

Leow Jit Yong - Project Portfolio

PROJECT: AddressBook - Level 3

Overview

The purpose of this document is to showcase my contributions to SecureIT, a project developed by my team under the Software Engineering module (CS2103T) in NUS. In this module, my team and I were tasked to enhance a basic command line interface desktop application called *Address Book 3 (AB3)*. We have decided to morph AB3 to become our current product - SecureIT.

Below is a snapshot of how SecureIT looks like!

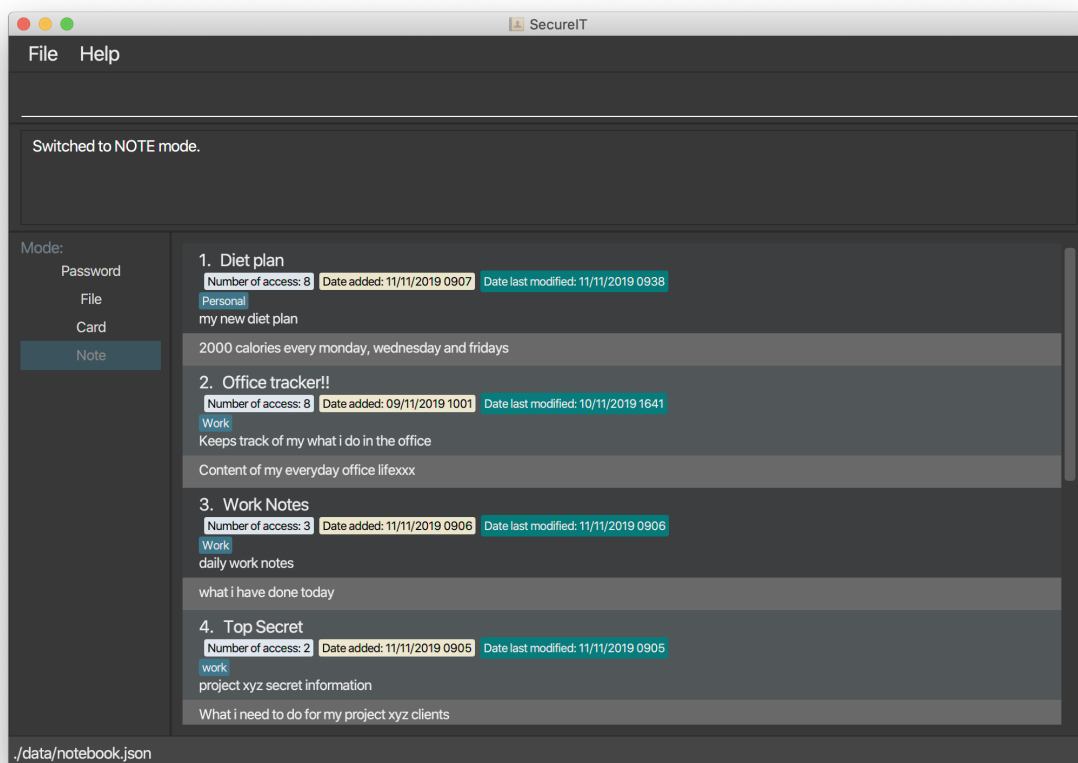




Figure 1. The graphical user interface for SecureIT in its notes component.

About this Project

SecureIT is a secure information management system that enables office workers to securely manage confidential data such as passwords, notes, files and credit card information completely offline. Users are able to interact with SecureIT via the Command-line Interface (CLI) which requires them to issue commands to the application in the form of successive lines of text. My role in this project was to design and implement all features related to notes within SecureIT.

In this document, I will be providing detailed descriptions and explanations to showcase my contributions to SecureIT's key features.

	This symbol indicates some useful tips that help you understand the features.
	This symbol indicates important information that you have to take note of.
open	Monospaced font indicates that this is a command that can be inputted into the command box and executed.
Note	Bolded monospaced font indicates a component, class or object in this application.

