John Doe – Project Portfolio for OASIS

About the project

My team and I were tasked with enhancing a basic command line interface addressbook for our Software Engineering project. We chose to morph it into an employee records management cum communication system called **OASIS**. This enhanced application enables office managers to file and recall employee data; manage employee work schedule and leave application; and email employees directly without opening an email application.

My role was to design and write the codes for the `undo` and `redo` features. The following sections illustrate these enhancements in more detail, as well as the relevant sections I have added to the user and developer guides in relation to these enhancements.

Summary of contributions

This section shows a summary of my coding, documentation, and other helpful contributions to the team project.

Enhancement added: I added the ability to undo and redo previous commands

- What it does: The `undo` command allows the user to undo a previous command. The user may reverse this `undo` command with the `redo` command.
- Justification: In the event that users have made a mistake or changed their minds about executing a command, the `undo` command enables them to revert to a version immediately before the mistaken command was executed. If they change their minds again and decide to execute the command after all, then the `redo` command enables them to do so easily.
- Highlights: This enhancement works with existing as well as future commands. An in-depth analysis of design alternatives was necessary to… The implementation was also challenging as it required changes to existing commands.
- Credits: *{mention here if you reused any code/ideas from elsewhere or if a third-party library is heavily used in the feature so that a reader can make a more accurate judgement of how much effort went into the feature}*

Code contributed: Please click these links to see a sample of my code: [Functional code] [Test code] *{give links to collated code files}*

Other contributions:

- Project management:
  - There were a total of 5 releases, from version 1.1 to 1.5. I managed releases versions 1.3 to 1.5 (3 releases) on GitHub.
- Enhancements to existing features:
  - Updated the GUI color scheme because/so that… (Pull requests #33, #34)
  - Wrote additional tests for existing features to increase coverage from 88% to 92% (Pull requests #36, #38)
- Documentation:
  - Made cosmetic improvements to the existing User Guide to make it more reader-friendly: #14
- Community:
  - Reviewed Pull Requests (with non-trivial review comments): #12, #32, #19, #42
  - Contributed to forum discussions (examples: 1, 2, 3, 4)
  - Reported bugs and offered suggestions for other teams in the class (examples: 1, 2, 3)

    - Some parts of the history feature I added was adopted by several other project teams in the class (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}

Contributions to the User Guide

We had to update the original addressbook User Guide with instructions for the enhancements that we had added. The following is an excerpt from our ***OASIS User Guide,*** showing additions that I have made for the `undo` and `redo` features.

This section also contains an excerpt for the data file encryption feature that I have planned for the next version (v2.0) of **OASIS**.

Note the following symbols and formatting used in the User Guide (and in the next section, the Developer Guide):

This symbol indicates important information.

`undo`

A grey highlight (called a mark-up) indicates that this is a command that can be inputted into the command line and executed by the application.

VersionedAddressBook

Blue text with grey highlight indicates a component, class or object in the architecture of the application.

Undoing a previous command: `undo`

This command restores **OASIS** to the state before the previous command was executed.

Example:
Let's say that you have been entering contact information into **OASIS**.

- You now decide that you do not need one of the contacts in your list after all, so you `delete` this contact.
- But for some reason, you change your mind immediately, and decide that you actually do need the contact after all, and want to undo the `delete` command you have just entered. To do so, type `undo` into the command line. This reverses the `delete` command and restores the contact you had earlier deleted.

The `undo` feature applies to only undoable commands.
Undoable commands are commands that modify **OASIS**'s content, such as `add`, `delete`, `edit` and `clear`.

The `undo` command only reverses undoable commands.
Let's say you have executed a `select` command to select and view the information for a contact in your list. If you were to now execute the `undo` command, this command would fail because `select` is not an undoable command.

The `undo` command reverses previous commands in reverse chronological order.
Let's say you have executed the `delete` command, followed by the `clear` command. Executing `undo` now will reverse the `clear` command. Executing `undo` again will now reverse the `delete` command.

