Chaitanya Baranwal - Project Portfolio

PROJECT: Insurelytics

Overview

Insurelytics is a desktop client management application for insurance agents. Insurelytics facilitates the tracking of client and policies, and also generate statistical analytics to provide insights for the agent. The user interacts with it using a CLI, and it has a GUI created with JavaFX. It is written in Java, and has about 85 kLoC.

Summary of contributions

- **Major enhancement**: added **the ability to undo/redo previous data changes, and view previous history of commands**
  - **What it does:** The features relating to this enhancement can be divided into two components:
    - **Undo/redo previous data changes**: Allows the user to undo/redo any data changes made to the insurance data in the current session.
    - **View a history of previously entered valid commands**: Allows the user to view a history of previously entered valid commands.
  - **Justification:** A user can make mistakes in commands and the app should provide a convenient way to rectify them. Given the sensitivity and size of data handled by a single insurance agent, it is quite possible to mistakenly enter erroneous data. The `history` command is also very useful, since the user would often want to see the previously entered commands to note the changes he/she made.

    - **Highlights:** This enhancement affects existing commands and commands to be added in future. The implementation too was challenging as it required changes to existing commands and key components like `ModelManager` and `LogicManager`, as well as implementation of completely new components.
    - **Credits:** The code for `StatefulAddressBook.java` is adapted from this repo, although I made sure that I completely understood the implementation and wrote the code for this application myself. The `history` command did not use any previously existing code.
- **Minor enhancement**: added a functionality that allows the user to navigate to previous commands using up/down keys.
    - Different from the `history` command, because this stores the entered commands in a separate list, and provides all commands entered regardless of whether they were valid or not. The idea is to provide incorrectly entered commands so that the user can quickly navigate to that command and perform the required changes to enter the correct command.
- **Code contributed**: tP Dashboard
- **Functional Code contributed**:
    - Model: [StatefulAddressBook] [CommandHistory]
    - UI: [HistoryListCard] [HistoryListPanel]
    - Commands: [UndoCommand] [RedoCommand] [HistoryCommand] and their parsers
- **Test Code Contributed**:
    - Model: [StatefulAddressBookTest] [CommandHistoryTest]
    - Commands: [UndoCommandTest] [RedoCommandTest] [HistoryCommandTest] [… other tests added to `ModelManager`, `LogicManager` and `AddressBookParser`]
- **Other contributions**:
    - Enhancements to existing features:
        - Added commands (complete with tests) for assigning and unassigning policies (Pull requests #88 and #94).
    - Documentation:
        - Updated the existing Developer Guide diagrams to reflect the new application and models #127
    - Community:
        - PRs reviewed (with non-trivial review comments): #95, #196, #132, #228

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.

Editing a person : `edit`

Edits an existing person in the address book.
Format: `edit INDEX [n/NAME] [ic/NRIC] [p/PHONE] [e/EMAIL] [a/ADDRESS] [dob/DATE_OF_BIRTH] [g/GENDER]`

- Edits the person at the specified `INDEX`. The index refers to the index number shown in the displayed person list. The index **must be a positive integer** 1, 2, 3, …
- At least one of the optional fields must be provided.
- Existing values will be updated to the input values.
- If editing a person’s details results in him being ineligible for a policy/multiple policies he currently possesses, they will be unassigned from him.

NOTE To edit tags, use `addTag` or `deleteTag`

Example:

```
edit 1 p/91234567 e/johndoe@example.com
```

Considering the first person is `Alex Yeoh`, sample output is:

```
Edited Person: Alex Yeoh
NRIC: S0000001A; Phone: 91234567; Email: johndoe@example.com; Address: Blk 30 Geylang
Street 29, #06-40; Date of Birth: 12 December 1998; Gender: Male
Policies: [Teenage]
Tags: [diabetic]
```

Adding policies to a person : `assignpolicy`

Assigns a policy to the person at the specified index.
Format: `assignpolicy INDEX pol/POLICY NAME`

- The index refers to the index number shown in the displayed person list. The index **must be a positive integer** 1, 2, 3, …
- The policy name refers to the name of the policy.
- A policy already assigned cannot be assigned again.
- Each policy name must match the policy exactly as it appears in the absolute policy list.
- Any number of policies can be added as long as the person is eligible for the policy.

