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 year 2 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.

`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.

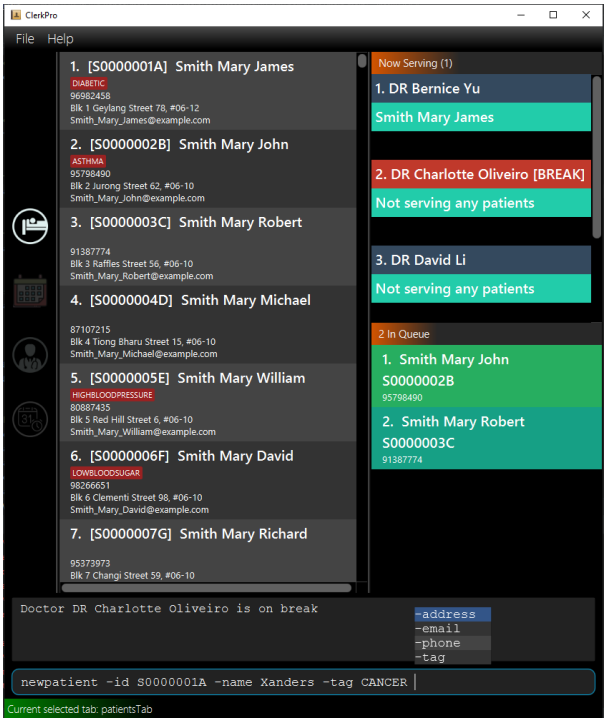


Figure 1. A quick glance at ClerkPro in action.

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]

{YET TO DO} * **Other contributions**:

- Project management:
    - Managed releases `v1.3` - `v1.5rc` (3 releases) on GitHub
- Enhancements to existing features:
    - Updated the GUI color scheme (Pull requests #33, #34)
    - Wrote additional tests for existing features to increase coverage from 88% to 92% (Pull requests #36, #38)
- Documentation:
    - Did cosmetic tweaks to existing contents of the User Guide: #14
- Community:
    - PRs reviewed (with non-trivial review comments): #12, #32, #19, #42
    - Contributed to forum discussions (examples: 1, 2, 3, 4)
    - Reported bugs and suggestions for other teams in the class (examples: 1, 2, 3)
    - Some parts of the history feature I added was adopted by several other class mates (1, 2)
- Tools:
    - Integrated a third party library (Natty) to the project (#42)
    - Integrated a new Github plugin (CircleCI) to the team repo

{you can add/remove categories in the list above}

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.

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.

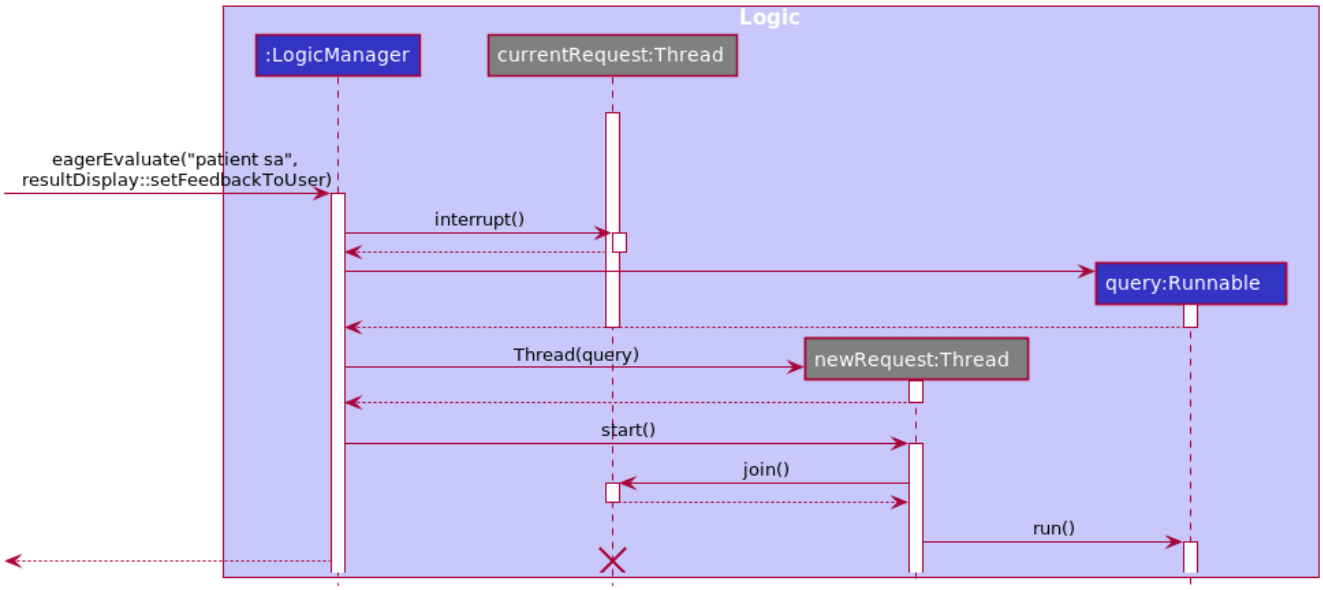


Figure 2. Sequence Diagram showing how the multi-threading is handled.

