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}

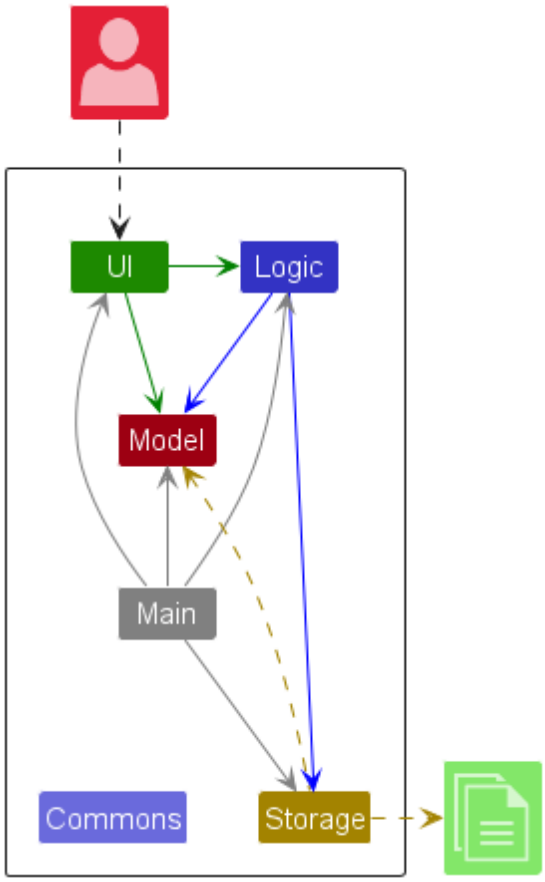
Setting up, getting started

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

Design

> 💡 **Tip:** The `.puml` files used to create diagrams in this document `docs/diagrams` folder. Refer to the *PlantUML Tutorial* at se-edu/guides to learn how to create and edit diagrams.

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.

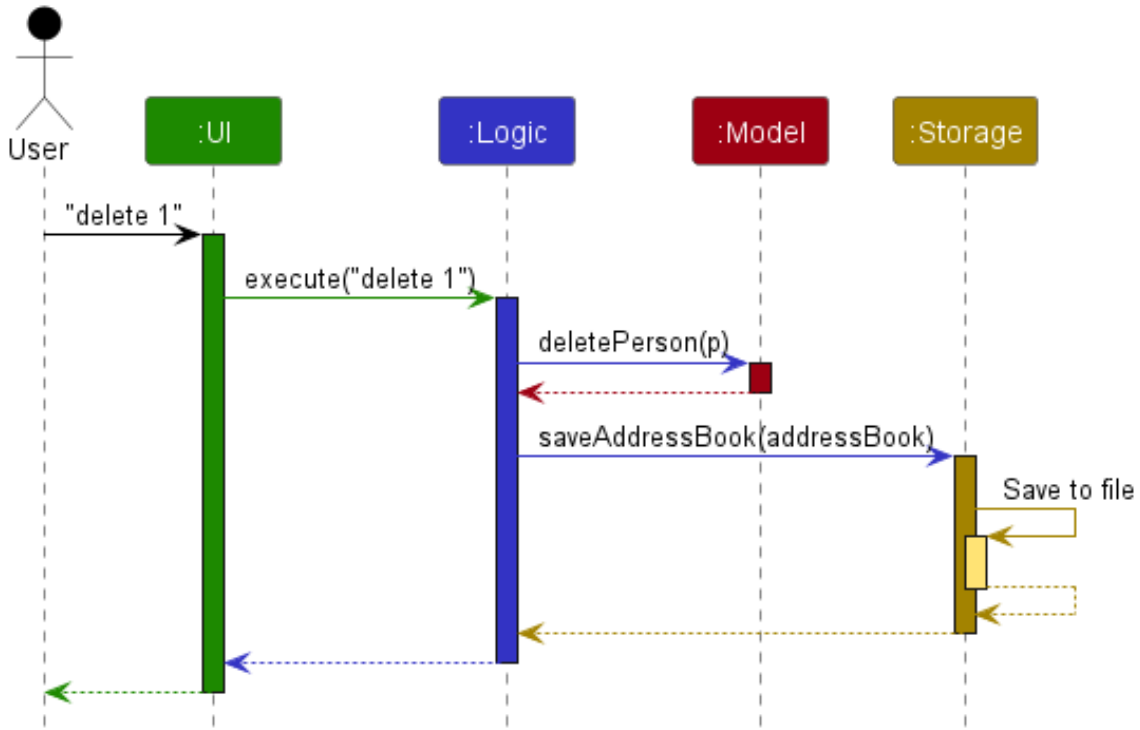
The bulk of the app's work is done by the following four components:

- `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

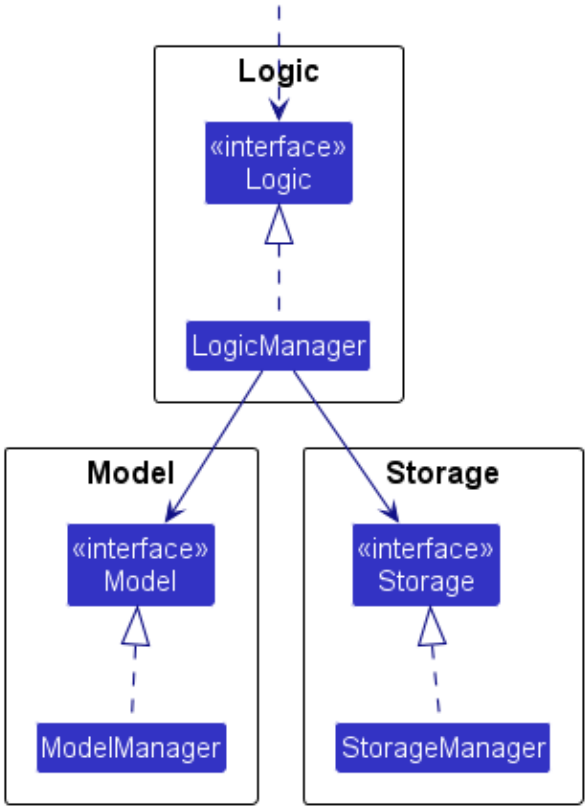
The *Sequence Diagram* below shows how the components interact with each other for the scenario where the user issues the command `delete 1`.