Examples:

```
`find Betsy`
`assignpolicy 1 pol/Accident Insurance`
```

Expected output:

```
Assigned Policy: Accident Insurance to Person: Betsy Kumar
```

Undo recently entered commands : `undo`

User can simply enter the command `undo` to undo the most recent address book data change.

Format: `undo`

Expected Output:

```
An undo has been performed!
```

NOTE An undo does not work for commands which do not make a change in the address book data (like `listpeople` for instance).

Redo recently undone commands : `redo`

User can simply enter the command `redo` to redo the most recent address book data change. A redo is possible only when an undo has been previously performed.

Format: `redo`

Expected Output:

```
A redo has been performed!
```

NOTE A redo does not work for commands which do not make a change in the address book data (like `listpeople` for instance).

Listing command history : `history`

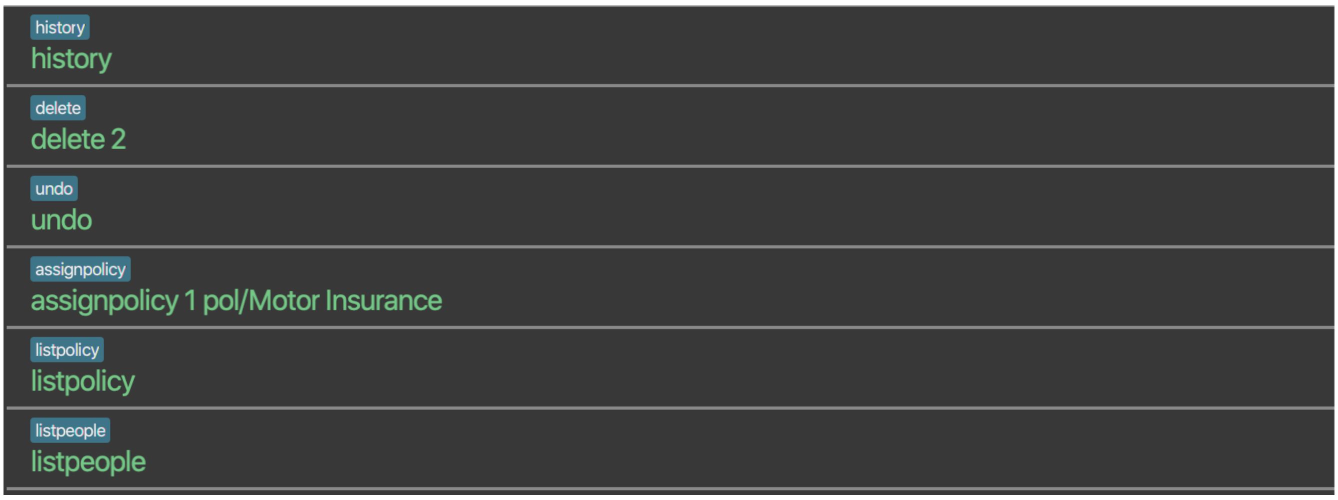
Shows a list of all previously entered (valid) commands in the right panel. Commands entered during the merging period are not included. `history` itself is taken as a valid command too.

Format: `history`

Example:

```
# input commands
>> listpeople
>> listpolicy
>> assignpolicy 1 pol/Motor Insurance
>> undo
>> delete 2
>> history
```

Expected Output:


```
history
history
delete
delete 2
undo
undo
assignpolicy
assignpolicy 1 pol/Motor Insurance
listpolicy
listpolicy
listpeople
listpeople
```

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.

