John Doe - Project Portfolio

PROJECT: AddressBook - Level 3

Overview

AddressBook - Level 3 is a desktop address book application used for teaching Software Engineering principles. The user interacts with it using a CLI, and it has a GUI created with JavaFX. It is written in Java, and has about 10 kLoC.

Summary of contributions

- **Major enhancement**: added **the ability to undo/redo previous commands**
  - What it does: allows the user to undo all previous commands one at a time. Preceding undo commands can be reversed by using the redo command.
  - Justification: This feature improves the product significantly because a user can make mistakes in commands and the app should provide a convenient way to rectify them.
  - Highlights: This enhancement affects existing commands and commands to be added in future. It required an in-depth analysis of design alternatives. The implementation too was 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}*
- **Minor enhancement**: added a history command that allows the user to navigate to previous commands using up/down keys.
- **Code contributed**: [Functional code] [Test code] *{give links to collated code files}*
- **Other contributions**:
  - Project management:
    - Managed releases `v1.3` - `v1.5rc` (3 releases) on GitHub
  - Enhancements to existing features:
    - Updated the GUI color scheme (Pull requests #33, #34)
    - Wrote additional tests for existing features to increase coverage from 88% to 92% (Pull requests #36, #38)
  - Documentation:
    - Did cosmetic tweaks to existing contents of the User Guide: #14
  - Community:
    - PRs reviewed (with non-trivial review comments): #12, #32, #19, #42

        - Contributed to forum discussions (examples: 1, 2, 3, 4)
        - Reported bugs and suggestions for other teams in the class (examples: 1, 2, 3)
        - Some parts of the history feature I added was adopted by several other class mates (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

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

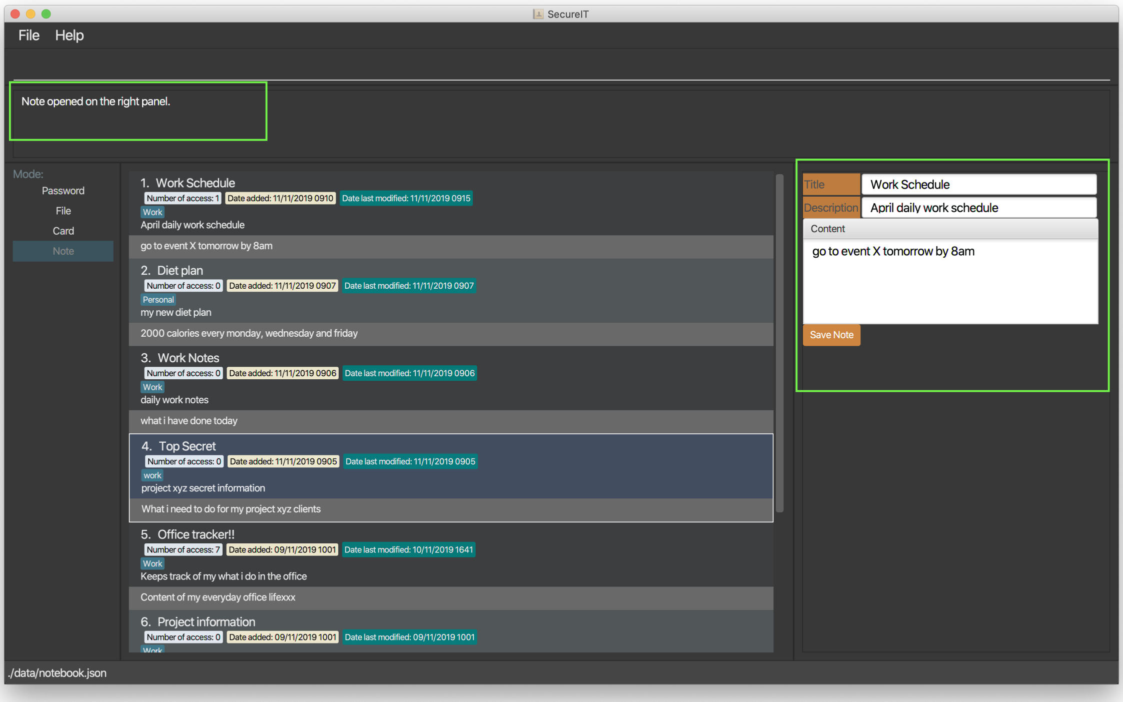
Accessing a note : `open`

Too much hassle to type out everything in the `edit` command to make a small change to your note? Want to read your note and edit your note easily? The `open` command is designed for you!

