Jerrold Tan - Project Portfolio

PROJECT: VISIT

Overview

VISIT is an open-source Customer Relationship Management (CRM) software for **House Call Doctors**. It is based on a Command-Line Interface (CLI) application called AddressBook3 by the SE-EDU Initiative. The UnrealUnity Team comprising of 4 people including myself introduced features that allow these doctors on-the-go to manage their appointments in an easy and enhanced fashion. It is written in Java with its GUI created in JavaFX, and has about 16 kLoC.

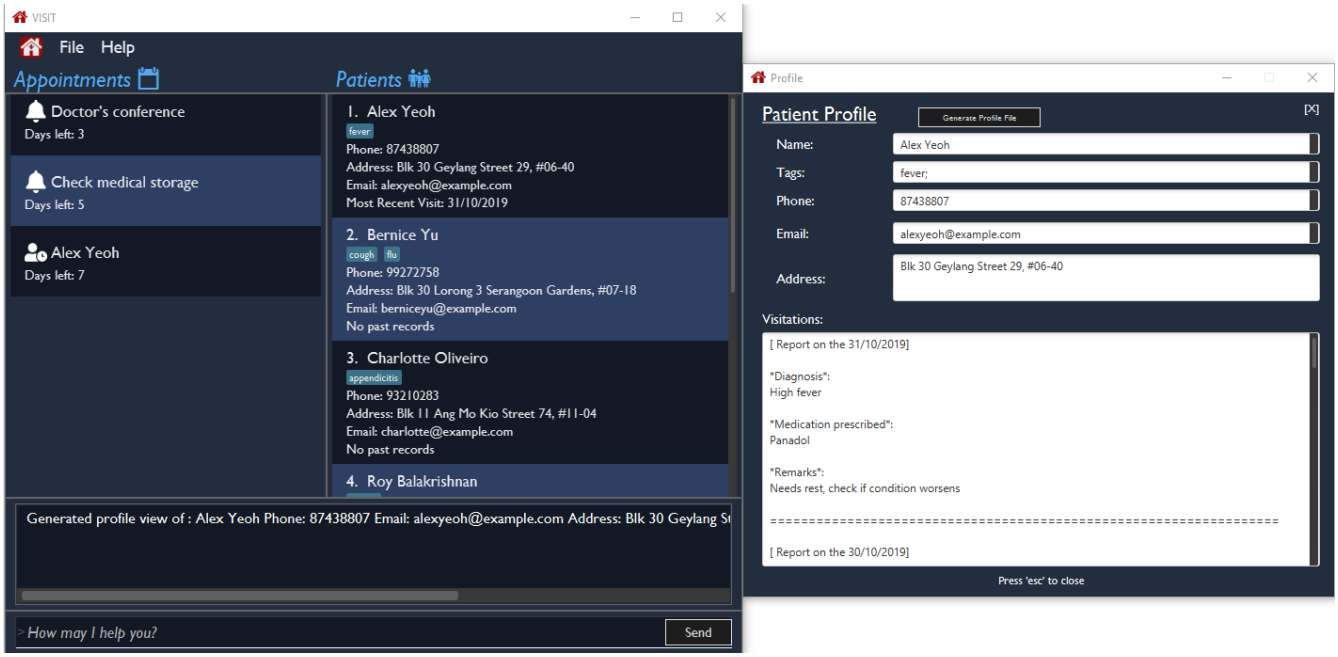


Figure 1. VISIT Main Window

I was tasked with conceptualizing and implementing the **Appointments** system comprising of **Reminders** and **Follow-ups**. This document highlights these features and provides the documentation for them from the User and Developer Guides.

Summary of contributions

- **Major enhancement**: Added a system for handling aliases
  - What it does: Allows the user to define, use and delete aliases.
  - Justification: When doctors visit patients, they need to do multiple things repeatedly, such as scheduling followups. With the inclusion of the alias feature, it streamlines doctors' workflow during visitations.
  - Highlights: This feature requires a dictionary data structure to store all the user-defined

aliases which are defined. It also matches the longest starting substring instead of wholesale matching and replacing of the aliases.