Command History Feature

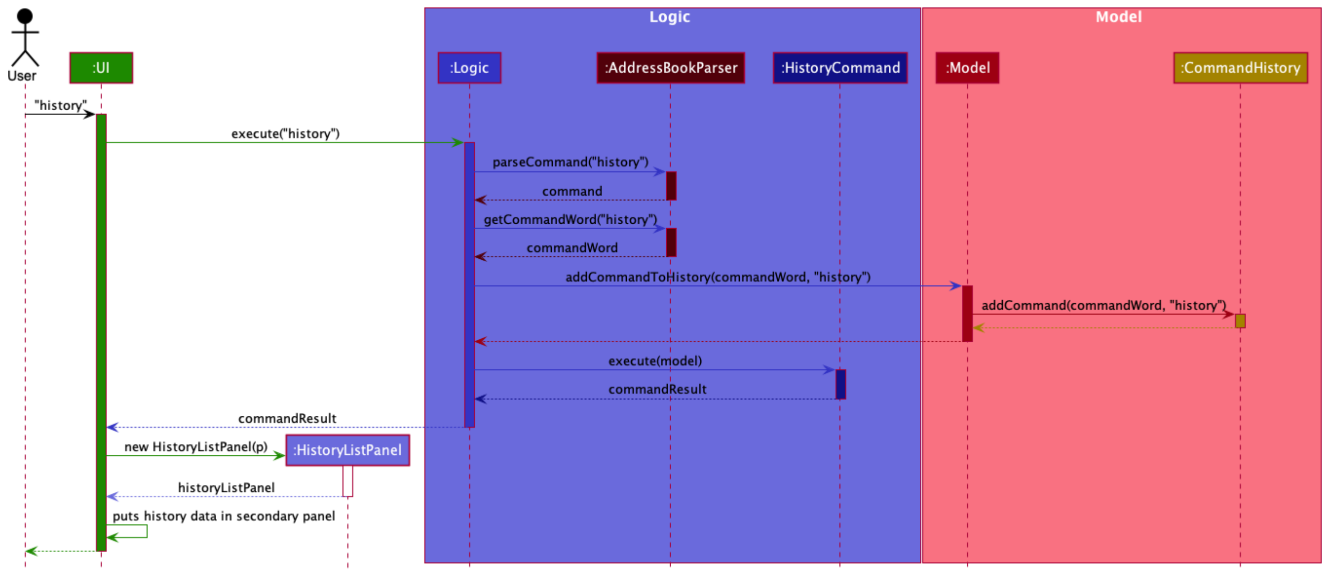
Implementation

To allow users to view the list of previously entered commands, a command history mechanism is implemented, which lists down all the previous (valid) commands entered by the user from the point of starting the application. Commands entered while merging are not added. The feature is supported by the `CommandHistory` class, an instance of which is stored as one of the memory objects inside `ModelManager`. The main operations implemented by this class are:

- `addCommand(commandWord, commandText)` — Adds the command with the command word `commandWord` and full command text `commandText` into the list of previously entered commands `userInputHistory`.
- `getHistory()` — Reverses the list of previously entered commands and returns the list.

The history view is accompanied by its associated `HistoryCard` and `HistoryListPanel` FXML. These views share a common CSS with other FXML files.

Following is the sequence diagram that shows the interaction between different components of the app when the command `history` is typed in by the user:



Design Considerations

Aspect: Where to store the CommandHistory object

- **Alternative 1:** Place the `CommandHistory` object directly in the `LogicManager` class
    - Pros: Since the user input is being parsed in `LogicManager`, storing the command history ensures that data is not being passed around between the `LogicManager` and the `ModelManager`.
    - Cons: By right the `LogicManager` is only concerned with handling logic, any data storage should be performed in `ModelManager`. Violates Single Responsibility Principle and Separation of Concerns.
- **Alternative 2 (current choice):** Place the `CommandHistory` object in model manager, parse user input in `LogicManager` and pass it to `ModelManager`.
    - Pros: Separation of concerns and Single Responsibility Principle for `LogicManager` and `ModelManager` is maintained.
    - Cons: Increases coupling between `LogicManager` and `ModelManager`, thereby violating the Law of Demeter.