Summary of contributions

- **Minor Enhancement:** Added **the ability to open a note**

- What it does: gives users the ability to open a note and edit it through the User Interface(UI).
- Justification: This feature helps reading and editing notes in the app much easier, which is useful for an enhanced user experience.
- Highlights: To implement this feature, an additional UI panel was created to display the note. This UI panel allows users to freely edit the note's contents and save the changes back to the note. The implementation was challenging because it required thorough understanding of how the UI interacted with the other components for this to work.

- **Major enhancement:** added **the ability to undo and redo previous commands**

- What it does: gives users the ability to undo and redo commands one at a time. Undo command undo the last undoable action. Redo command restores any actions previously undone by the undo command.
- Justification: This feature serves to be extremely important because users tend to make mistakes and the app should provide them with an easy way to undo their mistakes. This helps to enhance the user experience and can save users precious time and energy used to amend their mistakes manually.
- Highlights: This enhancement required a good understanding of data structures and algorithm to be able to come out with a clean and intuitive solution. This enhancement also required a thorough understanding of how the components interacted with each other as it had to integrate with several commands.

- **Code contributed:** [Reposense](#)

- **Other contributions:**

- Project management:
 - Contributed to the implementation the encryption and decryption mechanism for the app. (Pull request [#36](#))
- Enhancements to existing features:
 - Updated GUI to make it more interactive. (Pull requests [#68](#))
- Documentation:
 - Updated Model and UI components UML diagrams to accommodate new newly added components. (Pull request [#57](#))

- Community:
 - Reported bugs and suggestions for other teams (examples: [#5](#), [#7](#), [#4](#))

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.

Accessing a note : **open**

Too much hassle to type out everything in the **edit** command to make edits to your note? Want to be able to read and edit your note easily? The **open** command is designed for this!

The **open** command opens your note in a panel on the right for you to read and edit its contents easily.

Format: **open** INDEX



- Opening a note updates the number of times the notes is accessed.
- **open** command is undoable.

Example: **open** 1

Your note at index 1 will be opened, allowing you to read and edit it easily.

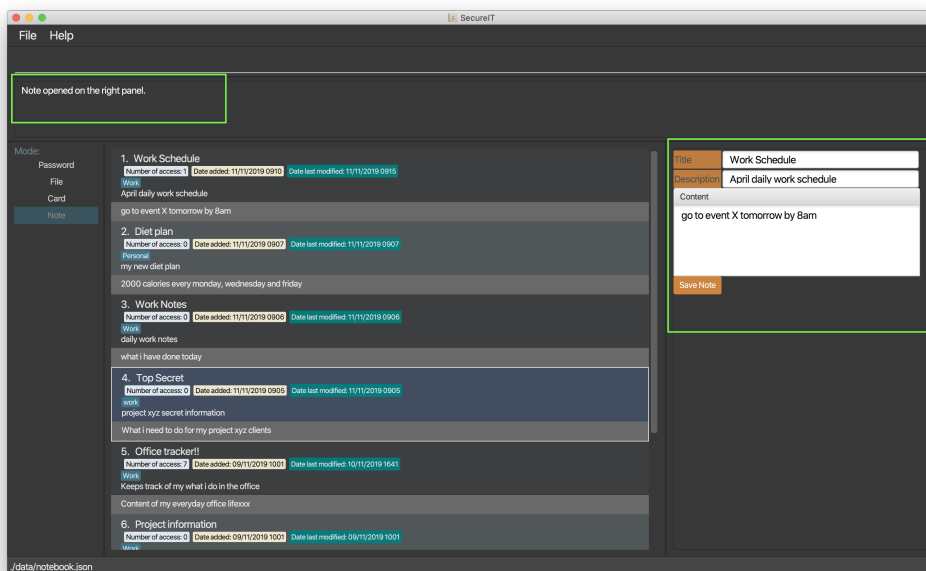


Figure 2. Note at index 1 opened on the right panel.

You can click on the Save Note button after editing to save your changes made to the note.