4.2. 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.

4.2.1. Current Implementation

We use a pairing of two commands to describe any action that can be undone. This pairing is encapsulated as a `ReversibleActionPairCommand`, which consists of 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.

Unlike implementing the undo functionality into every command that modifies the database of our system, we can mimic these undo functionality by pairing and executing existing commands which are available to us.

The following activity diagram summarises what happens when a user executes a new command:

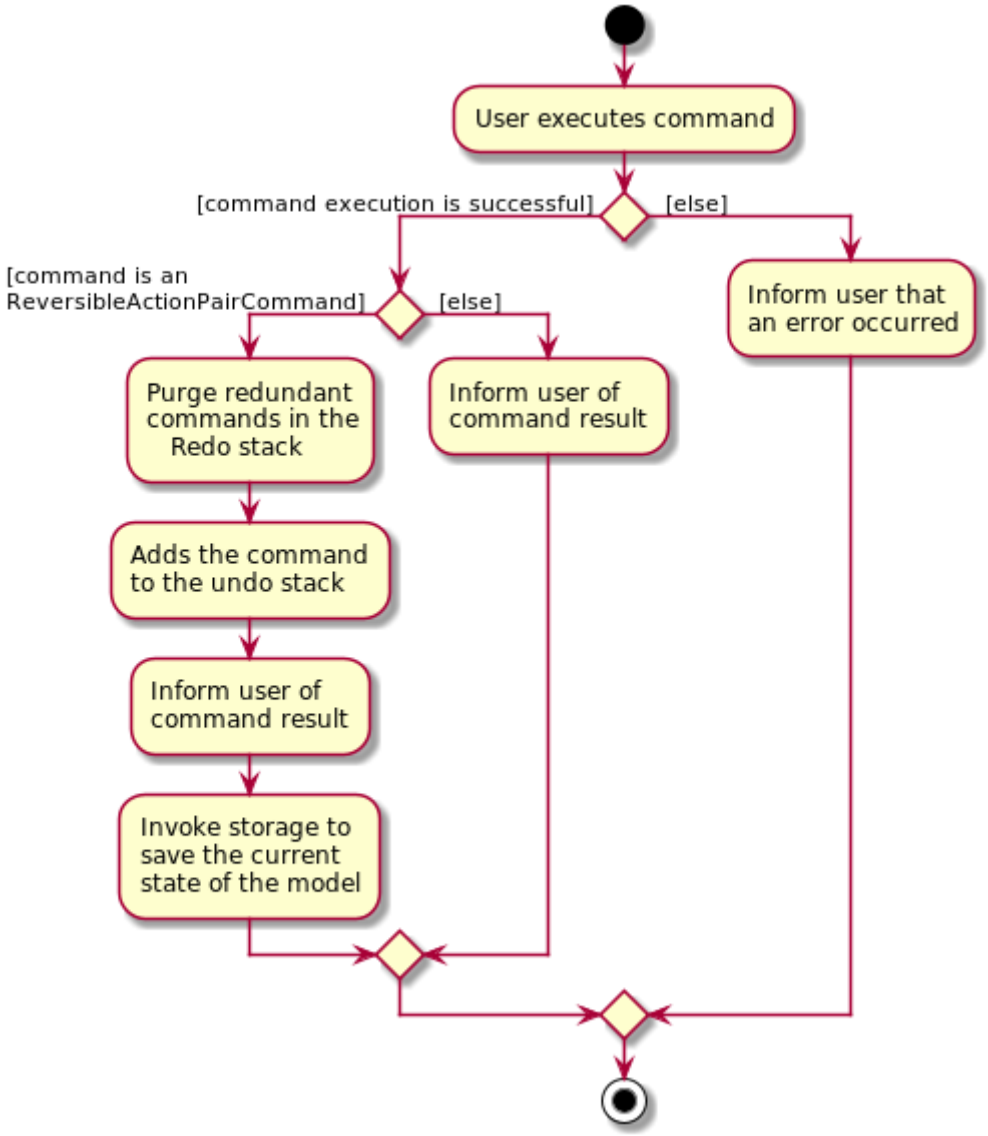


Figure 3. Commit Activity Diagram

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.