- **Code contributed**: [RepoSense]
- **Other contributions**:
  - Project management:
    - Vetted and assigned incoming Issues to team members.
  - Documentation:
    - Worked on documentation for the project, example: #96
  - Community:
    - Reported bugs and suggestions for other projects using AddressBook3 as a base (examples: Code review of NurseTraverse)

Contributions to the User Guide

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

Optimization for Command-Line Interface (CLI) Users

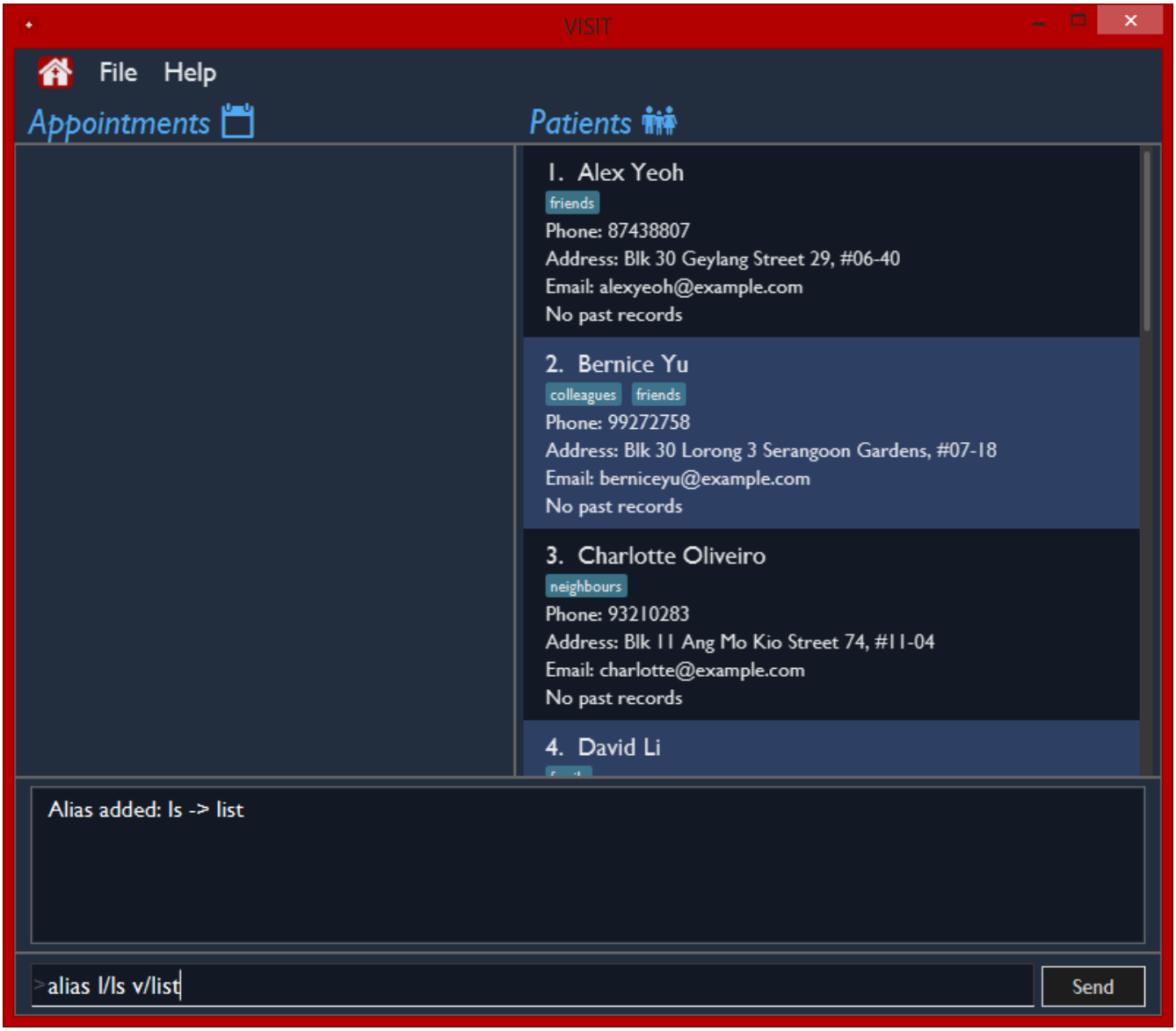
VISIT being an application optimized for doctors who are comfortable with CLI, has support for features such as aliasing for commands to enable command entries which are often used to be stored and used quickly and easily.

Setting a shorthand command / alias : `alias`

Set an alias for a command to enable faster command entry for a customized input. This command will override your previous mapping for the alias if there exists a prior mapping. Additionally, using existing commands as aliases is disallowed. These aliases are persistent and exists across multiple sessions.
Format: `alias l/SHORTHAND v/COMMAND`

Examples:

- `alias l/ls v/list`
  Typing `ls` now works equivalently as typing `list`.
- `alias l/display all patients v/list`
  Typing `display all patients` now works equivalently as typing `list`.