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.

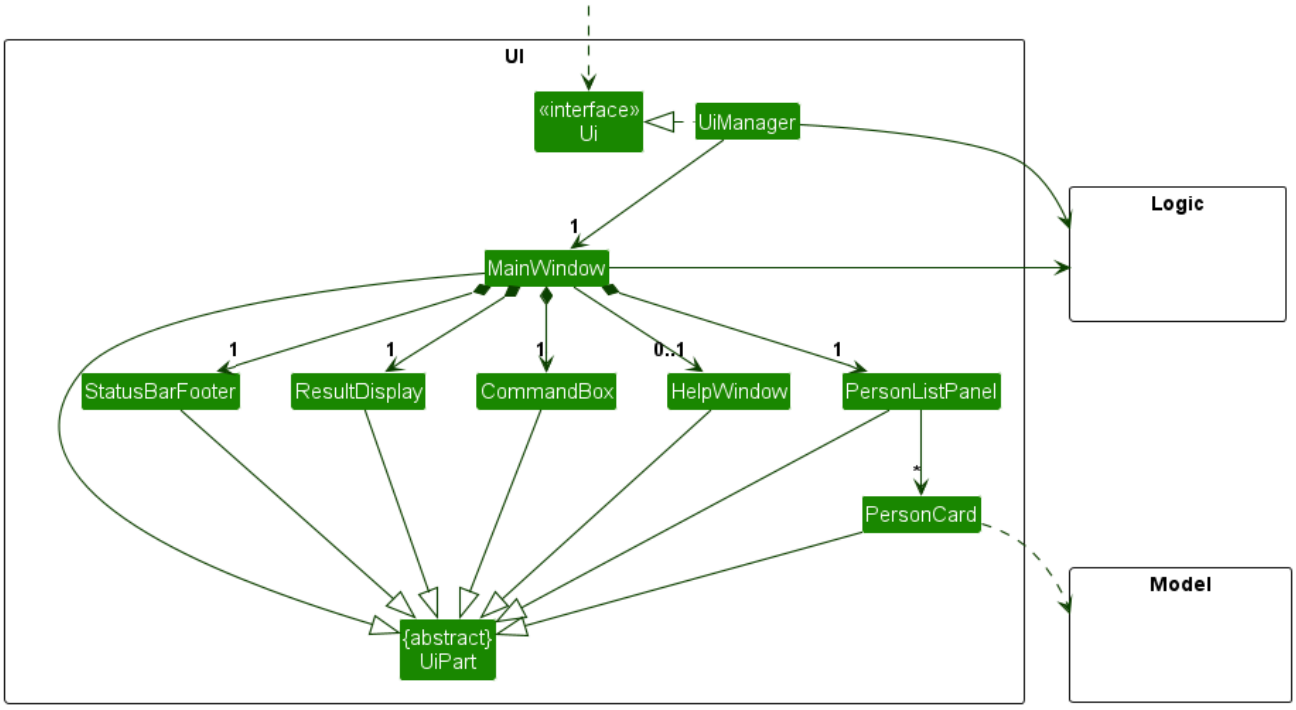
For example, the `Logic` component defines its API in the `Logic.java` interface and implements its functionality using the `LogicManager.java` class which follows the `Logic` interface. Other components interact with a given component through its interface rather than the concrete class (reason: to 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`

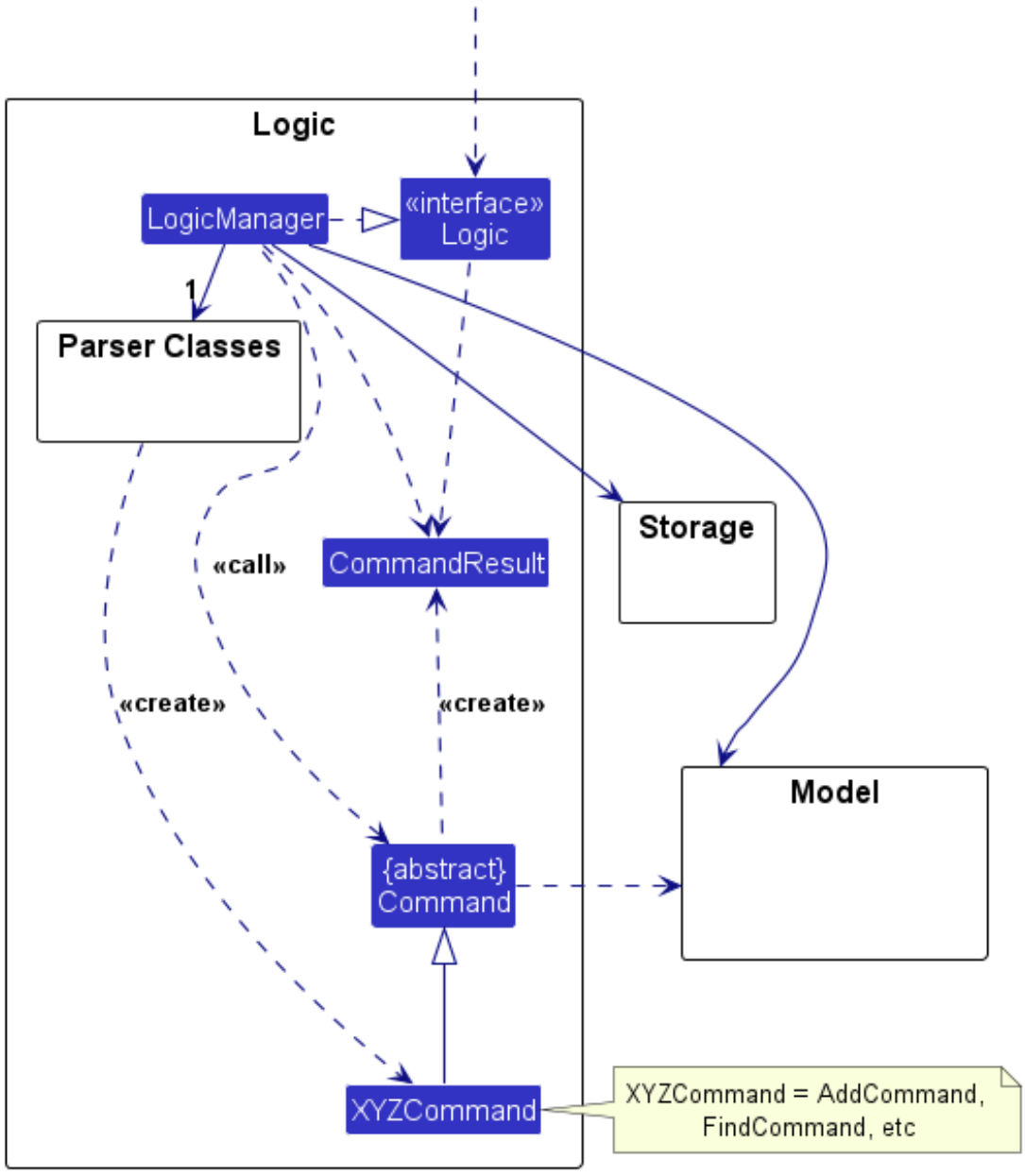
The `UI` component,

- 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`.

