layout: page title: Developer Guide

- Table of Contents {:toc}

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

: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.

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` 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.

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),

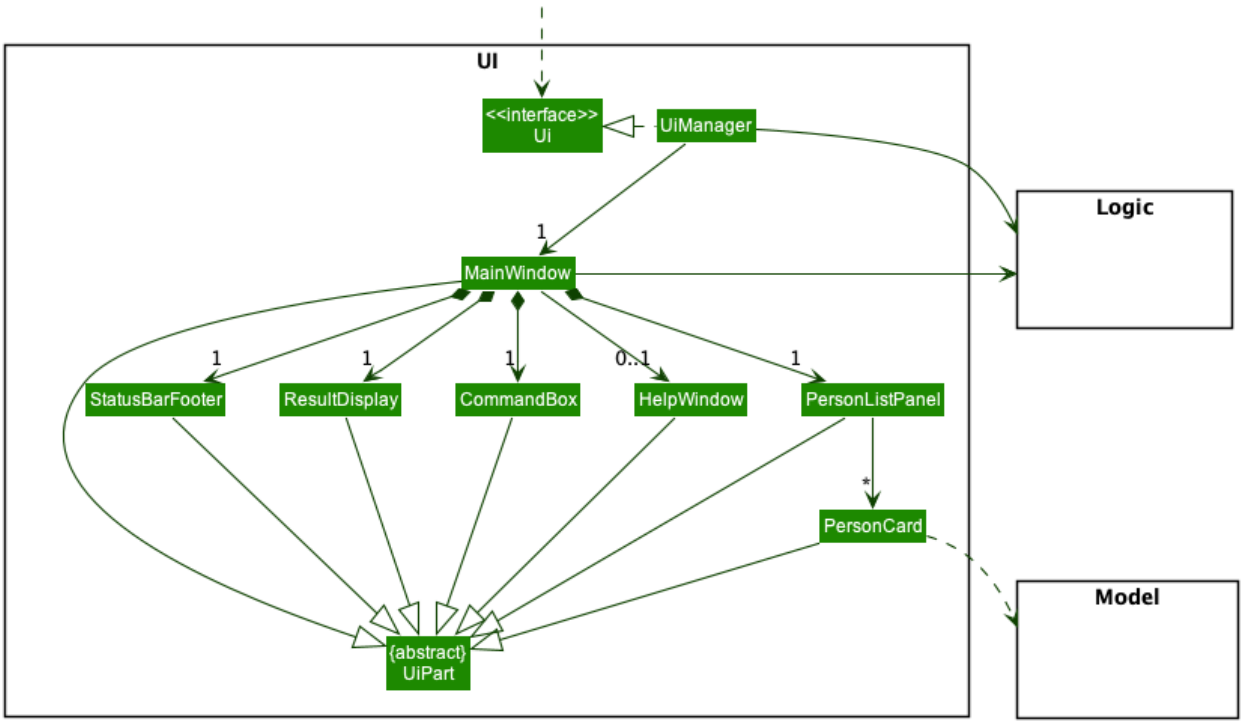
- defines its *API* in an `interface` with the same name as the Component.
- implements its functionality using a concrete `{Component Name}Manager` class (which follows the corresponding API `interface` mentioned in the previous point.

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

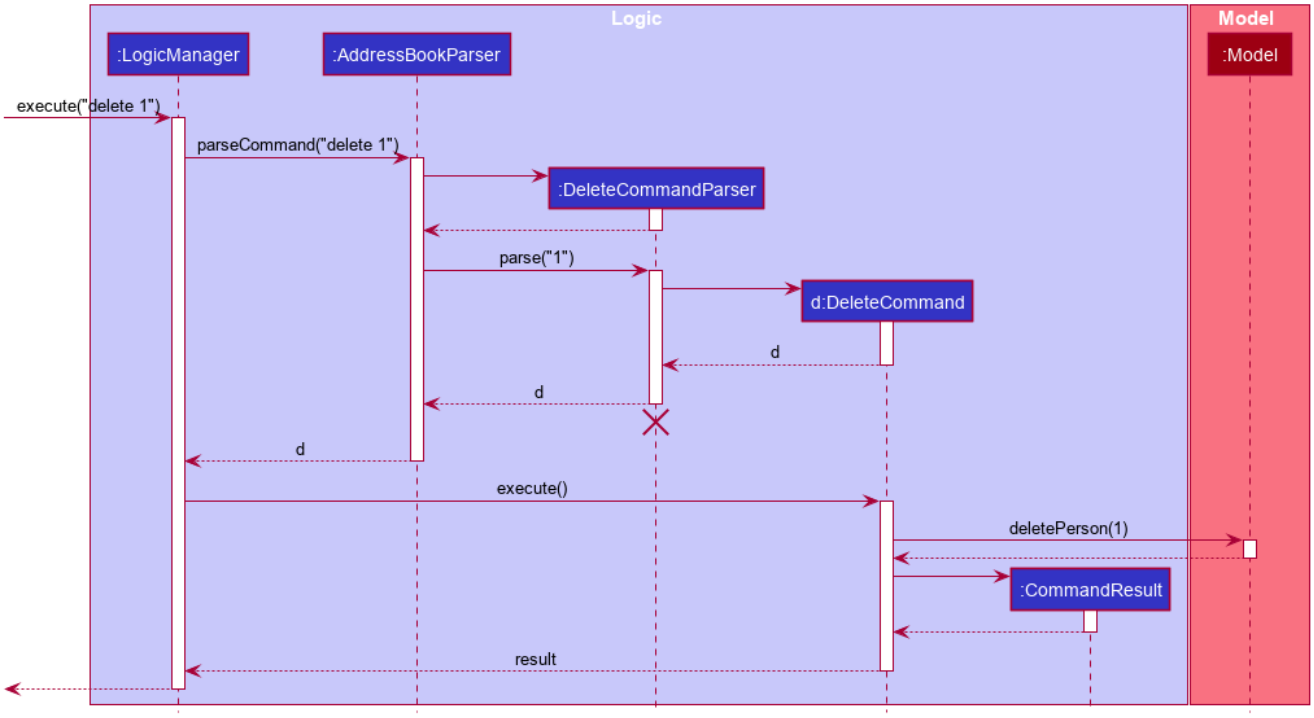
API : `Logic.java`

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

How the `Logic` component works:

1. When `Logic` is called upon to execute a command, it uses the `AddressBookParser` class to parse the user command.
2. This results in a `Command` object (more precisely, an object of one of its subclasses e.g., `AddCommand`) which is executed by the `LogicManager`.
3. The command can communicate with the `Model` when it is executed (e.g. to add a person).
4. The result of the command execution is encapsulated as a `CommandResult` object which is returned back from `Logic`.

