ClerkPro - Developer Guide

By: `Team T09-3` Since: `Sep 2019` Licence: `MIT`

1. Introduction

ClerkPro is an appointment and queue management system targeted at clerks working in small clinics. This developer guide is organised in a top-down approach, beginning with the Architecture Design. You may jump to any section by clicking on the heading in the contents page.

2. Setting up

Refer to the guide here.

3. Design

3.1. Architecture

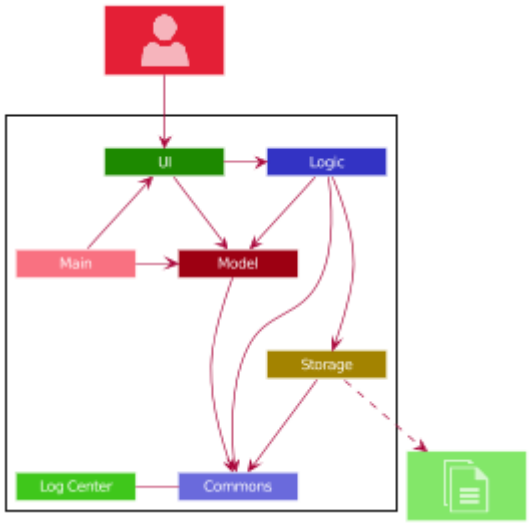


Figure 1. Architecture Diagram

The ***Architecture Diagram*** given above explains the high-level design of ClerkPro. Given below is a quick overview of each component.

`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 method where necessary.

`Commons` represents a collection of classes used by multiple other components. The following class plays an important role at the architecture level:

- `LogsCenter` : Used by many classes to write log messages to the App's log file.

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.

Each of the four components

- Defines its *API* in an `interface` with the same name as the Component.
- Exposes its functionality using a `{Component Name}Manager` class.

For example, the `Logic` component (see the class diagram given below) defines it's API in the `Logic.java` interface and exposes its functionality using the `LogicManager.java` class.

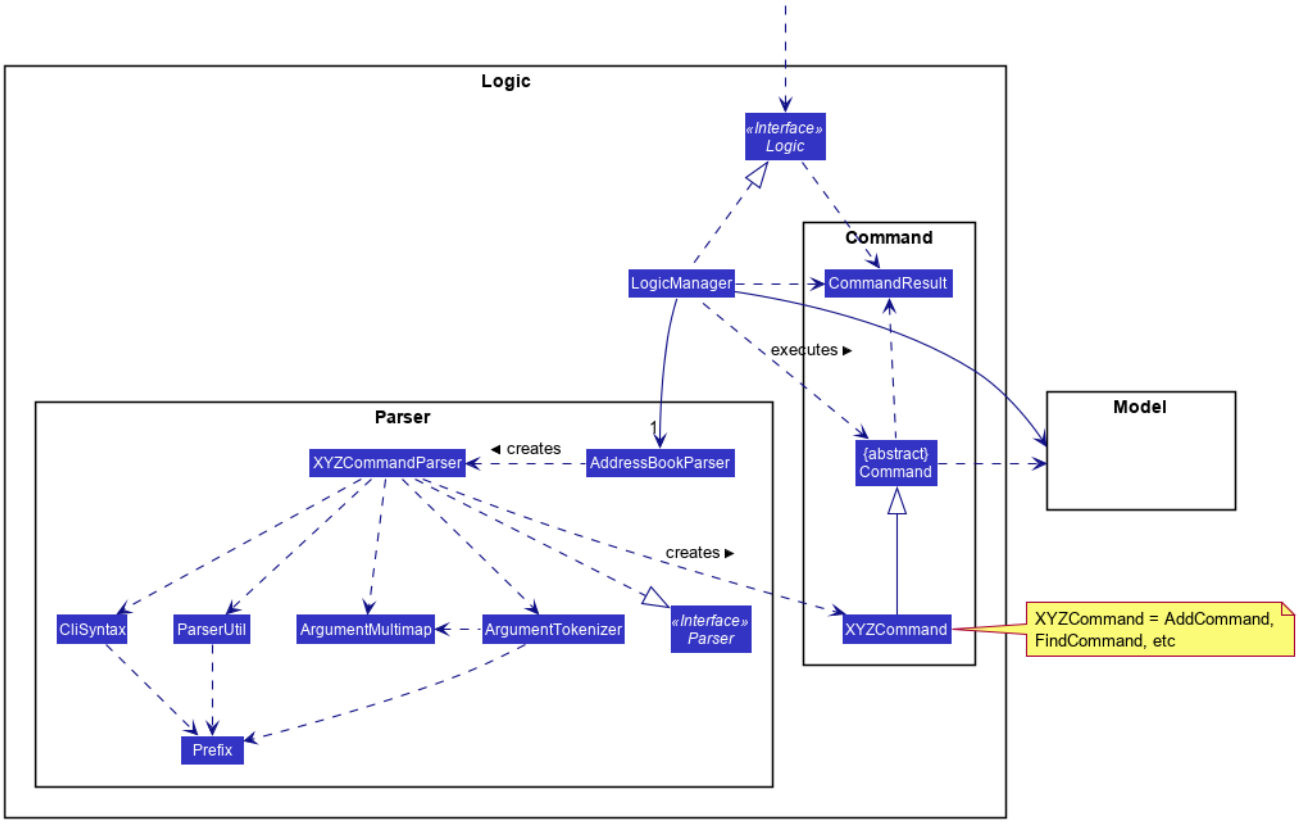


Figure 2. Class Diagram of the Logic Component

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`.

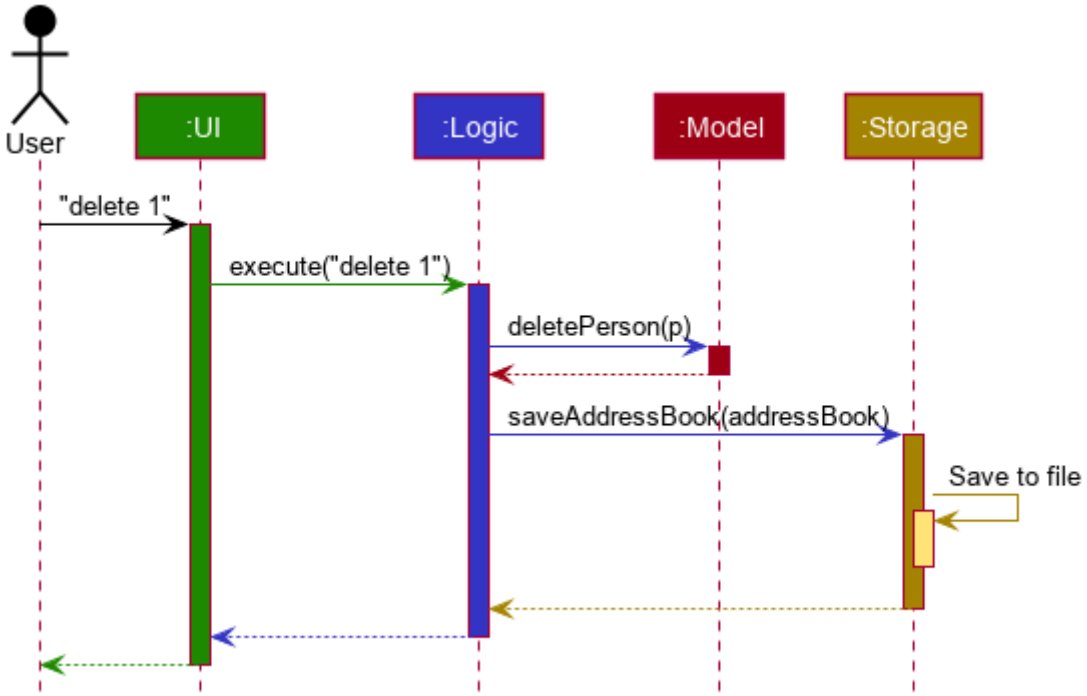


Figure 3. Component interactions for `delete 1` command

The sections below give more details of each component.

3.2. UI component

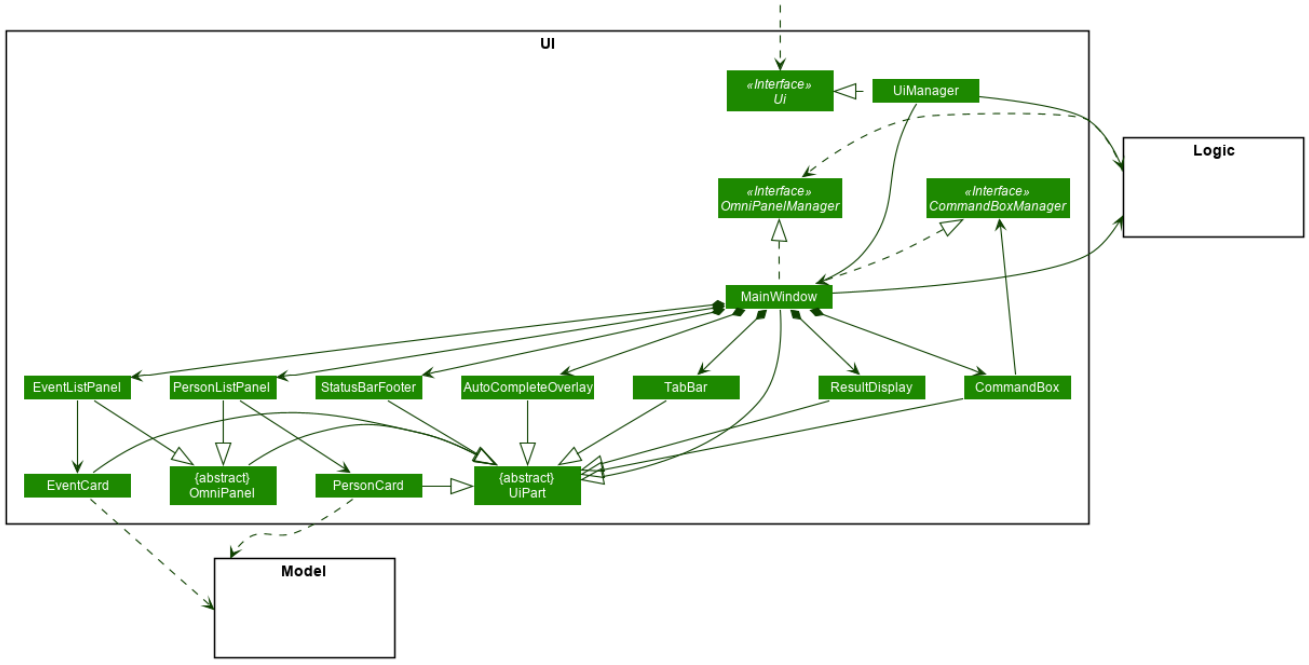


Figure 4. Structure of the UI Component

API : `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.

The `UI` component uses 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.

