Shawn Lum - Project Portfolio

1. Introduction

This project portfolio briefly introduces the project, ClerkPro and outlines my contributions to the project's key features.

1.1. About the Team

My team consists of 4 Computer Science Undergraduate students, including myself.

1.2. About the Project: ClerkPro

ClerkPro is a desktop application used for appointments scheduling and queue management system targeted at clerks working in small clinics. Our application is optimised for individuals who **prefer to work with a Command Line Interface (CLI)** while still having the benefits of a Graphical User Interface (GUI).

1.3. Key to the Icons and Formatting Used in the Document

This symbol indicates extra information or definition.

This symbol indicates an important design decision.

`Model` : Text with this font and grey highlight indicates a component, class or object in the architecture of the application. It also indicates a generic command format for the command box in the User Guide.

`command` : Text with this blue font and grey highlight indicates a command that can be inputted by the user.

1.4. Introduction to ClerkPro

This desktop application consists of 4 tabs, a right side panel to display the patients currently being served, a command box for users to input their commands and a response box. Each tab serves a different purpose, but are designed to facilitate an effortless experience in managing the scheduling of appointments for patients.

The following shows the various tabs and their respective purposes (from top to bottom):

1. Patient tab: Keeps track of patient particulars.
2. Appointments tab: Handles the scheduling of patients' appointments

3. Staff tab: Keeps track of particulars of staff doctors.
4. Duty Shift tab: Records the shift timings of which doctor is on duty.

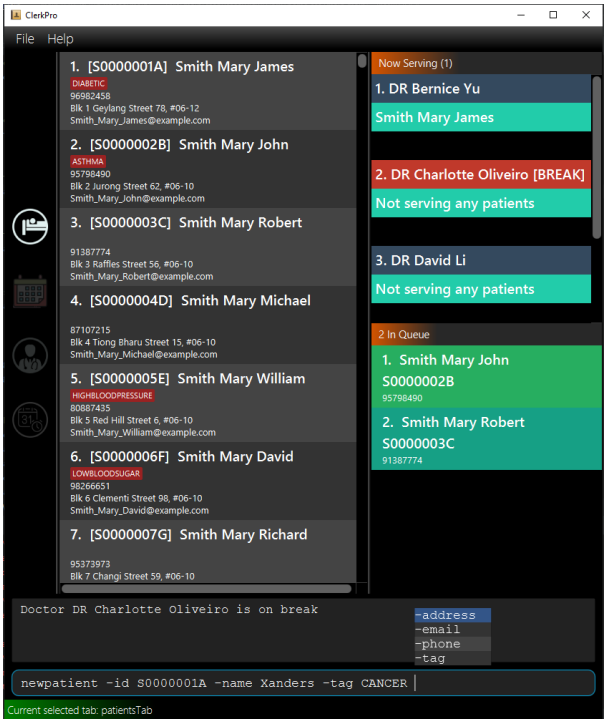


Figure 1. A quick glance at ClerkPro in action.

2. Summary of contributions

My responsibilities includes management of patient particulars and morphing the backend of the application to support the its appointment management features. This section will describe my contributions in greater detail.

- **Major enhancement**: Implementation of **Reactive Search (Multi-Threading and Logic)**
  - What it does: As the user types, an instant preview of search results is provided. This feature is used to search for registered patients, appointments, doctors and duty shifts.
  - Justification: Enhances user experience (UX) significantly as the system is able to narrow down search search results as the user is giving new inputs.
  - Highlights: Requires understanding the process of JavaFX main application thread and plan to defer updates in a way that is not disruptive to UX of other components.
  - Credits: Daniel Tan Jun Xian (team member for the proposal of the UX concept)
- **Major enhancement**: added **the ability to undo/redo previous commands**
  - What it does: Allows the user to undo all previous commands one at a time. Preceding undo commands can be reversed by using the redo command.
  - Justification: Provide a convenient way to rectify any mistakes made by the user.
  - Highlights: After in-depth analysis of design alternatives. We are able to mimic the undo/Redo behaviour without additional logic, by pairing the command to be executed with its inverse. The implementation was challenging as it required changes to existing commands.