The Sequence Diagram below illustrates the 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.

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)

: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` 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.addressbook.commons` package.

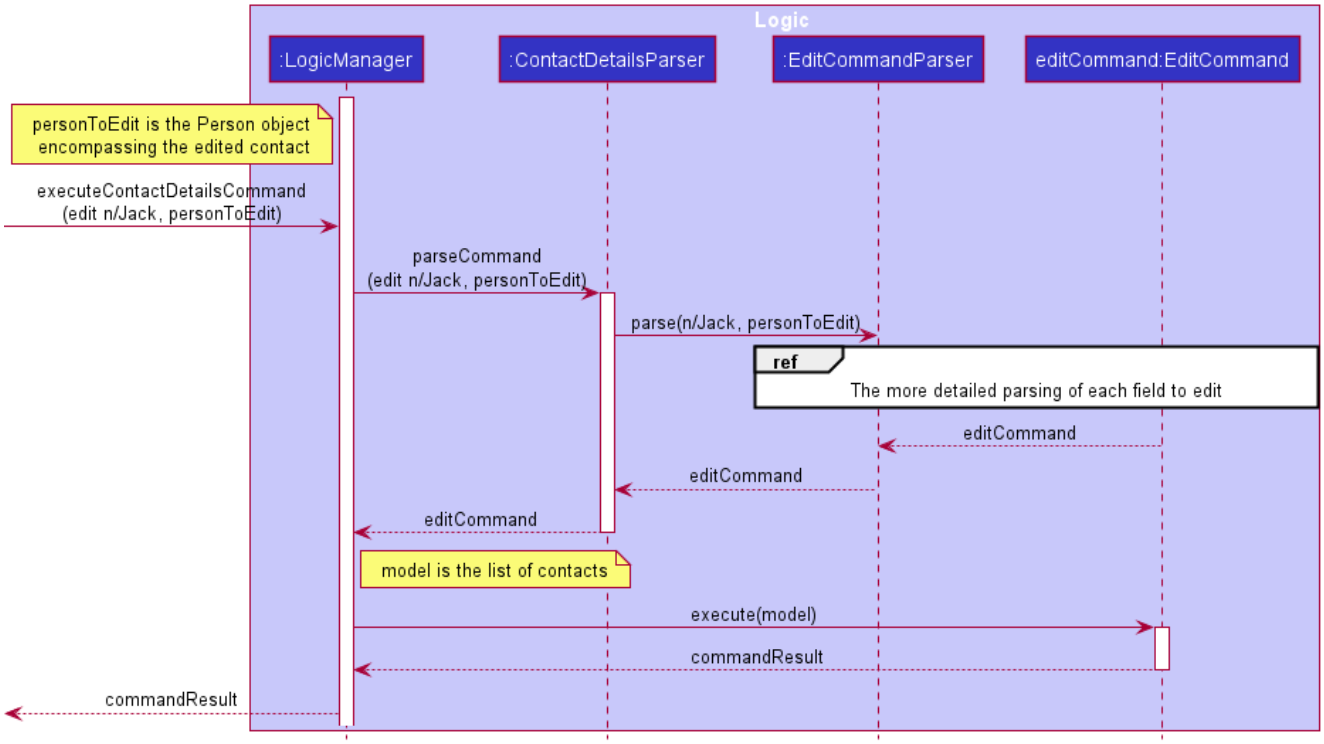
Implementation

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

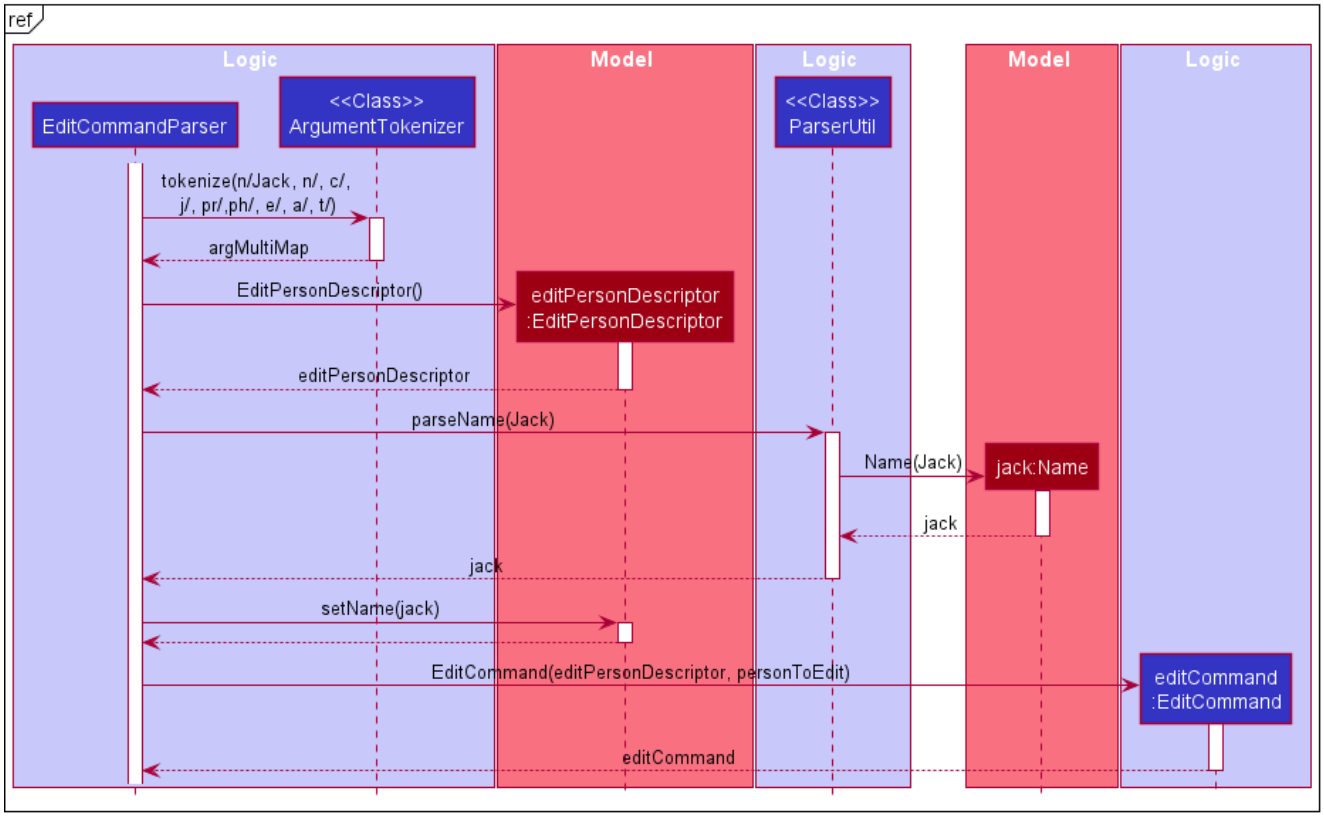
Edit feature

The edit mechanism is a feature used to change the details of the contacts. It is only allowed in the application after initiating an add command or view command and in other words, it is functional only in the contact details windows. It is facilitated mainly by the `ContactDetailsParser` , `EditCommandParser` and `EditCommand` classes.