3.3. Logic component

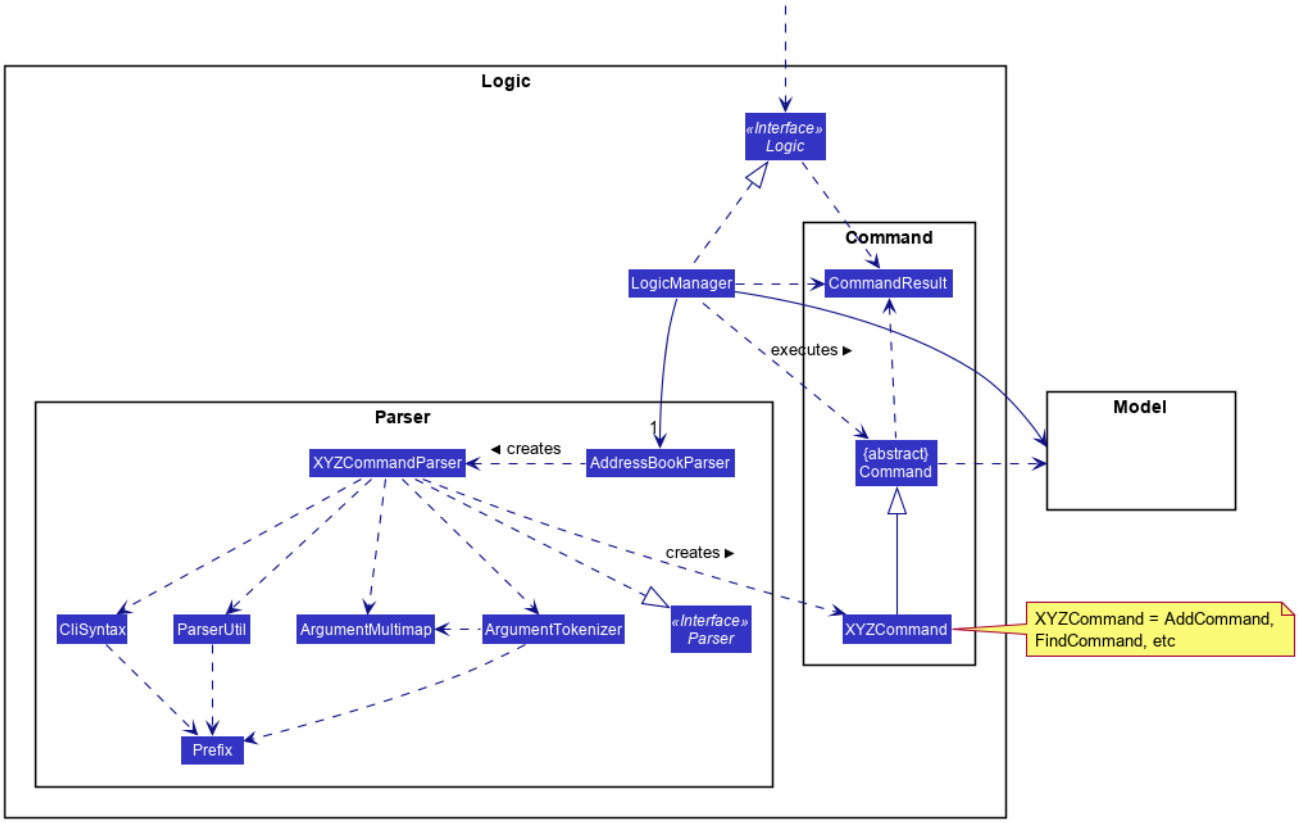


Figure 5. Structure of the Logic Component

API : `Logic.java`

1. `Logic` uses the `SystemCommandParser` class to parse the user command.
2. This results in a `Command` object which is executed by the `LogicManager`.
    - A `Command` object can be classified as one of two types, a `ReversibleCommand` and a `NonActionableCommand`.
    - A `ReversibleCommand` refers to any command which modifies the data in the system's model. To enable the user to revert their changes, the actions of such commands needs to be reversible.
    - Conversely, a `NonActionableCommand` only reads data from the system's model without modifying it.
3. The command execution can affect the `Model` (e.g. adding a patient).

- If the user intends to execute a reversible command, a `ReversibleActionPairCommand` is created and pushed into an undo stack. This action pair command contains a pairing of of the action itself and its inverse. (e.g. Pairing 'add person A' and 'delete person A' command).
- If the user intends to execute a `NonActionableCommand`, the command will be directly executed.

4. The result of the command execution is encapsulated as a `CommandResult` object which is passed back to the `Ui`.
5. In addition, the `CommandResult` object can also instruct the `Ui` to perform certain actions, such as displaying help to the user.
6. Handles the mutli-threading of reactive search requests by the user.
    - When the user is searching for an existing entry, the system will attempt to filter through the results as the user types.
    - Each key stroke will trigger a new reactive search request which is processed on a separate thread. This is done to avoid lagging the UI.
    - If a new reactive search request is triggered before the previous request has been completed. The previous request thread will be interrupted before the new request thread is allowed to be executed.
    - The execution of a reactive search differs from the normal command, in the way that it only allows the execution of `NonActionableCommand` types. Hence, there is no modification of any data in the system's model when executing a reactive search.

Given below is the Sequence Diagram for interactions within the `Logic` component for the `enqueue E0000001A` API call.

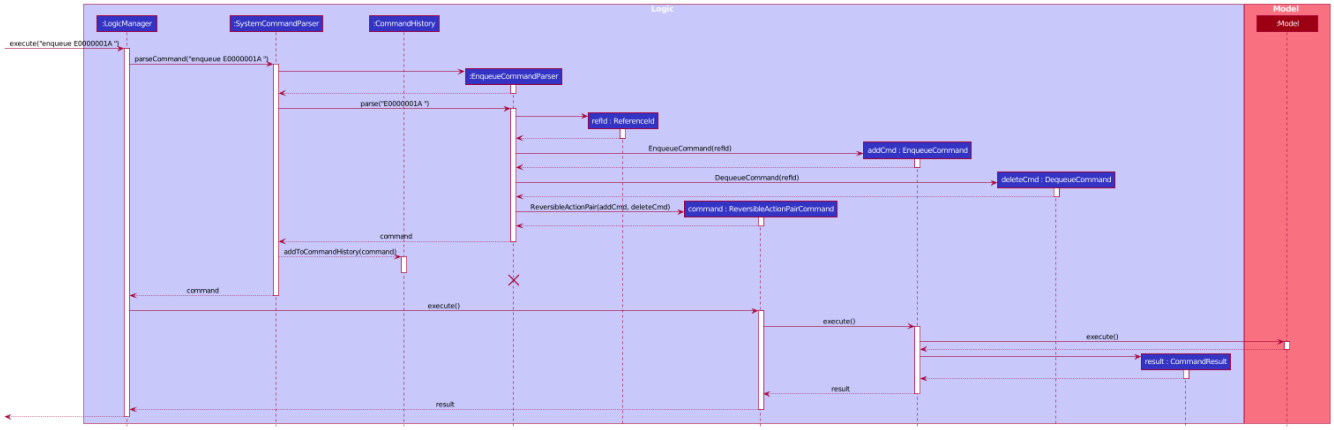


Figure 6. Interactions inside the Logic Component for the `enqueue E0000001A` Command.

3.4. Model component

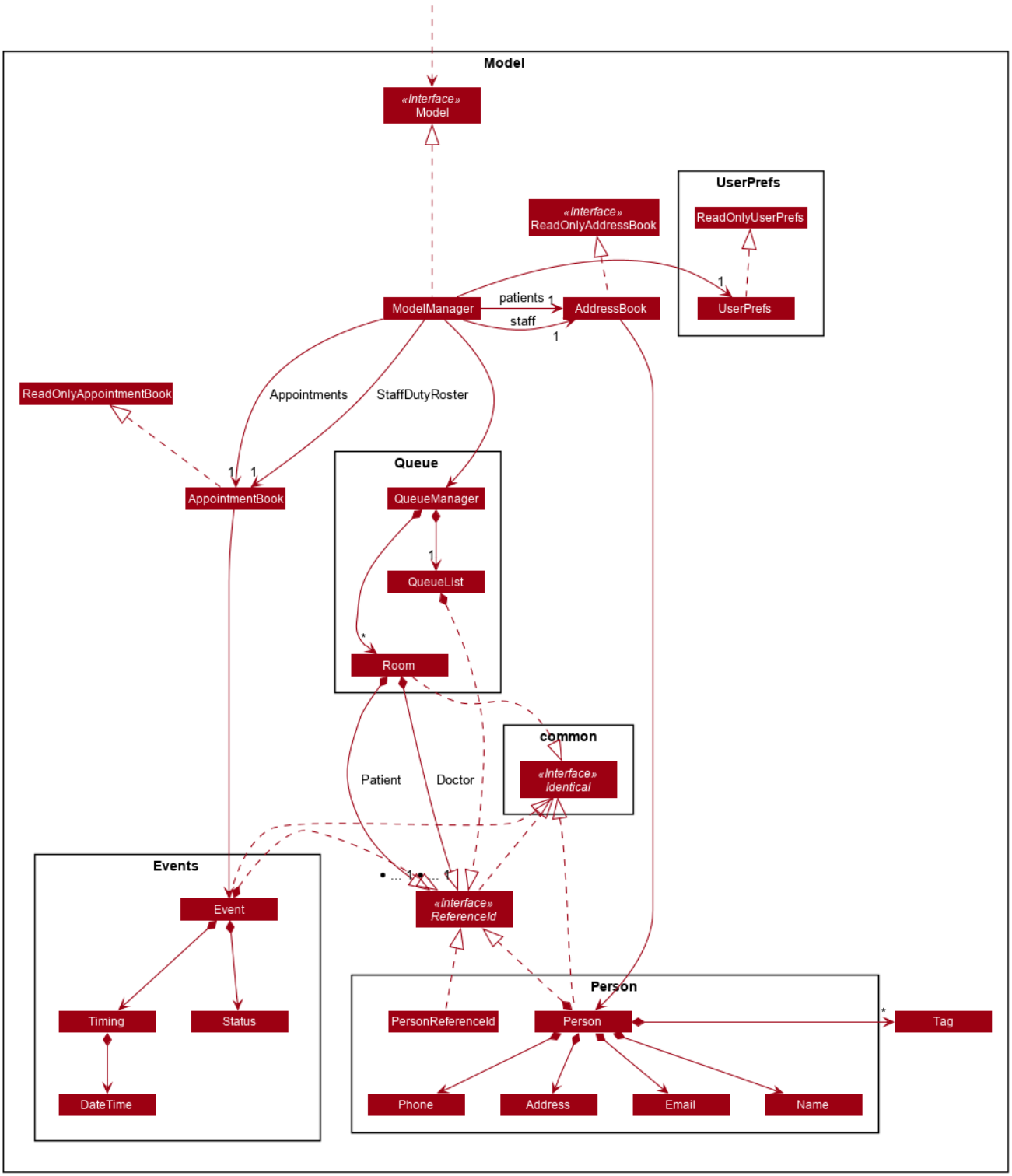


Figure 7. Structure of the Model Component

API : `Model.java`

3.4.1. Overall Model component

The `Model`,

- stores a `UserPref` object that represents the user's preferences.
- does not depend on any of the other three components: `Logic`, `Ui`, and `Storage`.

- consists of 3 sub-components: `QueueManager`, `AddressBook` and `AppointmentBook`.
- stores the details of patients and staff in 2 separate instances of `AddressBook`.
- stores the patients' appointments and duty shifts of staff doctors in 2 separate instances of `Appointment Book.
- exposes an unmodifiable `ObservableList<Person>` and `ObservableList<Event>` 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 unique list of `Tags` and `ReferenceId`, which `Person` can reference. This would allow `Address Book` to only require one `Tag` object per unique `Tag`, instead of each `Person` needing their own `Tag` object.

The `QueueManager`,

- stores a `QueueList` object that represents the queue.
- stores a `UniqueElementList<Room>` which represents a list of consultation rooms where staff doctors are stationed.

The `AddressBook`,

- stores a `Person` object that, depending on its usage, can represent either a staff doctor or patients.
- stores a `UniquePersonList` which represents a list of all registered doctors or patients.

The `AppointmentBook`,

- stores a `Event` object that, depending on its usage, can represent either a patient's appointment and staff duty shift.
- stores a `UniquePersonList` which represents a list of all registered appointments or shifts.