Format: `open INDEX`

Example: `open 1`

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



Note opened on the right panel.

1. Work Schedule
Number of access: 1 Date added: 11/11/2019 0910 Date last modified: 11/11/2019 0915
Work
April daily work schedule
go to event X tomorrow by 8am

2. Diet plan
Number of access: 0 Date added: 11/11/2019 0907 Date last modified: 11/11/2019 0907
Personal
my new diet plan
2000 calories every monday, wednesday and friday

3. Work Notes
Number of access: 0 Date added: 11/11/2019 0906 Date last modified: 11/11/2019 0906
Work
daily work notes
what i have done today

4. Top Secret
Number of access: 0 Date added: 11/11/2019 0905 Date last modified: 11/11/2019 0905
work
project xyz secret information
What i need to do for my project xyz clients

5. Office tracker!!
Number of access: 7 Date added: 09/11/2019 1001 Date last modified: 10/11/2019 1641
Work
Keeps track of my what i do in the office
Content of my everyday office lifexxx

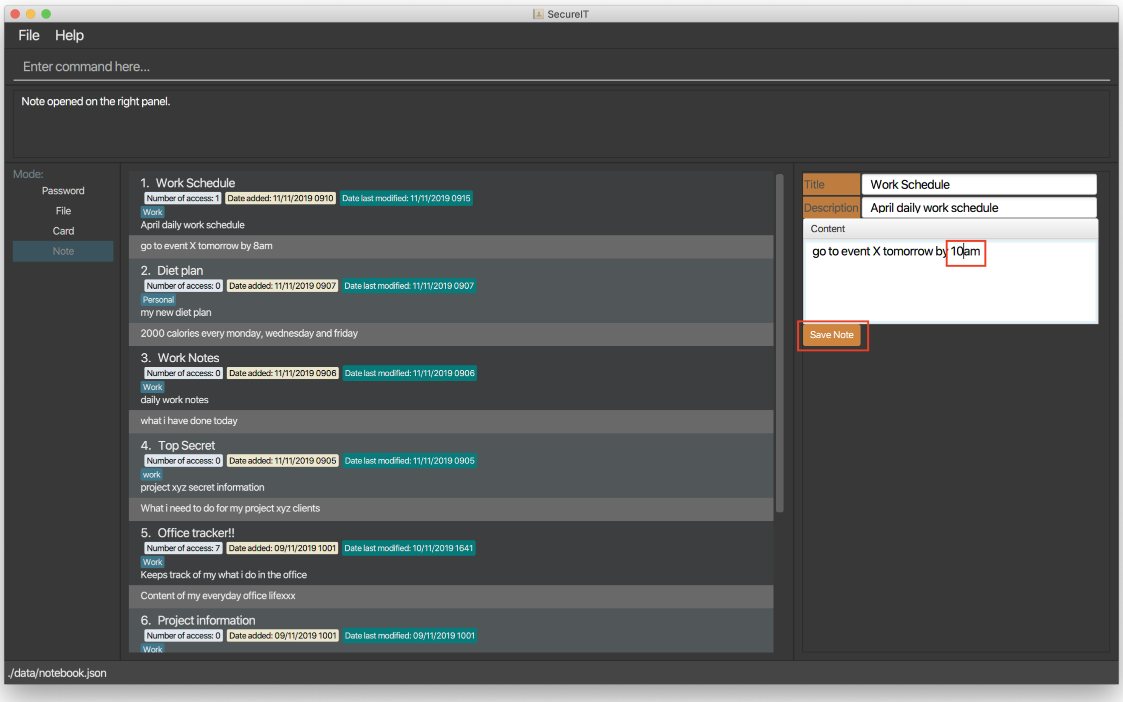
6. Project information
Number of access: 0 Date added: 09/11/2019 1001 Date last modified: 09/11/2019 1001
Work

Title: Work Schedule
Description: April daily work schedule
Content: go to event X tomorrow by 8am

./data/notebook.json

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

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



Note opened on the right panel.

