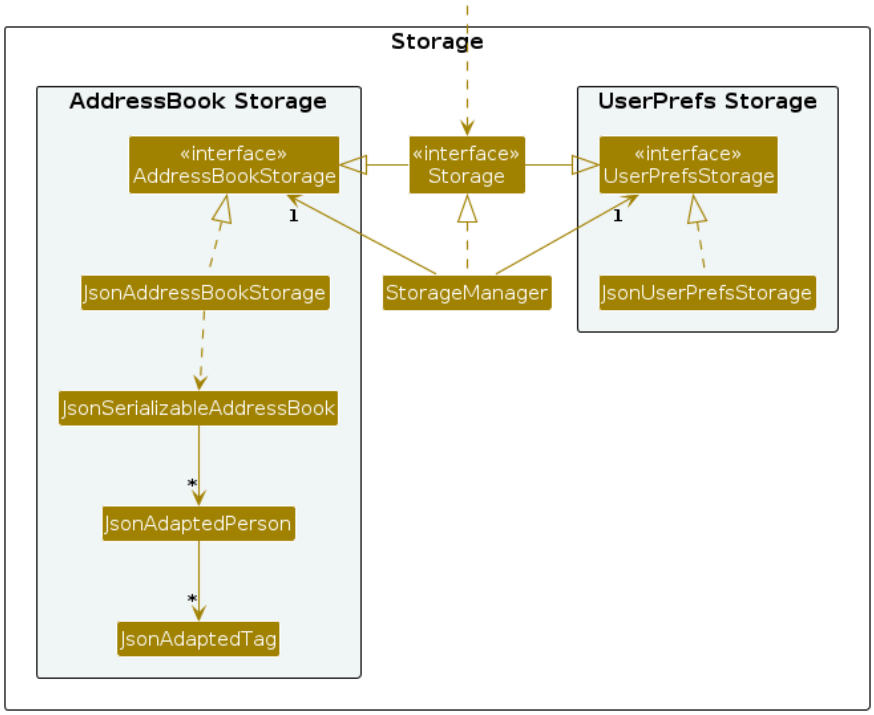
- stores the address book data i.e., all `Person` objects (which are contained in a `UniquePersonList` object).
- stores the currently 'selected' `Person` objects (e.g., results of a search query) as a separate *filtered* list which is exposed to outsiders as an unmodifiable `ObservableList<Person>` 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.
- stores a `UserPref` object that represents the user's preferences. This is exposed to the outside as a `ReadOnlyUserPref` objects.
- does not depend on any of the other three components (as the `Model` represents data entities of the domain, they should make sense on their own without depending on other components)

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` objects.



Storage component

API : `Storage.java`



The `Storage` component,

- can save both address book data and user preference data in JSON format, and read them back into corresponding objects.
- inherits from both `AddressBookStorage` and `UserPrefStorage`, which means it can be treated as either one (if only the functionality of only one is needed).
- depends on some classes in the `Model` component (because the `Storage` component's job is to save/retrieve objects that belong to the `Model`)

Common classes

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

Implementation

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

[Proposed] Undo/redo feature

Proposed Implementation

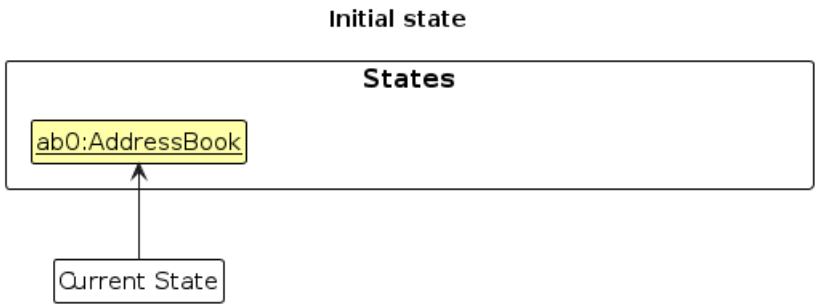
The proposed undo/redo mechanism is facilitated by `VersionedAddressBook`. It extends `AddressBook` with an undo/redo history, stored internally as an `addressBookStateList` and `currentStatePointer`. Additionally, it implements the following operations:

- `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.