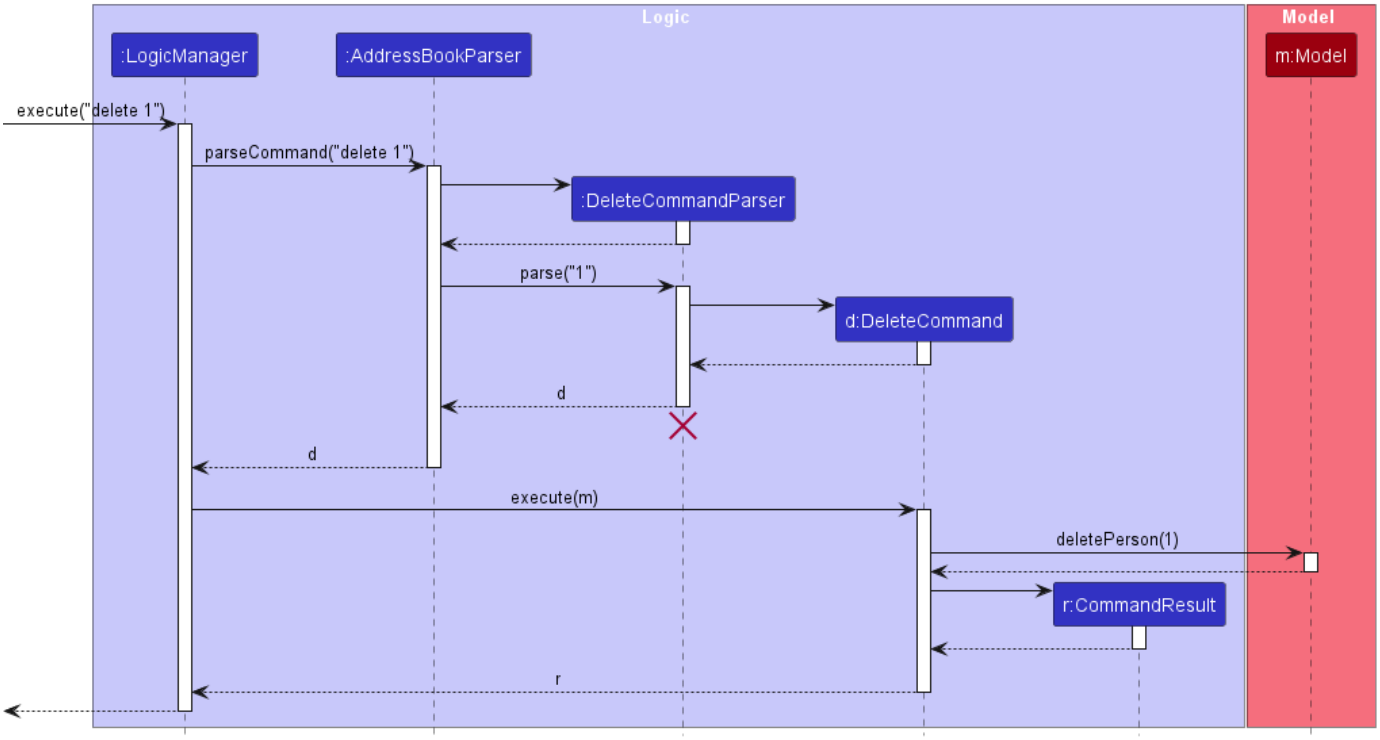
Logic component

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.