1. Work Schedule
Number of access: 1 | Date added: 11/11/2019 0910 | Date last modified: 11/11/2019 0915
Work
April daily work schedule
go to event X tomorrow by 8am

2. Diet plan
Number of access: 0 | Date added: 11/11/2019 0907 | Date last modified: 11/11/2019 0907
Personal
my new diet plan
2000 calories every monday, wednesday and friday

3. Work Notes
Number of access: 0 | Date added: 11/11/2019 0906 | Date last modified: 11/11/2019 0906
Work
daily work notes
what i have done today

4. Top Secret
Number of access: 0 | Date added: 11/11/2019 0905 | Date last modified: 11/11/2019 0905
work
project xyz secret information
What i need to do for my project xyz clients

5. Office tracker!!
Number of access: 7 | Date added: 09/11/2019 1001 | Date last modified: 10/11/2019 1641
Work
Keeps track of my what i do in the office
Content of my everyday office lifexxx

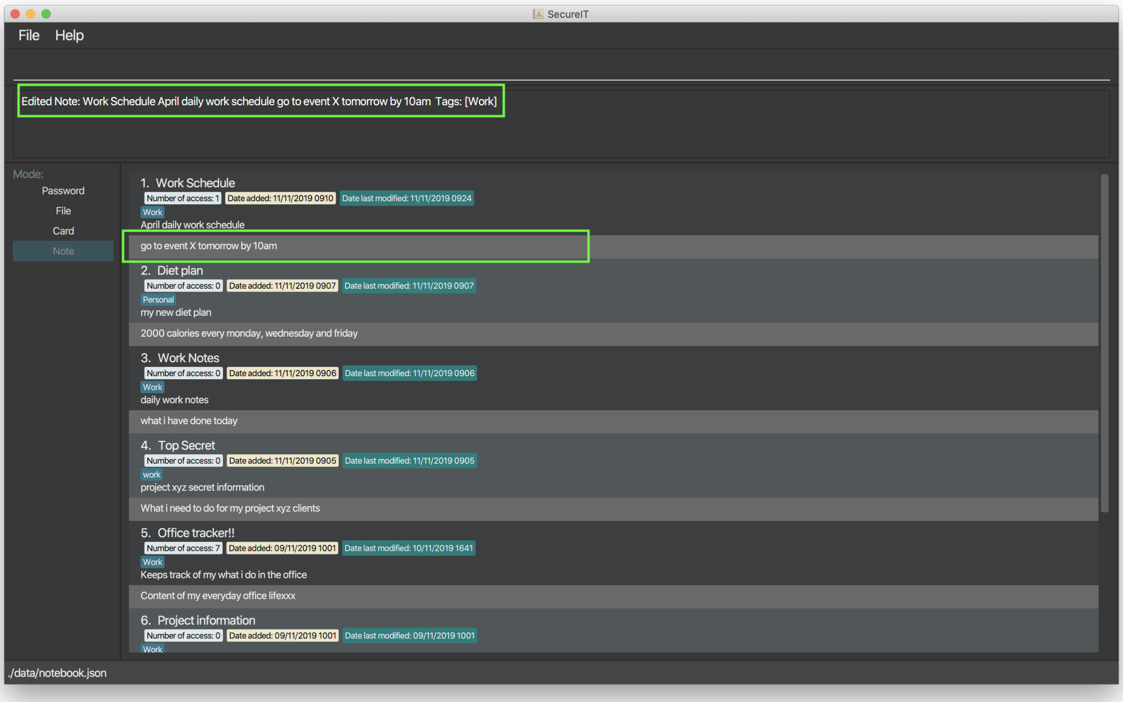
6. Project information
Number of access: 0 | Date added: 09/11/2019 1001 | Date last modified: 09/11/2019 1001
Work

Title: Work Schedule
Description: April daily work schedule
Content: go to event X tomorrow by 10am

./data/notebook.json

Figure 2. Edits can be made to text and the Save button can be clicked.