Redoing an undone command: `redo`

This command reverses the most recent `undo` command.

Example:
Let's say you have executed the `delete` command to delete a contact in your list.

- You may undo this action and restore the contact by executing the `undo` command.
- Then, if you decide that you want the contact to remain deleted after all, you may execute the `redo` command to reverse the `undo` command that you had just executed.

The `redo` command reverses only the `undo` command.
Let's say you have executed the `delete` command to delete a contact in your list. Then you execute the redo command. This command will fail because no `undo` command was entered before this.

The `redo` command reverses previous `undo` commands in reverse chronological order.
Let's say that you have executed the `delete` command, followed by the `clear` command.

- Executing `undo` now will reverse the `clear` command. Executing `undo` again will then reverse the `delete` command as well.
- Following this, executing `redo` will now reverse the last `undo` command and reapply the `delete` command. Executing `redo` again will reverse the second last `undo` command and reapply the `clear` command.

Encrypting data files [coming in v2.0]

{explain how the user can enable/disable data encryption}

Contributions to the Developer Guide

The following section shows my additions to the ***OASIS Developer Guide*** for the `undo` and `redo` features.

Undo/Redo feature

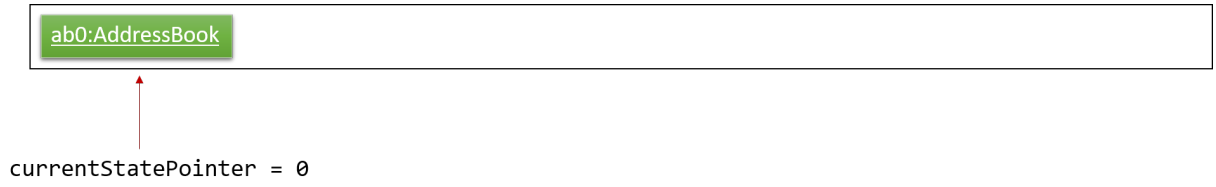
The undo/redo mechanism is facilitated by VersionedAddressBook. It extends AddressBook with an undo/redo history, stored internally as an addressBookStateList and currentStatePointer. Additionally, it implements the following operations:

- VersionedAddressBook#commit() — Saves the current address book state in its history.
- VersionedAddressBook#undo() — Restores the previous address book state from its history.
- VersionedAddressBook#redo() — Restores a previously undone address book state from its history.

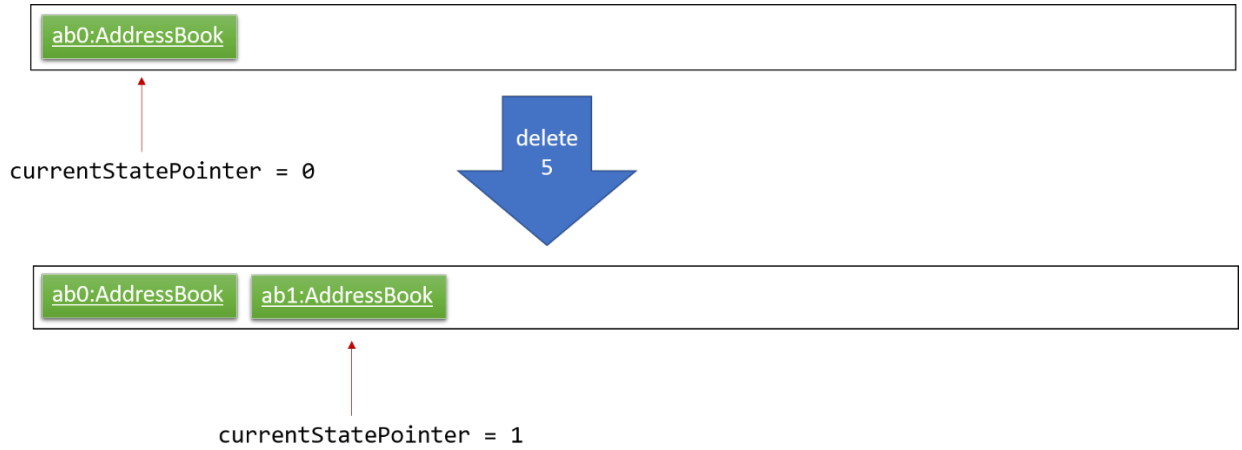
These operations are exposed in the Model interface as Model#commitAddressBook(), Model#undoAddressBook() and Model#redoAddressBook() respectively