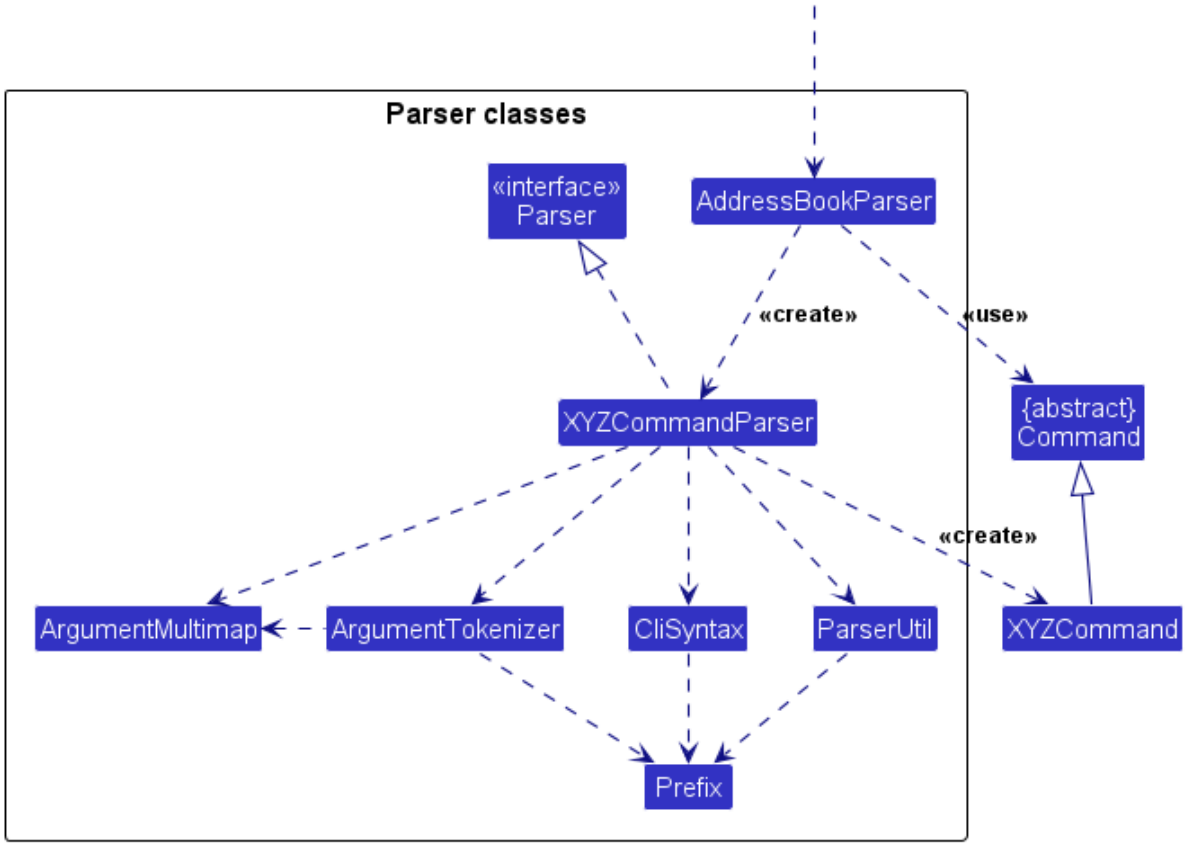


ℹ️ **Note:** The lifeline for `DeleteCommandParser` should end at the destroy marker (X) but due to a limitation of PlantUML, the lifeline continues till the end of diagram.

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:

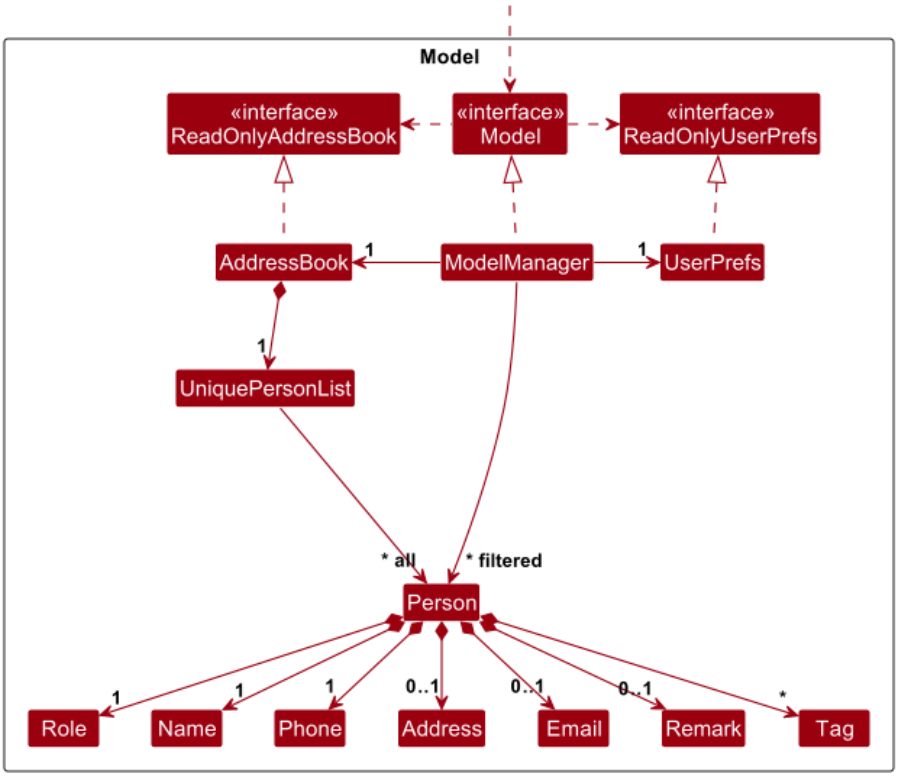


How the parsing works:

- 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` , `FindByNameCommandParser` ...) inherit from the `Parser` interface so that they can be treated similarly where possible e.g, during testing.