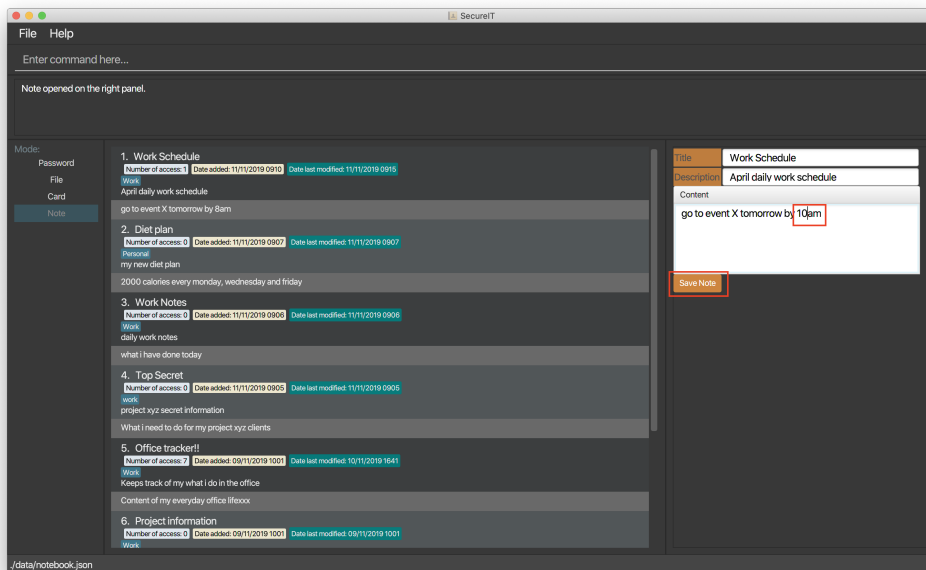


Figure 3. Edits can be made to the text and the Save Note button can be clicked.

After clicking on the Save Note button, your edits made to the note will be saved!

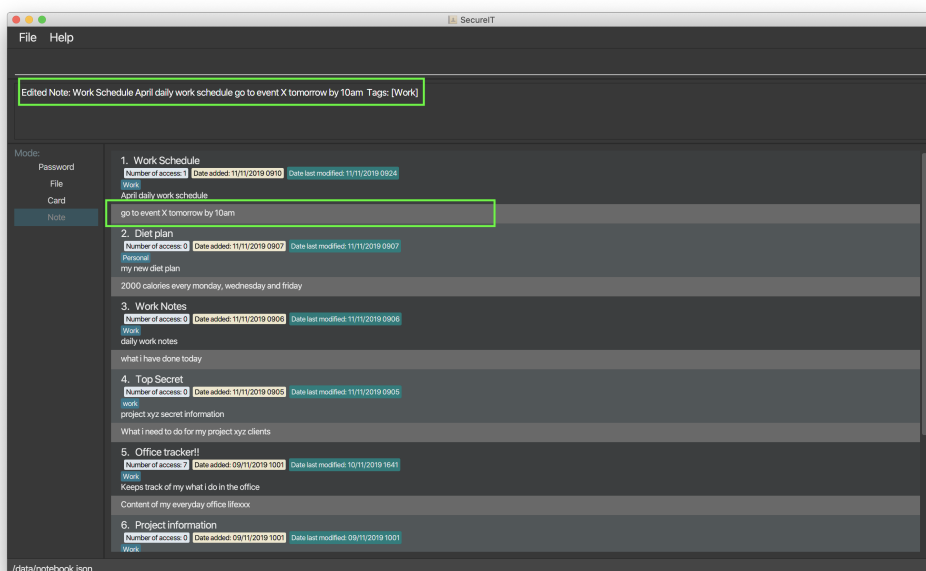


Figure 4. Note at index 1 is updated via the corresponding **edit** command after clicking on the Save Note button.

Undo notes action: **undo**

Made a mistake in the notes feature? Do not worry because you can **undo** your mistake!