The undo/redo mechanism is facilitated by the `CommandHistory` class, which is found in the `logic` component. The history class stores commands that can be undone as a `ReversibleActionPairCommand`, which describes a pairing of two `ReversibleActionCommand`, the first command being the action to be executed and its inverse. (e.g. Pairing 'add event A' and 'delete event A' command).

With such an implementation, our design can conform to the Command pattern without violating the Single Responsibility Principle. Moreover, this implementation is less likely to face performance issues in terms of memory usage, compared to the Memento Pattern, which implements the undo capability by storing an instance of every previous state of the application in memory.

The design of `CommandHistory` uses the Command pattern, a common design pattern often used in software engineering. It implements the following operations:

- `CommandHistory#addToCommandHistory()` — Saves the most recent command that modifies the

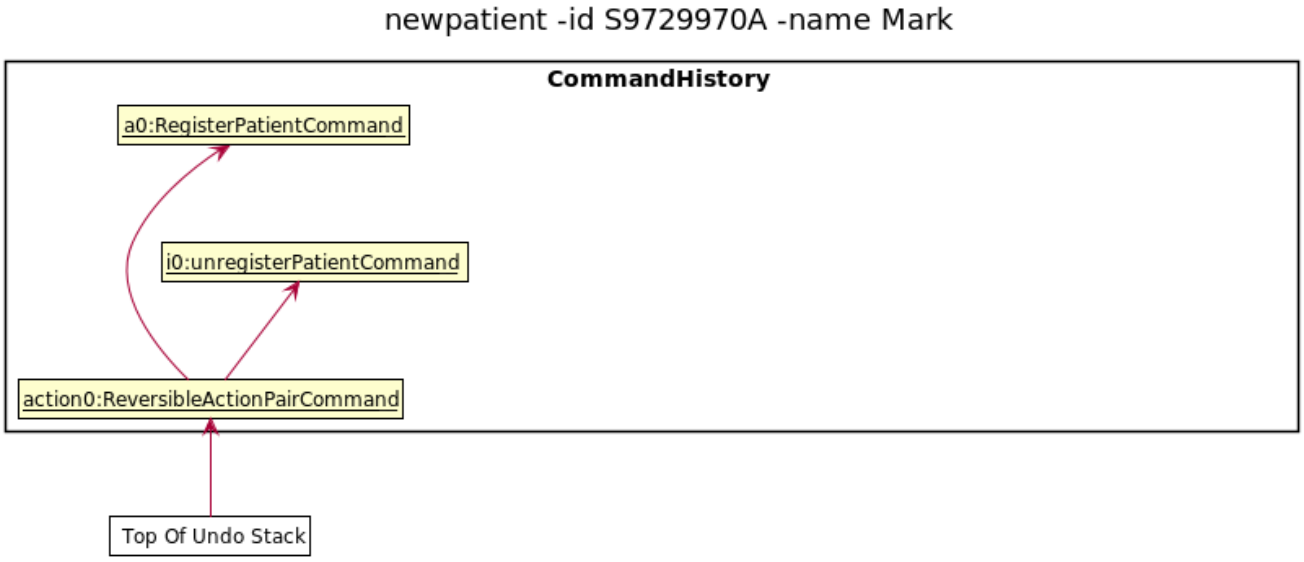
system's model in its undo history.

- `CommandHistory#performUndo()` — Performs the inverse operation to restore the system to its previous state.
- `CommandHistory#performRedo()` — Restores a previously undone state by re-executing the respective undone command.
- `CommandHistory#canUndo()` — Checks if there are previous states to be restored
- `CommandHistory#canRedo()` — Checks if the a previously undone state can be restored