Your edits made to the opened note is saved!



Edited Note: Work Schedule April daily work schedule go to event X tomorrow by 10am Tags: [Work]

1. Work Schedule
Number of access: 1 | Date added: 11/11/2019 0910 | Date last modified: 11/11/2019 0924
Work
April daily work schedule
go to event X tomorrow by 10am

2. Diet plan
Number of access: 0 | Date added: 11/11/2019 0907 | Date last modified: 11/11/2019 0907
Personal
my new diet plan
2000 calories every monday, wednesday and friday

3. Work Notes
Number of access: 0 | Date added: 11/11/2019 0906 | Date last modified: 11/11/2019 0906
Work
daily work notes
what i have done today

4. Top Secret
Number of access: 0 | Date added: 11/11/2019 0905 | Date last modified: 11/11/2019 0905
work
project xyz secret information
What i need to do for my project xyz clients

5. Office tracker!!
Number of access: 7 | Date added: 09/11/2019 1001 | Date last modified: 10/11/2019 1641
Work
Keeps track of my what i do in the office
Content of my everyday office lifexxx

6. Project information
Number of access: 0 | Date added: 09/11/2019 1001 | Date last modified: 09/11/2019 1001
Work

./data/notebook.json

Figure 3. Note at index 1 is updated via the corresponding `Edit` command after clicking on the Save button.

Undo notes action: `undo`

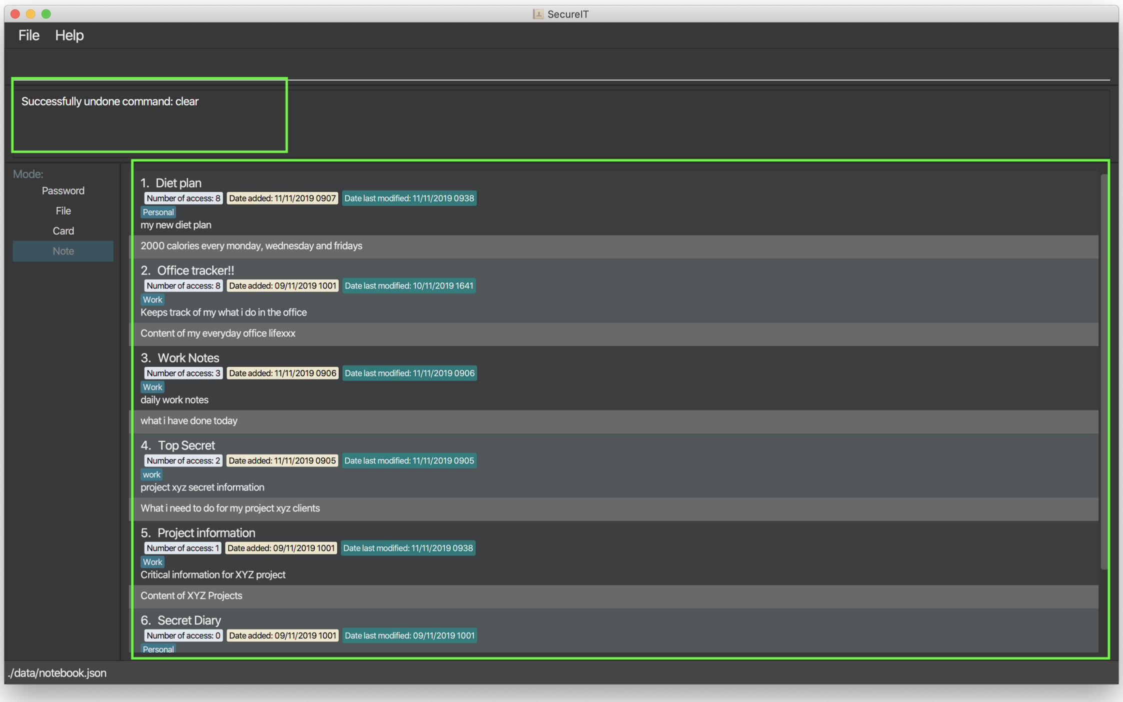
Made a mistake in the notes feature? Worry not for you can `undo` your mistake!