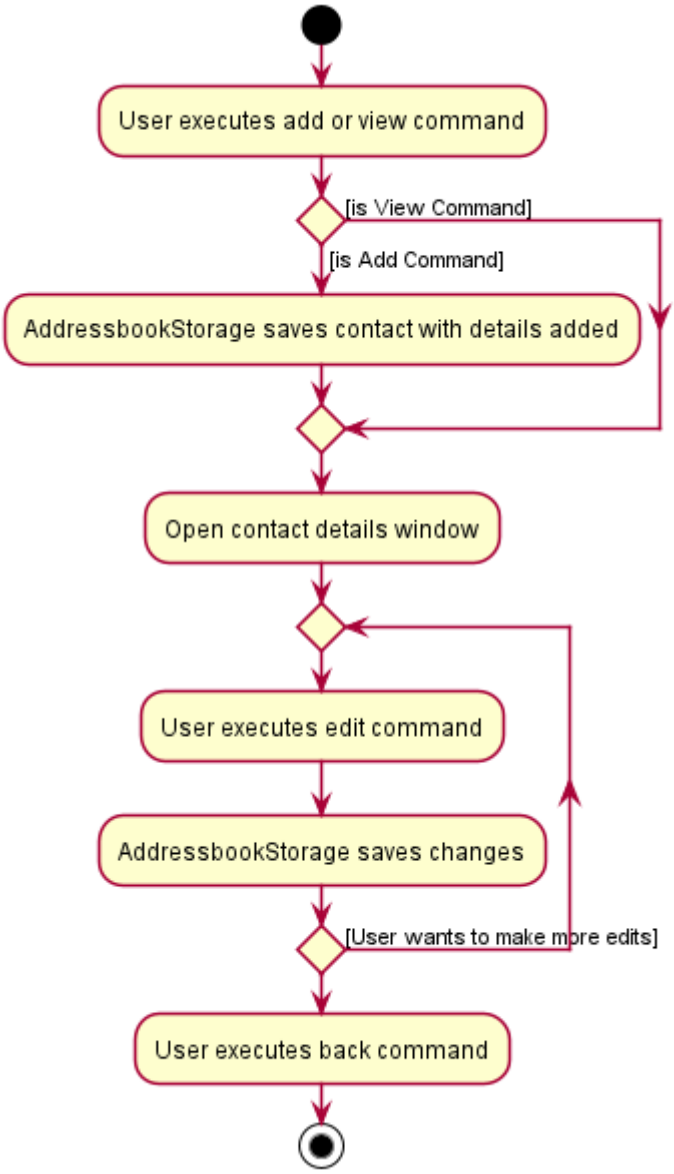
The following sequence diagram shows how the edit operation works:



- Here, the user executes an add command which takes in a new name input "Jack" which is tagged with a prefix "n/" for input type identification.
- If a user were to execute a view command instead, the only difference would be that the editing is done on an existing contact instead of a new one.



Below is an activity diagram summarising the possible paths for an edit command:



Design considerations:

Aspect: How edit saves:

- **Alternative 1 (current choice):** Each edit is saved immediately.
  - Pros: Prevents data loss from system crashes.
  - Cons: May have performance issues in terms of memory usage.
- **Alternative 2 :** All edits are saved only after executing a save command (not implemented feature).
  - Pros: Allows user to revert their changes
  - Cons: System crashes will not save the edits.

Delete fields feature

The **delete fields** feature can be used to delete fields stored for the contacts. This feature is also restricted to the Contact Details screen (in *edit* mode), which can be accessed after the *add* or *view* commands. It is mainly facilitated by the `ContactDetailsParser`, `DeleteFieldCommandParser` and `DeleteFieldCommand` classes.

Note: This feature is different from the **delete contacts** feature, which is only accessible on the Person Details screen (in *default* mode).

The *delete fields* operation works similar to the *edit* operation, except for:

1. the deletion functionality
2. the parsing of the command

The deletion functionality sets the fields to return to their initial empty values. It is illustrated in the code snippets below:

```
// Single-valued fields
if (argMultimap.getValue(PREFIX_COMPANY).isPresent()) {
    deleteFieldDescriptor.setCompany(null);
}
```

```
// Multi-valued fields
if (deleteFieldDescriptor.getNumbers().isPresent()) {
    Collection<String> numbersToBeDeleted = argMultimap.getAllValues(PREFIX_PHONE);

    requireNonNull(numbersToBeDeleted);

    if (CollectionUtil.hasEmptyString(numbersToBeDeleted)) {
        deleteFieldDescriptor.setNumbers(new HashMap<String, Phone>());
    } else {
        Map<String, Phone> numbers = new HashMap<>(deleteFieldDescriptor.getNumbers()
                .orElse(new HashMap<>()));
        numbersToBeDeleted.forEach(numbers::remove);
        deleteFieldDescriptor.setNumbers(numbers);
    }
}
```

Design considerations:

Since certain fields allow for multiple values to be stored, the user needs to specify the label of the value (or the value itself for non-labelled fields) they want to delete along with the field to be deleted for such fields.