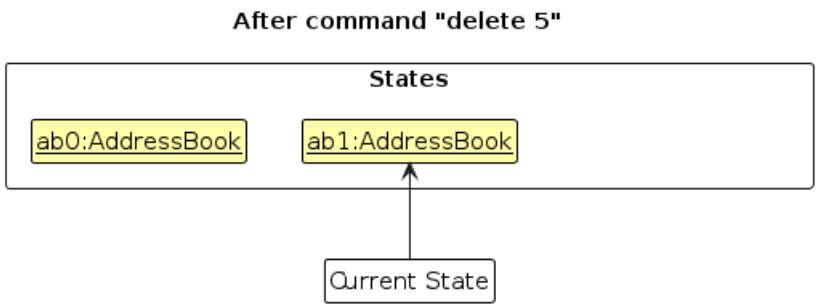
These operations are exposed in the `Model` interface as `Model#commitAddressBook()`, `Model#undoAddressBook()` and `Model#redoAddressBook()` respectively.

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

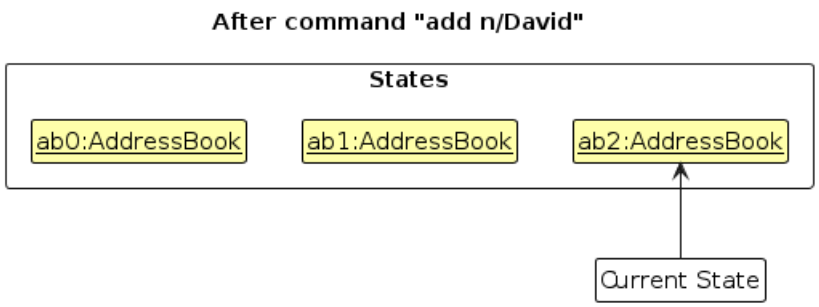
Step 1. The user launches the application for the first time. The `VersionedAddressBook` will be initialized with the initial address book state, and the `currentStatePointer` pointing to that single address book state.



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.

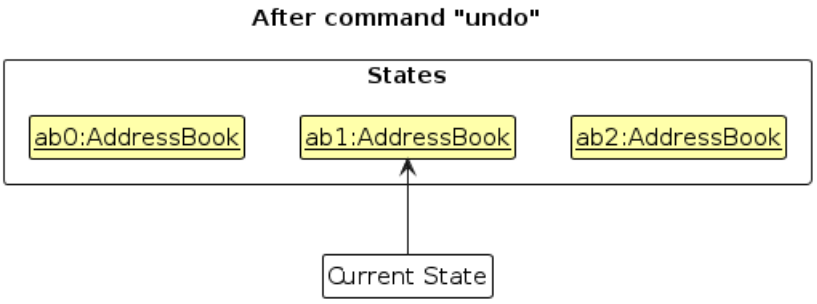


Step 3. The user executes `add n/David …` to add a new person. The `add` command also calls `Model#commitAddressBook()`, causing another modified address book state to be saved into the `addressBookStateList`.



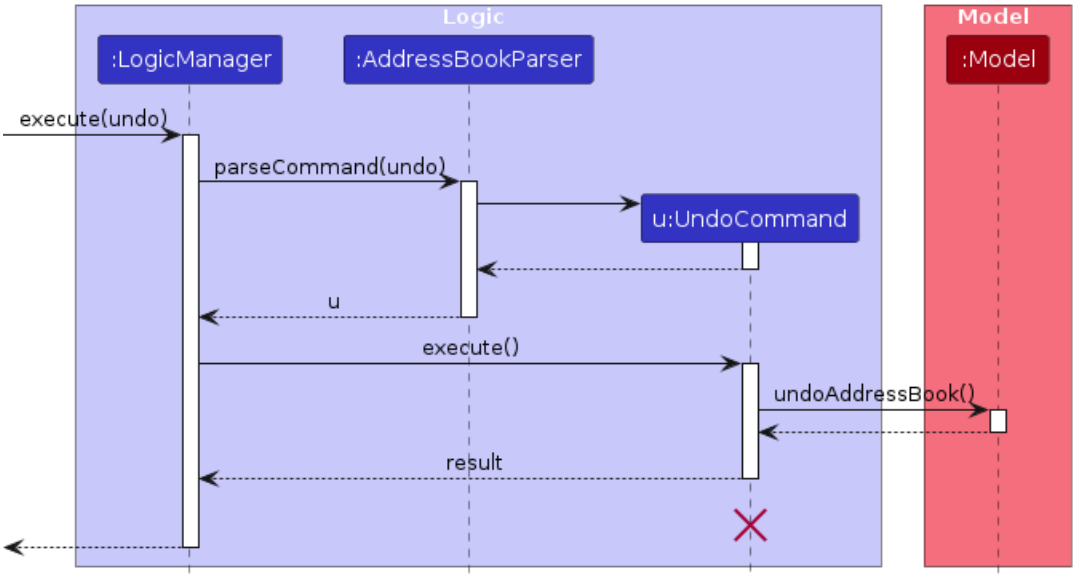
> **Note:** If a command fails its execution, it will not call `Model#commitAddressBook()`, so the address book state will not be saved into the `addressBookStateList`.