Using our `undo` feature 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:



Successfully undone command: clear

1. Diet plan
Number of access: 8 Date added: 11/11/2019 0907 Date last modified: 11/11/2019 0938
Personal
my new diet plan
2000 calories every monday, wednesday and fridays

2. Office tracker!!
Number of access: 8 Date added: 09/11/2019 1001 Date last modified: 10/11/2019 1641
Work
Keeps track of my what i do in the office
Content of my everyday office lifexxx

3. Work Notes
Number of access: 3 Date added: 11/11/2019 0906 Date last modified: 11/11/2019 0906
Work
daily work notes
what i have done today

4. Top Secret
Number of access: 2 Date added: 11/11/2019 0905 Date last modified: 11/11/2019 0905
work
project xyz secret information
What i need to do for my project xyz clients

5. Project information
Number of access: 1 Date added: 09/11/2019 1001 Date last modified: 11/11/2019 0938
Work
Critical information for XYZ project
Content of XYZ Projects

6. Secret Diary
Number of access: 0 Date added: 09/11/2019 1001 Date last modified: 09/11/2019 1001
Personal

./data/notebook.json

Figure 4. All deleted notes from previous `clear` command is added back into the note book.

💡

Tip

- You can only perform undo on undo-able commands. Undoable commands are: `add`, `edit`, `delete`, `clear` and `sort`
- You can undo as many commands as you wish until there are no more commands that can be undone.

Redo notes action: `redo`

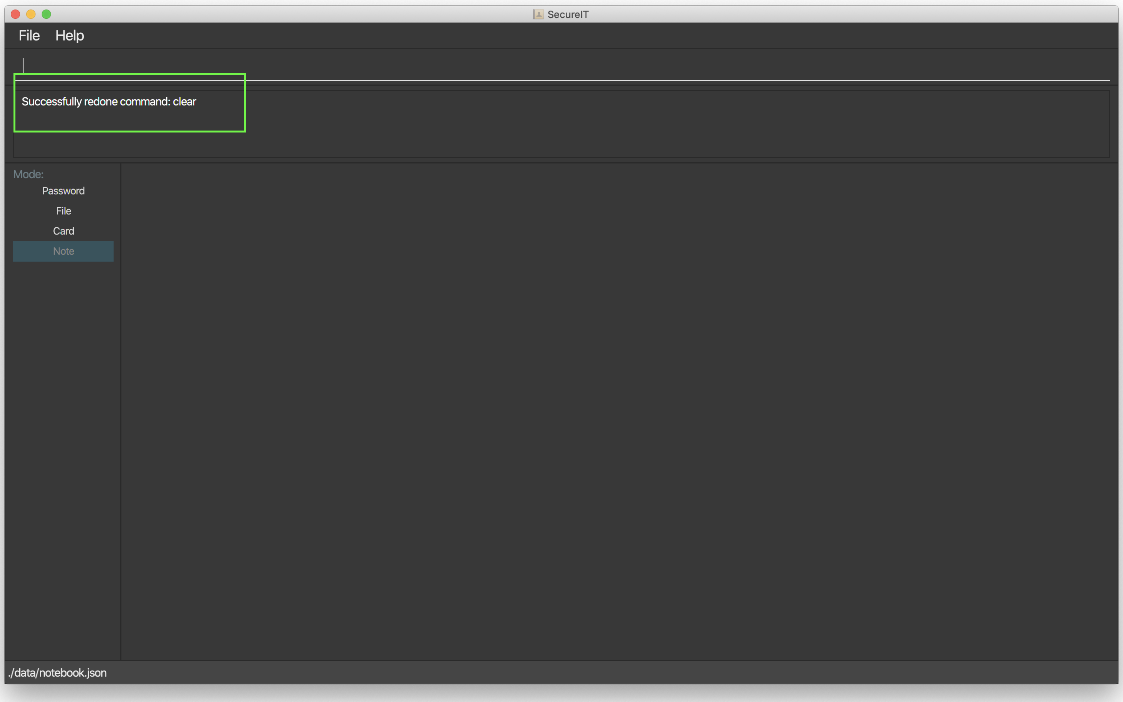
Regret making that undo action? Once again, worry not for you can `redo` your action!