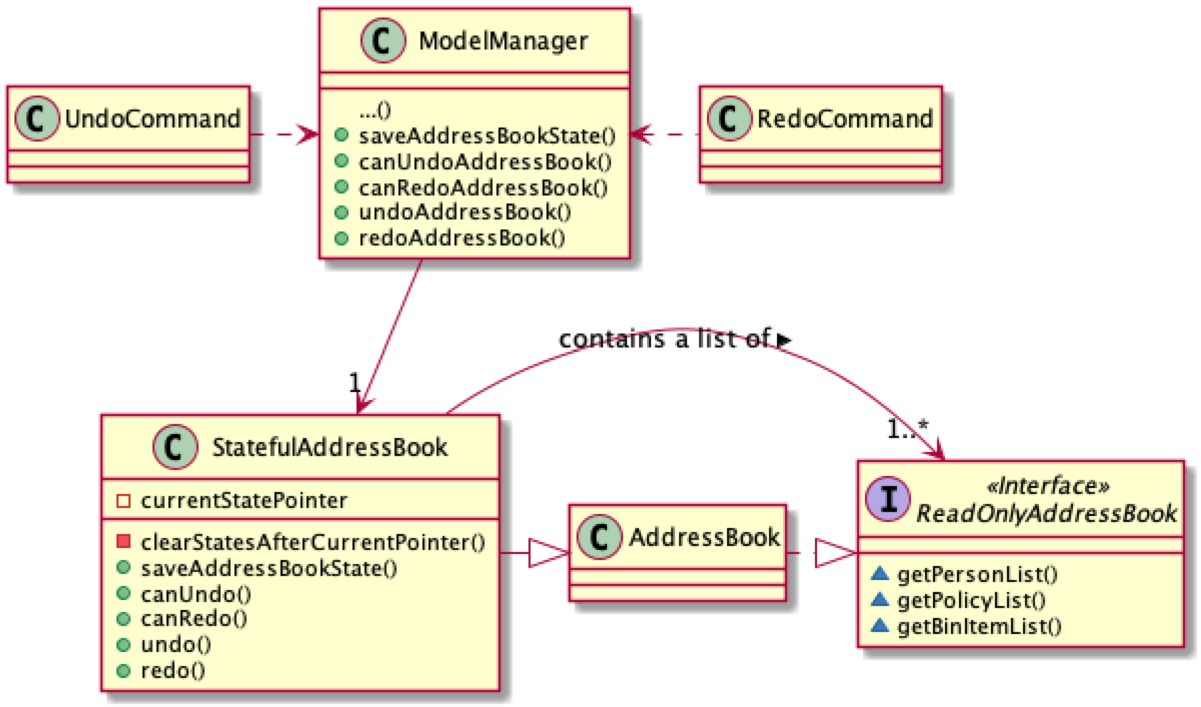
Aspect: Which commands to display in command history

- **Alternative 1:** Display only those commands which will be relevant to user if she is considering undo/redo (therefore, display only data changes)
    - Pros: Only commands which are more relevant shown to user.
    - Cons: Which commands are relevant depends on the user, moreover limits the application of command history feature to undo/redo application.
- **Alternative 2 (current choice):** Display every valid command entered by the user
    - Pros: More accurate representation of command history.
    - Cons: Clutters the command history with potentially unnecessary commands.

Undo/Redo Feature

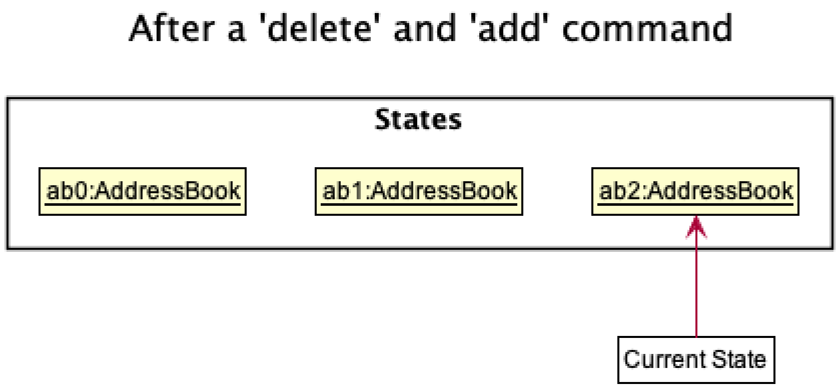
Implementation

To allow users to revert/return to a previous/future state of the address book, an undo/redo mechanism is implemented, which undoes/redoes the last data change made in the application. The feature is supported by the `StatefulAddressBook` class, an instance of which is stored as one of the memory objects inside `ModelManager`.