Model component

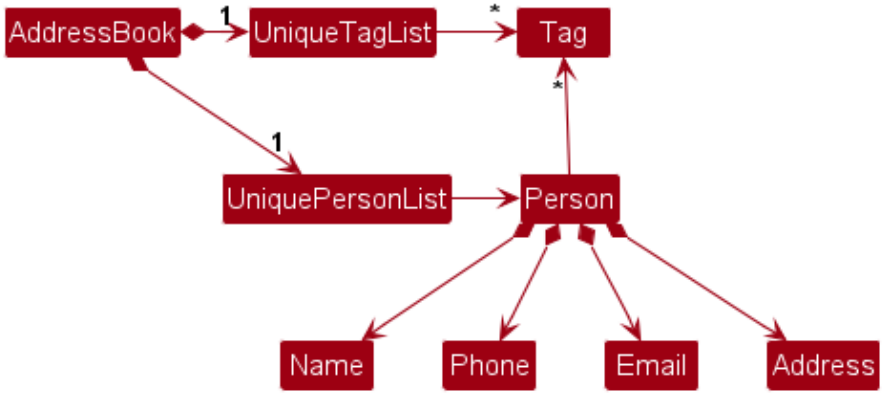
API : `Model.java`



The `Model` component,

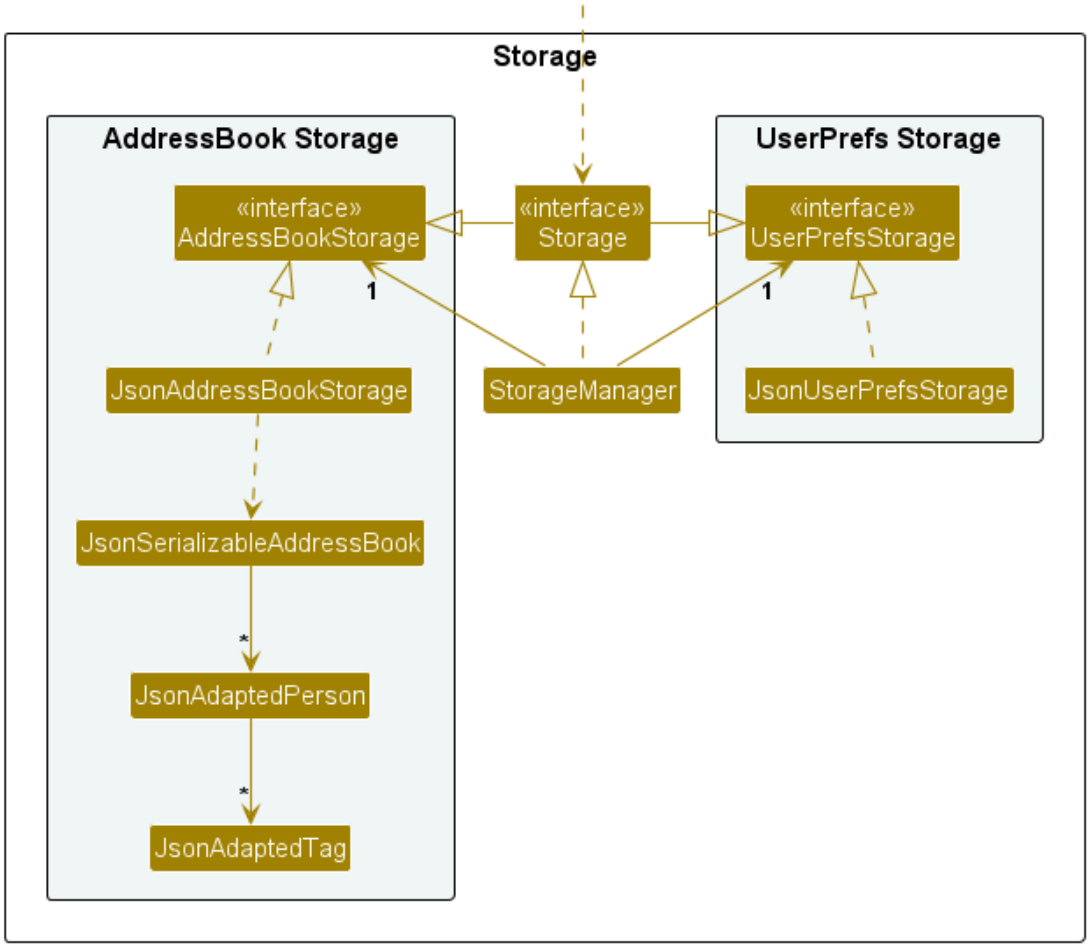
- stores the address book data i.e., all `Person` objects (which are contained in a `UniquePersonList` object).
- stores the currently 'selected' `Person` objects (e.g., results of a search query) as a separate *filtered* list which is exposed to outsiders as an unmodifiable `ObservableList<Person>` 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)

ℹ️ **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

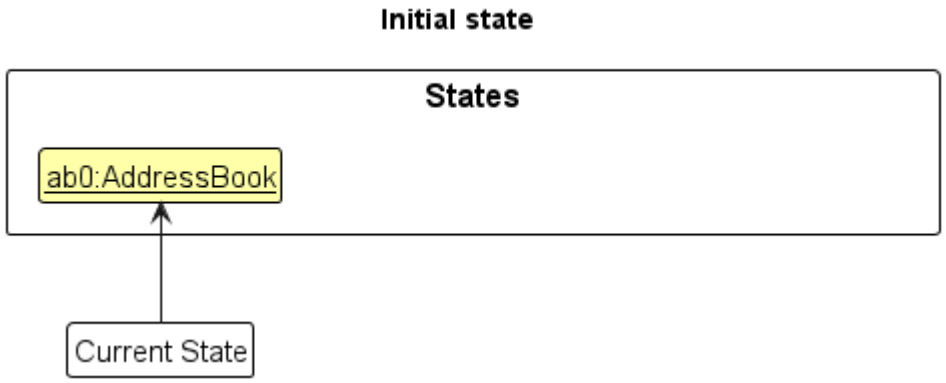
The proposed undo/redo mechanism is facilitated by `VersionedAddressBook`. It extends `AddressBook` with an undo/redo history, stored internally as an `addressBookStateList` and `currentStatePointer`. Additionally, it implements the following operations:

- `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.

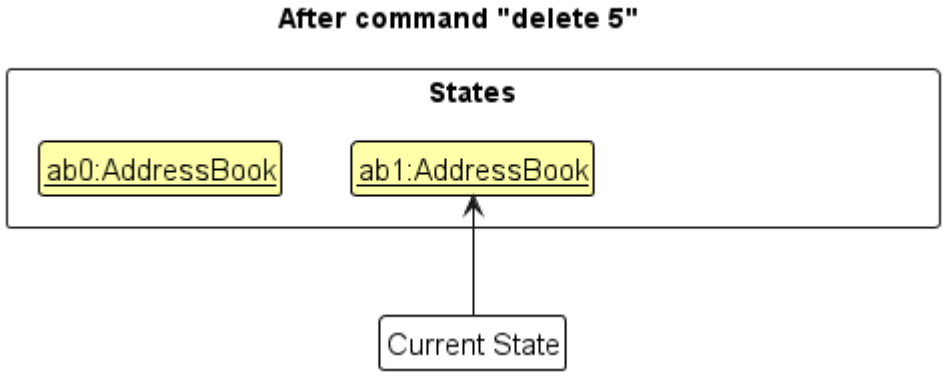
These operations are exposed in the `Model` interface as `Model#commitAddressBook()`, `Model#undoAddressBook()` and `Model#redoAddressBook()` respectively.

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

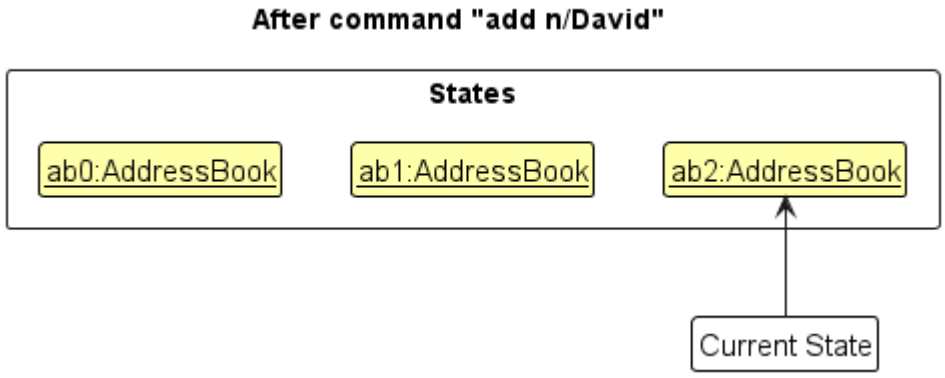
Step 1. The user launches the application for the first time. The `VersionedAddressBook` will be initialized with the initial address book state, and the `currentStatePointer` pointing to that single address book state.