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.



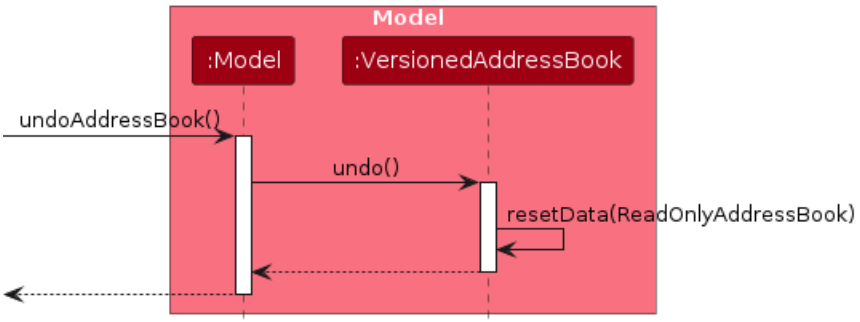
ℹ **Note:** If the `currentStatePointer` is at index 0, pointing to the initial AddressBook state, then there are no previous AddressBook states to restore. The `undo` command uses `Model#canUndoAddressBook()` to check if this is the case. If so, it will return an error to the user rather than attempting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

The following sequence diagram shows how an undo operation goes through the `Logic` component:



ℹ **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.

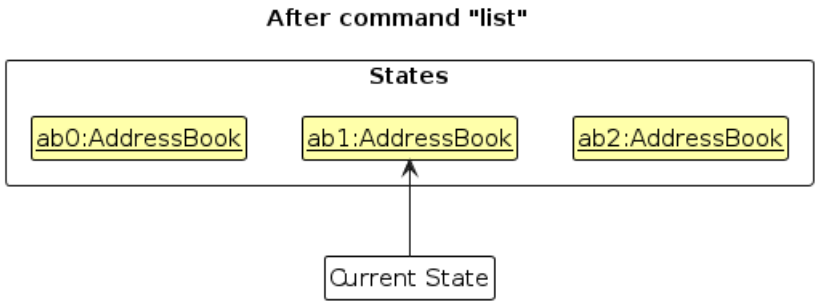
Similarly, how an undo operation goes through the `Model` component is shown below:



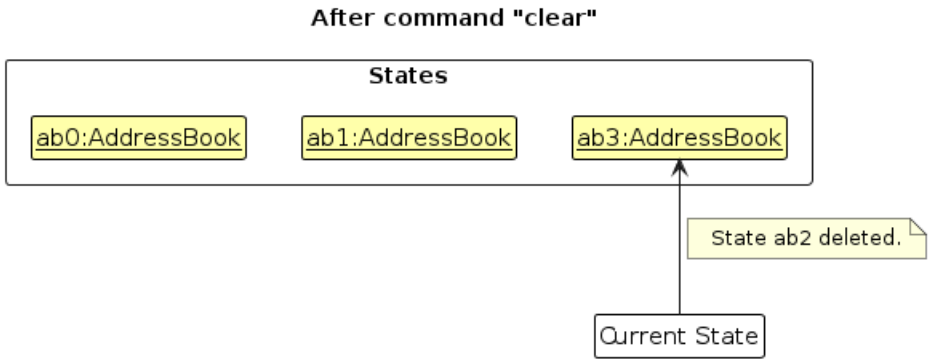
The `redo` command does the opposite — it calls `Model#redoAddressBook()`, which shifts the `currentStatePointer` once to the right, pointing to the previously undone state, and restores the address book to that state.

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

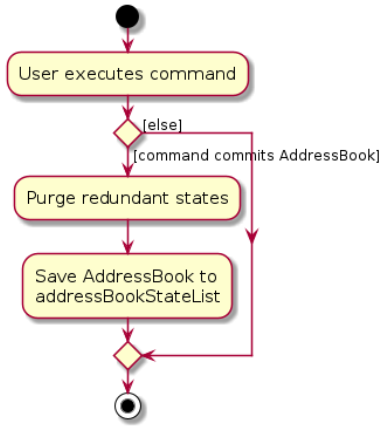
Step 5. The user then decides to execute the command `list`. Commands that do not modify the address book, such as `list`, will usually not call `Model#commitAddressBook()`, `Model#undoAddressBook()` or `Model#redoAddressBook()`. Thus, the `addressBookStateList` remains unchanged.