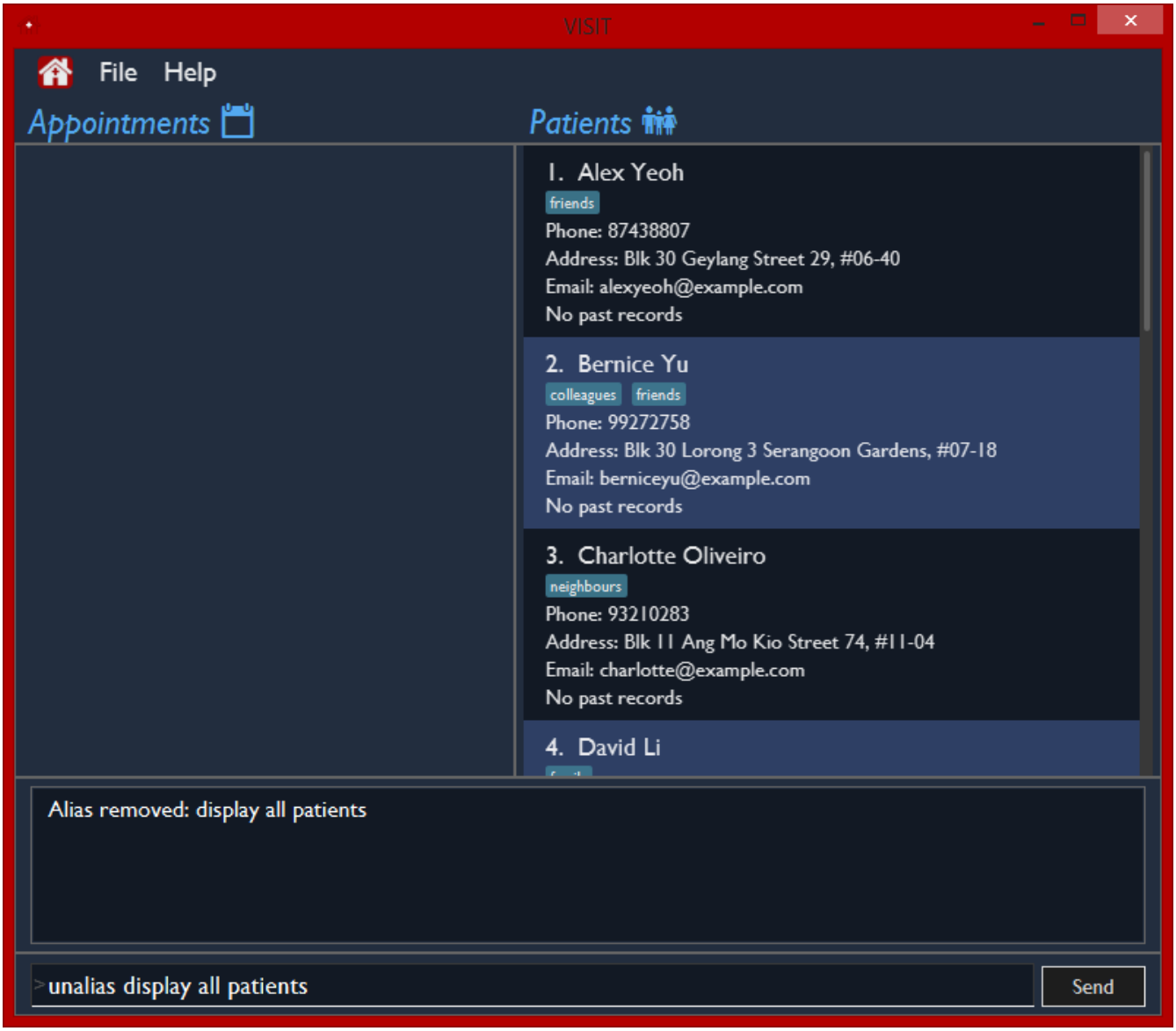


Removing a shorthand command / alias : `unalias`

Delete an existing alias, if it exists.
Format: `unalias SHORTHAND`

Examples:

- `unalias ls`
  Typing `ls` will no longer be equivalent to typing `list`.