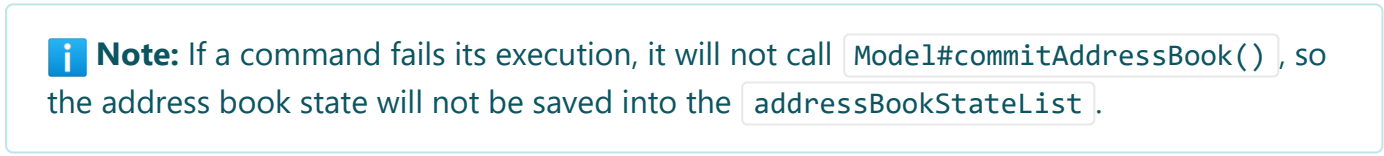


Step 2. The user executes `delete 5` command to delete the 5th person in the address book. The `delete` command calls `Model#commitAddressBook()`, causing the modified state of the address book after the `delete 5` command executes to be saved in the `addressBookStateList`, and the `currentStatePointer`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

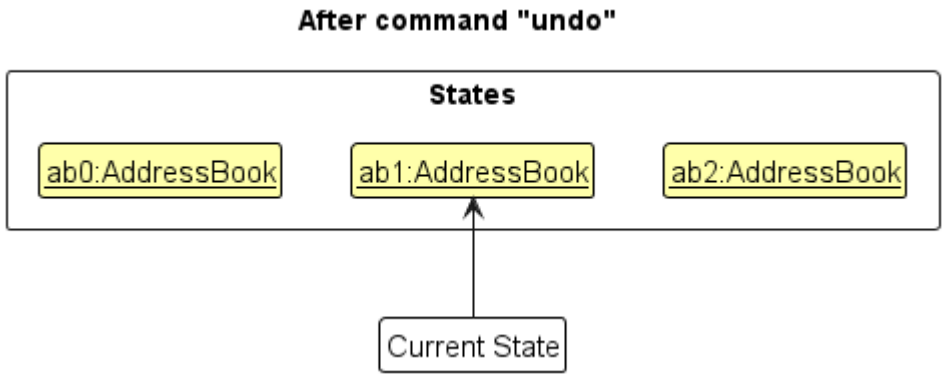


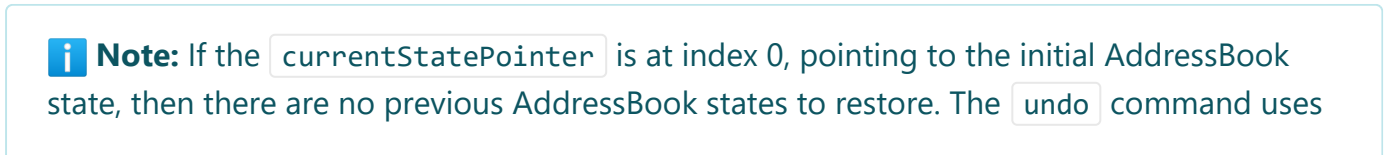
Step 3. The user executes `add n/David ...` to add a new person. The `add` command also calls `Model#commitAddressBook()`, causing another modified address book state to be saved into the `addressBookStateList`.





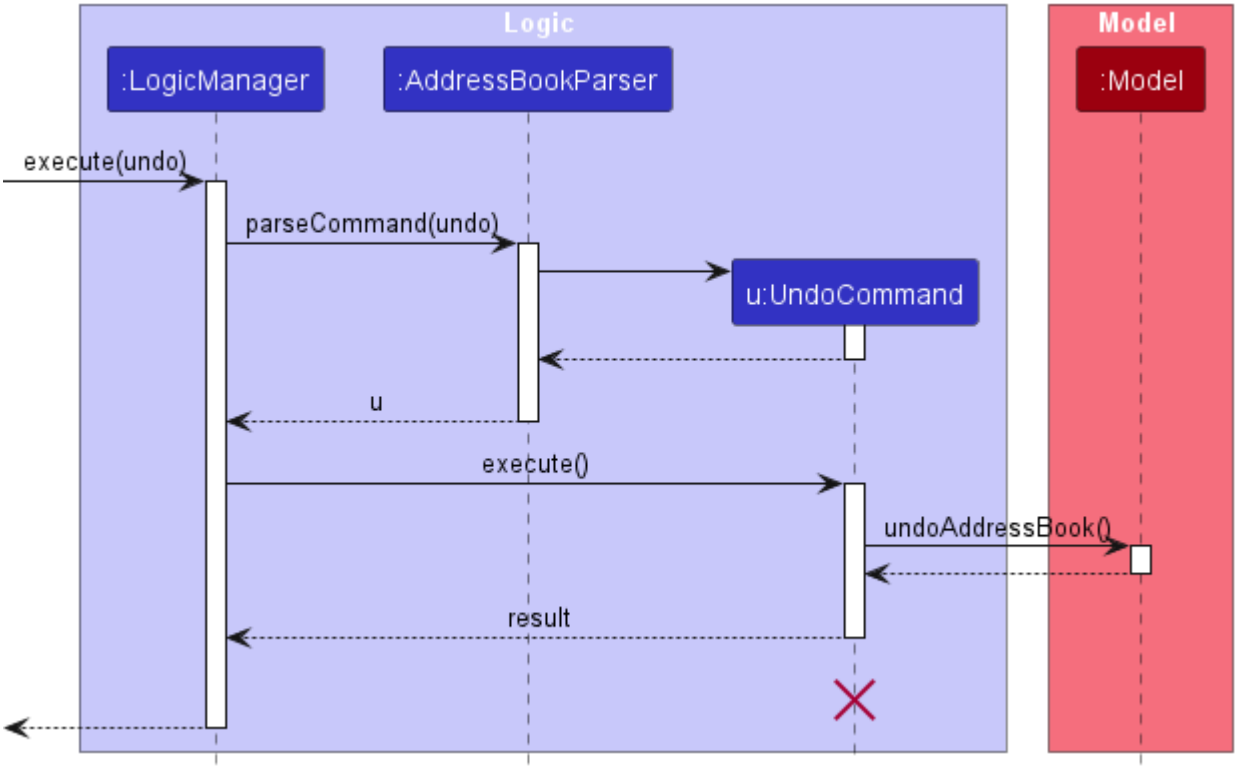
Step 4. The user now decides that adding the person was a mistake, and decides to undo that action by executing the `undo` command. The `undo` command will call `Model#undoAddressBook()`, which will shift the `currentStatePointer` once to the left, pointing it to the previous address book state, and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