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:



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.

{more aspects and alternatives to be added}

[Proposed] Data archiving

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

Documentation, logging, testing, configuration, dev-ops

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

Appendix: Requirements

Product scope

Target user profile:
Healthcare administrators in clinics

Value proposition:
HealthSync allows healthcare staff to efficiently organize patient details and key contacts in one unified platform. With quick access to updated information, administrators can easily connect with medical staff and patients' families, ensuring smooth communication and prompt action, especially when managing recovery progress and treatment schedules.

User Stories

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

Priority	As a ...	I want to ...	So that I can...
`* * *`	healthcare administrator	quickly locate patient emergency contacts	promptly notify their next of kin in life-threatening medical situations
`* * *`	healthcare administrator	accurately upload and store hospital patients' contact details	ensure their information is reliably available for communication and record-keeping
`* * *`	healthcare professional	quickly identify a patient's diseases and all	understand the patient's condition and give better advice
`* * *`	new user	have sample data to work with	understand how to use the application
`* * *`	healthcare administrator	efficiently update and manage patient schedules	ensure that doctor's schedules are efficiently maintained and there is no overlap between patients
`* *`	healthcare administrator	undo the last operation	recover from unintentional modifications or deletions
`* *`	healthcare administrator	redo the last undone operation	reverse an undo operation if it was done in error
`*`	user with many persons in the address book	sort persons by name	locate a person easily

Priority	As a …	I want to …	So that I can…
`* * *`	healthcare administrator	identify patient's allergies by tags	give appropriate medicines and prevent any cases of malpractice

Use Cases

(For all use cases below, the **System** is `HealthSync` and the **Actor** is the `Healthcare Administrator`, unless specified otherwise.)

UC01 - Add Patient

Main Success Scenario (MSS):

1. Healthcare Administrator chooses to add a new patient.
2. HealthSync requests patient details.
3. Healthcare Administrator enters the requested details.
4. HealthSync asks the Healthcare Administrator to confirm the addition.
5. Healthcare Administrator confirms the addition details.
6. HealthSync successfully adds the patient.
7. Use case ends.

Extensions:

- **3a.** Healthcare Administrator enters incomplete or invalid details.
  - **3a1.** HealthSync highlights the errors and requests corrections.
  - **3a2.** Healthcare Administrator provides corrected details.
  - Steps **3a1-3a2** are repeated until all details are valid.
  - Use case resumes from step **4**.
- **3b.** A patient with the same unique identifier (e.g., National ID, Patient ID) already exists.
  - **3b1.** HealthSync notifies the Healthcare Administrator of the duplication.
  - **3b2.** Healthcare Administrator can choose to:
    - Update the existing record (transition to **UC02 - Edit Patient Details**).
    - Cancel the addition (Use case ends).
- **3c.** Healthcare Administrator chooses to assign an emergency contact while adding a patient.
  - **3c1.** HealthSync requests emergency contact details.
  - **3c2.** Healthcare Administrator provides emergency contact details.
  - Use case resumes from step **5**.
- **4a.** Healthcare Administrator chooses to cancel the operation.
  - **4a1.** Healthcare Administrator confirms the cancellation.
  - Use case ends.

UC02 - Edit Patient Details

Main Success Scenario (MSS):

1. Healthcare Administrator requests to edit a patient's details.
2. HealthSync prompts the user to enter the following details:
   - Index of the patient to be edited.
   - New information for the field to be updated.
3. Healthcare Administrator inputs the new information.
4. HealthSync asks for confirmation.
5. Healthcare Administrator confirms the edit.
6. HealthSync updates the patient's details successfully.
7. Use case ends.

Extensions:

- **2a.** The given index is invalid (out of range).
  - **2a1.** HealthSync informs the user of the invalid index.

    - Use case resumes from step **2**.
- **2b.** The entered details are invalid (e.g., phone number contains letters).
    - **2b1.** HealthSync displays an error message.
    - **2b2.** Healthcare Administrator corrects the details and resubmits.
    - Use case resumes from step **2**.
- **6a.** Healthcare Administrator chooses to cancel the edit.
    - **6a1.** Healthcare Administrator confirms the cancellation.
    - Use case ends.