3.5. Storage component

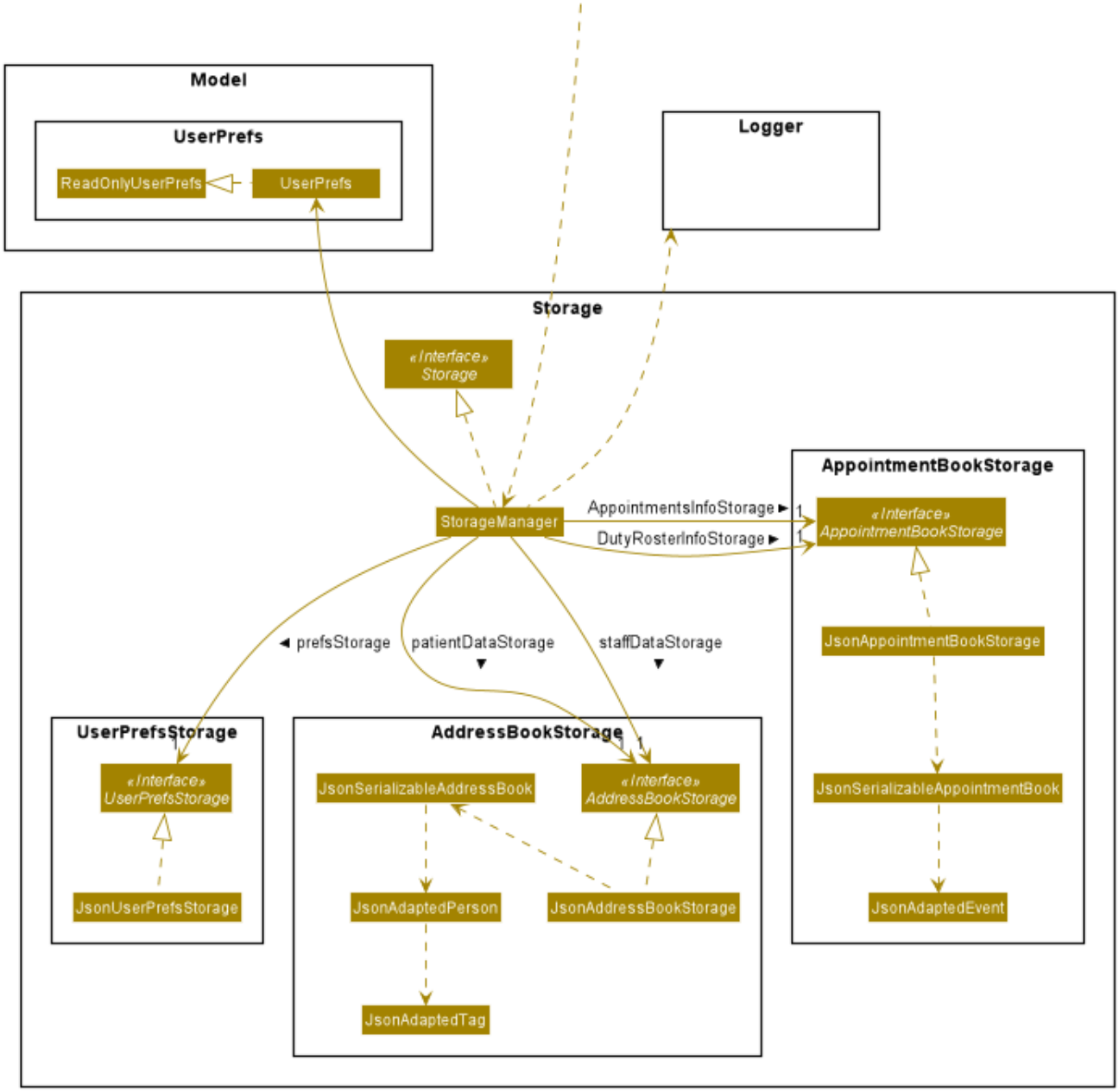


Figure 8. Structure of the Storage Component

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.
- reuses `JsonAddressBookStorage` to store the details of doctors and patients.
- reuses `JsonAppointmentBookStorage` to store the details of staff duty shifts and patient's appointments.

3.6. Common classes

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

4. Implementation

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

4.1. Unique Reference Identification

In ClerkPro, each person is assigned a unique `ReferenceId`. Appointments and duty shifts are tagged to the respective patients and doctors through the use of `ReferenceId`.

This unique reference identifier consist of two parts:

1. a unique case-insensitive string, which consist of 9 alphanumeric characters, referring to its unique ID
2. a boolean referring to whether the reference identifier belonged to a person who is registered as a patient or a staff doctor.

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. Implementation

The undo/redo mechanism is facilitated by the `CommandHistory` class, which is found in the `logic` component. The history class stores undoable commands as a `ReversibleActionPairCommand`, which pairs 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).

A `Command` object can be classified as one of two types, a `ReversibleCommand` and a `NonActionableCommand`. A `ReversibleCommand` refers to any command which modifies the data in the system's model. Commands that only reads data from the model without modifying it, will not be added to the undo history stack. Conversely, a `NonActionableCommand` only reads data from the system's model without modifying it.

Consider that the undo functionality of an 'add event' command is similar to the action of the 'delete event' command. The execution of the 'delete event' command could be done in place of the undo functionality of the 'add event' command. Hence, with such an implementation, we can avoid unnecessary duplication of code.

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

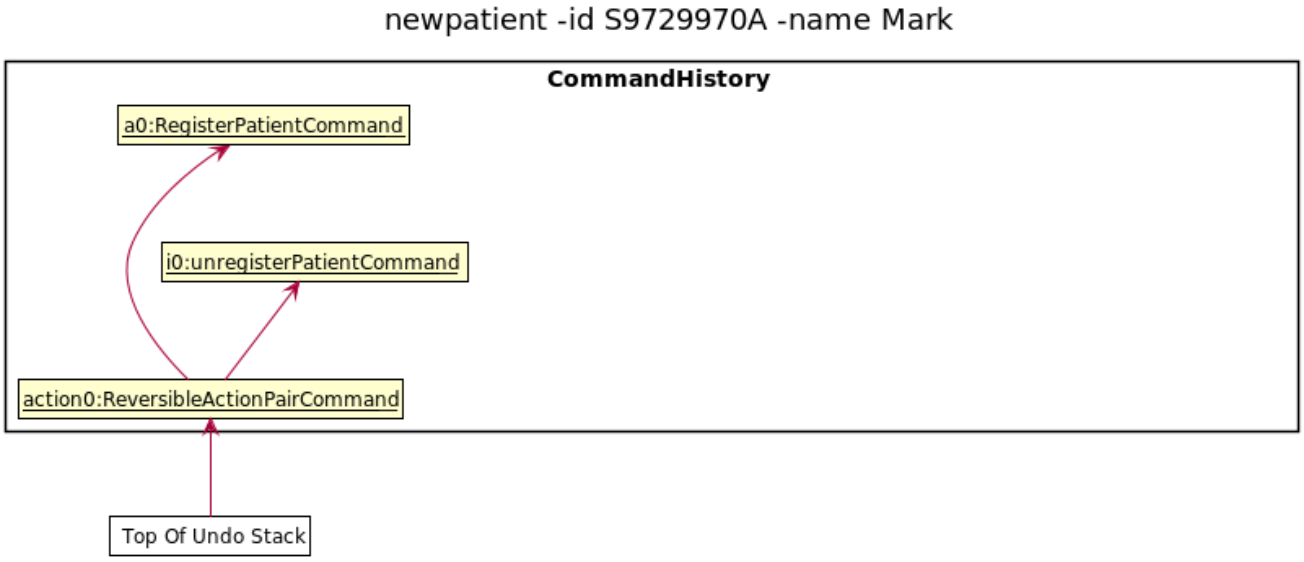
system's model in its undo history.

- `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

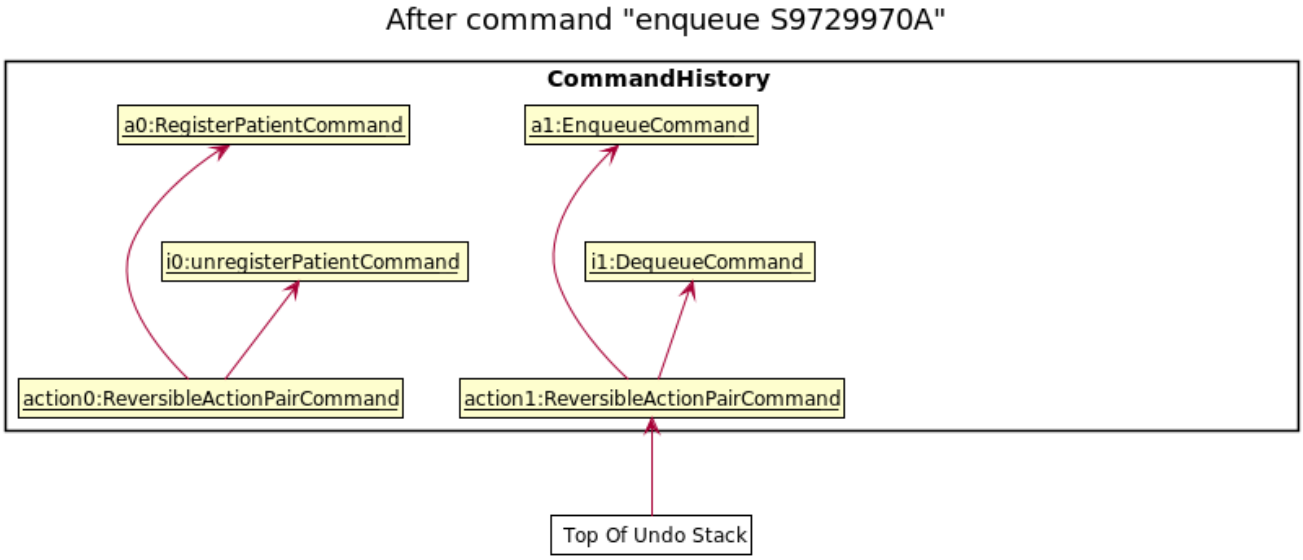
These operations are all contained within the `Logic` component and do not depend on any other components. Given below is an example usage scenario and how the undo/redo mechanism behaves at each step.

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

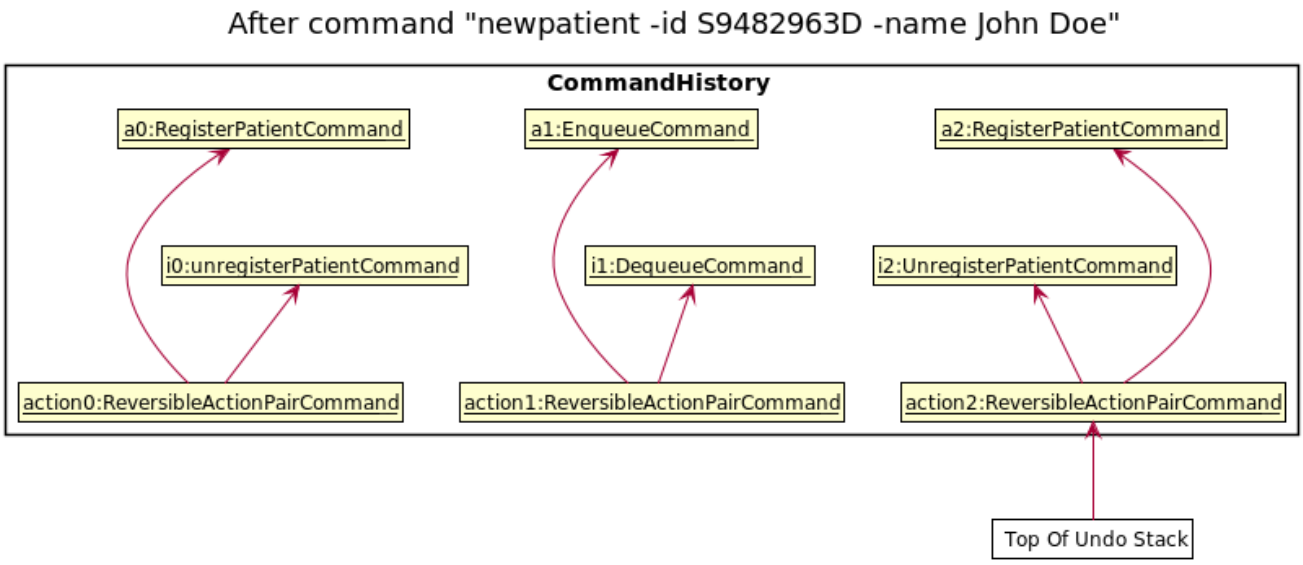
Step 2. The user executes `newpatient ⋯ -name Mark` command which registers a new patient named `Mark` with the unique id of `S9729970A`. The `newpatient` command creates a `ReversibleActionPairCommand`, which pairs of a `RegisterPatientCommand` and `UnregisterPatientCommand`. After the invoking execution 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.