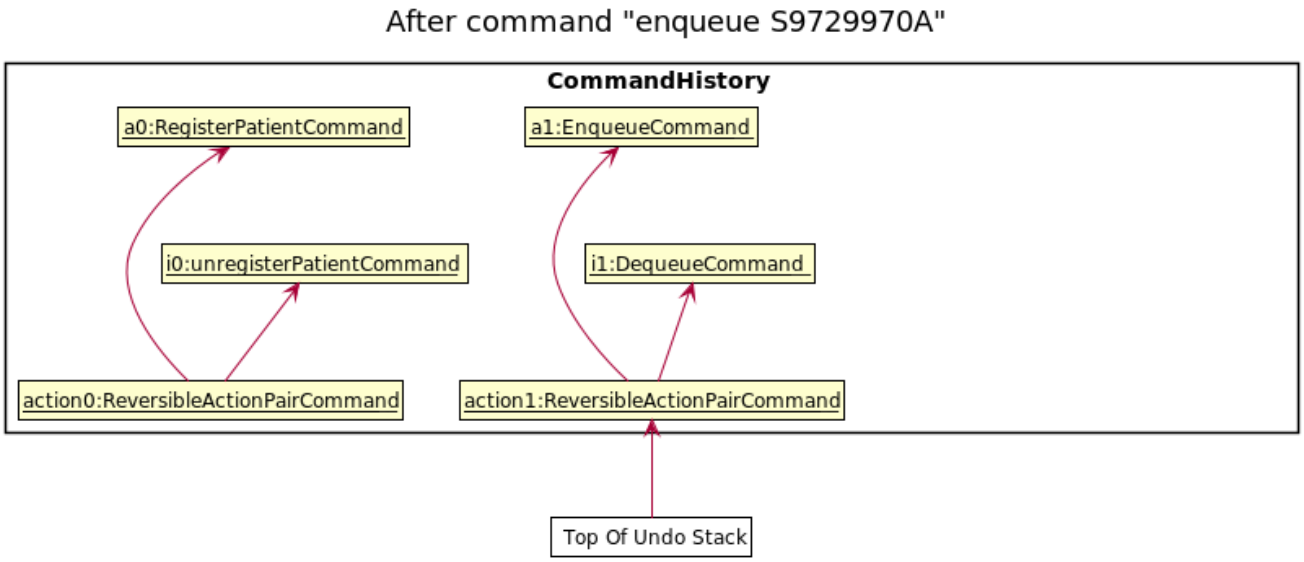
These operations are all contained within the `Logic` component and do not depend on any other components. Given below is an example usage scenario and how the undo/redo mechanism behaves at each step.

Step 1. The user launches the application for the first time. The `CommandHistory` will be initialised with an empty undo and redo stack.

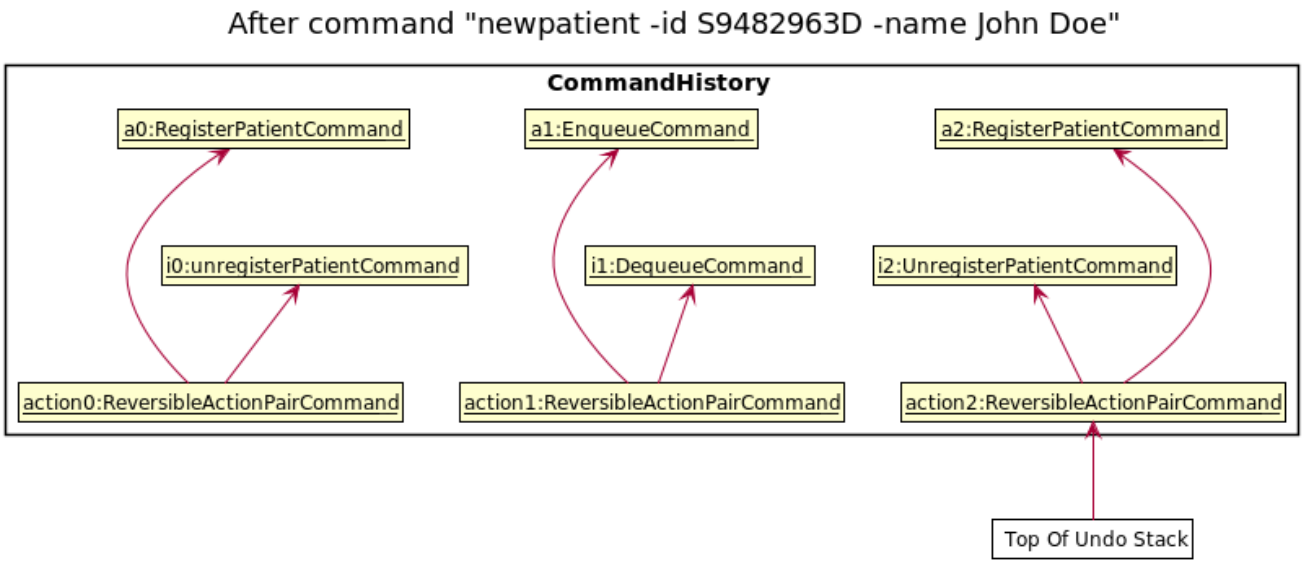
Step 2. The user executes `newpatient ⋯ -name Mark` command which registers a new patient named `Mark` with the unique id of `S9729970A`. The `newpatient` command creates a `ReversibleActionPairCommand`, which pairs of a `RegisterPatientCommand` and `UnregisterPatientCommand`. After the invoking execution of the `RegisterPatientCommand`, the whole `ReversibleActionPairCommand` is pushed to the undo history stack via the `CommandHistory#addToCommandHistory()`



Step 3. The user executes `enqueue S9729970A` command to add a patient whose reference id matches `S9729970A`. The `enqueue` command also calls the `CommandHistory#addToCommandHistory()`, which pushes the enqueue action pair into the command history stack.