Using our **undo** command will not only undo your last undoable command, it will also inform you of exactly what action is being undone.

Format: **undo**

Example: **undo**

Here is what it would look like to **undo** a **clear** command:

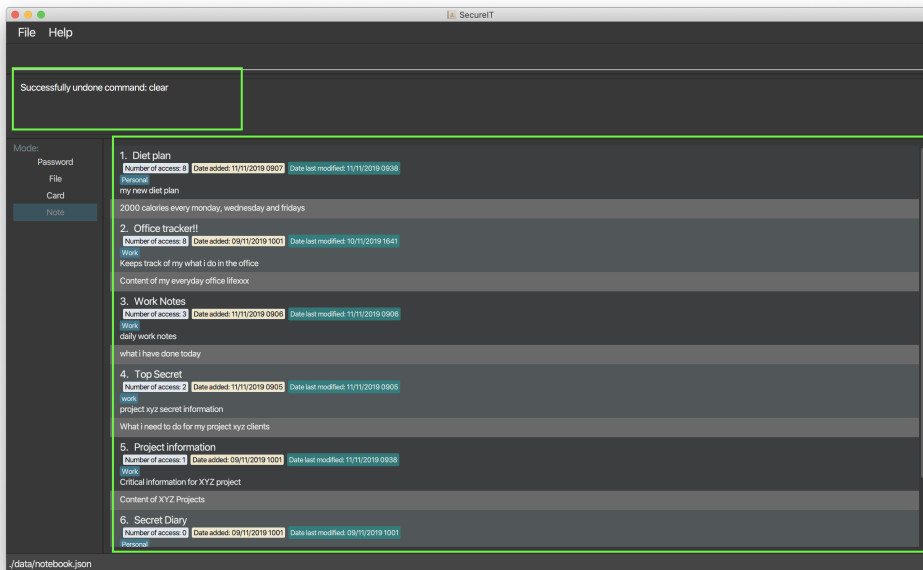


Figure 5. All deleted notes from the previous **clear** command is added back into the note book.



- You can only perform undo on undoable commands. Undoable commands are: **add**, **edit**, **delete**, **clear** and **open**.
- You can undo as many times as you wish until there are no more commands that can be undone.
- **open** command is undoable because it changes the number of times the notes is accessed.

Redo notes action: **redo**

Regret making that undo action? Once again, do not worry because you can **redo** your action!

Using the **redo** feature will not only redo your last undone action, it will also inform you of exactly what action is being redone.

Format: **redo**

Example: **redo**

Here is what it would look like to **redo** a **clear** command:

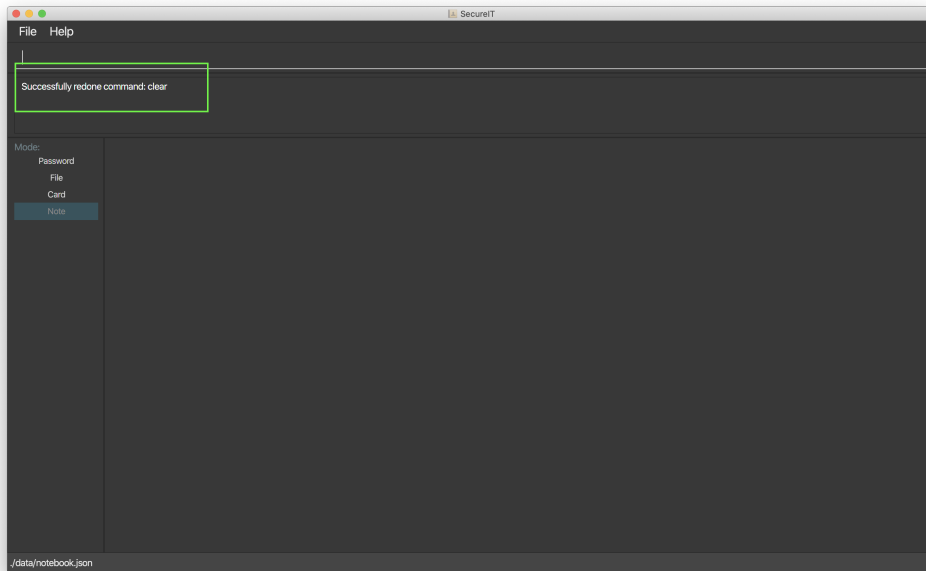


Figure 6. All notes are cleared from the note book from redoing the `clear` command.