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

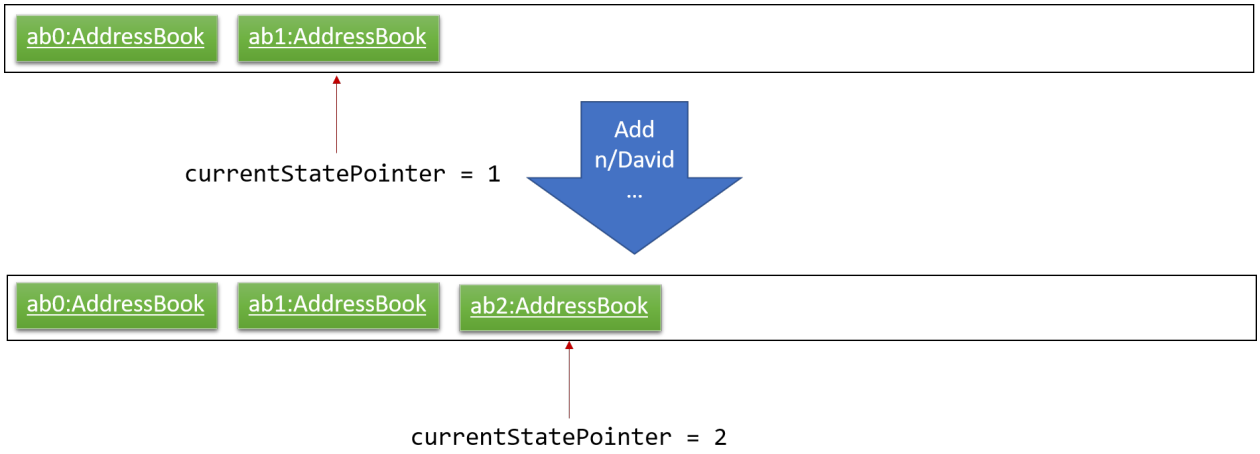
Step 1. The user launches the application for the first time. The VersionedAddressBook will be initialized with the initial address book state, and the currentStatePointer pointing to that single address book state.



Step 2. The user executes `delete 5` command to delete the 5th person in the address book. The `delete` command calls Model#commitAddressBook(), causing the modified state of the address book after the delete 5 command executes to be saved in the addressBookStateList, and the currentStatePointer is shifted to the newly inserted address book state.