Using our `redo` feature 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:



Successfully redone command: clear

./data/notebook.json

Figure 5. All notes are deleted from the note book from redoing the `clear` command.

Tip

- You can redo as many commands as you wish until there are no more commands that can be redone.
- If you perform a new undo-able command, all existing redo-able 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 the note book 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.

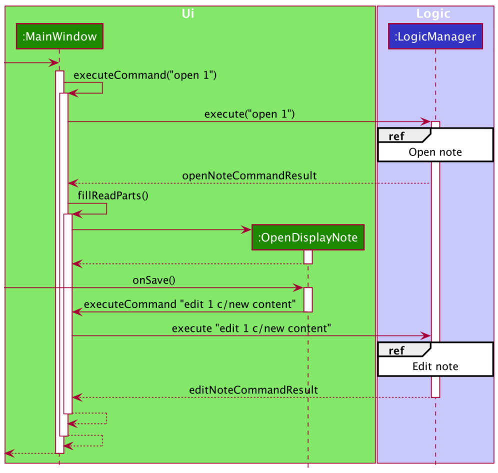


Figure 6. Diagram illustrates how note is retrievable and editable through the UI.

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

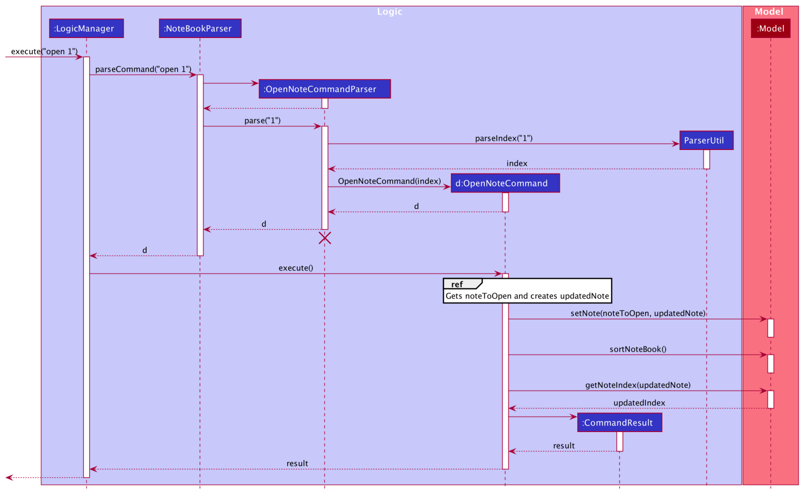


Figure 7. Sub-diagram to illustrate how a note is retrieved through the `open` command.

- Note that the sequence diagram for the `edit` command is similar to the `open` command shown above, hence is omitted.

Given below is an example usage of scenario 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 then executes the command `open 1`.
2. The app recognises that it is an `open` command and 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 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 a undo/redo history. This history is stored in two stacks - the undo stack and the redo stack. The undo stack stores the states of NoteBooks before the current state and the redo stack which stores the states of NoteBooks after the current state. The key operations that support this mechanisms are:

`VersionedNoteBook#commit()` — Saves the current NoteBook state and its corresponding command in the undo stack.

`VersionedNoteBook#undo()` — Restores the previous note book state from the undo stack 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 redo stack.

`VersionedNoteBook#redo()` — Restores a previously undone note book state from the redo stack 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 undo stack.