Aspect: What happens when the user does not specify a label or value:

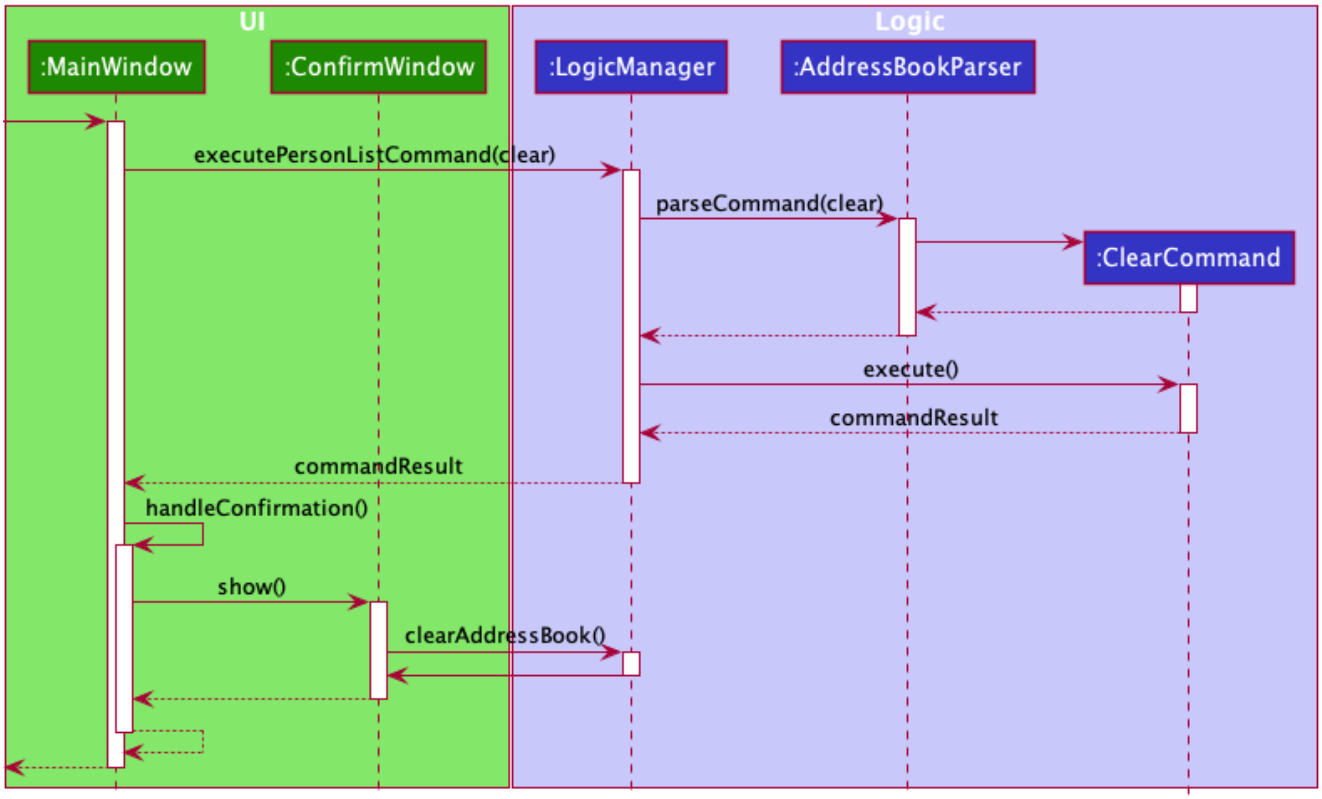
- **Alternative 1 (current choice):** Delete all the values stored for this field immediately.
  - Pros:
    - Seems to be the most intuitive approach.
    - Easier to implement.
    - Faster to execute command.
  - Cons:
    - User may have forgotten to mention the label or field, which could lead to unintended loss of data.
- **Alternative 2 :** Confirm that the user wants to delete all values for this field
  - Pros:
    - Allows user to cancel the command if it was unintentional.
  - Cons:
    - Slower to execute command.
    - Difficult to implement, since the current implementation does not store command history.

We picked *alternative 1* since the focus of our CLI app is on speed and efficiency. Additionally, *alternative 2* required a lot of changes to the existing implementation which would not be very helpful for executing other commands.

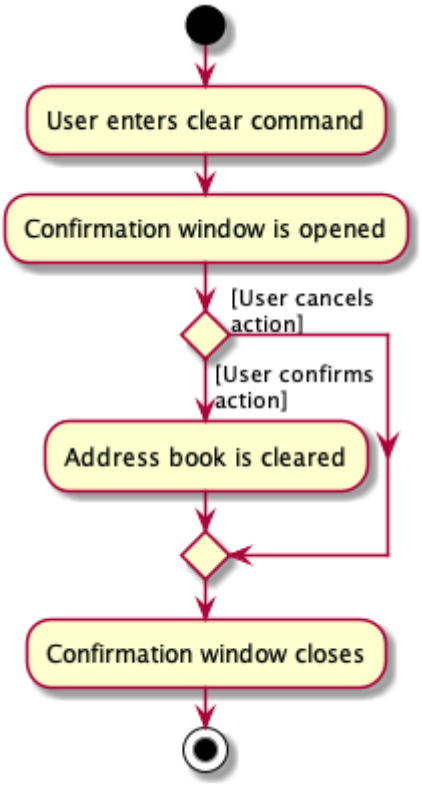
Clear address book feature

The **clear address book** feature can be used to delete all the contacts stored by the user and to start with a new address book. Since deleted data cannot be recovered, the app opens a pop-up window asking for confirmation that the user wants to delete all of their stored contacts.

The following sequence diagram shows how the clear operation works:



This activity diagram summarises the possible paths of executing the *clear* command:



View person feature

The view feature allows the user to view the full contact details of a specified person in the address book. The command is only available from the person list window,and is thus facilitated by the `AddressBookParser`, `ViewCommandParser`, and `ViewCommand`. Additionally, it implements the following operation:

- `MainWindow#LoadContactScreen(Person personToDisplay)` — Constructs and shows a `ContactDetailsPanel`, which displays the full details of the `Person` provided as argument.