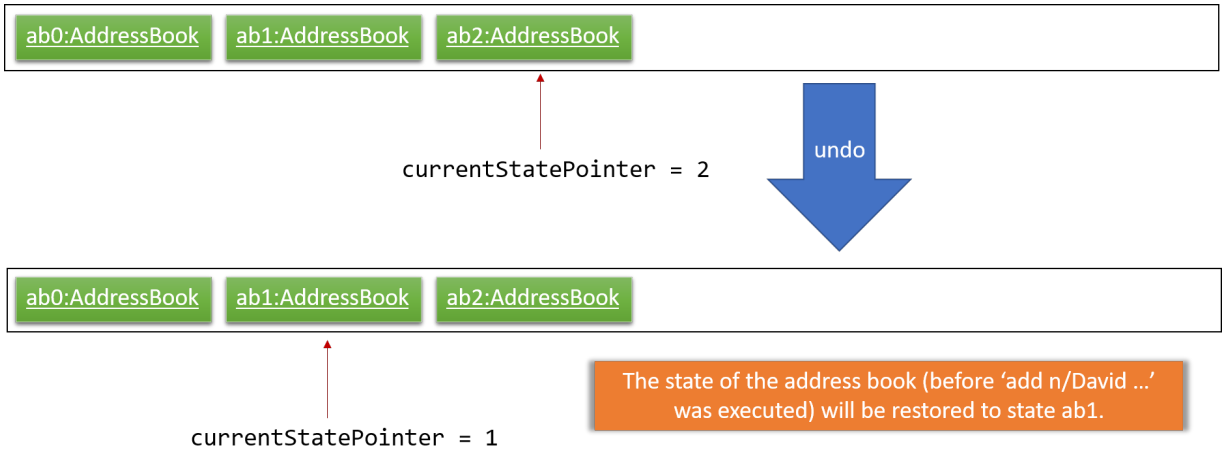


Step 3. The user executes `add n/David ...` to add a new person. The `add` command also calls Model#commitAddressBook(), causing another modified address book state to be saved into the addressBookStateList.



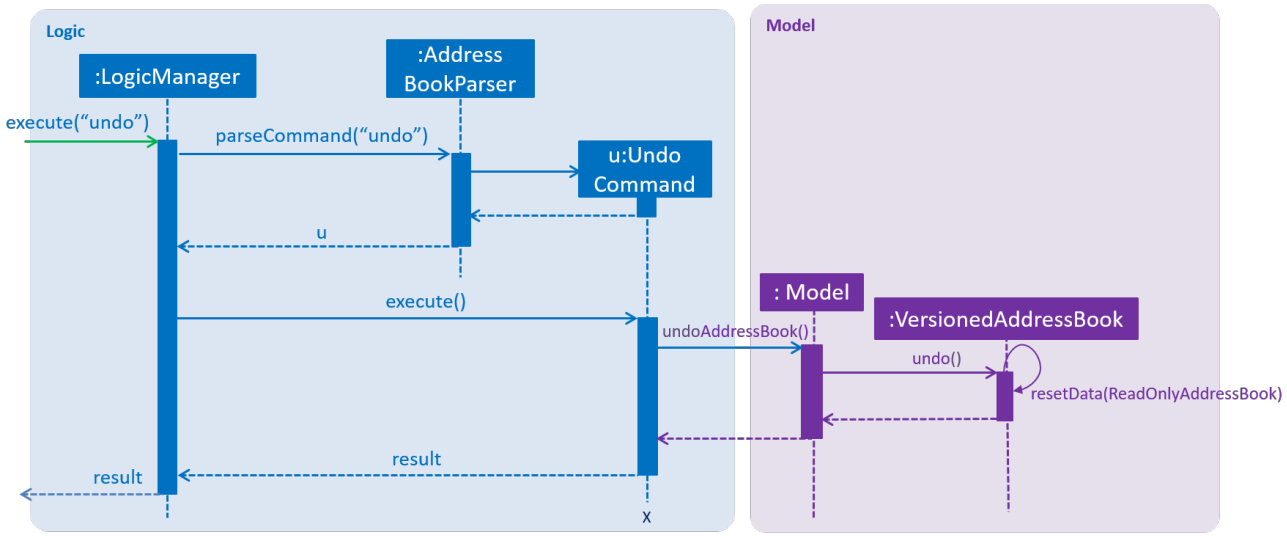
If a command fails its execution, it will not call Model#commitAddressBook(), so the address book state will not be saved into the addressBookStateList.

Step 4. The user now decides that adding the person was a mistake, and decides to undo that action by executing the `undo` command. The `undo` command will call Model#undoAddressBook(), which will shift the currentStatePointer once to the left, pointing it to the previous address book state, and restores the address book to that state.