- You can only perform redo on a command that has been undone.
- You can redo as many times as you wish until there are no more commands that can be redone.



- If you perform a new undoable command, all existing redoable commands will be cleared.

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.

Notes Feature

Open Note

The user is able to open a note in a separate panel to easily read and edit its contents.

The following sequence diagram illustrates how the note is retrieved and edited through the UI via the `open` command.

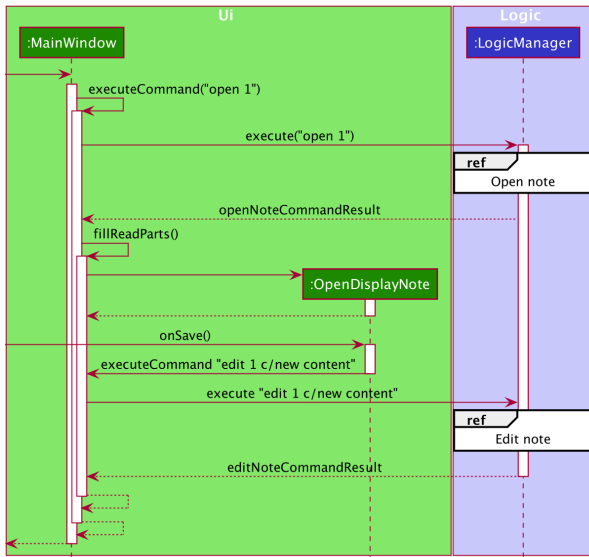


Figure 7. Diagram illustrates how note is retrievable and editable through the UI.

Below is the sub-diagram for retrieving the note through the **open** command.

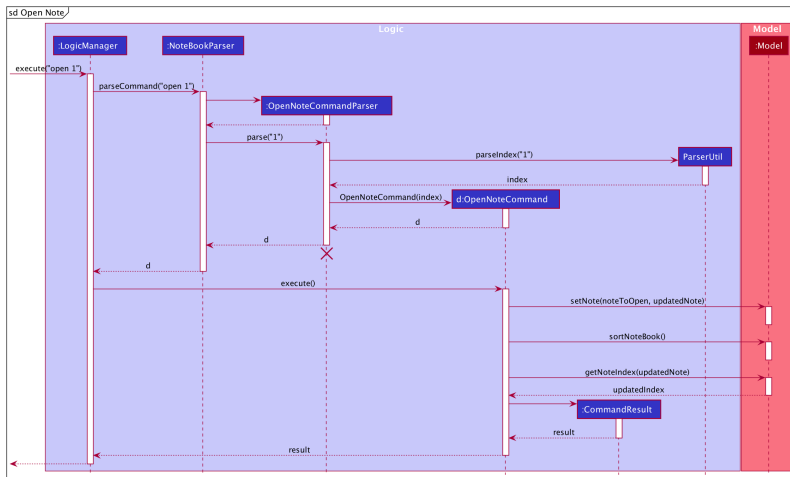


Figure 8. Sub-diagram to illustrate how a note is retrieved through the **open** command.

Given below is an example usage of the **open** command in notes, where a user intends to open a note to read or edit its contents.

1. Through the **goto note** command, the user arrives at the notes component of the app. The user can then executes the command **open 1**.
2. This invokes the **Model#OpenNoteCommand()** command which proceeds to check if the index **1** given is a valid index using the **ParserUtil#parseIndex()** method.
3. If the index given by the user is valid, the corresponding note will be retrieved. This note is shown to the user in a panel on the right within the app.
4. The user can then proceeds to edit the Title, Description and Content field of the note through this panel and saves his edits using the save button. This executes the **Logic#EditNoteCommand()** which edits the note in the note book.
5. The note book is now updated with the note edited by the user.