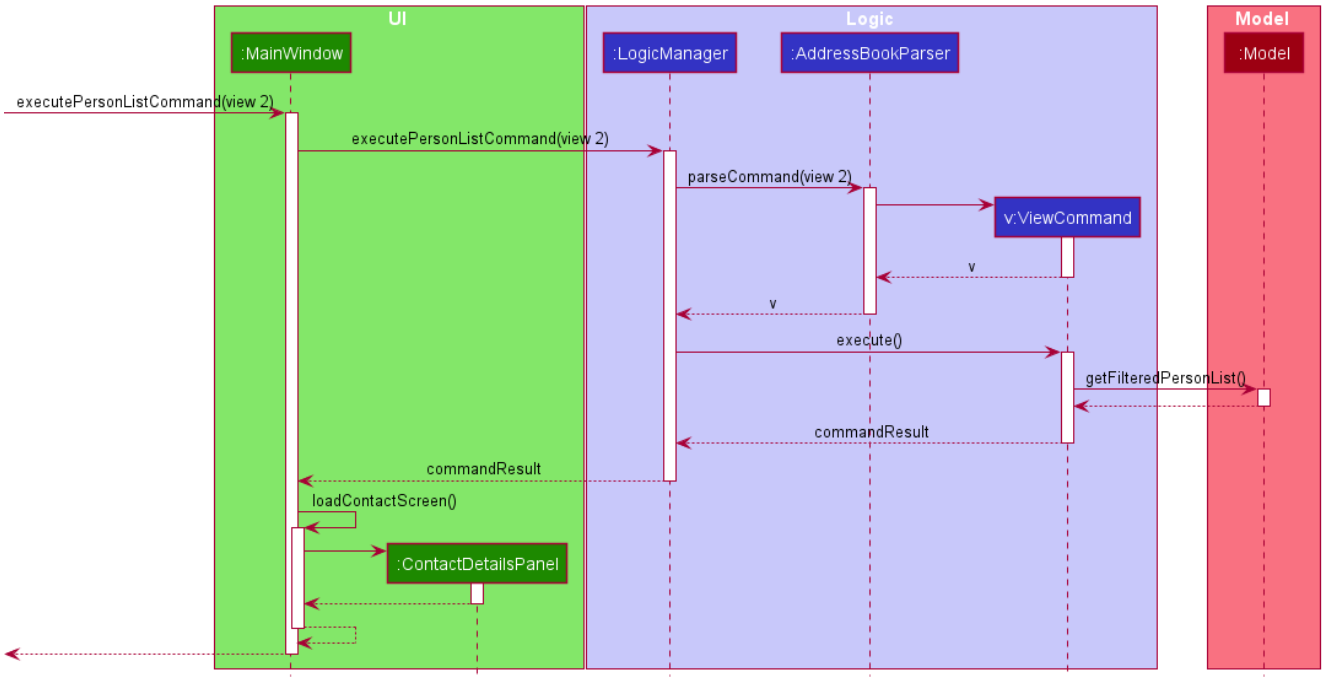
Given below is an example usage scenario and how the view mechanism behaves at each step.

Step 1. From the person list window, the user executes `view 2` to view the contact details of the second person in the address book. A `ViewCommand` is constructed with the index of the person to de displayed.

Step 2. The `ViewCommand` is executed, and the person that corresponds to the provided index is returned to the `MainWindow` inside a `CommandResult`.

Step 3. `MainWindow#loadContactScreen(Person personToDisplay)` is executed with the specified person passed as argument, which constructs and displays the respective `ContactDetailsPanel`.

The following sequence diagram shows how the view feature works:



Design considerations:

Aspect: Where to display a person's contact details:

- **Alternative 1:** Display all contact information in the person list screen.
  - Pros:
    - Easy to implement
    - Does not require the user to navigate to a new screen, which speeds up program use
  - Cons:
    - Clutters the person list screen with a lot of information for each person
- **Alternative 2 (current choice):** Navigate to a new screen to for contact information
  - Pros:
    - Greatly reduces clutter in the person list screen
    - Reduces the size of the person list, making it easier to scroll through
  - Cons:
    - Difficult to implement
    - Slower program use due to the addition of an additional navigation step

We chose alternative 2 because its benefit to the visual clarity of the address book and thus the ease of its use outweighs the cost of including an additional navigation step.

[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:

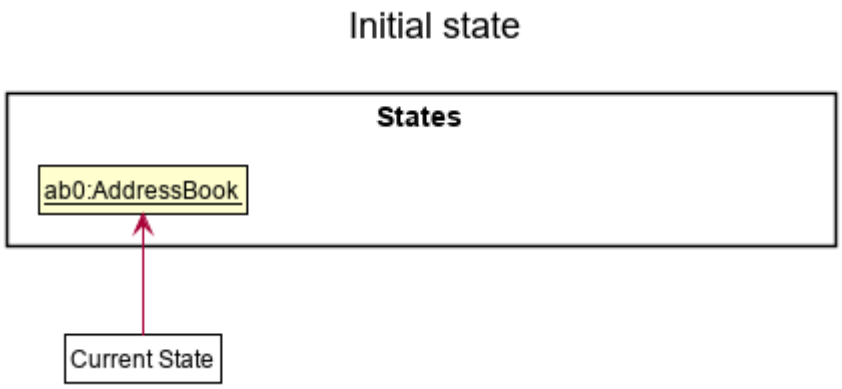
- `VersionedAddressBook#commit()` — Saves the current address book state in its history.
- `VersionedAddressBook#undo()` — Restores the previous address book state from its history.

- `VersionedAddressBook#redo()` — Restores a previously undone address book state from its history.

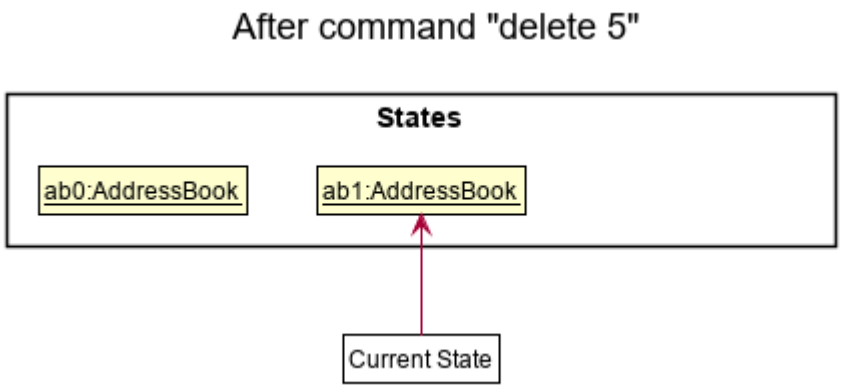
These operations are exposed in the `Model` interface as `Model#commitAddressBook()`, `Model#undoAddressBook()` and `Model#redoAddressBook()` respectively.

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.

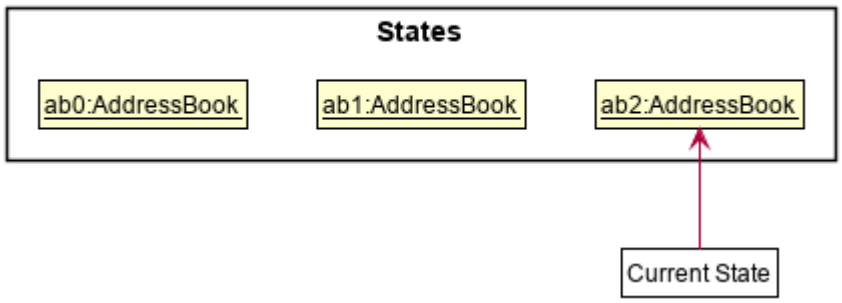


Step 2. The user executes `delete 5` command to delete the 5th person in the address book. The `delete` command calls `Model#commitAddressBook()`, causing the modified state of the address book after the `delete 5` command executes to be saved in the `addressBookStateList`, and the `currentStatePointer`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

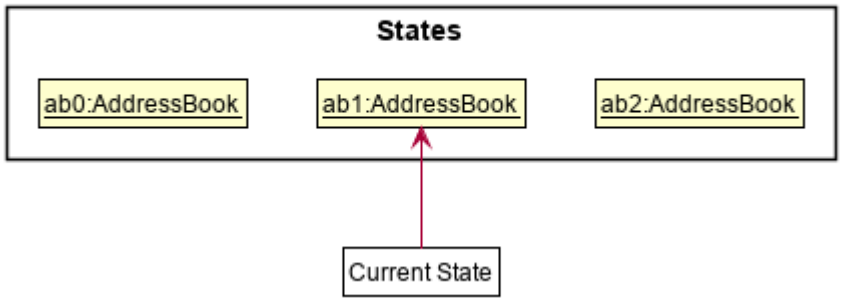
After command "add n/David"



: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`.