- **Major enhancement**: **prevents over scheduling of appointments**
    - What it does: Ensures that user do not mistakenly schedule more patient appointments than available on-duty doctors at any given appointment of time.
    - Justification: This feature assists the user to avoid mistakenly over scheduling appointments for patients.
    - Highlights: Through the custom implementation of a TreeList, we are able find the number of scheduled appointments in a given time range, with a O(log(n)) time complexity. This same functionality can be used to determine the number of doctors on-duty during a given timing. Allowing the system to enforce that the number of scheduled appointments cannot be more than the on-duty doctors at point of time. Due to the limitation imposed the JavaFx's implementation of a non-extendable `FilteredList<E>`, which inherits from the abstract `TransformationList<E>` class, a custom implementation of a TreeSet that implements a `List<E>` was necessary.
- **Minor enhancement**: Patients and doctors can be identified using a unique reference id.
- **Minor enhancement**: Modified fields to be optional for patients and doctors. These optional fields include phone number, home address and email.
- **Code contributed**: [Functional code] [Test code]
- **Other contributions**:
    - Project management:
        - Managed releases `v1.1` and `v1.2` (2 releases) on GitHub
        - Set up team repository.
    - Enhancements to existing features:
        - CRUD for appointment management (Pull requests #41, #74)
        - Implements undo and redo functionality (Pull requests #3, #73)
        - Implements optional fields for person (Pull requests #115)
        - Implements reactive search (Pull requests #114, #145)
    - Documentation:
        - Updates to User and developer Guide: (Pull requests #99, #167, #235, #247)
    - Community:
        - PRs reviewed (with non-trivial review comments): #173, #188
        - Reported bugs and suggestions for other teams in the class (issue 1, 2, 3)
    - Tools:
        - Set up Travis and Codacy for team repo

3. Contributions to the User Guide

Given below are some sections I contributed to the User Guide. They showcase my ability to write documentation targeting end-users.

Reactive search

- The listing automatically filters to display only the most relevant results when searching for an existing entry.
- When searching for a specific patient or staff, using the `patient` or `doctor` command, the results are filtered based on whether the person's reference Id, name or phone number contains the given keyword.
- When searching for a specific appointment or duty shift, using the `appointments` or `shifts` command, the results are filtered based on whether the events involves a person whose reference Id contains the given keyword.

=== === Patient Management

3.1. Searches patient by arguments: `patient`

Filters patients whose reference ids, names or phone number contains the given keyword.
If no search keyword is provided, all registered patients will be displayed.

Format: `patient [<SEARCH_KEYWORD>]`

- e.g. `patient S0000001A`

3.2. Registers a new patient: `newpatient`

Registers a new patient. Only patient's reference id and name are compulsory fields

Format: `newpatient -id <PATIENT_REFERENCE_ID> -name <PATIENT_NAME> [-phone <PHONE_NUM>] [-email <EMAIL>] [-address <ADDRESS>] [-tag <Tags>]…`

- e.g. `newpatient -id E0000001A -name Edmond Halley -phone 85732743 -email halley@example.com -address 12, Kent ridge Ave 3, #01-11 -tag AIDS`

As a design decision, undoing the registration of a patient via the `undo` command is allowed. However, though our target users are clerks working at clinics, our clients would be the employers and stakeholders of said clinics. Hence, the user **should not have administrative privilege to unregister any patient already in the system.**

3.3. Updates patients' profiles: `editpatient`

Updates the particulars of a patients

Editing tags will overwrite all existing tags. Patients particulars should not be edited while patient is in queue or being served.

Format: `editpatient -entry <ENTRY_ID> [-id <PATIENT_REFERENCE_ID>] [-name <NAME>] [-phone <PHONE_NUM>] [-email <EMAIL>] [-address <ADDRESS>] [-tag <Tags>]…`

- e.g. `editpatient -entry 1 -phone 91200567 -email edmond@example.com`

4. Contributions to the Developer Guide

Given below are some sections I contributed to the Developer Guide. They showcase my ability to write technical documentation and the technical depth of my contributions to the project.