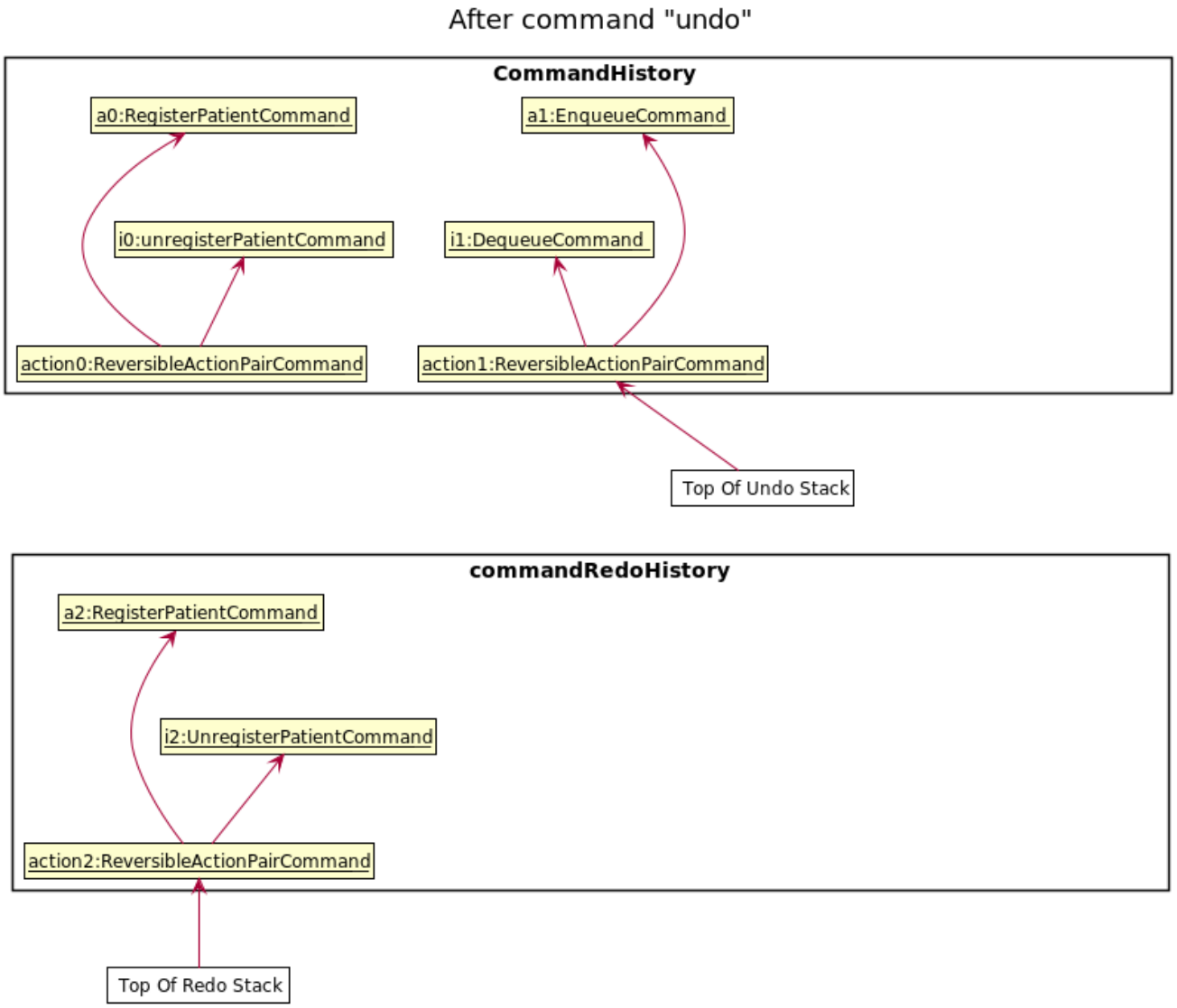


Step 4. The user executes `newpatient … -name John Doe` to register a new patient. Similar to the previous command, also causes another action pair to be added into the command history undo stack.



If a command fails its execution, it will not call `CommandHistory#performRedo()`. Hence, the command will not be saved into the command history.

Step 5. 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 `CommandHistory#performUndo()`. This invokes the `UnregisterPatientCommand` which reverts the system to its previous state, and moves the action from the top undo stack to the top of the redo history stack.



If the undo stack is empty then 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.