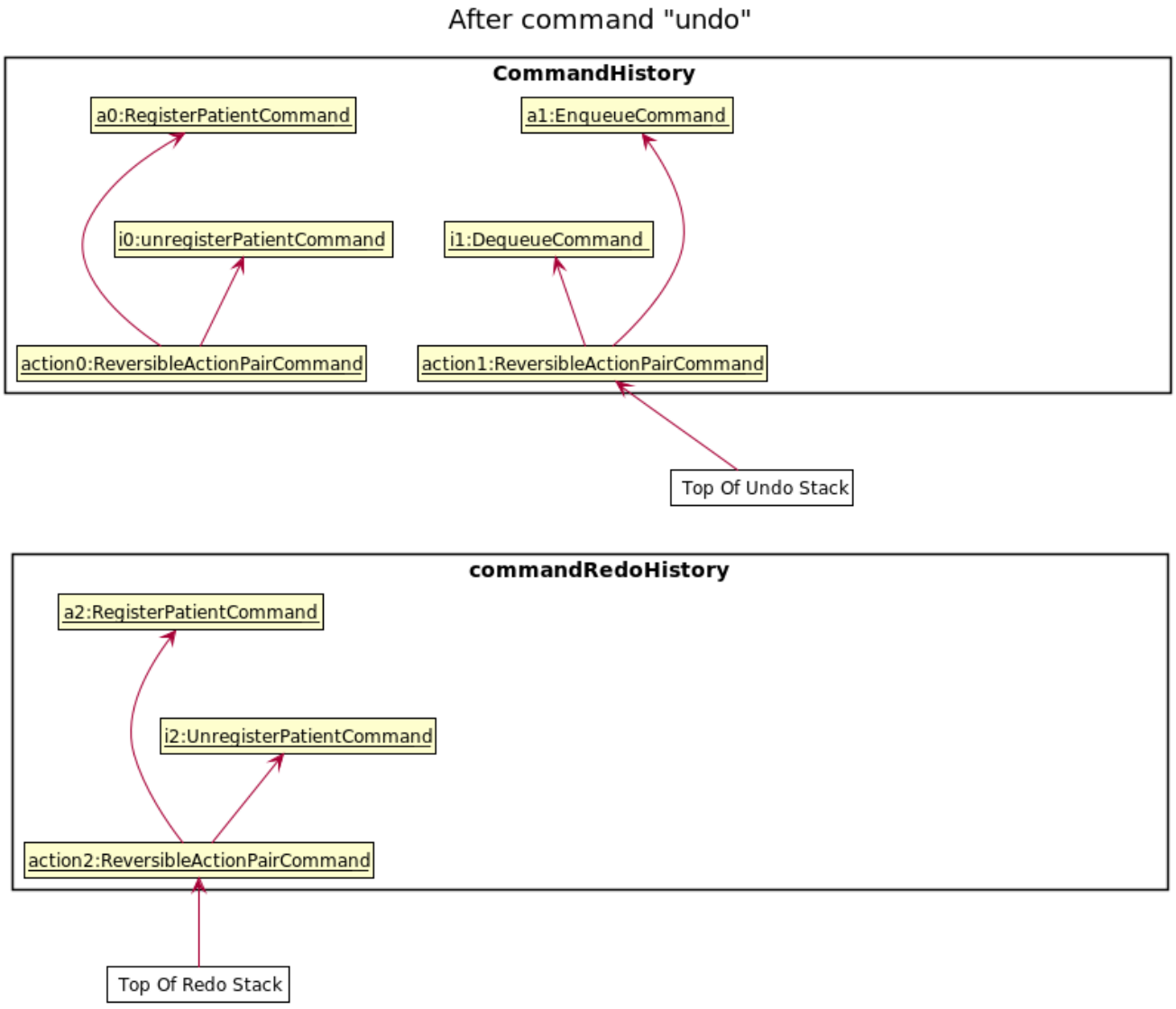


Step 4. The user executes `newpatient … -name John Doe` to register a new patient. Similar to the previous command, also causes another action pair to be added into the command history undo stack.



NOTE 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.



NOTE 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:

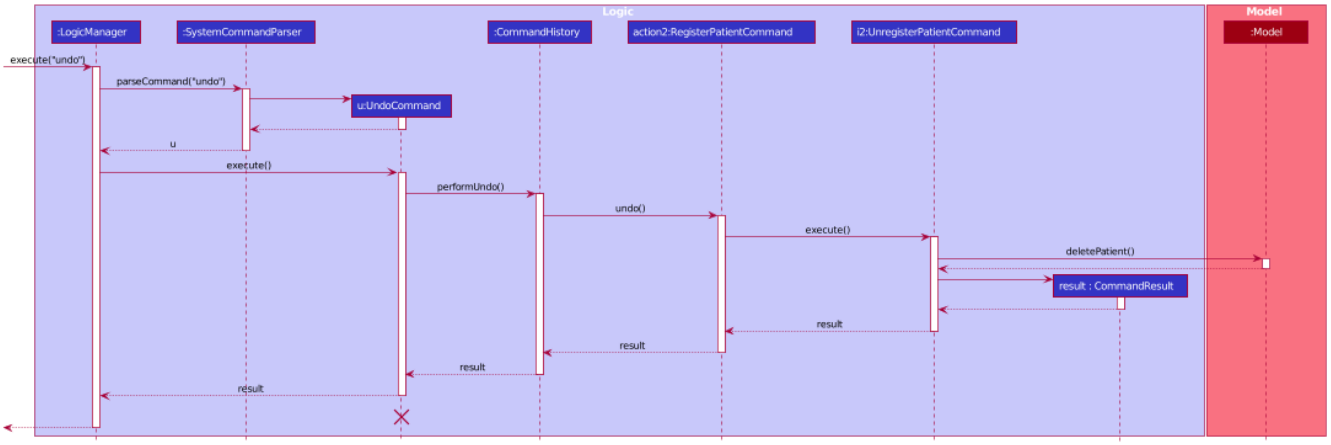


Figure 9. Undo Sequence Diagram

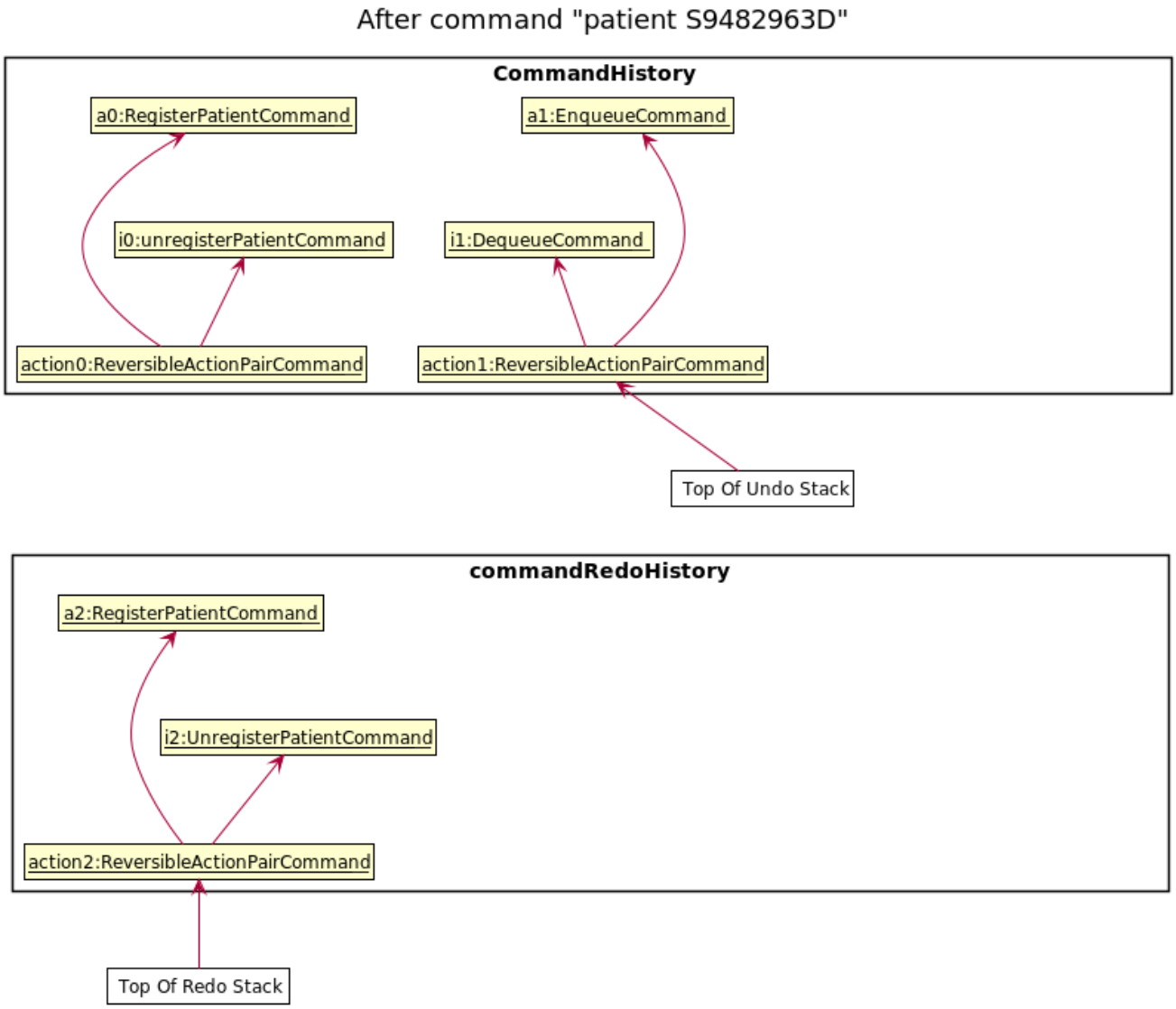
NOTE	The lifeline for `UndoCommand` should end at the destroy marker (X) but due to a limitation of PlantUML, the lifeline reaches the end of diagram.

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

NOTE	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.

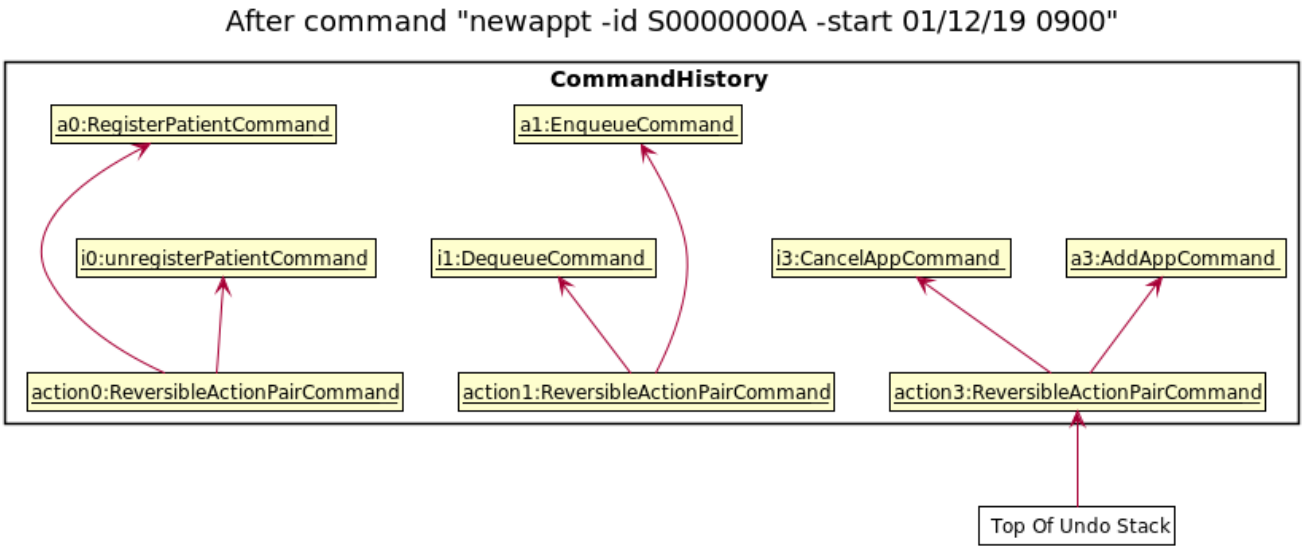
Step 5. The user then decides to execute the command `patient S9482963D`. A command that searches for a patient whose id matches `S9482963D`, only reads and does not modify any data from the model. Such commands will not call `CommandHistory#addToCommandHistory()`, `CommandHistory#performUndo()`, `CommandHistory#performRedo()`.

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.