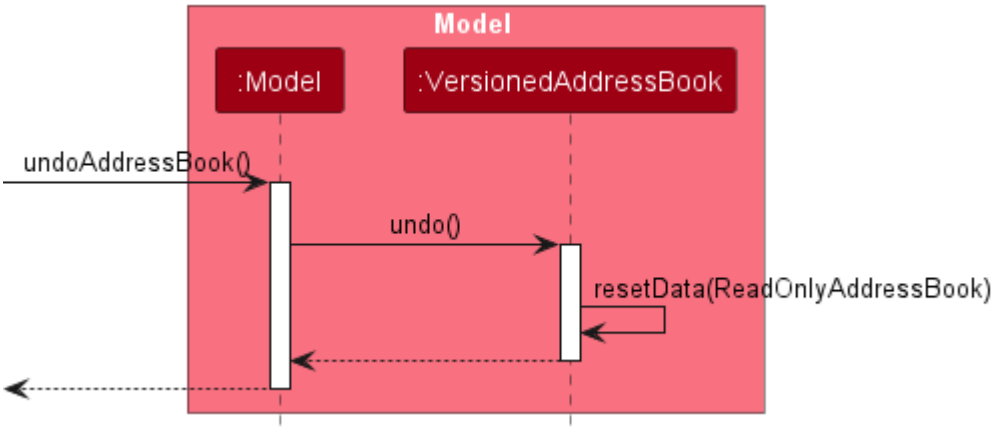
`Model#canUndoAddressBook()` to check if this is the case. If so, it will return an error to the user rather than attempting to perform the undo.

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



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.

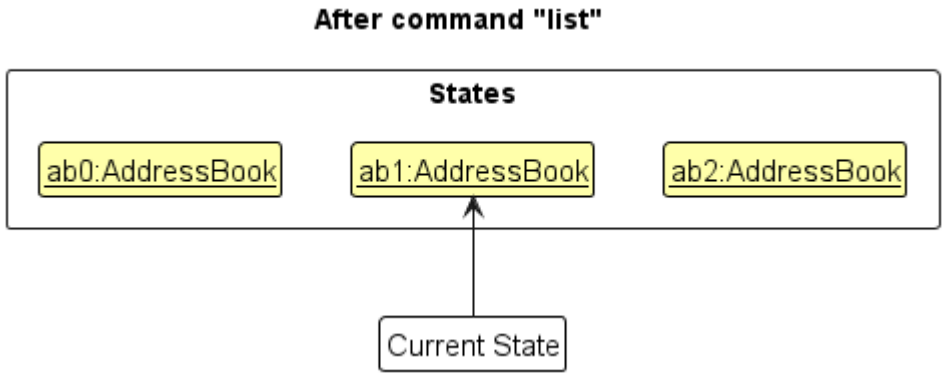
Similarly, how an undo operation goes through the `Model` component is shown below:



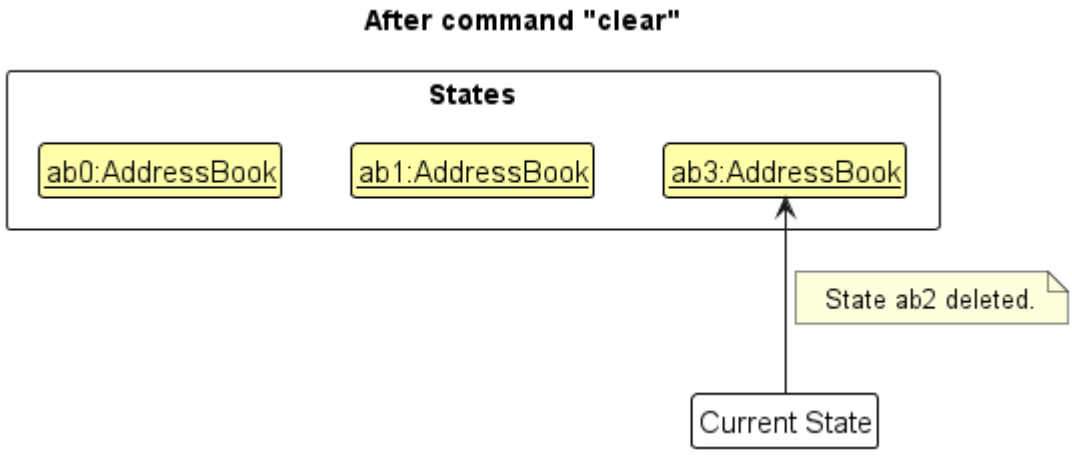
The `redo` command does the opposite — it calls `Model#redoAddressBook()`, which shifts the `currentStatePointer` once to the right, pointing to the previously undone state, and restores the address book to that state.

ℹ️ **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.

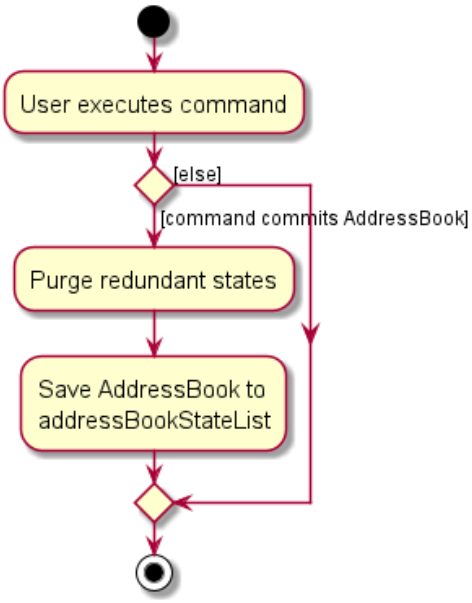
Step 5. The user then decides to execute the command `list`. Commands that do not modify the address book, such as `list`, will usually not call `Model#commitAddressBook()`, `Model#undoAddressBook()` or `Model#redoAddressBook()`. Thus, the `addressBookStateList` remains unchanged.