4.1. Reactive Search

The main concept behind reactive searching is to provide feedback to the user of their search results for a given keyword as they type. However, filtering larger data sets may be a time consuming process. Hence, the queries are processed on a separate thread to avoid blocking the `UI` thread.

If the `UI` triggers another reactive search request before previous request is completed, the thread of the previous request is first interrupted and joined to the thread of the new request. Only after the previous thread has been successfully interrupted, the new request will be processed. This is done to ensure that the UI thread only displays the results of the latest request.

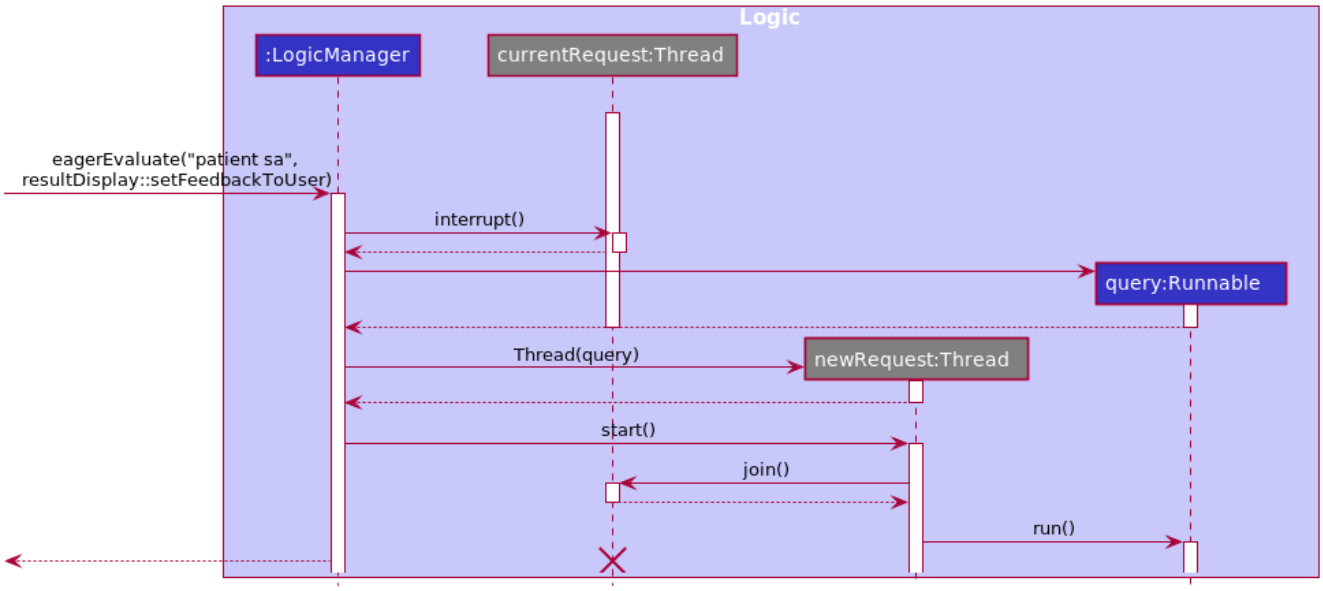


Figure 2. Sequence Diagram showing how the multi-threading is handled.