Undo/Redo

Implementation

The undo/redo mechanism is facilitated by the **VersionedNoteBook**, which extends **NoteBook** with an undo/redo history. This history is stored in two stacks - the **UndoStack** and the **RedoStack**. The **UndoStack** stores the states of note books before the current state while the **RedoStack** stores the states of note books after the current state. The key operations that support these mechanisms are:

VersionedNoteBook#commit() — Saves the current note book state and its corresponding command in the **UndoStack**.

VersionedNoteBook#undo() — Restores the previous note book state from the **UndoStack** and returns its corresponding command to inform the user what command is undone. It also stores the current state of the note book into the **RedoStack**.

VersionedNoteBook#redo() — Restores a previously undone note book state from the **RedoStack** and returns its corresponding command to inform the user what command is redone. It also stores the current state of the note book into the **UndoStack**.

Below is a class diagram to illustrate how the classes mentioned above interact with one another.

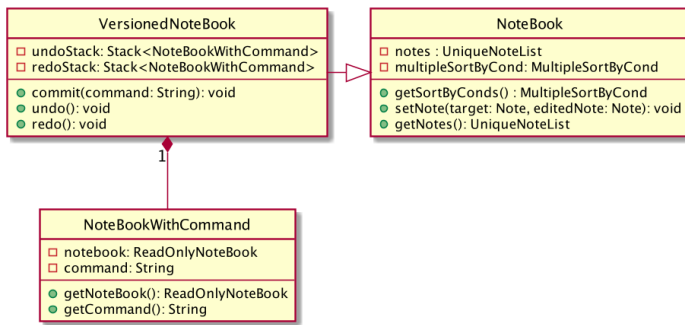


Figure 9. Class diagram of **VersionedNoteBook**, **NoteBook** and **NoteBookWithCommand**.

Below is a comprehensive activity diagram to illustrate how the undo & redo mechanism works.