If the currentStatePointer is at index 0, pointing to the initial address book state, then there are no previous address book states to restore. 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`.

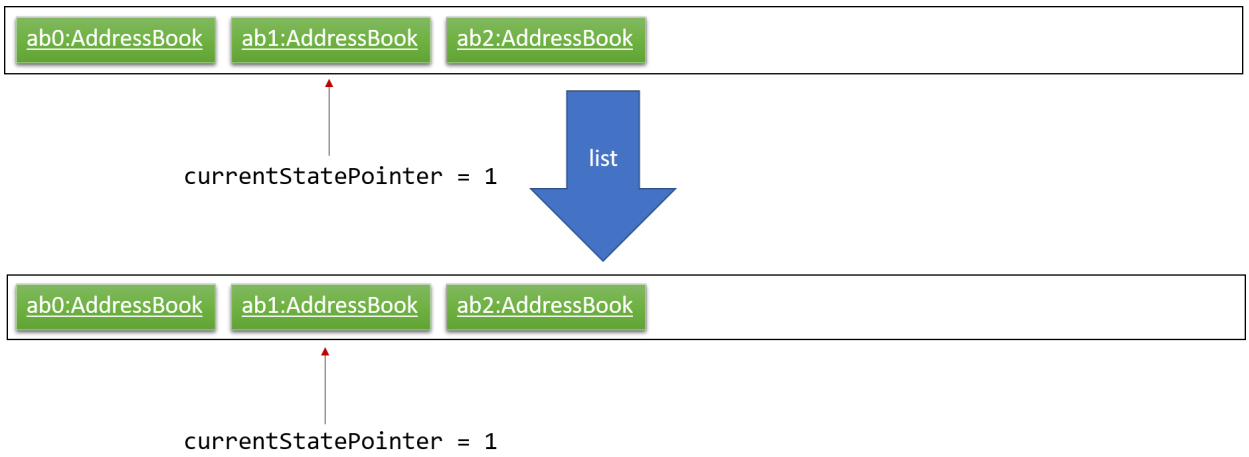
The following sequence diagram shows how the undo operation works:



The `redo` command does the opposite — it calls Model#redoAddressBook(), which shifts the currentStatePointer once to the right, pointing to the previously undone state, and restores the address book to that state.

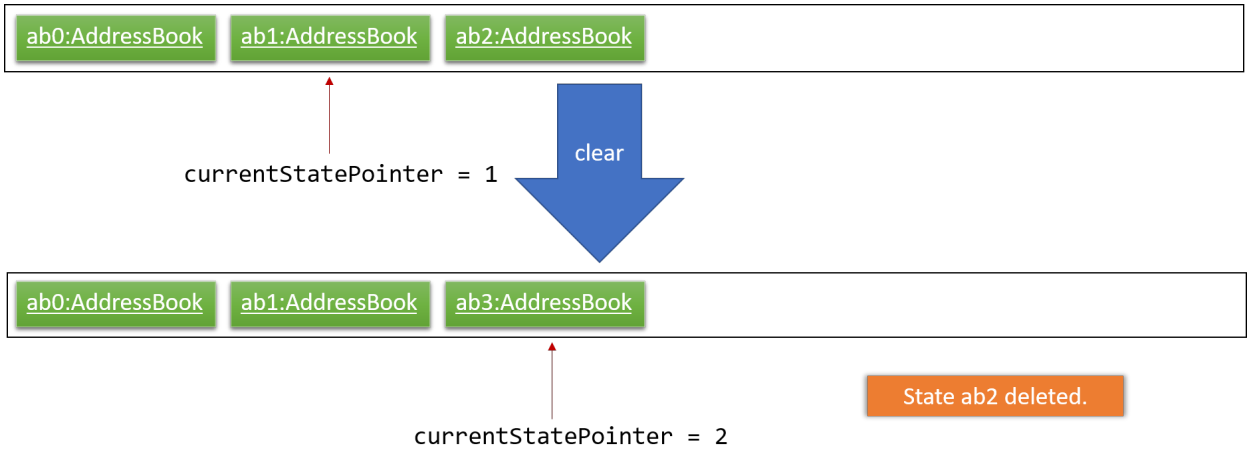
If the currentStatePointer is at index addressBookStateList.size() - 1, pointing to the latest address book state, then there are no undone address book states to restore. The `redo` command uses Model#canRedoAddressBook() to check if this is the case. If so, it will return an error to the user rather than attempting to perform the `redo`.