4.2. Storage component

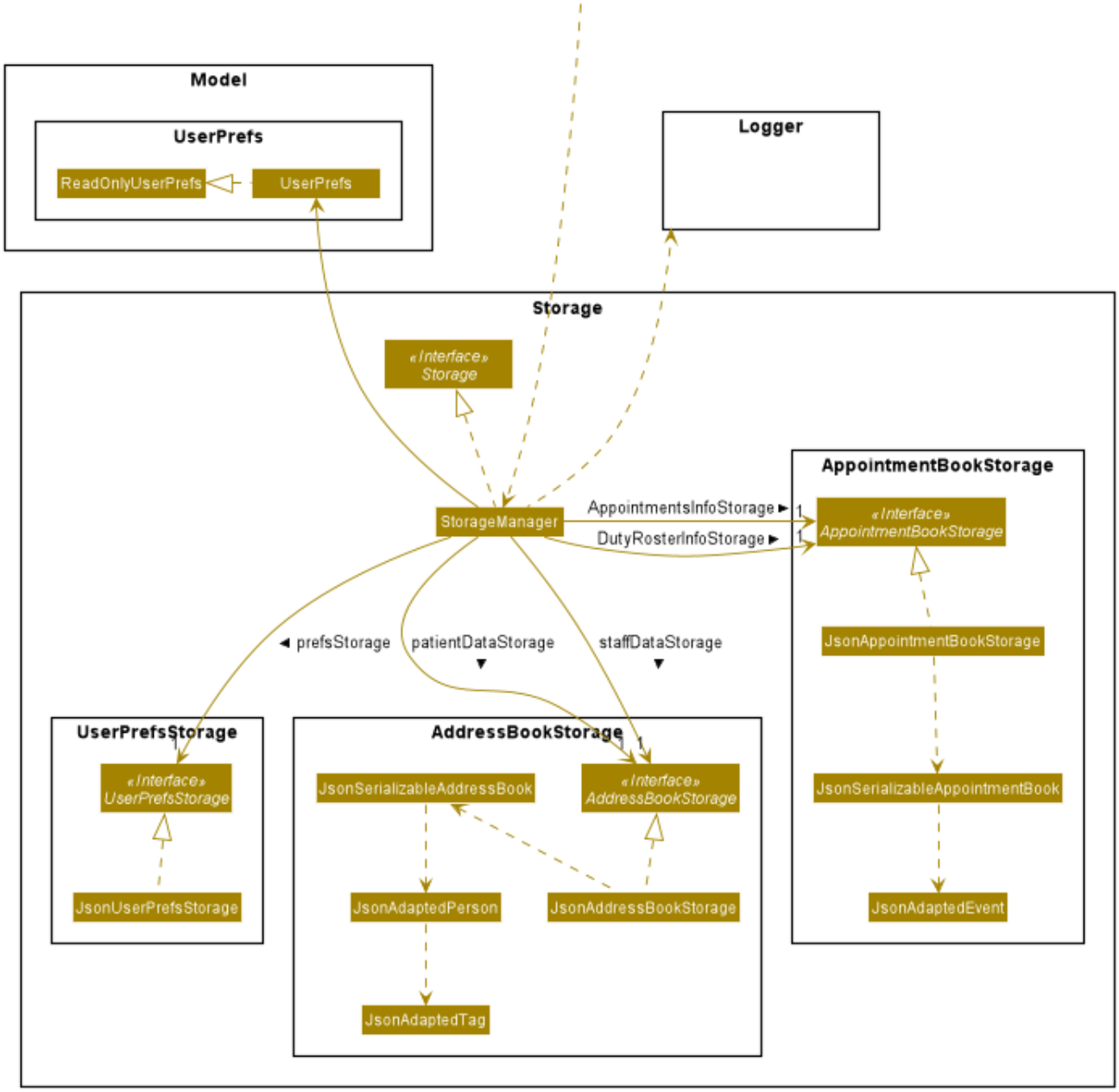


Figure 3. Structure of the Storage Component

API : `Storage.java`

The `Storage` component,

- saves `UserPref` objects in json format and read it back.
- saves and read data of the particulars of patients and doctors in json format.
- saves and read data of the Appointments and duty shifts data in json format.

`JsonAddressBookStorage` class is reused to store the details of both doctors and patients. Likewise, the `JsonAppointmentBookStorage` class can be reused to store the details of both staff duty shifts and patient's appointments.

4.3. Undo/Redo feature

The undo/redo feature allows users to revert the action of a command or redo a command action that has been undone.