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

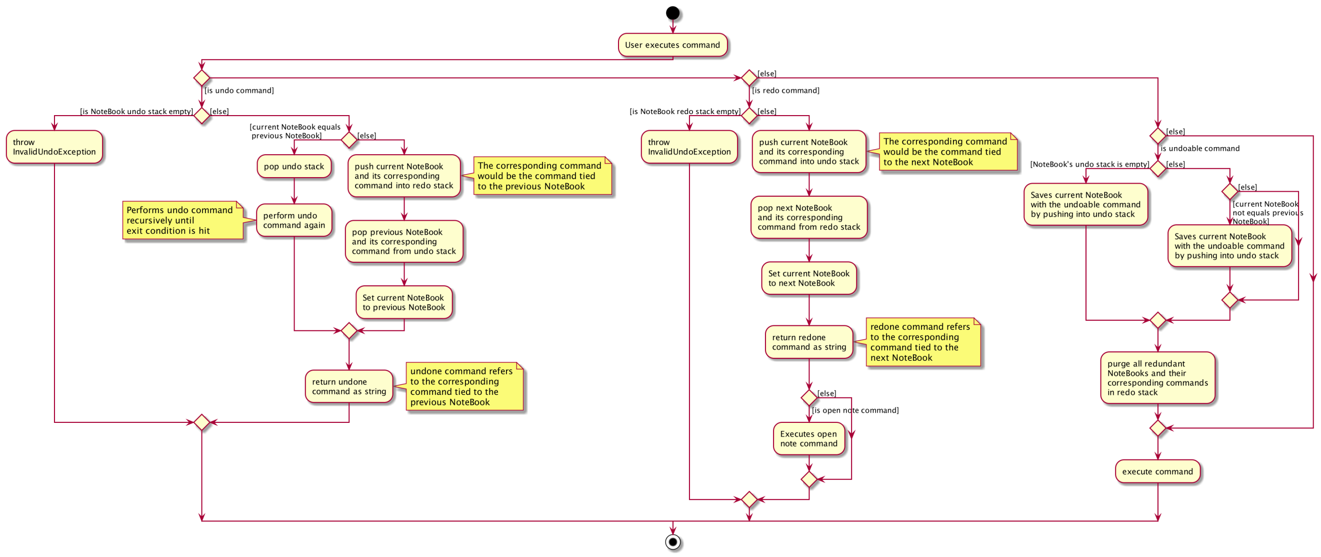

User executes command
[is undo command]
[else]
[is NoteBook undo stack empty]
[else]
throw InvalidUndoException
[current NoteBook equals previous NoteBook]
[else]
pop undo stack
push current NoteBook and its corresponding command into redo stack
The corresponding command would be the command tied to the previous NoteBook
Performs undo command recursively until exit condition is hit
perform undo command again
pop previous NoteBook and its corresponding command from undo stack
Set current NoteBook to previous NoteBook
return undone command as string
undone command refers to the corresponding command tied to the previous NoteBook
[is redo command]
[is NoteBook redo stack empty]
[else]
throw InvalidUndoException
push current NoteBook and its corresponding command into undo stack
The corresponding command would be the command tied to the next NoteBook
pop next NoteBook and its corresponding command from redo stack
Set current NoteBook to next NoteBook
return redone command as string
redone command refers to the corresponding command tied to the next NoteBook
[else]
[is open note command]
Executes open note command
[else]
is undoable command
[NoteBook's undo stack is empty]
[else]
Saves current NoteBook with the undoable command by pushing into undo stack
[else]
[current NoteBook not equals previous NoteBook]
Saves current NoteBook with the undoable command by pushing into undo stack
purge all redundant NoteBooks and their corresponding commands in redo stack
execute command

Figure 8. Activity diagram to illustrate how the undo & redo executes.

Not all commands are undo-able. Undo-able commands are those that modifies the note book which includes: `add`, `delete`, `clear`, `edit` and `open`.

- The `open` command is considered undo-able because it updates the number of access to the note that is opened.