UC03 - Delete a Patient

Main Success Scenario (MSS):

1. Healthcare Administrator requests to delete a patient's contact.
2. HealthSync prompts the user to enter the index of the patient to be deleted.
3. Healthcare Administrator provides the patient index.
4. HealthSync asks for confirmation.
5. Healthcare Administrator confirms the deletion.
6. HealthSync deletes the patient's contact from the system.
7. HealthSync confirms successful deletion.
8. Use case ends.

Extensions:

- **2a.** The given index is invalid (out of range or does not exist).
    - **2a1.** HealthSync informs the user of the invalid index.
    - Use case resumes from step **2**.
- **4a.** Healthcare Administrator cancels the deletion.
    - **4a1.** HealthSync aborts the deletion process.
    - Use case ends.

UC04 - Update Patient Schedule

Main Success Scenario (MSS):

1. Healthcare Administrator requests to update a patient's appointment.
2. HealthSync prompts the user to enter:
    - Patient identifier (e.g., index, name, or ID).
    - New or modified appointment details (e.g., appointment date, time, doctor, location).
3. Healthcare Administrator inputs the updated appointment.
4. HealthSync asks for confirmation.
5. Healthcare Administrator confirms the update.
6. HealthSync updates the patient's appointment successfully.
7. Use case ends.

Extensions:

- **2a.** The provided patient identifier is invalid.
    - **2a1.** HealthSync informs the user.
    - Use case resumes from step **2**.
- **2b.** The entered appointment details are invalid (e.g., overlapping appointments, incorrect format).
    - **2b1.** HealthSync displays an error message.
    - **2b2.** Healthcare Administrator corrects the details and resubmits.
    - Use case resumes from step **2**.
- **4a.** Healthcare Administrator chooses to cancel the update.
    - **4a1.** Healthcare Administrator confirms the cancellation.
    - Use case ends.

UC05 - Add an Emergency Contact

Main Success Scenario (MSS):

1. Healthcare Administrator requests to add an emergency contact for a patient.
2. HealthSync prompts the user to enter:
    - Patient index.
    - Emergency contact's name.
    - Emergency contact's email.
    - Emergency contact's phone number.
    - Emergency contact's address.
    - Relationship to the patient.
3. Healthcare Administrator provides the details.
4. HealthSync asks for confirmation.
5. Healthcare Administrator confirms the addition.
6. HealthSync adds the emergency contact to the patient's record.
7. HealthSync confirms successful addition.
8. Use case ends.

Extensions:

- **2a.** The given patient index is invalid.
    - **2a1.** HealthSync informs the user.
    - Use case resumes from step **2**.
- **2b.** The entered details are invalid.
    - **2b1.** HealthSync displays an error message.
    - **2b2.** User corrects the details and resubmits.
    - Use case resumes from step **2**.
- **4a.** Healthcare Administrator chooses to cancel the operation.
    - **4a1.** Healthcare Administrator confirms the cancellation.
    - Use case ends.

UC06 - Sort Patients by Name

Main Success Scenario (MSS):

1. Healthcare Administrator requests to sort the list of patients by name.
2. HealthSync prompts the user to choose the sorting order:
    - Ascending (A-Z).
    - Descending (Z-A).
3. Healthcare Administrator selects the preferred sorting order.
4. HealthSync asks for confirmation.
5. Healthcare Administrator confirms the sorting.
6. HealthSync sorts the patient list accordingly.
7. HealthSync displays the sorted patient list.
8. Use case ends.

Extensions:

- **4a.** Healthcare Administrator chooses to cancel the sorting request.
    - **4a1.** HealthSync aborts the sorting process.
    - Use case ends.

Non-Functional Requirements

- Should work on any mainstream OS as long as it has **Java 17 or above** installed.
- Should be able to handle up to **50 concurrent patients** without noticeable sluggishness in performance for typical use.
- A user with **above-average typing speed** for regular English text (i.e., not code or system admin commands) should be able to accomplish most tasks **faster using commands than with the mouse**.
- **HealthSync must comply with relevant healthcare data protection regulations**, such as **PDPA**.
- The architecture should support **modular extensions**, allowing for additional features (e.g., appointment scheduling, integration with electronic health records).

Appendix: 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.

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 a person

1. Deleting a person while all persons are being shown
    1. Prerequisites: List all persons using the `list` command. Multiple persons 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 person 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 … }*

Glossary

1. **Mainstream OS**: Windows, Linux, Unix, MacOS
2. **Private contact detail**: A contact detail that is not meant to be shared with others