4.3.1. Current Implementation

A Command can be classified as one of two types, a `ReversibleCommand` or `NonActionableCommand`. A `ReversibleCommand` refers to any command which modifies the data stored by the system. Conversely, a `NonActionableCommand` only reads data from the system's model without modifying it.

By pairing of two `ReversibleCommand` commands together, we can describe an action that has the undone.

This pairing is encapsulated as a `ReversibleActionPairCommand`, which consists of two `ReversibleCommand` classes: the command to be executed and its inverse.

Consider that our appointment management system provides the user the ability to add, cancelling and reschedule an appointment using the commands named `AddApptCommand`, `CancelApptCommand`, `EditApptCommand` respectively.

By executing the `CancelApptCommand`, we can mimic the undo functionality of the `AddApptCommand`, which is simply cancelling the same appointment the `AddApptCommand` had scheduled.

The following activity diagram describes the sequence of events when a user executes a new command:

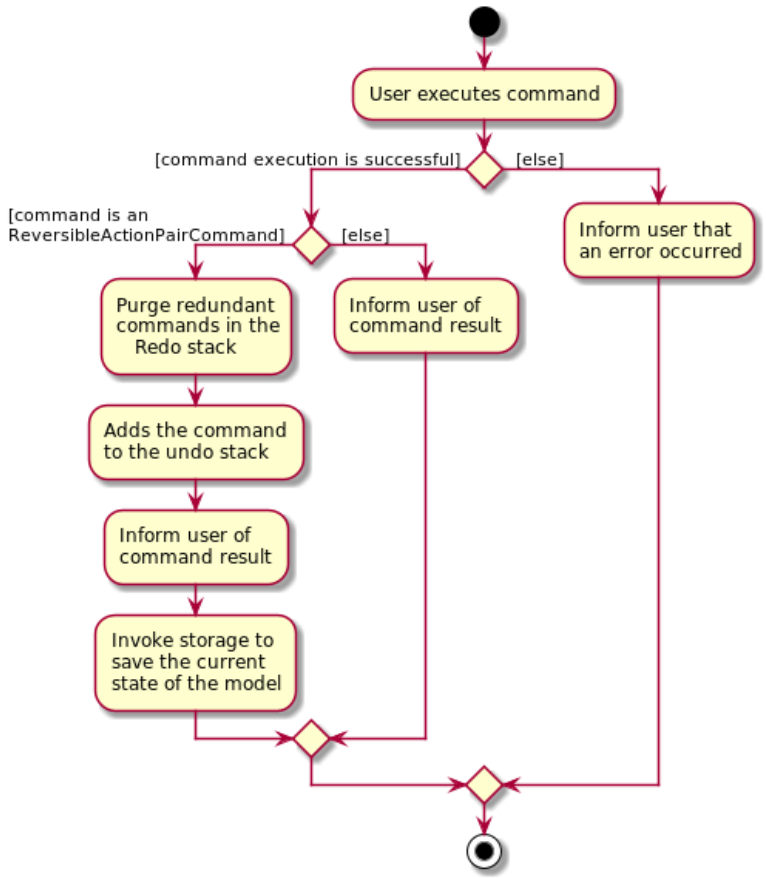


Figure 4. Commit Activity Diagram

The undo/redo mechanism is facilitated by the `CommandHistoryManager` class, which is found in the `logic` component. The history stores actions that can be undone as a `ReversibleActionPairCommand` and implements the following operations:

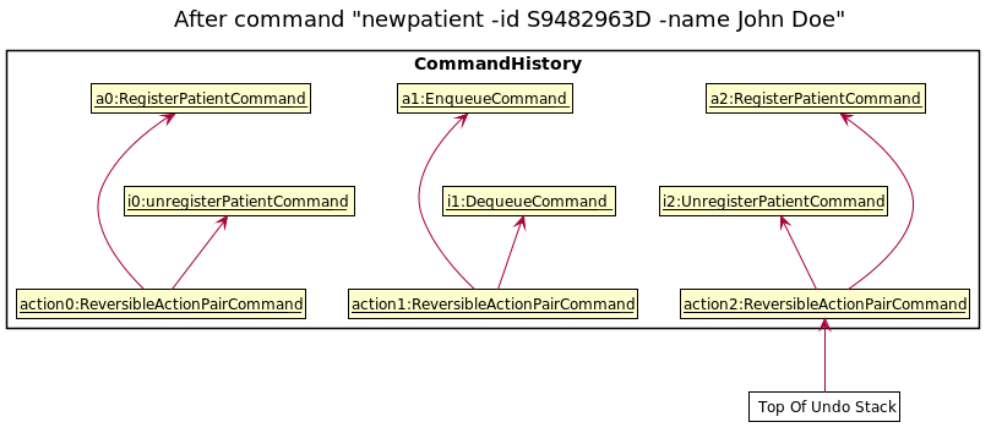
- `CommandHistoryManager#addToCommandHistory()` — Saves the most recent command that modifies the system's model in its undo history.
- `CommandHistoryManager#performUndo()` — Performs the inverse operation to restore the system to its previous state.
- `CommandHistoryManager#performRedo()` — Restores a previously undone state by re-executing the respective undone command.
- `CommandHistoryManager#canUndo()` — Checks if there are previous states to be restored
- `CommandHistoryManager#canRedo()` — Checks if the a previously undone state can be restored

These operations are all contained within the `Logic` component and do not depend on any other components.

On initialisation of the application, the undo and redo stack of the `CommandHistoryManager` would be empty. As the user executes commands that modifies the data stored by the system, the commands invokes `CommandHistoryManager#addToCommandHistory()` and are pushed in the undo stack of the `CommandHistoryManager`.

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

Step 1. Suppose that the user has already executed several commands earlier, with the latest command being `newpatient ⋯ -name John Doe`, which registers a patient named 'John Doe'.



Step 2. The user now decides that adding the patient was a mistake, and decides to undo that action by executing the `undo` command. The undo command will call `CommandHistoryManager#performUndo()`, which in turn invokes the `UnregisterPatientCommand` which reverts the system to its previous state.