Below is an example run through of how the undo command is executed:

1. User executes `undo` command which calls the `VersionedNoteBook#undo()`.
2. This method call first checks if the undo stack is empty.

    a. The undo stack will be empty if no undo-able command were called prior to calling the `undo` command.
    b. The undo mechanism will only proceed only if the undo stack is not empty, else an `InvalidUndoException` will be thrown.
3. 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 undo stack 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 undo stack is empty.
    b. In the event that they are different, proceed on to step 4.
4. The current state of the note book is first stored in the redo stack.
5. The note book is then reverted to the previous note book state. This same previous note book state is removed from the undo stack.
6. Undo command is complete.

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

Design Considerations

Aspect: How undo & redo executes

- Alternative 1 (current choice): Saves the entire note book.
    - Pros: Less complex and easy 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 aim to introduce 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, one 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 less complex to implement and the code is much cleaner for better readability which also helps in future extensions.

Sort Note feature

Sorting of notes is handled by the MultipleSortByCond class, which allows the notes in the NoteBook to be sorted in three different ways - by date modified, date added or number of access.

Below is a class diagram to illustrate how the `NoteBook` class and MultipleSortByCond class interact with one another.

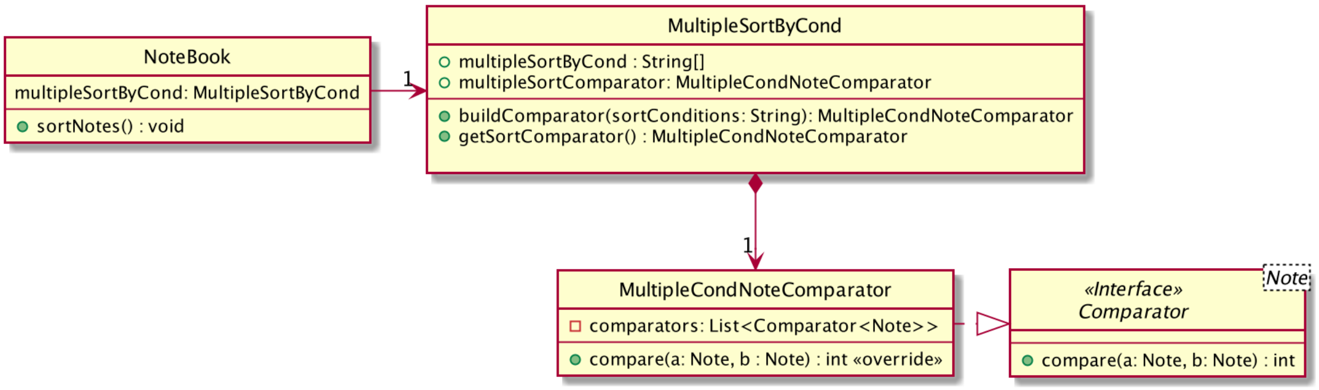

NoteBook
multipleSortByCond: MultipleSortByCond
sortNotes() : void
1
MultipleSortByCond
multipleSortByCond : String[]
multipleSortComparator: MultipleCondNoteComparator
buildComparator(sortConditions: String): MultipleCondNoteComparator
getSortComparator() : MultipleCondNoteComparator
1
MultipleCondNoteComparator
comparators: List<Comparator<Note>>
compare(a: Note, b : Note) : int «override»
«Interface»
Comparator
Note
compare(a: Note, b: Note) : int

Figure 9. Class diagram to illustrate how the `MultipleSortByCond` class interacts the other classes.

Sorting the note book rearranges the notes according to the sort conditions provided.

More than one of the conditions can be used to sort the notes in the note book at one time, with the first condition having the greatest precedence.

Example: `sort by/NumOfAccess DateAdded`

- NumOfAccess and DateAdded are the sort conditions, with NumOfAccess having greater precedence over DateAdded.

This is handled by the `MultipleSortByCond#buildComparator()`, which takes in the sort conditions specified by the user and returns a `MultipleCondNoteComparator` object that is used to sort the list of notes in NoteBook.

PROJECT: PowerPointLabs

{Optionally, you may include other projects in your portfolio.}