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

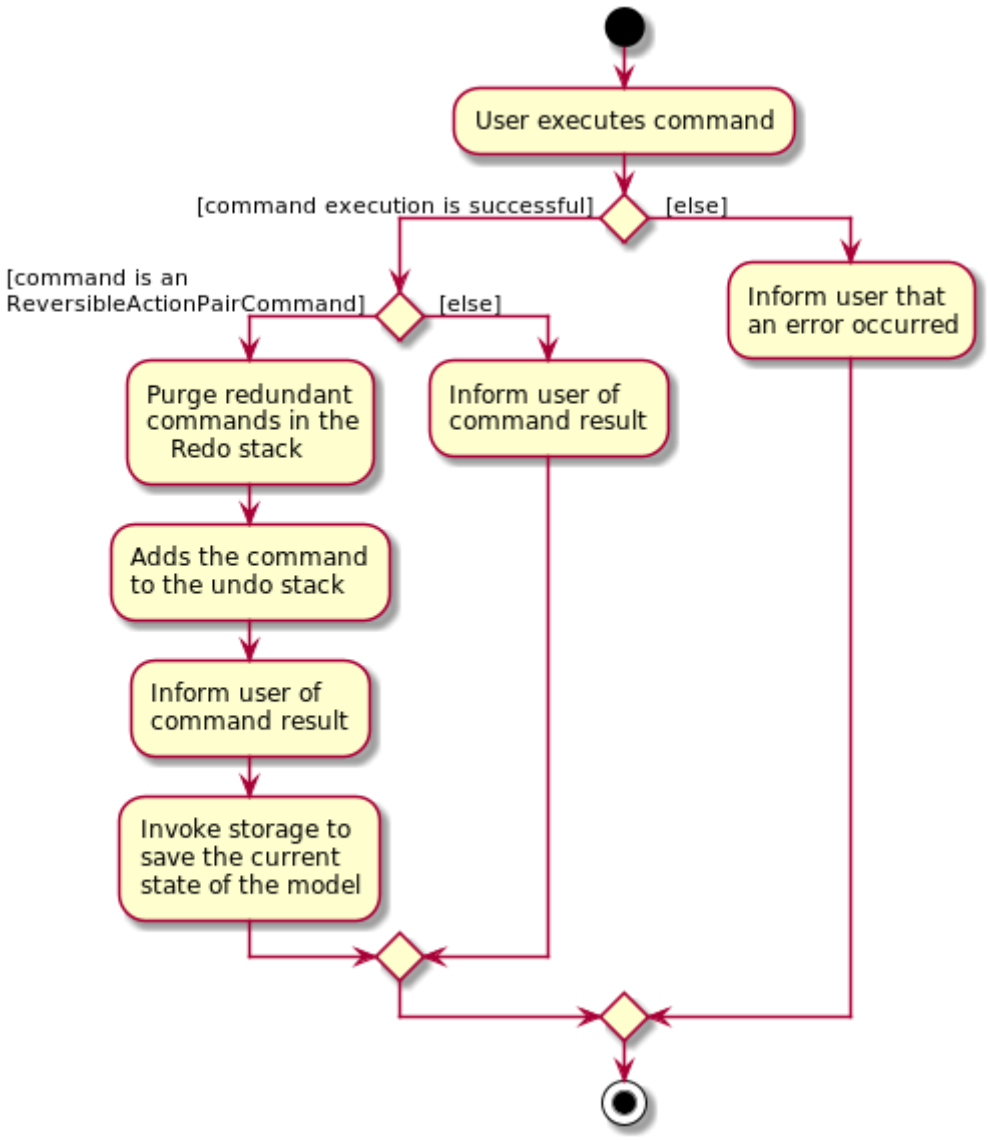


Figure 10. Commit Activity Diagram

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.

- **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. Reactive Search

The main concept behind reactive searching is that we would like time-consuming searching to be UI non-blocking and executed as rapidly as possible so that user is instantly provided with results for a given keyword.

Similar to eager evaluation, Reactive Search attempts to process commands if it does not mutate the storage. Otherwise it does a simple redirect to the tab relevant to the command.

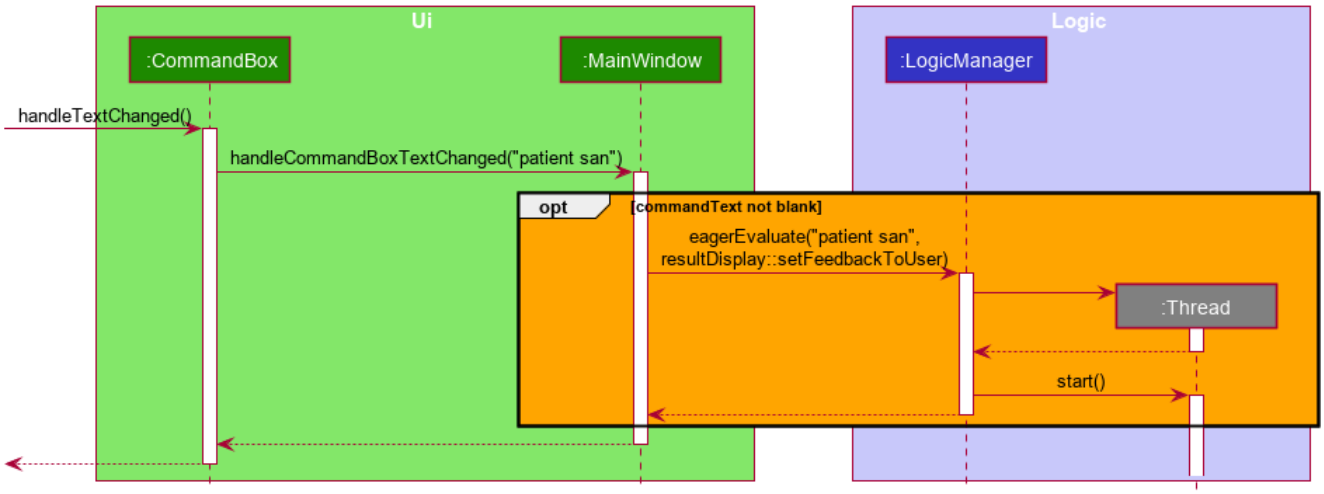


Figure 11. Reactive Search Sequence Diagram showing how a thread is created to do the expensive processing.

After the newly created thread starts, it will attempt to kill any previous thread that is still processing asynchronously before starting its own processing.

After which, changes to the ObservableList will trigger the updateItem listener in UI. And since each graphical update is fast and can only be executed from the JavaFX main application thread, we defer each of them via Platform.run() so that the FX main application thread can execute them in order whenever it is free.

```
class EventListViewCell extends ListCell<Event> {
    @Override
    protected void updateItem(Event event, boolean empty) {
        super.updateItem(event, empty);

        if (empty || event == null) {
            Platform.runLater(() -> setGraphic(null));
        } else {
            Platform.runLater(() -> setGraphic(new EventCard(event, getIndex() + 1,
displayStatus).getRoot()));
        }
    }
}
```

Hence, this ensures the main FX application thread is always polling for user inputs or graphical updates and handles them rapidly as they are lightweight tasks.

4.4. AutoComplete

We are using Prefix Tree which is known as a Trie for AutoComplete. This gives us a worst case time complexity of O(m), where m is length of the search string. While the space complexity is O(bm), where b is number of unique characters used, m is length of longest word stored.

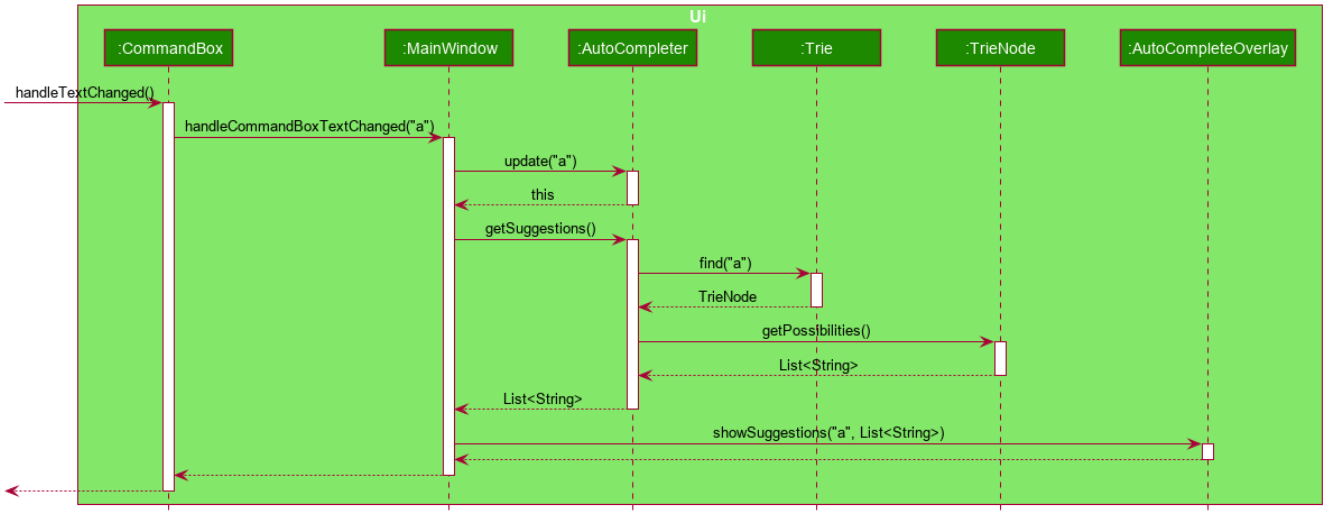


Figure 12. AutoComplete Sequence Diagram

Aspect: Data structure to implement AutoComplete

- **Alternative 1 (current choice):** Using Trie (as known as Prefix Tree).

Pros	1. Time Complexity Efficient 2. One of the most natural data structure for text prediction.
Cons	1. Requires initialisation. 2. Requires rebuilding of strings.

- **Alternative 2:** Using ArrayList.

Pros	1. Extremely simple to implement.
Cons	1. Time Complexity Inefficient.

4.5. Queue feature

The queue feature allows the user to enqueue and dequeue a patient from the queue.

- e.g. `enqueue 003A` - Enqueues the patient with `referenceId` 003A.
- e.g. `next 1` Serves the next patient in queue and allocates him/her to room 1.

Queue supports a few basic commands:

- Enqueue — Enqueues a patient into the queue.
  Format: `enqueue <PATIENT_REFERENCE_ID>`
- Dequeue — Dequeues a patient from the queue.
  Format: `dequeue <QUEUE_INDEX>`
- Next — Assigns the next patient in the queue to a doctor.
  Format: `next <ENTRY_ID>`
- Break — Avoids directing patients to a doctor. e.g. Doctor is on a lunch break
  Format: `break <ENTRY_ID>`

- Resume — Allows patients to be directed to a doctor. e.g. Doctor is back from his/her break. Format: `resume <ENTRY_ID>`

4.5.1. Current Implementation

The queue will be displayed in a list.

The following activity diagram summarizes what happens when a user executes an enqueue command:

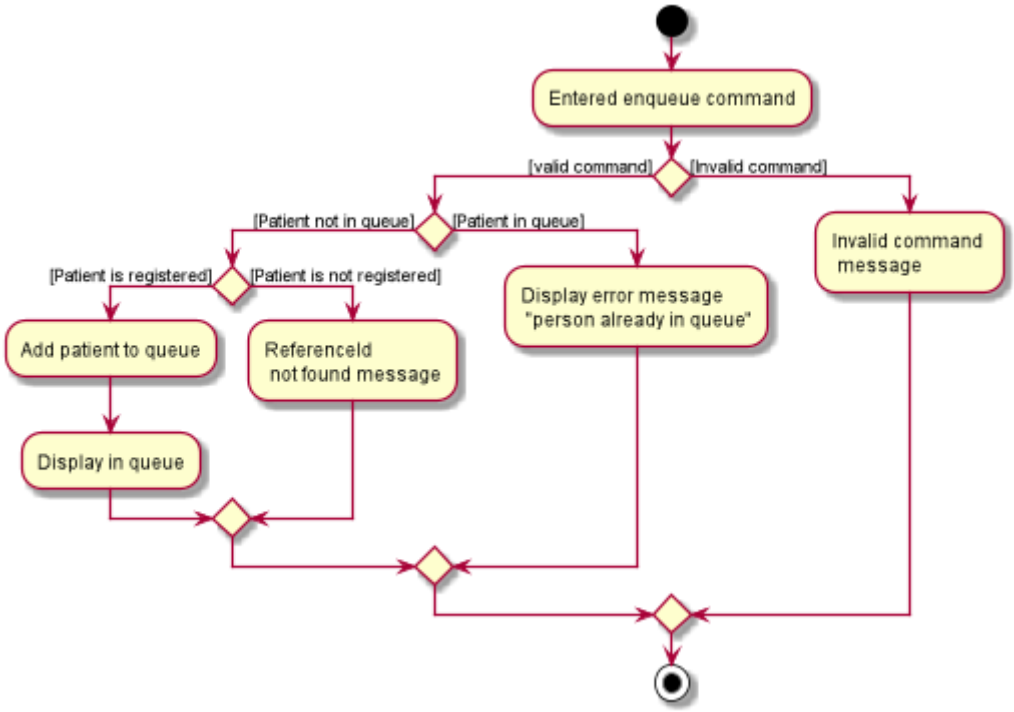


Figure 13. Enqueue Activity Diagram

The following activity diagram summarizes what happens when a user executes a next command:

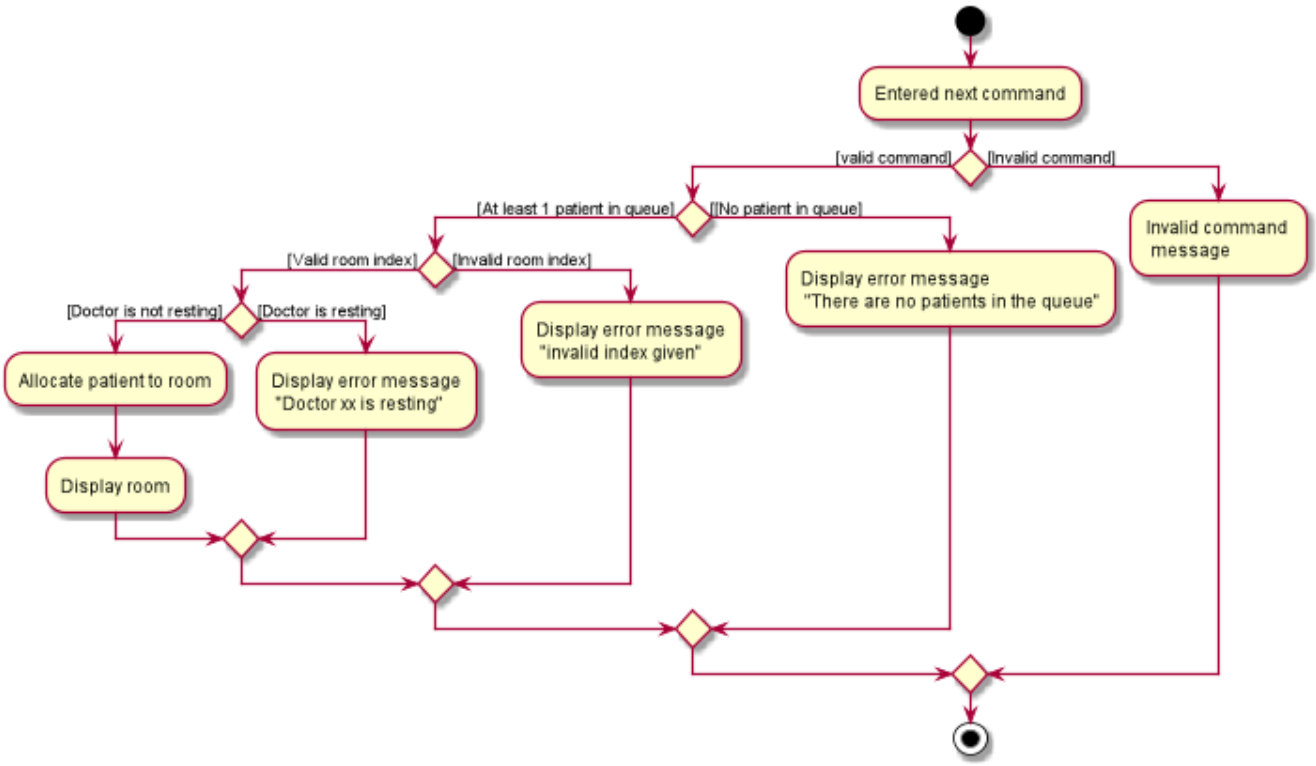


Figure 14. Next Activity Diagram

Below is an example usage of the queue feature.

Step 1: User enters the `enqueue E0000001A` command.

Step 2: The command then calls Model#enqueuePatient to enqueue the patient into the queue.

Step 3: Patient will then displayed in the queue.

4.6. Appointment feature

The Appointment feature enables users to manage appointments for patients by providing basic Create, Read, Update, Delete (CRUD) of appointments. User is also able to find missed appointments and settle each missed appointment efficiently.

- e.g. `newappt -id E0000001A -start 01/12/19 0900 -end 01/12/19 0940 -reoccur m -num 2` - creates two monthly reoccurring appointments to patient whose `referenceId` is E0000001A.
- e.g. `editappt -entry 1 -start 01/12/19 1000 -end 01/12/19 1040` - edits a patient's first appointment timing to be the input timing if there is no conflict with doctor's duty shift.

The number of scheduled appointments cannot be more than the number of on-duty staff doctors at any given time.

4.6.1. Current Implementation

The current appointment feature is implemented using a list which works like a balanced binary search tree. This allows us to search for an appointment within O(logn) instead of O(n) time, where n is the number of appointments in the list.

The Appointment feature contains multiple operations to indirectly manipulate the `UniqueEventList`. The implemented operations include:

`newappt` Command - Creates an new appointment or reoccurring appointments for a patient.

`ackappt` Command - Acknowledges an appointment whenever the patient checks in with the clerk.

`appointments` Command - Lists all upcoming appointments or appointments which involves the patient whose referenceId contains a certain keyword.

`editappt` Command - Edits an appointment timing.

`cancelappt` Command - Cancels an appointment.

`missappt` Command - Lists all missed appointments.

`settleappt` Command - Removes a missed appointment.

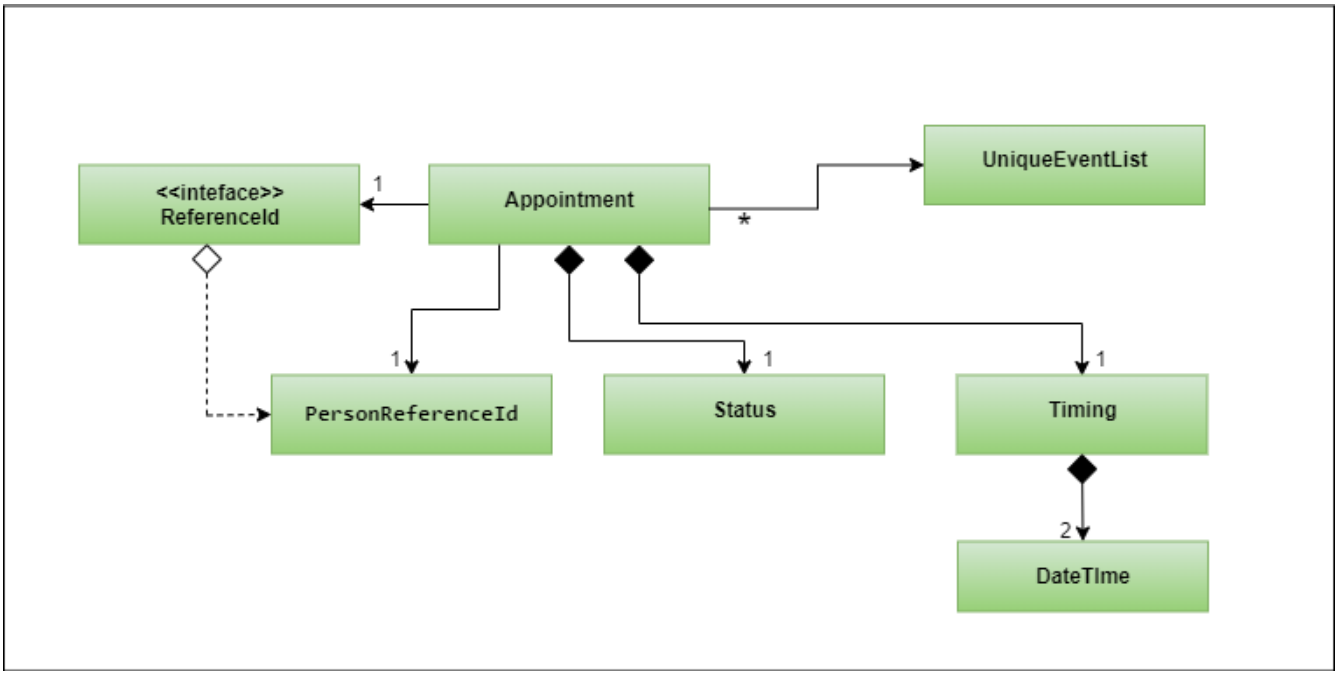


Figure 15. Appointment class diagram illustrating interactions between the Appointment class and associated classes.

he appointment will be rejected by the system, if there are insufficient staff doctors on duty at the time of the appointment.
Each `Appointment` object consists of a `PersonReferenceId`, `Timing`, `Status`. `Timing` class has 2 `DateTime` objects which indicates the start and end time of the `appointment`. The `UniqueEventList` contains 0 or more appointments.

The current implementation of `Appointment` checks with `patient` object by the unique `referenceId` and also checks the timing with doctors' dutyRoster. If the `referenceId` exists within the Model#UniquePersonList and the timing is valid, then the `Appointment` object is constructed. This ensures that the patient is registered before making any appointments and the appointment's timing is valid.

The appointment will be rejected by the system, if there are insufficient staff doctors on duty at the

time of the appointment.

Before an appointment can be scheduled, the system first checks the total number of staff doctors on duty in that timeslot. Next, the system checks the existing appointments in that timeslot. If the number of appointments in that timeslot is less than the number of doctors, the appointment will be scheduled. Otherwise, the appointment will not be scheduled.

4.6.2. `newappt` Command

The `newappt` command is similar to the `new` command of patient and doctor. The command takes in the parameters required to construct `ReferenceId`, `DateTime` and `Status`. The image below shows how the `Appointment` object is constructed.

The following activity diagram summarizes what happens when a user executes a `AddAppointment` command:

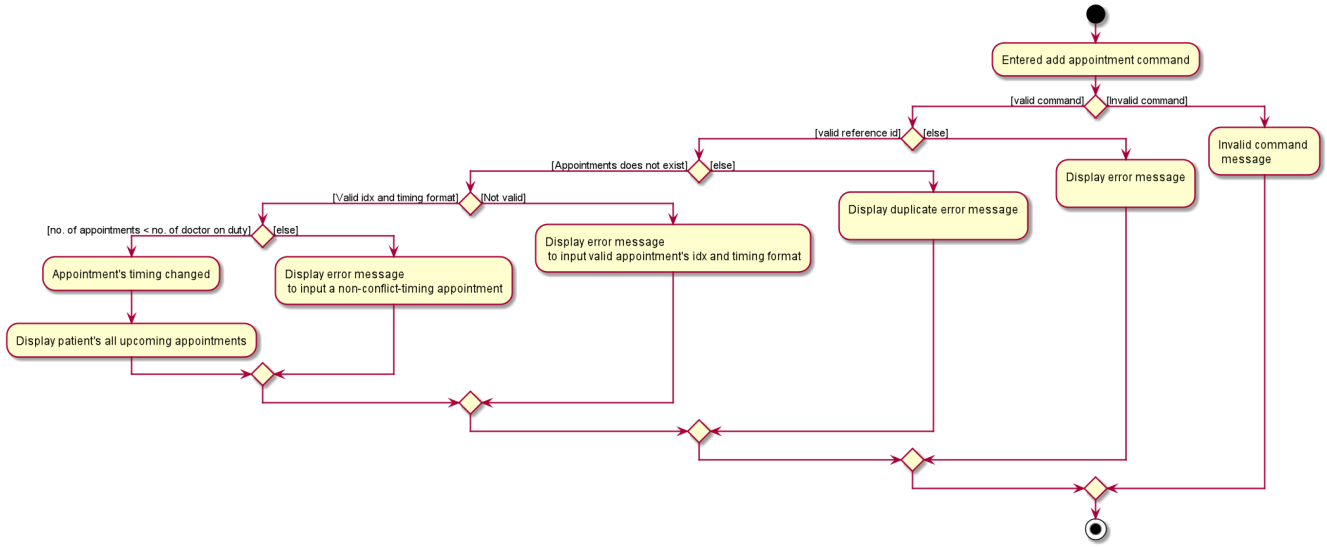


Figure 16. Interactions Inside the Logic Component when a user executes a `newappt` command

4.6.3. `ackappt` command

The `ackappt` command marks the patient's the most upcoming appointment as acknowledged only if it is on the same day and it is before the appointment's end time and also updates `UniqueEventList` to display the rest appointments belonging to the patient.

4.6.4. `appointments` Command

The `appointments` command works in two different ways.

Case 1: `appointments referenceId`

The `appointments` command searches for appointments that belong to the patient based on the given `referenceId`. The filtered appointments are found in `ModelManager`. The list is instantiated by filtering the `UniqueEventList` using `EventContainsKeywordPredicate` which is created from the `ReferenceId` argument supplied by the user.

Case 2: `appointments`

If the `appointments` command is executed without any other arguments, it executes with the predicate `EventContainsApprovedStatusPredicate`. updateFilteredAppointmentList() is called and the entire list of upcoming appointments is shown to the user.