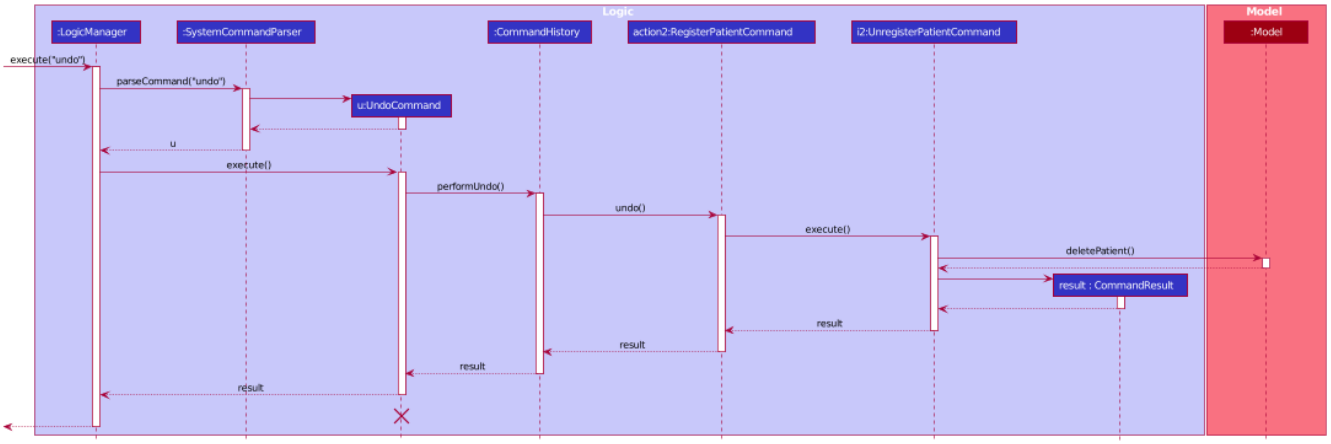
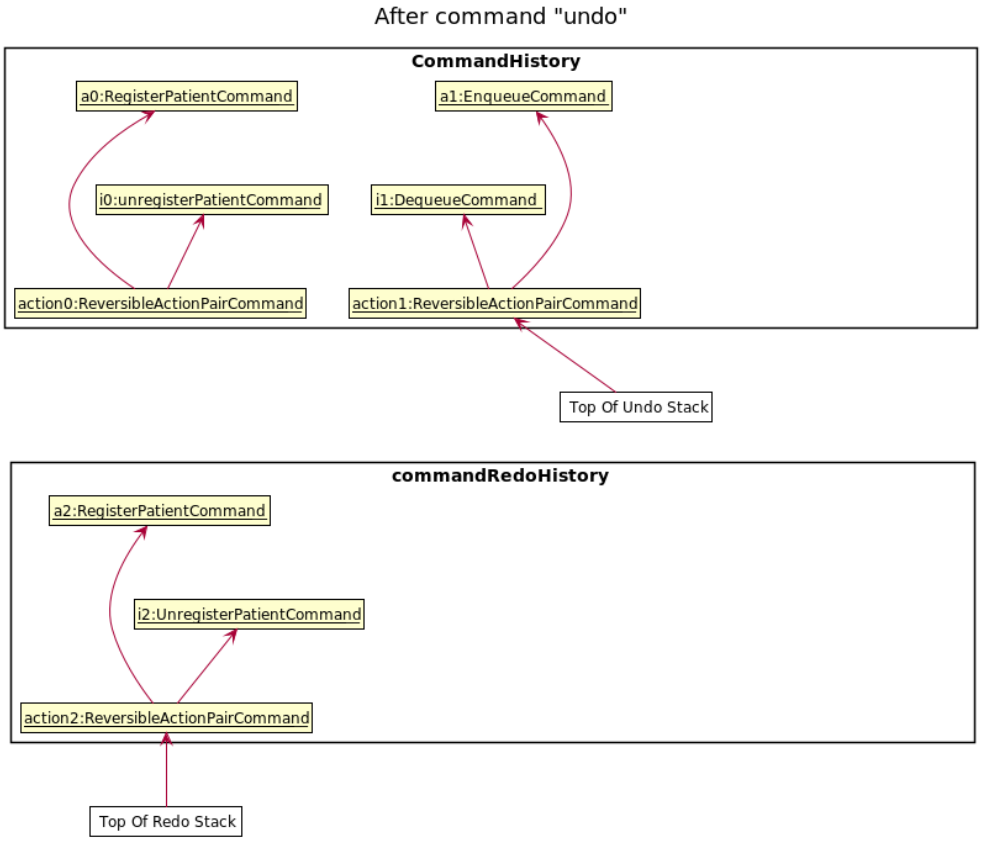


Figure 5. The sequence diagram shows how the undo operation works:

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.

An empty undo stack implies that there are no previous states to be restored. 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.

After the successful execution of the `UnregisterPatientCommand` command, the action from the top undo stack is transferred to the top of the redo history stack.



Conversely, the `redo` command does the opposite — it calls `CommandHistoryManager#performRedo()` which restores the address book to that state by invoking the original command again. Similarly, if the `commandRedoHistory` is empty, then there are no undone 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 3. The user then decides to execute the command `patient S9482963D`. A command that searches for a patient whose id matches `S9482963D`, only reads and does not modify any data from the model. Such commands will not invoke `CommandHistoryManager#addToCommandHistory()`, `CommandHistoryManager#performUndo()`, `CommandHistoryManager#performRedo()`.

Thus, the undo and redo stacks remain unchanged.

Step 4. The user executes `newappt …` to schedule a new appointment for a patient. This action invokes `CommandHistoryManager#addToCommandHistory()`, pushing the new action pair command in the undo stack. However, the commands in the redo stack will be purged.

We designed it this way because it no longer makes sense to redo the `newpatient … -name John Doe` command. This is the behavior that most modern desktop applications follow.