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:

- Has a need to manage a significant number of contacts
- Prefers desktop apps over other types of applications
- Can type fast and prefers typing to mouse interactions
- Is comfortable using command-line interface (CLI) apps for efficiency

Value proposition: Our product streamlines patient coordination by providing instant access to patient details, tagging patients to assigned doctors or caretakers, and tracking guardian or next-of-kin information. It also keeps contact information for doctors, nurses, and staff up-to-date, ensuring faster communication, improved workflows, and enhanced patient care.

User stories

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

| Priority | As a … | I want to … | So that I can… | |————-| ——————————— | ——————————— | ——————————————————————————- | | `* * *` | Patient Care Coordinator | add a new patient’s contact information | easily store and access important patient details for future reference and communications | | `* * *` | Patient Care Coordinator | delete patient’s contact information | remove obsolete patient data to keep the address book organized and clean | | `* * *` | Patient Care Coordinator | see a list of assigned patients when logged in | quickly manage and retrieve patient details without navigating multiple screens | | `* * *` | Patient Care Coordinator | add contact details for doctors, nurses, and other medical staff | easily store and access important staff details for future reference and communications | | `* * *` | Patient Care Coordinator | delete contact details for doctors, nurses, and other medical staff | remove obsolete staff data to keep the address book organized and clean | | `* * *` | Patient Care Coordinator | see contact details for doctors, nurses, and other medical staff | have immediate access to accurate contact information for quick outreach and coordination | | `* * *` | Patient Care Coordinator | tag a patient to a primary doctor | keep track of which medical professional is assigned to each patient for better organization and ensure that the right medical professional is notified for every patient | | `* * *` | Patient Care Coordinator | save contact

details in a file locally | ensure patient details are preserved even after the application is closed | | `* * *` | Patient Care Coordinator | load contact details from a local file | load patient details at startup, preventing the need to re-enter information | | `* *` | Patient Care Coordinator | search for a specific contact by name, role, etc. | quickly locate the right person or organization when urgent communication is needed | *{More to be added}*

Use Cases

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

Use case: Add a new patient's contact information

Main Success Scenario (MSS)

1. User requests to add a new patient's contact information.
2. ACaringBook validates the input details.
3. ACaringBook stores the patient's details.
4. ACaringBook confirms the successful addition.

Use case ends.

Extensions

- 2a. The input format is incorrect.
  - 2a1. ACaringBook shows an error message.
  - 2a2. User re-enters the correct details.
  - Use case resumes at step 2.
- 3a. The patient already exists in the ACaringBook.
  - 3a1. ACaringBook prompts the user to either update, delete, or keep both entries.
  - 3a2. User makes a choice and ACaringBook proceeds accordingly.
  - Use case resumes at step 4.

Use case: Delete a patient's contact information

MSS

1. User requests to list patients.
2. ACaringBook shows a list of patients.

3. User requests to delete a specific patient in the list.
4. ACaringBook deletes the patient and confirms the deletion.

Use case ends.

Extensions

- 2a. The list is empty.
  - Use case ends.
- 3a. The given index is invalid.
  - 3a1. ACaringBook shows an error message.
  - Use case resumes at step 2.

Use case: View list of patients

MSS

1. User requests to view a list of patients.
2. ACaringBook displays all patients in a tabular format.

Use case ends.

Extensions

- 2a. No patients are found in the ACaringBook.
  - 2a1. ACaringBook displays “No patient data is found.”
  - Use case ends.
- 2b. Patient data is corrupted.
  - 2b1. ACaringBook displays “Patient data corrupted!”
  - Use case ends.

Use case: Add healthcare staff contact

MSS

1. User requests to add a new healthcare staff contact.
2. ACaringBook validates the input details.
3. ACaringBook stores the staff contact.
4. ACaringBook confirms the successful addition.

Use case ends.

Extensions

- 2a. The input format is incorrect.
  - 2a1. ACaringBook shows an error message.
  - 2a2. User re-enters the correct details.
  - Use case resumes at step 2.
- 3a. The staff contact already exists in the ACaringBook.
  - 3a1. ACaringBook rejects the entry and displays an error message: "Duplicate entry. This contact already exists."
  - Use case ends.

Use case: Delete healthcare staff contact

MSS

1. User requests to delete a healthcare staff contact.
2. ACaringBook validates the input staff name.
3. ACaringBook checks if the staff exists.
4. ACaringBook deletes the contact.
5. ACaringBook confirms the successful deletion.

Use case ends.

Extensions

- 2a. The input staff name is missing.
  - 2a1. ACaringBook shows an error message: "Error: Missing required fields. Format: delete staff NAME."
  - 2a2. User re-enters the correct staff name.
  - Use case resumes at step 2.
- 3a. The staff member with the provided name does not exist.
  - 3a1. ACaringBook shows an error message: "Error: Staff contact not found. Name: John_Doe does not exist."
  - Use case ends.
- 4a. An error occurs while deleting the staff contact.
  - 4a1. ACaringBook shows an error message: "Error: Failed to delete the staff contact. Please try again."
  - Use case ends.

{More to be added}

Non-Functional Requirements

1. Should work on any *mainstream OS* as long as it has Java `17` or above installed.
2. Should be able to hold up to 1000 contacts 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.
4. Should only allow use by a single user (patient care coordinator).
5. Should return search results within 0.5 seconds to ensure quick lookup.
6. Should consume less than 200MB of RAM and low CPU usage on standard hardware.
7. Data of the patients should be in a text file that is human editable.
8. Should not require any installation by the user.
9. Should be able to load data without internet connection.

Glossary

- **Mainstream OS**: Windows, Linux, Unix, MacOS
- **Private contact detail**: A contact detail that is not meant to be shared with others
- **Above average typing speed**: Typing speeds above the global average of 41.4 words per minute
- **Patient care coordinator**: A healthcare professional responsible for managing patient information and coordinating communication between medical staff and patients
- **Contact**: An entry representing a patient, doctor, nurse or medical staff member within ACaringBook
- **NOK (Next-of-Kin)**: Primary emergency contact for a patient

---

Appendix: Instructions for manual testing

Given below are instructions to test the app manually.

> ℹ️ **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 … }*