The following sequence diagram shows how the undo operation works:

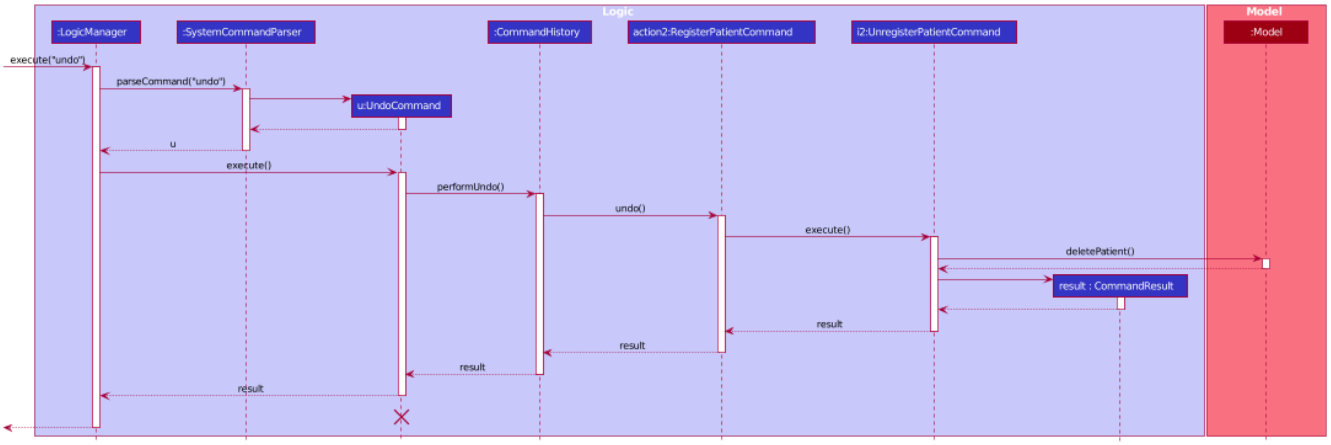


Figure 4. Undo Sequence Diagram

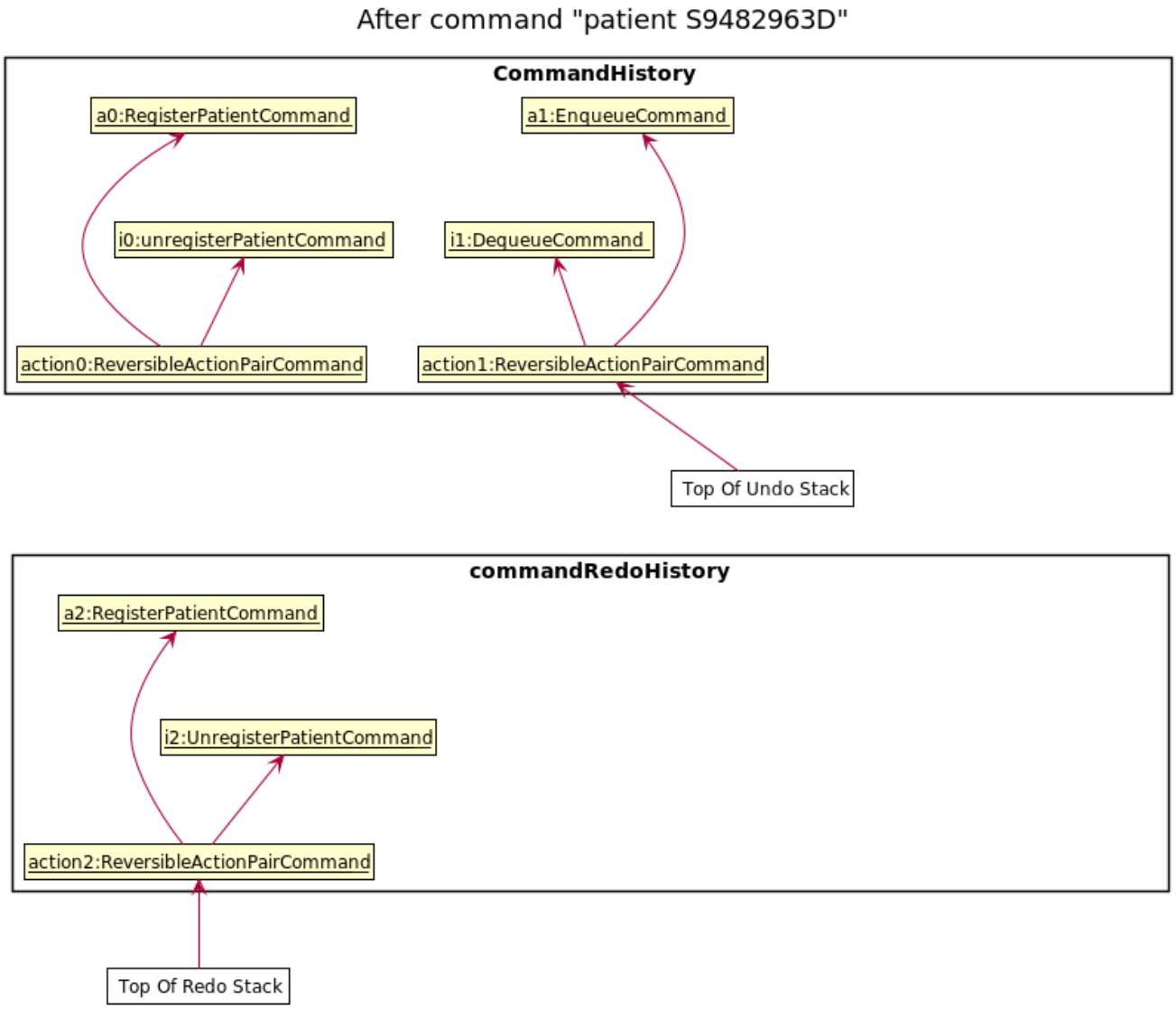
The lifeline for `UndoCommand` should end at the destroy marker (X) but due to a limitation of PlantUML, the lifeline reaches the end of diagram.