4.6.5. `editappt` Command

The following activity diagram summarizes what happens when a user executes a `editappt` command:

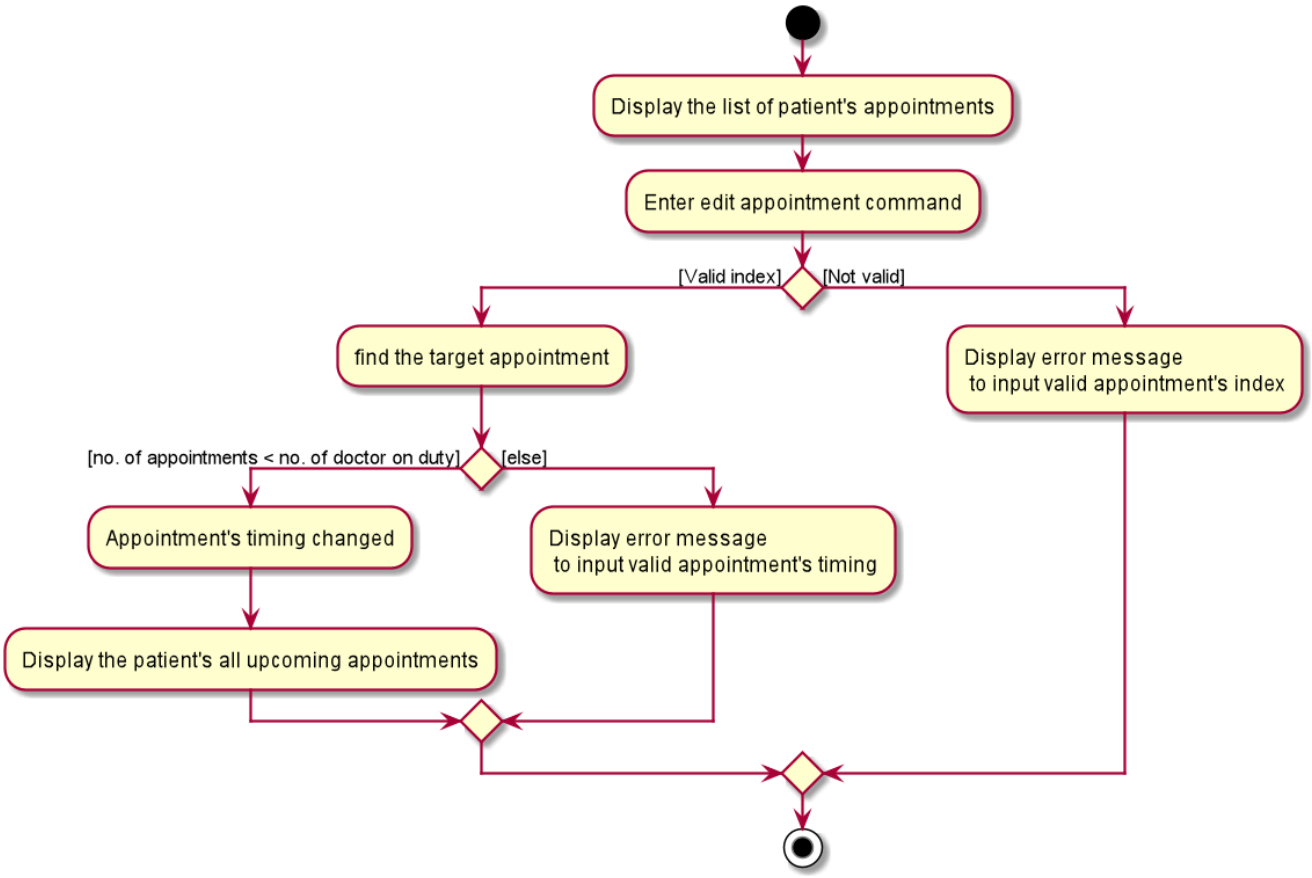


Figure 17. Interactions Inside the Logic Component when a user executes a `editappt` command

4.6.6. `cancelappt` Command

`cancelappt` simply takes in the index of the target appointment to cancel according to the displayed appointment list.

Given below is the sequence diagram for interactions within the `Logic` component for the `execute("cancelappt 1")` API call.

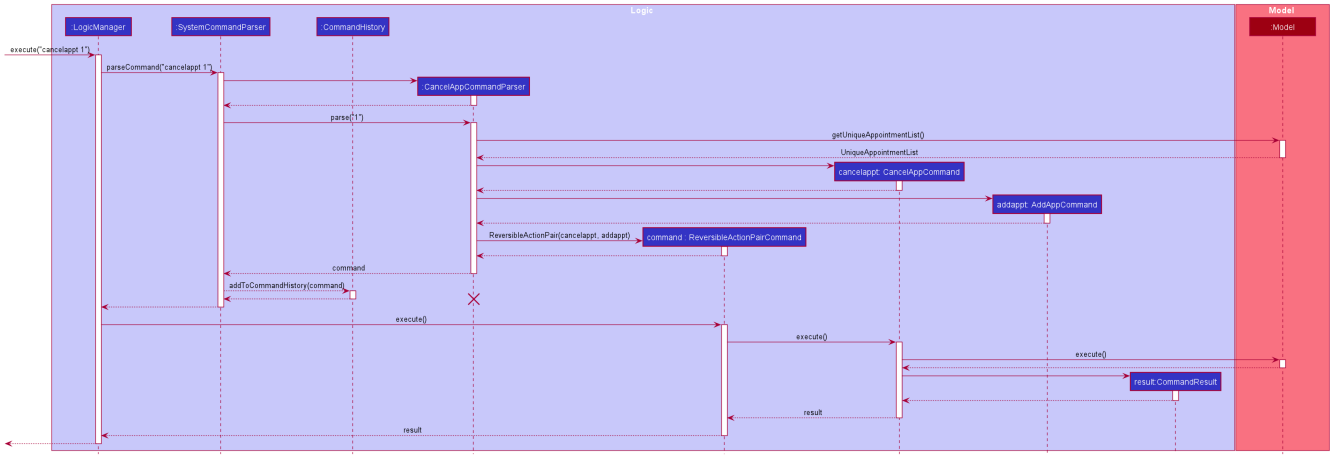


Figure 18. Interactions Inside the Logic Component for the `cancelappt 1` Command

4.6.7. `missappt` Command

The `missappt` command displays appointments that patients did not attend. The filtered appointments are found in `ModelManager`. The list is instantiated by filtering the `UniqueEventList` using `EventsMissedPredicate` which checks all APPROVED-Status appointments' end times with current time.

4.6.8. `settleappt` Command

The `settleappt` command helps users to remove any missed appointments once users have settled it. It will also update `UniqueEventList` to display the rest of the missed appointments.

In future implementations, i.e. v2.0, the valid timing slot will be given based on the doctor's availability. This ensures users can easily choose a slot to arrange appointments for patients.

4.6.9. Design Considerations

Aspect: How to store Timing fields

- **Alternative 1 (current choice):** Create `DateTime` and `Timing` class to store
  - Pros: Makes it easier to calculate timings and clashes between multiple appointments with different timing field.
  - Cons: Requires additional code to implement Timing class and interact with other common methods that rely on DateTime.
- **Alternative 2:** Store as Strings
  - Pros: Makes it easier to implement.
  - Cons: Requires additional code to convert into `DateTime` class when carrying out calculating methods.

Aspect: display appointments

- **Alternative 1 (current choice):** Display appointments in a tab
  - Pros: Creates an intuitive and easily navigable screen to display appointments.

    - Cons: Decreases the efficiency of CLI by having to use GUI inputs.
- **Alternative 2:** Keeps the onscreen clutter at a minimum and stays in line with the CLI concept.
    - Pros: Makes it easier to implement.
    - Cons: Increases difficulty in freely accessing appointments.

4.7. Duty Shift Scheduling

The duty shift scheduling provides users the ability to schedule duty shifts for doctors. It can help doctors to check, add, edit duty shifts efficiently.

- e.g. `newshift -id W0000001A -start 01/11/19 0900 -end 01/12/19 2100 -reoccur m -num 2` - allows the user to create two monthly reoccurring duty shifts to doctor whose `referenceId` is W0000001A.
- e.g. `editshift -entry 1 -start 02/12/19 0900 -end 02/12/19 2100` - allows the user to change a doctor's first duty shift to be the input timing if there is no conflict with appointments.

4.7.1. Current Implementation

The duty shift scheduling contains multiple operations to indirectly manipulate the `UniqueEventList`. The implemented operations include:

`newshift` Command - Adds a duty shift or reoccurring duty shifts to a doctor.

`shifts` Command - Lists all duty shifts involving the doctor's referenceId which contains the keyword.

`editshift` Command - Change a current duty shift's timing.

`cancelshift` Command - Cancels duty shift.

Each Duty Shift is an `Event` object consists of a `PersonReferenceId`, `Timing`, `Status`. `Timing` class has 2 `DateTime` object as they indicate the start and end times of the duty shift.

The current implementation of duty shift checks with doctor object by the unique `referenceId` and also checks the timing with appointments. If the `referenceId` exists within the Model#UniquePersonList and the timing is valid, then the duty shift is constructed. This ensures that the doctor is registered and the duty shift's timing is valid before making any duty shifts.

The duty shift will be rejected by the system, if there are insufficient staff doctors on duty at the given time.

Before a duty shift's time can be edited, the system first checks the total number of staff doctors on duty in that timeslot. Next, the system checks the existing appointments in that timeslot. If the number of appointments in that timeslot is less than the number of doctors, the duty shift's time will be changed. Otherwise, the duty shift will not be allowed to edit.

4.7.2. `newshift` Command

The `newshift` command behaves similarly to the `new` command used for patient and doctor. The command takes in the parameters required to construct `ReferenceId`, `DateTime` and `Status`.

4.7.3. `shifts` Command

The `shifts` command works in two different ways.

Case 1: `shifts ReferenceId`

The shifts command searches for duty shifts that belong to the doctor based on the given `ReferenceId`. The filtered shifts are found in `ModelManager`. The list is instantiated by filtering the `UniqueEventList` using `EventContainsKeywordPredicate` which is created from the `referenceId` argument supplied by the user.

Case 2: `shifts`

The shifts behaves similarly to `shifts ReferenceId` when it does not take in any other arguments. Instead, it automatically executes with the predicate `EventContainsApprovedStatusPredicate`. updateFilteredEventList() is called and the entire list of the upcoming duty shifts is shown to the user.

4.7.4. `editshift` Command

The following activity diagram summarizes what happens when a user executes a `editshift` command:

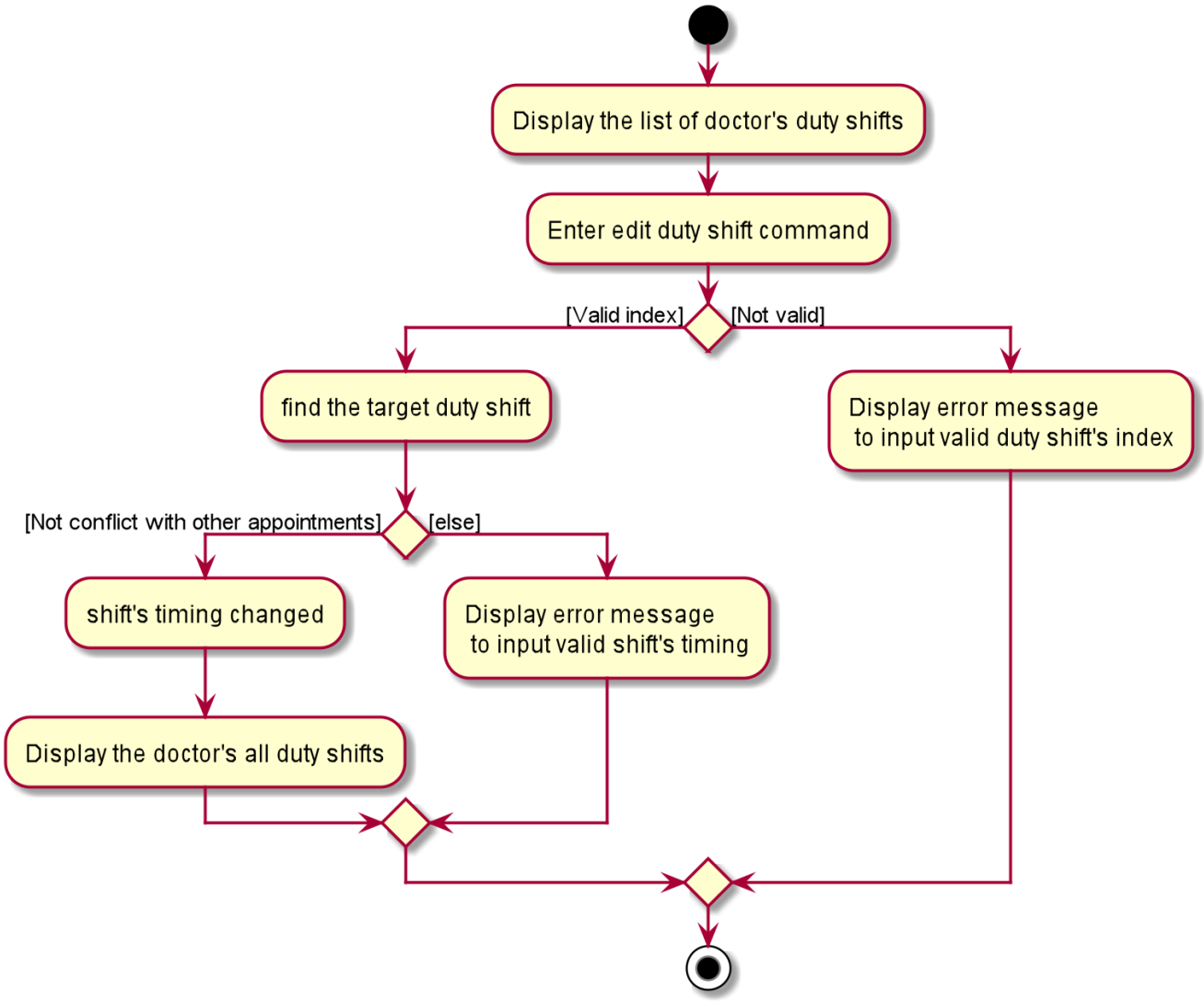


Figure 19. Interactions Inside the Logic Component when a user executes a `editshift` command

4.7.5. `cancelshift` Command

`cancelshift` simply takes in the index of the target duty shift to cancel according to the displayed shift list.

4.8. Logging

We are using `java.util.logging` package for logging. The `LogsCenter` class is used to manage the logging levels and logging destinations.

- The logging level can be controlled using the `logLevel` setting in the configuration file (See Section 4.9, “Configuration”)
- The `Logger` for a class can be obtained using `LogsCenter.getLogger(Class)` which will log messages according to the specified logging level
- Currently log messages are output through: `Console` and to a `.log` file.

Logging Levels

- `SEVERE` : Critical problem detected which may possibly cause the termination of the application

- `WARNING` : Can continue, but with caution
- `INFO` : Information showing the noteworthy actions by the App
- `FINE` : Details that is not usually noteworthy but may be useful in debugging e.g. print the actual list instead of just its size

4.9. Configuration

Certain properties of the application can be controlled (e.g user prefs file location, logging level) through the configuration file (default: `config.json`).

5. Documentation

Refer to the guide here.

6. Testing

Refer to the guide here.

7. Dev Ops

Refer to the guide here.

Appendix A: Product Scope

Target user profile:

- has a need to manage a significant number of patients and doctors
- is a clerk/receptionist working at a small clinic
- prefer desktop apps over other types
- can type fast
- prefers typing over mouse input
- is reasonably comfortable using CLI apps

Value proposition: manage queue and appointments faster than a typical mouse/GUI driven app

Appendix B: 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	see usage instructions	refer to instructions when I forget how to use the App
* * *	clerk	find out the upcoming appointments for given patients	
* * *	clerk	update the doctors' details by typing commands and user details	
* * *	clerk	add new doctors into system	
* * *	clerk	edit patients' details	keep their particulars up to date
* * *	clerk	register new patients with optional fields	
* * *	clerk	add ad-hoc patients to the queue	
* * *	clerk	search for patients using their name or phone number	
* * *	clerk	look up how many patients are in the queue, on a side panel	recommend estimated time that the patient will be attended to

Priority	As a …	I want to …	So that I can…
* * *	clerk	look up patient using a reference id	
* * *	clerk	reschedule appointments of patients	
* * *	clerk	search for appointment slots easily	schedule appointments for patients easily
* * *	clerk	assign a queue number to each patient in the queue	
* * *	clerk	use the appointment scheduler	schedule appointments for my patients
* * *	clerk	add reoccurring appointments	schedule new reoccurring appointments for my patients
* * *	clerk	save time managing the queue	have more time to do my own work
* * *	clerk	take note of the doctors that are on-shift	effectively direct patients to available doctors
* *	clerk	remove a patient from the queue if they leave.	
* *	clerk	view the number of patients who visited the clinic today	

Priority	As a …	I want to …	So that I can…
* *	clerk	schedule patient's follow up appointments	
* *	clerk	find all patients who have missed their appointments	keep track of the list of patients whom I need to inform
* *	clerk	see relevant information only	so that my focus is not lost
* *	clerk	use auto-complete to predict my commands	save time on verifying its existence and correctness
* *	clerk	quick-fill the command box with the suggestions of Auto-Complete	so that it reduces typing of the entire command
* *	clerk	refer to command history	review entered commands that maybe incorrect
* *	clerk	quick-fill the command box with history commands	inputting last few commands is easier
* *	receptionist	use the undo and redo feature	to remedy any mistakes
* *	clerk	acknowledge appointments if patients are present for their appointments	keep track of patients who came for their appointments

Priority	As a …	I want to …	So that I can…
* *	clerk	tag patient with known allergies	keep track of their allegies
* *	clerk	cancel appointments for patients	free up appointment time slots

Appendix C: Use Cases

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

Use case: Add patient into queue (UC1)

MSS

1. New patient arrives at the clinic
2. User wants to add new patient into the queue
3. System adds the patient into the queue

Use case ends.

Extensions

- 2a. User inputs invalid format
    - 2a1. System requests for correct input format.

      Use case resumes at step 2.

Use case: Remove person from queue (UC2)

MSS

1. Patient wants to leave
2. User requests to remove patient from the queue
3. System removes the patient from queue

Use case ends.

Extensions

2a. Person is not in queue

Use case ends.

3a. The given index is invalid.

3a1. System shows an error message.

Use case resumes at step 2.

Use case: Serve next patient (UC3)

MSS

1. Patient exits from room 1
2. User requests to allocate patient into room 1
3. System removes the patient from queue and allocates him/her to room 1

Use case ends.

Extensions

2a. Doctor is resting

Use case ends.

3a. The given index is invalid.

3a1. System shows an error message.

Use case resumes at step 2.

Use case: Doctor takes a break (UC4)

MSS

1. User requests to avoid directing patients to the doctor in room 1
2. System sets the doctor to be on break

Use case ends.

Extensions

1a. Doctor is already on break

- 1a1. System shows an error message.

Use case ends.

2a. The given index is invalid.

2a1. System shows an error message.

Use case resumes at step 1.

Use case: Doctor resumes his/her duty (UC5)

Pre-condition: Doctor is on break

MSS

1. User requests to start directing patients to the doctor in room 1
2. System sets the doctor to be on duty

Use case ends.

Extensions

1a. Doctor is already on duty

- 1a1. System shows an error message.

  Use case ends.

2a. The given index is invalid.

2a1. System shows an error message.

Use case resumes at step 1.

Use case: Add new a appointment (UC6)

Pre-condition: Patient exists in the system

MSS

1. Patient wants to have a new appointment
2. User wants to add a appointments for the patient
3. ClerkPro adds this appointment

Use case ends.

Extensions

- 2a. User inputs invalid format
  - 2a1. ClerkPro requests for correct input format
  - 2a2. User inputs correct format

    Steps 2a1-2a2 are repeated until the appointment has the correct format

Use case resumes at step 2

- 2b. User inputs a appointment which is conflicted with other appointments and the appointment cannot be assigned to a doctor as all doctors has other appointments with other

patients at that time.

- 2b1. User ask patient to make provide a new appointment timing
- 2b2. patient give a new appointment date

  Steps 2b1-2b2 are repeated until the appointment is not conflicted

Use case resumes at step 2.

Use case: Change appointment date (UC7)

Pre-condition: Patient's appointment exists and Application is displaying the patient's list of appointments

MSS

1. User provides a new time slots for a current apointment to change.
2. ClerkPro updates the appointment date of the patient.

Use case ends.

Extensions

- 2a. Current appointment date is invalid format
  - 2a1. ClerkPro requests for correct input format
  - 2a2. User inputs correct format

    Steps 2a1-2a2 are repeated until the appointment has the correct format
- 2b. Appointment date is conflict with other appointments
  - 2b1. User ask patient to make a new appointment
  - 2b2. patient change appointment to another date

    Steps 2b1-2b2 are repeated until the appointment is not conflicted

    Use case resumes at step 2

Use case: Find patients appointment (UC8)

MSS

1. User requests to find patient's appointment list
2. System retrieve and display patient's appointments

Use case ends.

Extensions

- 2a. patient is not exist
    - System displays an error message "No such patient"

Use case ends

C.1. Use case: cancel patient’s appointment (UC9)

Pre-condition: Patient’s record exists

MSS

1. User requests to retrieve patient’s appointment list
2. System finds and display patient’s appointments
3. User request to cancel patient’s appointment’s timing.
4. System updates patient’s appointments and patient’s appointment list
5. System displays success message of cancelling appointment’s timing

Use case ends.

Extensions

- 2a. The system cannot find the requested patient’s record
    - 2a1. System displays an error message. "No such appointment"

Use case ends

Use case: Indicate that a doctor is on-duty and able to tend to patients (UC10)

Pre-condition: Details of the doctor is already registered in system.

MSS

1. User finds the doctor using either his/her name or staff id.
2. User assigns the on-duty doctor to a consultation room.
3. System updates the ui to display the available consultation rooms and doctors.

Use case ends.

Extensions

2a. Consultation room has already been taken.

2a1. System shows an error message. Informing the user that the room has already been assigned to another doctor.

Use case ends.

2b. Doctor has already been assigned to a consultation room.

2b1. System shows an error message. Informing the user that the doctor has already been assigned to a room.

Use case ends.

Use case: AutoCompleter (UC11)

Actor: Clerk

Guarantees:

1. Display suggestions of commands available from whatever has been typed.
2. Autofill of commands selected from AutoCompleter into Command Box.

MSS:

1. Clerk types "a" into the Command Box.
2. ClerkPro shows suggestions of commands available for "a".
3. Clerk selects a command from AutoCompleter.
4. ClerkPro auto-fills the selected command into Command Box.

Use case ends.

Use case: History (UC12)

Actor: Clerk

Guarantees:

1. Autofill of commands while traversing History.

MSS:

1. If AutoCompleter is not suggesting, Clerk can traverse history commands.
2. While traversing, the command box is auto-filled with the history command.

Use case ends.

Appendix D: Non Functional Requirements

1. Should work on any mainstream OS as long as it has Java 11 or above installed.
2. Should be able to hold up to 1,000,000 persons and 1,000,000 events without noticeable lag in User Interface within 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. No internet required.
5. System allows user to undo incorrect or accidental destructive actions
6. App can be downloaded and run via a jar file

Appendix E: Glossary

Mainstream OS

Windows, Linux, Unix, OS-X

Private contact detail

A contact detail that is not meant to be shared with others

Appendix F: Product Survey

Product Name

Author: …

Pros:

- …
- …

Cons:

- …
- …

Appendix G: Instructions for Manual Testing

Given below are instructions to test the app manually.

NOTE	These instructions only provide a starting point for testers to work on; testers are expected to do more *exploratory* testing.

G.1. Launch and Shutdown

1. Initial launch
   a. Download the jar file and copy into an empty folder
   b. 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

    a. Resize the window to an optimum size. Move the window to a different location. Close the window.

    b. Re-launch the app by double-clicking the jar file.
    Expected: The most recent window size and location is retained.

{ more test cases … }

G.2. Deleting a person

1. Deleting a person while all persons are listed

    a. Prerequisites: List all persons using the `list` command. Multiple persons in the list.

    b. 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.

    c. Test case: `delete 0`
    Expected: No person is deleted. Error details shown in the status message. Status bar remains the same.

    d. Other incorrect delete commands to try: `delete`, `delete x` (where x is larger than the list size) *{give more}*
    Expected: Similar to previous.

{ more test cases … }

G.3. Saving data

1. Dealing with missed/corrupted data files

    a. *{explain how to simulate a missed/corrupted file and the expected behavior}*

{ more test cases … }