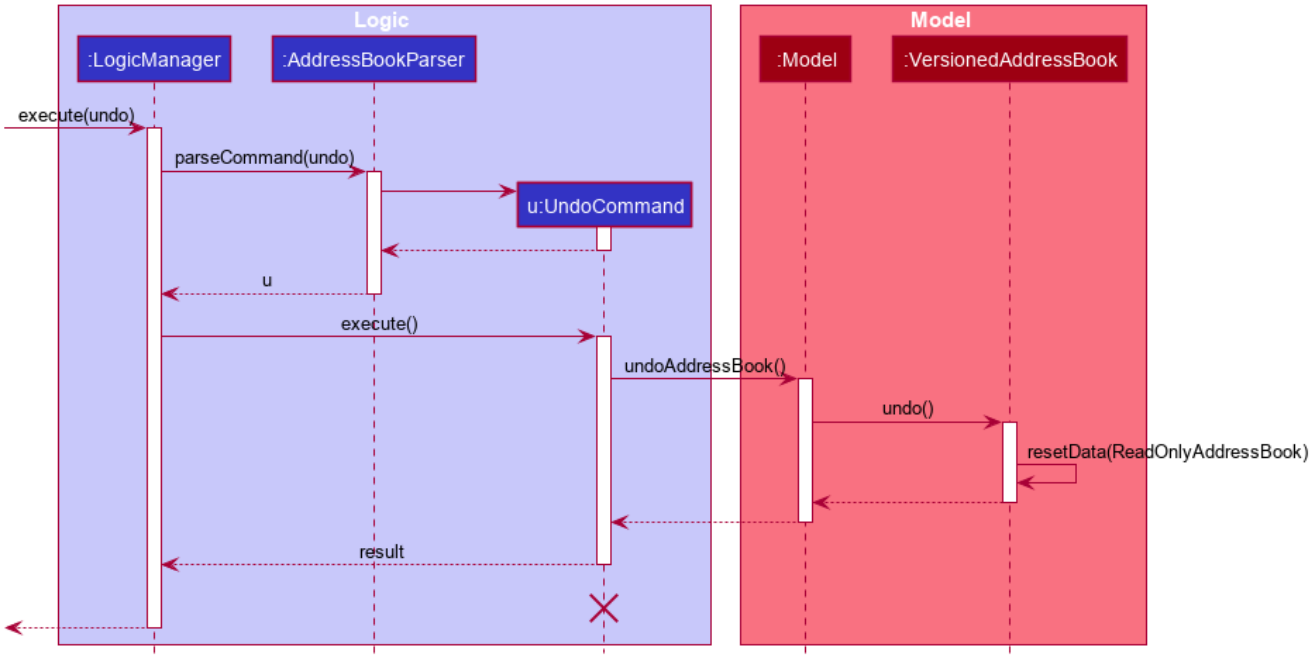
Step 4. The user now decides that adding the person was a mistake, and decides to undo that action by executing the `undo` command. The `undo` command will call `Model#undoAddressBook()`, which will shift the `currentStatePointer` once to the left, pointing it to the previous address book state, and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

After command "undo"



: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:

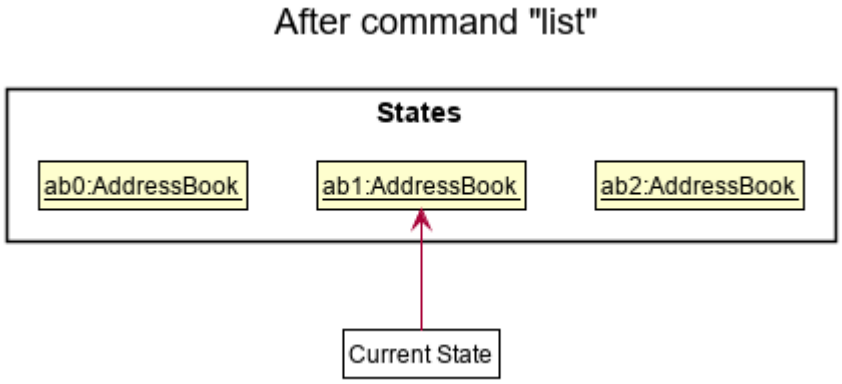


:information_source: **Note:** The lifeline for `UndoCommand` should end at the destroy marker (X) but due to a limitation of PlantUML, the lifeline reaches the end of diagram.

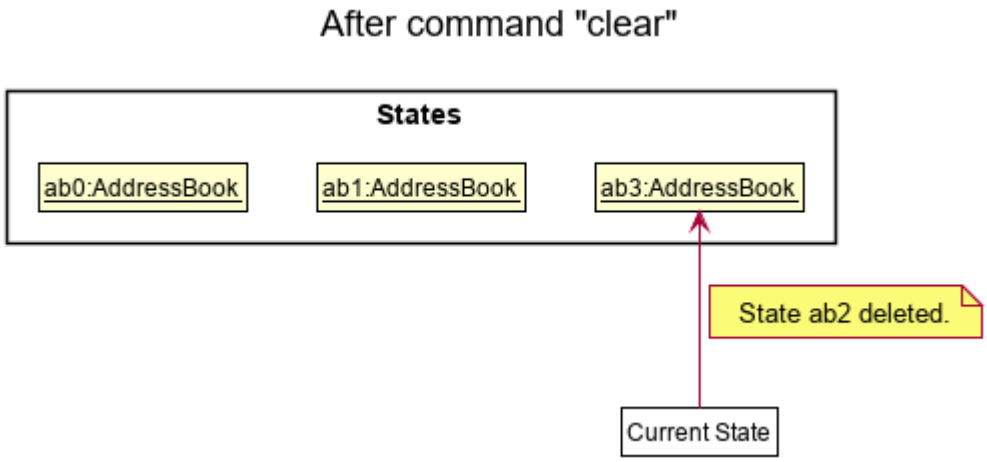
The `redo` command does the opposite — it calls `Model#redoAddressBook()`, which shifts the `currentStatePointer` once to the right, pointing to the previously undone state, and restores the address book to that state.