The `redo` command does the opposite — it calls `CommandHistory#performRedo()` which restores the address book to that state by invoking the original command again.

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.

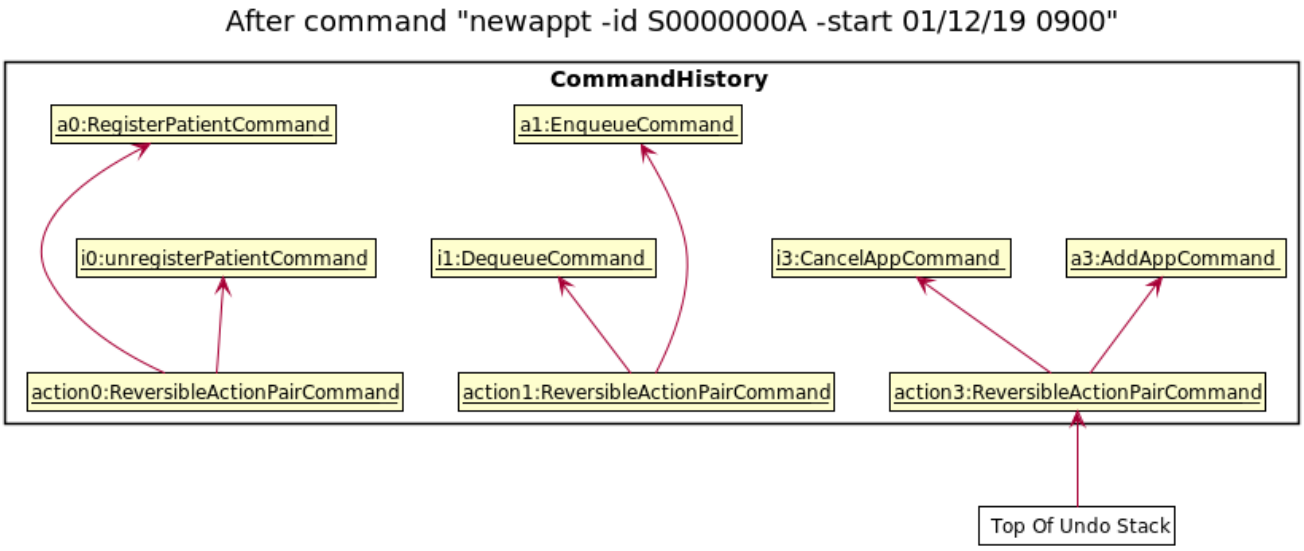
Step 5. The user then decides to execute the command `patient S9482963D`. A command that searches for a patient whose id matches `S9482963D`, only reads and does not modify any data from the model. Such commands will not call `CommandHistory#addToCommandHistory()`, `CommandHistory#performUndo()`, `CommandHistory#performRedo()`.

Thus, the undo and redo stacks remain unchanged.



Step 6. The user executes `newappt ⋯` to schedule a new appointment for a patient. This action invokes `CommandHistory#addToCommandHistory()`, pushing the new action pair command in the undo stack. However, the commands in the `commandRedoHistory` 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.



4.2.2. Design Considerations

Aspect: How undo & redo executes

- **Alternative 1 (current choice):** Pair Individual commands with their inverse action.

Pros	1. Use less memory (e.g. for `delete`, just save the person being deleted).
Cons	1. Must ensure that the implementation of each individual command are correct. 2. Must ensure that commands are paired

- **Alternative 3:** Individual command knows how to undo/redo by itself.

Pros	1. Use less memory (e.g. for `delete`, just save the person being deleted).
Cons	1. Must ensure that the implementation of each individual command are correct.

- **Alternative 2:** Save the entire address book.

Pros	1. Implementation is easy.
Cons	1. May have performance issues in terms of memory usage.