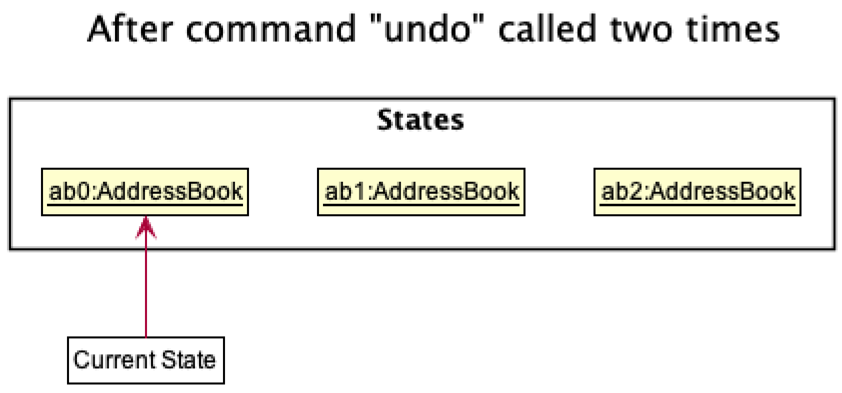


Given below is an example usage scenario and how the undo-redo mechanism works at each step. Let us assume that `StatefulAddressBook` has just been initialised.

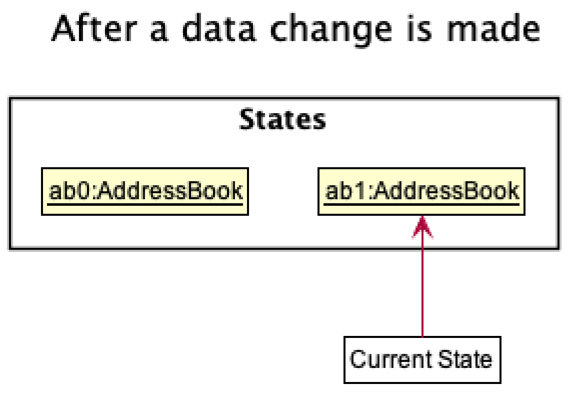
Step 1. The user makes some data changes. This adds a list of states to our `StatefulAddressBook`, and updates the `currentStatePointer`.



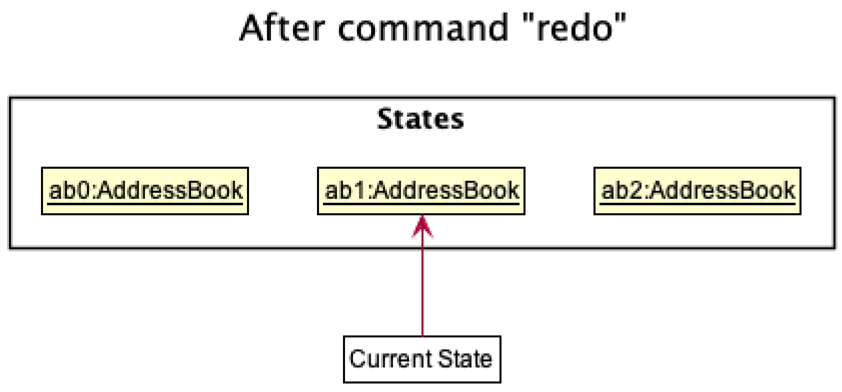
Step 2. User types `undo`. The `currentStatePointer` is decremented, and the address book is reset to the one being pointed to by `currentStatePointer`.



Step 3a. The use can perform another data change, following which states after `currentStatePointer` are erased, and a new state is added.