: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:

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

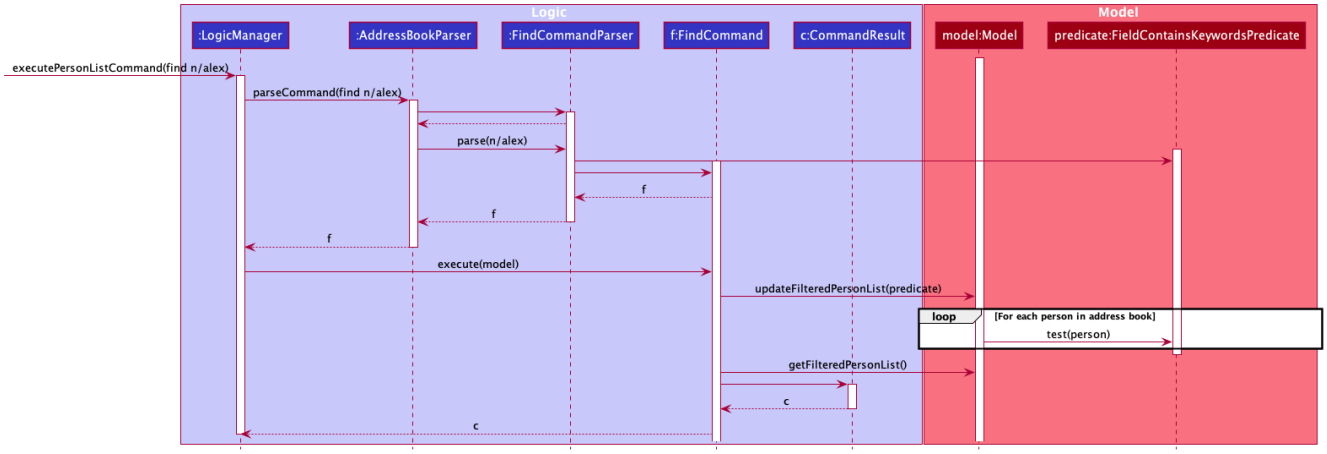
{Explain here how the data archiving feature will be implemented}

Find feature

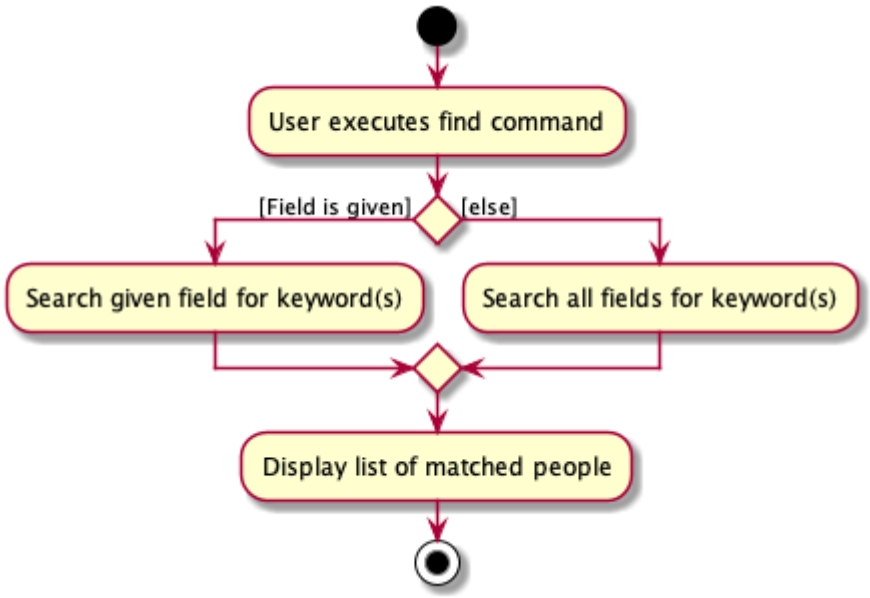
Implementation

The find command is used to search for people based on certain criteria.
{More details to be added later}

Below is a sequence diagram summarising the mechanism of find command:



Below is an activity diagram summarising the possible paths for a find command:



Design Considerations

Aspect: How keywords are matched

- **Alternative 1 (Current Choice):** Ignore case and full match is required
  - Pros:
    - Easy to implement.
    - It gives the best performance if the user remembers the exact keyword they are searching for.
  - Cons: Weak matching, i.e., `abc` does not match with `ab`. The current implementation would be less useful if the user remembers only some part of the search keywords.
- **Alternative 2:** Ignore case but full match is not required
  - Pros: Strong matching, would be especially helpful if the user remembers only bits and pieces of search keywords.
  - Cons: Difficult and time-consuming to implement.

Aspect: What happens when user does not specify a field

Note:
For evaluating the usefulness of the alternatives these are the assumptions made as to why the user does

not specify the field:
a) they forgot,
b) they do not want to restrict their search to one field, or
c) they do not remember which field they want to search.

Alternative 1 (Current Choice): Search all fields for the keyword

- Pros:
  - This is the most intuitive approach.
  - For all above mentioned scenarios a-c, this alternative is will produce the most useful result.
- Cons:
  - If there is a lot of data it will take more time to search all fields for every person.
  - Requires the most complex implementation among all alternatives.
  - Performs a lot of unnecessary comparisons (`alex` will never match any phone number, likewise `659347563` will never match any name).

Alternative 2: Use name as the default search field

- Pros: Simple implementation. Since searching people by their name is the most probable and intuitive use of this command, this is likely to produce a useful result.
- Cons: Useless for scenario b) and c).

Alternative 3: Produce a command syntax error and ask user to enter field

- Pros: Simple implementation. Useful in scenario a) above.
- Cons: Useless for scenario b) and c).

Manage meetings

[Coming soon]

Documentation, logging, testing, configuration, dev-ops

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

Appendix: Requirements

Product scope

Target user profile:

- really busy working professional
- has a need to manage a significant number of contacts
- prefer desktop apps over other types
- can type fast
- prefers typing to mouse interactions
- is reasonably comfortable using CLI apps

User persona:

Value proposition:

We help *busy working professionals* manage their large list of contacts by providing an **easy-to-use interface to store contacts** and help organize meetings. Our product will help users organize contacts by their companies, job titles, etc., and navigate their professional network quickly and efficiently to find who they are looking for.