Aspect: Data structure to support the undo/redo commands

- **Alternative 1 (current choice):** Use a list to store the history of address book states.

Pros	1. Easy for new Computer Science student undergraduates to understand, who are likely to be the new incoming developers of our project.
Cons	1. Logic is duplicated twice. For example, when a new command is executed, we must remember to update both `HistoryManager` and `VersionedAddressBook`.

- **Alternative 2:** Use `HistoryManager` for undo/redo

Pros	1. We do not need to maintain a separate list, and just reuse what is already in the codebase.
Cons	1. Requires dealing with commands that have already been undone: We must remember to skip these commands. Violates Single Responsibility Principle and Separation of Concerns as `HistoryManager` now needs to do two different things.

4.3. Storage component

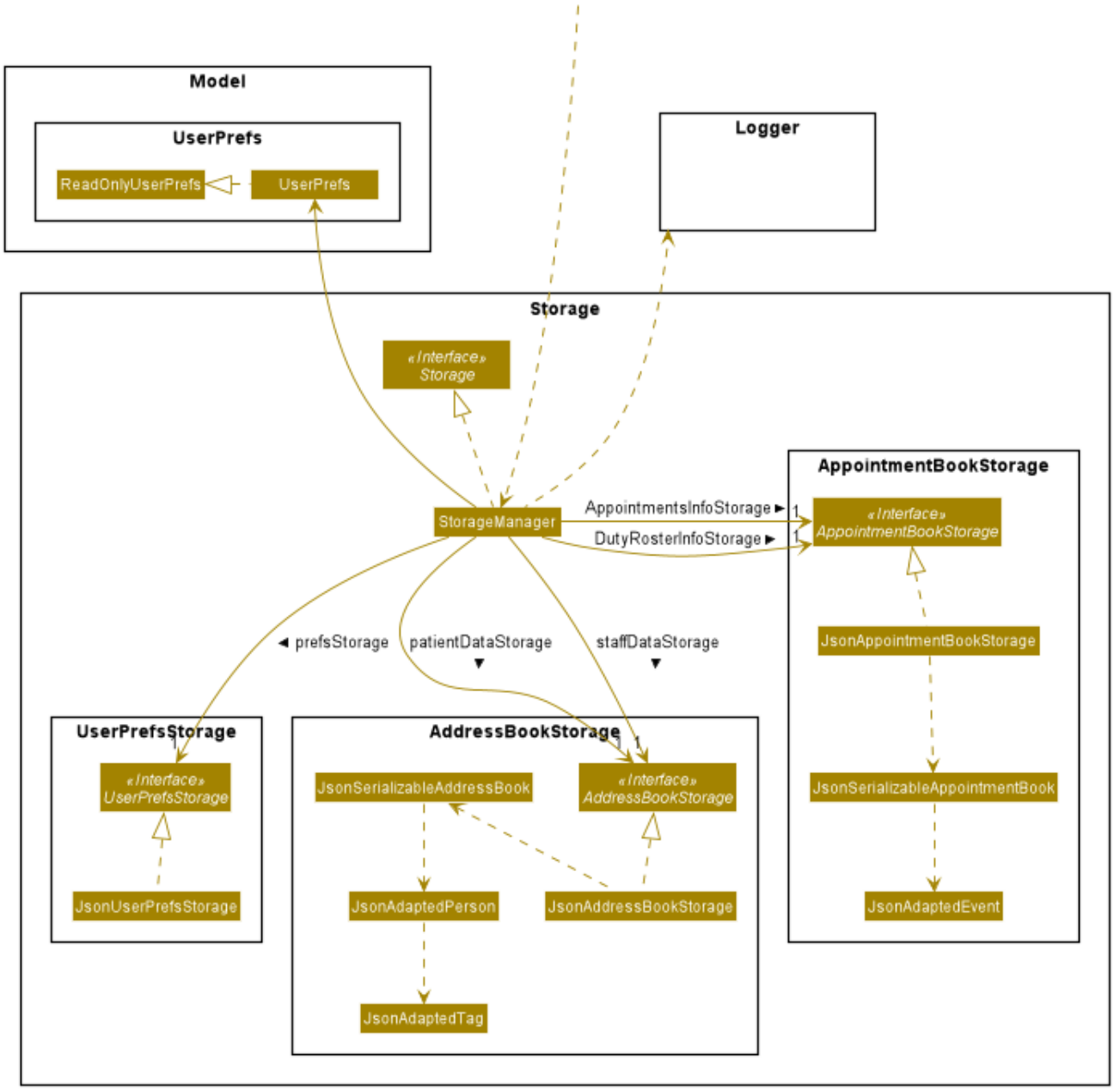


Figure 5. Structure of the Storage Component

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.