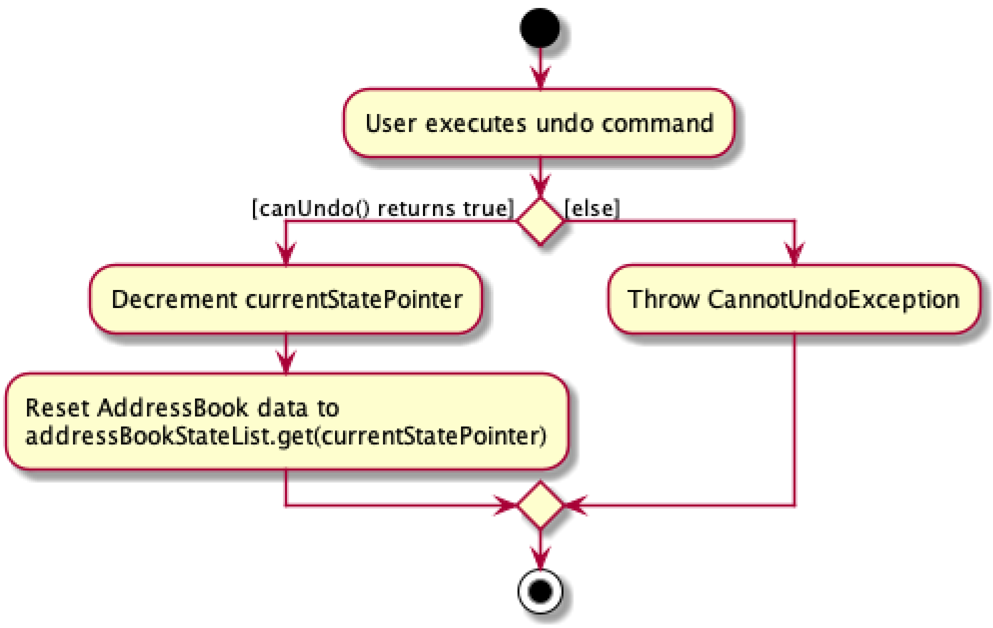


Step 3b. If a command which does not perform a data change is called, `StatefulAddressBook` stays the same as seen in `Step 2`. If a `redo()` is called, then the `currentStatePointer` is incremented.



Step 4. If the address book data is reset to another state's, then all the data inside the application is reloaded, with the resulting changes now reflected.

Following is the activity diagram. The diagram for a `redo` command will be similar.


User executes undo command
[canUndo() returns true]
[else]
Decrement currentStatePointer
Throw CannotUndoException
Reset AddressBook data to addressBookStateList.get(currentStatePointer)

Design Considerations

Aspect: How undo & redo executes

- **Alternative 1 (current choice):** Saves the entire address book.
  - Pros: Easy to implement, takes way lesser code.
  - Cons: May have performance issues in terms of memory usage.
- **Alternative 2:** Individual command knows how to undo/redo by itself.
  - Pros: Will use less memory (e.g. for delete, just save the person being deleted).
  - Cons: We must ensure that the implementation of each individual command are correct.
- **Alternative 3:** Do not store entire address book, but a PersonList, PolicyList and BinList depending on what is changed.
  - Pros: Will use less memory (e.g. for delete, just save the person list).
  - Cons: We must ensure that the implementation of each individual command are correct. Several cases to consider when we try to undo a command.

Use case: Edit person

MSS

1. User requests to list persons.
2. Insurelytics shows a list of persons.
3. User requests to edit a specific person in the list.
4. Insurelytics edits the person.
5. The person’s edited details are now visible in the address book.

   Use case ends.

Extensions

2a. The list is empty.

Use case ends.

3a. The given index is invalid.

3b. The given edited details are invalid.

3*1. Insurelytics shows an error message.

Use case resumes at step 2.

Use case: Assigning a policy to a person

MSS

1. User requests to list persons.
2. Insurelytics shows a list of persons.
3. User requests to assign a policy to a specific person.
4. The policy gets assigned to the person.

Use case ends.

Extensions

2a. The list is empty.

Use case ends.

3a. The given index is invalid.

3b. The person is not eligible for the policy.

3c. The policy is not present in the global list of policies.

3*1. Insurelytics shows an error message.

Use case resumes at step 2.