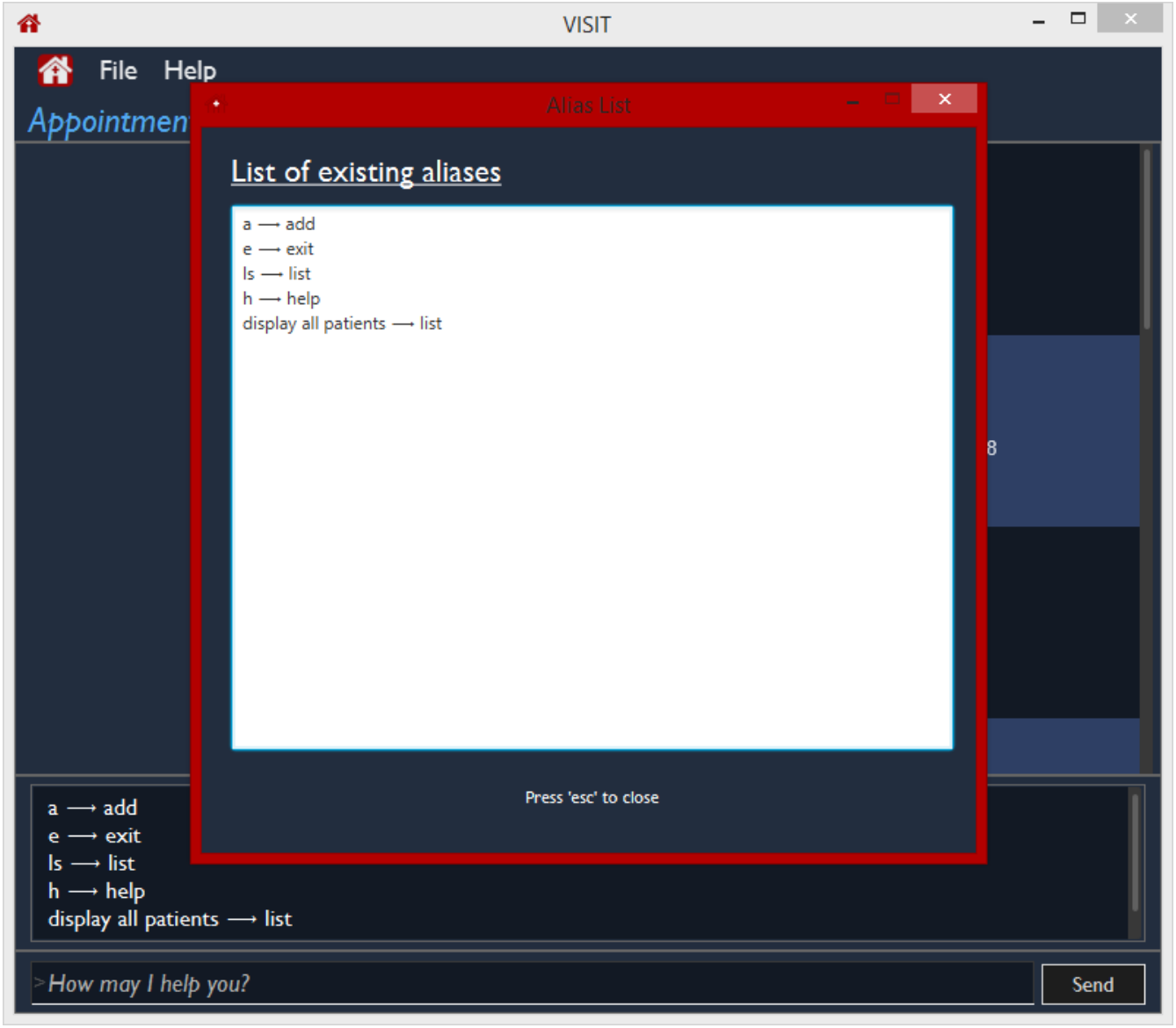


Showing all existing shorthand commands / aliases : `aliaslist`

Show all existing aliases.
Format: `aliaslist`

Examples:

- `aliaslist`
  Shows all the existing aliases.



Contributions to the Developer Guide

Given below are 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.