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	view a help page with commands and usage instructions	understand how to use the application
＊＊＊	user	add a new person to the contact list	expand my contacts list
＊＊＊	user	delete a person from the contact list	remove contacts I no longer require
＊＊＊	user	label and store a person's phone numbers	know how to contact them via phone and which number to use (personal, office, etc.)
＊＊＊	user	label and store a person's email addresses	know how to contact them via email and which email to use (personal, office, etc.)
＊＊＊	user	label and store a person's addresses	know how to find them via address and which address to use (home, office, etc.)
＊＊＊	user	store a person's company	check which company they work at
＊＊＊	user	store a person's job title	check what job they have

Priority	As a …	I want to …	So that I can…
* * *	user	store a person's pronouns	check how they prefer to be addressed
* * *	user	assign custom tags to a person	identify them by the tags I give them
* * *	user	edit a person's contact information	update their contact information without having to delete and create a new contact
* * *	user	delete all contacts from the contact list	remove all contacts when I no longer require them and start with a fresh contact list
* * *	user	view all my contacts as a list	scroll the list to view all contacts or find the one I want
* * *	user	save my data automatically	reduce the risk of my data being lost
* * *	advanced user	save all my contacts in an editable file	edit my contacts directly from the data file
* *	new user	be provided suggested commands when I am adding contact information	know what kind of information I am able to add
* *	new user	view sample contacts when I first launch the application	see how the application looks when in use
* *	new user	easily remove existing sample contact information	begin adding my own contacts without confusion
* *	user	be warned when I create a contact with a name that exists	make sure I do not accidentally create a duplicate contact
* *	user with many contacts	search for a contact by name	find the contact I am looking for without having to scroll through a long list
* *	user with many contacts	search for a contact by their contact information	find the person I am looking for when I do not remember their name
*	user	change the colour scheme of the application	personalise my experience
*	user	be able to undo my previous command	undo a command if I make a mistake

Priority	As a …	I want to …	So that I can…
*	user	save my contacts' addresses as Google Maps links	use Google Maps for directions
*	user	see information about my business dealings with my contacts	continue my business with them
*	user with many contacts	be provided a history of my most searched-for contacts	easily find the contacts I use more often
*	user with many contacts	access my recent search history	can easily search for a previously searched contact

Use cases

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

Use case: UC1 - Add a contact

MSS:

1. User requests to add a contact by their name.
2. Reache goes into ‘edit’ mode.
3. User edits the contact’s details (UC2).
4. Reache displays the newly added contact in the list of contacts.

Use case ends.

Extensions:

1a. User inputs using the wrong format.
  1a1. Reache displays an error message.
  Use case resumes from step 1.

2a. Reache informs that the contact name already exists.
Use case resumes at step 1.

Use case: UC2 - Edit contact details

MSS:

1. User requests to add details for specific field(s) of the contact.
2. Reache saves the specified details along with their respective field(s).
  Repeat steps 1 and 2 until satisfied.
3. User requests to leave ‘edit’ mode.
4. Reache returns to ‘default’ mode.

Use case ends.

Extensions:

1a. User inputs the wrong format.
  1a1. Reache displays an error message.
  Use case resumes from step 1.

Use case: UC3 - Delete a contact

MSS:

1. User requests to delete a contact.
2. Reache asks for confirmation.
3. User confirms deletion.
4. Reache deletes the contact.
Use case ends.

Extensions:

1a. The requested contact does not exist.
  1a1. Reache displays an error message.
  Use case resumes at step 1.

3a. User chooses to cancel the deletion.
  3a1. Reache cancels the deletion.
  Use case ends.

Use case: UC4 - Find contacts by field

MSS:

1. User requests to find contacts by a given value for a field.
2. Reache shows all contacts that match the find criterion.
Use case ends.

Extensions:

1a. No contacts match the find criterion.
  1a1. Reache alerts that no contacts were found.
  Use case ends.

Use case: UC5 - View contact's full details

MSS:

1. User requests to view a contact's full details.
2. Reache displays the contact's full details.
Use case ends.

Extensions:

1a. The requested contact does not exist.
  1a1. Reache displays an error message.

Use case resumes at step 1.

Use case: UC6 - List all contacts

MSS:

1. User requests to see a list of all contacts.
2. Reache displays the list.

Use case ends.

Extensions:

1a. There are no contacts.
  1a1. Reache alerts that contact list is empty.
  Use case ends.

Use case: UC7 - Clear all contacts

MSS:

1. User requests to see a list of all contacts.
2. Reache asks for confirmation.
3. User confirms the action.
4. Reache clears all contacts.

Use case ends.

Extensions:

1a. There are no contacts.
  1a1. Reache alerts that contact list is empty.
  Use case ends.

3a. User chooses to cancel clearing contacts.
  3a1. Reache cancels the clearing.
  Use case ends.

Non-Functional Requirements

Technical requirements:

1. The product should work on any *mainstream OS* as long as it has Java `11` installed.

Quality requirements:

1. A user with above average typing speed for plain English text should be able to accomplish most of the tasks using commands faster than with the mouse.

Testability requirements:

1. The product should be only for a single user for higher testability.

Data requirements:

1. Data should be stored locally and on a human-editable file.
2. The final JAR file size should not exceed 100MB.
3. The PDF file size for the DG and UG should not exceed 15 MB per file.

Usability requirements:

1. The DG and UG must be PDF-friendly.
2. The GUI should not cause any resolution-related inconveniences to the user for:
    i. standard screen resolutions 1920x1080 and higher
    ii. screen scales 100% and 125%.
   In addition, all functions in the GUI should be usable even if the user experience is not optimal for:
    i. resolutions 1280x720 and higher
    ii. screen scales 150%.

Glossary

- **Mainstream OS:** Windows, MacOS, Linux
- **Busy working professionals:** Someone who has to manage a large number of interpersonal relationships for success at work and life
- **Personal and professional network:** Friends, family, neighbors, acquaintances, co-workers, clients, mentors, mentees
- **'Default' mode:** Allows the user to view their list of contacts
- **'Edit' mode:** Allows the user to edit contact details

{More to be added}

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
    i. Download the jar file and copy into an empty folder
    ii. 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
    i. Resize the window to an optimum size. Move the window to a different location. Close the window.
    ii. 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
    i. Prerequisites: List all persons using the `list` command. Multiple persons in the list.
    ii. 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.
    iii. Test case: `delete 0`
    Expected: No person is deleted. Error details shown in the status message. Status bar remains the same.
    iv. 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
    i. *{explain how to simulate a missing/corrupted file, and the expected behavior}*
2. *{ more test cases … }*