Step 5. The user then decides to execute the command list. Commands that do not modify the address book, such as `list`, will usually not call Model#commitAddressBook(), Model#undoAddressBook() or Model#redoAddressBook(). Thus, the addressBookStateListremains unchanged.

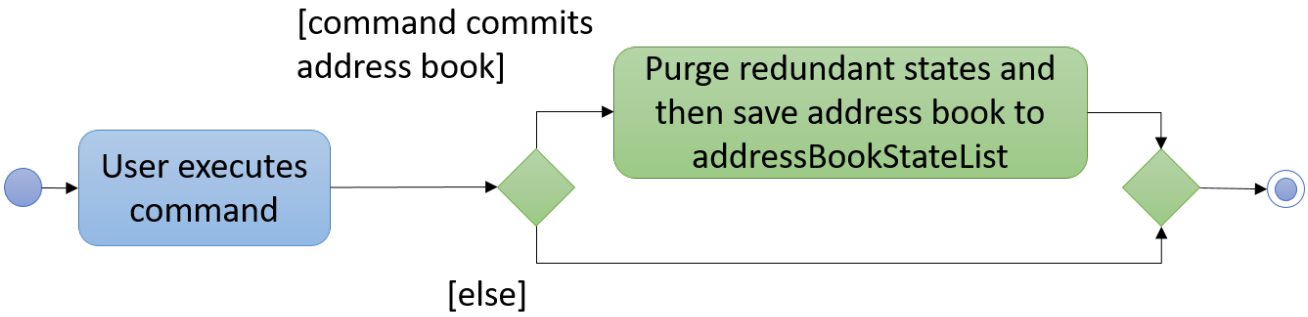


Step 6. The user executes `clear`, which calls Model#commitAddressBook(). Since the currentStatePointer is not pointing at the end of the addressBookStateList, all address book states after the currentStatePointer will be purged. We designed it this way because it no longer makes

sense to `redo` the `add n/David …` command. This is the behaviour that most modern desktop applications follow.



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



Design Considerations

When designing the `undo` and `redo` functions, I had to make decisions on how best to execute the commands and what data structure to support the commands. The following is a brief summary of my analysis and decisions.

Aspect	Alternative 1	Alternative 2
How `undo` and `redo` executes	The system saves a copy of the entire address book whenever the commands are executed. • **Pros:** Easy to implement. • **Cons:** May have performance issues in terms of memory usage. I decided to proceed with this option because…	Individual command knows how to undo/redo by itself. • **Pros:** Will use less memory (e.g. for `delete`, just saves the person whose info is being deleted). • **Cons:** We must ensure that the implementation of each individual command is correct.
Data structure to support the `undo` / `redo` commands	The system uses a list to store the history of address book states. • **Pros:** Easy for a new Computer Science undergraduate student like me to understand. New incoming	Use HistoryManager for `undo` / `redo` • **Pros:** We do not need to maintain a separate list, and instead just reuse what is already in the codebase. • **Cons:** Requires dealing with commands that have already been

	developers of our project are likely to be from this group. • **Cons:** Logic is duplicated twice. For example, when a new command is executed, we must remember to update both HistoryManager and VersionedAddressBook. I decided to proceed with this option because…	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.

[Proposed feature] Data Encryption (coming in v2.0)
{Explain here how the data encryption feature will be implemented}