Alias feature

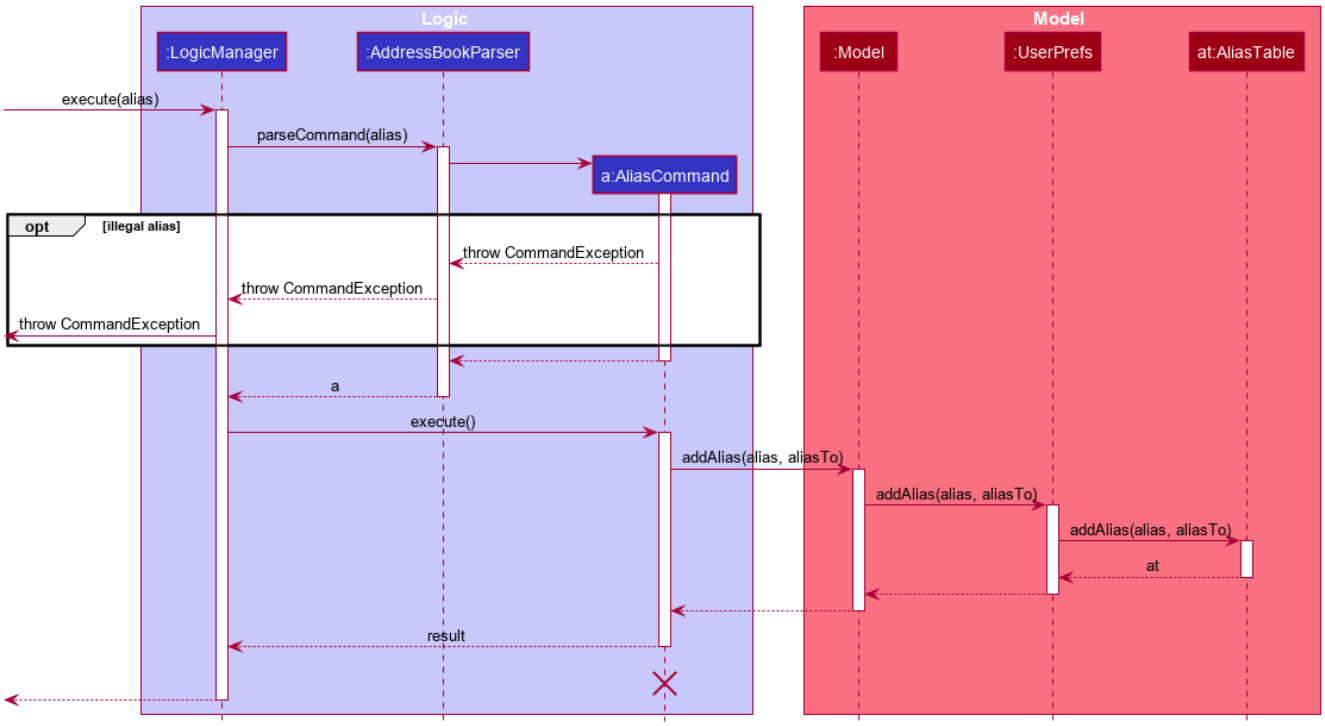
Implementation

The alias mechanism is facilitated by `AliasTable`. At a lower level, saving of aliases is facilitated by use of a `HashMap`. It is contained within `UserPrefs` and can be serialized together with the rest of the members in `UserPrefs`. Additionally, it implements the following operations:

- `AliasTable#applyAlias(commandText)` — Applies the longest stored aliases to the supplied command.
- `AliasTable#addAlias(alias, aliasTo)` — Adds a new alias to the alias table.
- `AliasTable#removeAlias(alias)` — Removes a existing alias from the alias table.

These operations are exposed in the `Model` interface as `Model#applyAlias(commandText)`, `Model#addAlias(alias, aliasTo)` and `Model#removeAlias(alias)` respectively.