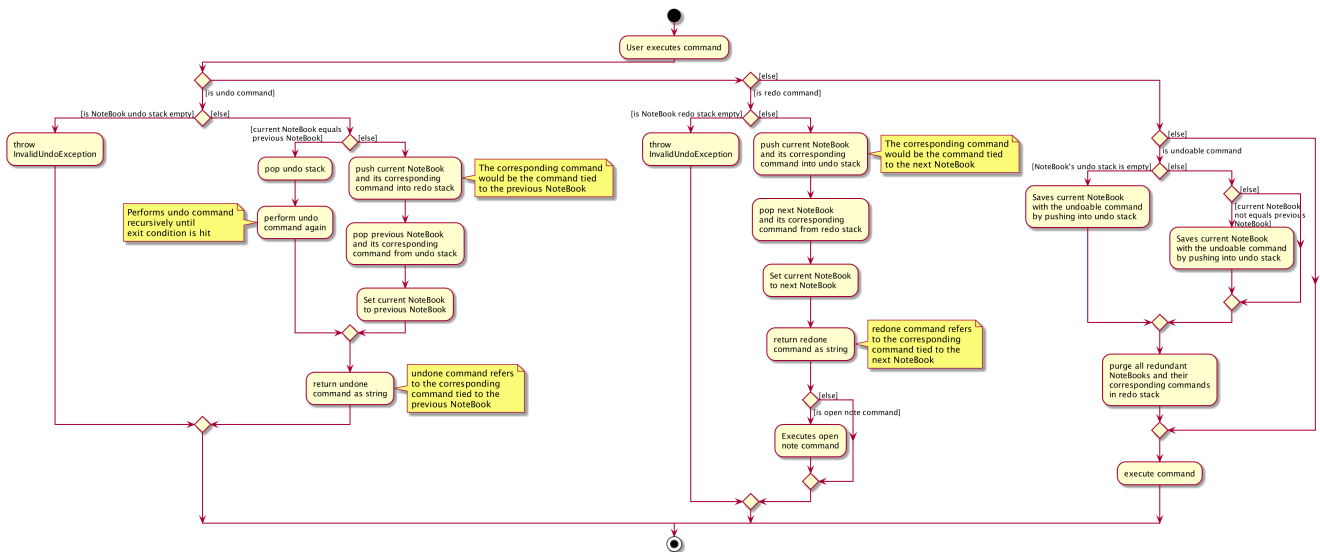


Figure 10. Activity diagram to illustrate how undo & redo executes.



Warning

Not all commands are undoable. Undo-able commands are commands that modifies the note book. They include: `add`, `delete`, `clear`, `edit` and `open`.

Below is an example to illustrate how the undo command is executed:

1. The user launches the application for the first time. The `VersionedNoteBook` will be initialized with the initial note book state, with empty `UndoStack` and `RedoStack`.
2. The user executes `delete 5` command to delete the 5th note in the note book. The delete command invokes the `Model#commitNoteBook()`, causing the state of the note book before the `delete 5` command is executed to be saved in the `UndoStack`. The current state of the note book is then updated to the state after the `delete 5` command executes.
3. The user now decides that deleting the note was a mistake, and wants to undo that action by executing the `undo` command.
4. This will call the `Model#undo()`, which will first check if the `UndoStack` is empty.
 - a. The `UndoStack` will be empty if no undoable command were called prior to calling the `undo` command.
 - b. The undo mechanism will proceed only if the `UndoStack` is not empty, else an error will be returned to the user rather than attempting to perform the undo.
5. The current state of the note book is then checked against the previous state of the note book.
 - a. In the event that they are the same, the previous note book state is popped from the `UndoStack` and the `undo` command is called recursively (starting from step 2). This recursive call is performed until either the current and previous note book state are different or until the `UndoStack` is empty.
 - b. In the event that they are different, proceed on to step 6.
6. The current state of the note book is stored in the `RedoStack`.
7. The note book is then reverted to the previous note book state. This same previous note book state is popped from the `UndoStack`.
8. The undo command is complete.

Note:

- Step 3 is to prevent users from being able to perform `undo` actions when there is no change to be undone.

For the redo command, a key point to note is that it is not logical to redo a command from the `RedoStack` after an undoable command is executed. Therefore, anytime an undoable command is executed, the `RedoStack` is purged. This behavior follows that of most modern desktop applications as well.

As the rest of the mechanisms of the **redo** command is similar to that of the **undo** command but opposite, its details are omitted.

Design Considerations

Aspect: How undo & redo executes

- Alternative 1 (current choice): Saves the entire note book.
 - Pros: Less complex and easier to implement.
 - Cons: May have performance issues in terms of memory usage.
- Alternative 2: Individual command knows how to undo/redo by itself.
 - Pros: Makes use of lesser memory.
 - Cons: More complex to implement and more prone to bugs as each individual command must be correct to work correctly.

The reason why we chose alternative 1: By adopting a simpler implementation, we are able improve testability of the command. This helps in introducing less bugs into system. Furthermore, this implementation better supports future extensions as undoable commands can be added much easier.

Aspect: Data structure to support the undo/redo commands

- Alternative 1 (current choice): Use two stacks to store the history of note book states - one for undo, and the other for redo.
 - Pros: Very easy to implement.
 - Cons: Using two data structures may incur additional overhead in terms of memory.
- Alternative 2: Use a single linked list to store the history of note book states which supports both undo and redo.
 - Pros: May incur less overhead in terms of memory usage as only one data structure is used.
 - Cons: More complex to implement.

The reason why we chose alternative 1: It is easier to visualize the implementation of this method and is less complex to implement. As the code is much cleaner, it is more readable which also helps in improving testability and future extensions as well.