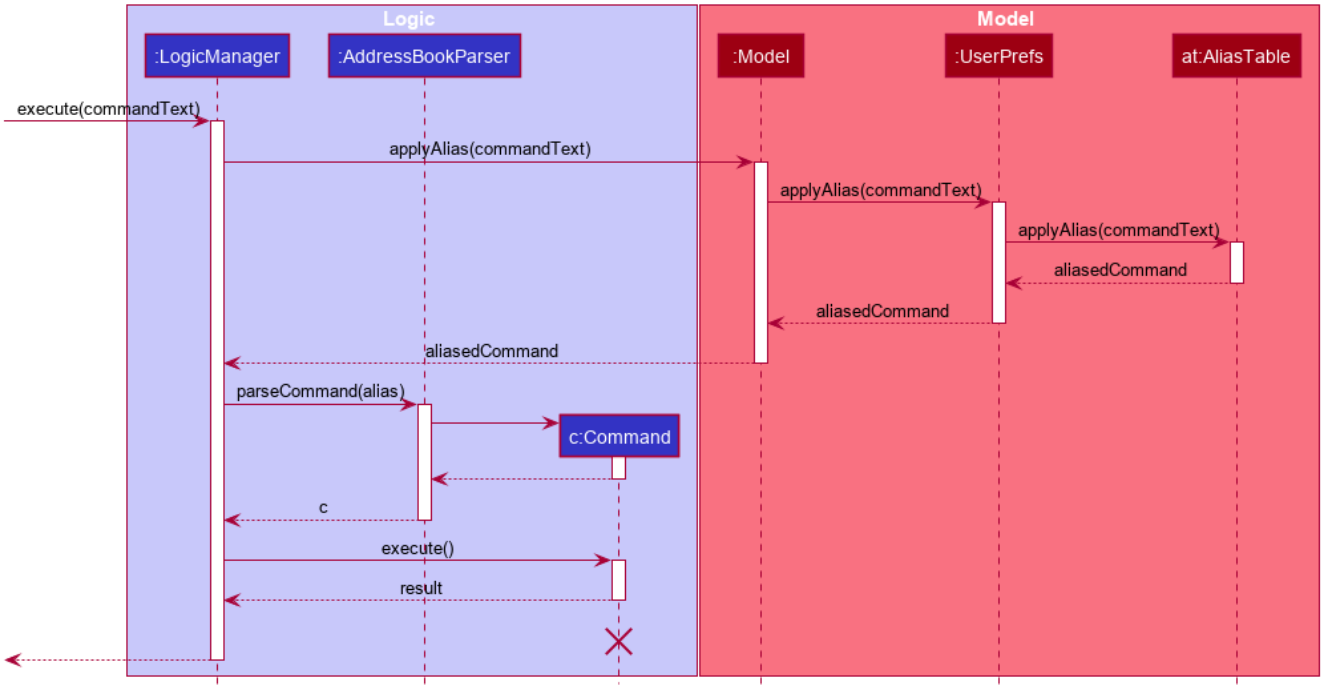
The following sequence diagram shows how adding an alias works:



NOTE The check for and illegal alias consists of creating a new `AddressBookParser` and parsing the given alias into it, which is not shown in this diagram.

The `unalias` command does the opposite — it calls `Model#removeAlias(alias)` instead which calls `Model#UserPref(alias)` and `AliasTable#removeAlias(alias)`

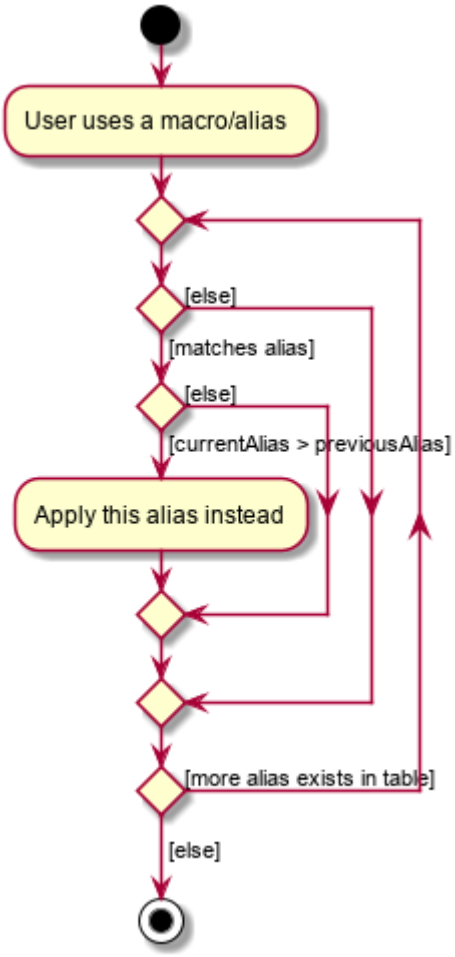
The following sequence diagram shows how applying alias works:



NOTE This is a generalized diagram which depicts what happens between `execute(commandText)` and `LogicManager#parseCommand(alias)`. This process is not shown in other sequence diagrams.

A user-defined alias is considered a match with the user input if the alias is a substring, that starts from the beginning, of the user input. Specifically, `AliasTable` uses the regex `(ALIAS)($| ).*` to check if it is a match. Following that, it picks the longest matching alias to apply to the user input.

The following activity diagram shows how applying alias picks which alias to apply:



NOTE An unfortunate side effect to matching the longest matching macro increases the time complexity of this operation to O(n) from O(1) if we used wholesale matching instead.