Ahmad Hatziq - Project Portfolio

PROJECT: T.A.rence

Hello there, I’m *Hatziq*. I am currently pursuing a major in Industrial & Systems Engineering, with minors in Computer Science and Statistics. at National University of Singapore (NUS).

I aim to build useful applications that aims to leverage on the large amount of data we have at our disposal today.

Data-driven decision making is the main principle that I abide by.

Overview

T.A.rence is a class management system for teaching assistants in NUS.

Given that many students in NUS serve as TAs and take up multiple
classes in a single semester, this application will allow for a smoother personal management process.

T.A.rence is designed to be a simple and intuitive application that allows users to interact by typing the commands using the Command Line Interface (CLI).

In addition, T.A.rence also comes with a clean Graphical User Interface (GUI) that presents information in an organized manner.

Summary of contributions

Major Enhancement: added Storage

All Storage functionality (serializing and de-serializing of Application & Json files)

- Summary of the features:
  - Allows the user to save the application data and read off from the data after the application is closed. such as students, tutorials, modules, attendance and assignments to be displayed in a easy-to-understand form.
- Justification:
  - Allows the user to efficiently carry out their work without having to memorise their previous commands.
- Highlights:
  - This enhancement affects existing commands and commands to be added in the future.
    Given the storage is dependant on the underlying structure of the model,

    - collaboration with other components of the application was highly needed.
    - The implementation too was challenging as it required to work with a variety of data structures, so as to maintain the efficiency and effectiveness of data processing.
    - Given the many layers of information contained within T.A.rence, information had to carefully extracted without compromising other related systems.

Major Enhancement: added Undo command

- Summary of the features:
    - Allows the user to undo the previous command.
- Justification:
    - Allows the user to undo any command if they had erroneously added a command.
- Highlights:
    - This enhancement is different from the one suggested in AB3. During the operation of the application, temporary state files will be stored on the hard disk. This modification allows for multiple states to be stored, without affecting valuable RAM.
    - Once an undo command is executed, the previous state is then loaded into the system memory from the saved json state file.

Minor Enhancement: added commands for "addModule" and "addTutorial".

- Added the basic skeleton functionality for TArence.

Other contributions

- Project management:
    - Used Github for milestone, bug and issue tracking
    - Assisted to review peer code before merging

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.

Undo Previously-entered Commands : `undo`

Undos a specified number of actions.

Format: `undo` u/[NUMBER_OF_ACTIONS]

Undoes any state-altering command from the application.

Format: `undo u/NUMBER_OF_STATES_TO_UNDO`

Example:

`undo u/2`

- Resets the application state to the previous state.
- Undo can only be applied to states that are present in that particular session (from application start-up)
- The `NUMBER_OF_STATES_TO_UNDO` refers to the previous number of states to undo in that session
- The `NUMBER_OF_STATES_TO_UNDO` **must be a positive integer**; 1, 2, 3, …

Clearing all entries : `clear`

Clears all entries from the T.A.rence.

Format: `clear`

Example:

`clear`

- Resets the application state to a clean state.

Saving the data

All data in T.A.rence is saved in the hard disk automatically after any command that changes the data.
There is no need to save manually.

Data is saved to "application.json".

Temporary state files are created in "/data/states/". On closure of the application, the "/data/states/" folder will be automatically deleted.

Adding a Module: `addModule`

Adds a module to T.A.rence.
Format: `addModule m/MODULE_CODE`.

Examples:

- `addModule m/CS1010`
- `addModule m/ST2132`

TIP Command synonyms: `addMod`, `addmod`, `addmodule`

Adding a Tutorial Slot : `addTutorial`

Adds a tutorial slot into the specified module.
Pre-condition: Module must already exist inside application.
Explanation: Adds a tutorial called Tutorial-01 which starts at 1PM, lasts for 60 minutes, and occurs every Monday during weeks 1,2, and 3 into module CS1010.

Format: `addTutorial tn/[TUTORIAL_NAME] st/[START_TIME] dur/[TUTORIAL_DURATION] d/[TUTORIAL_DAY] w/[TUTORIAL_WEEKS] m/[MODULE_CODE]`

Example:

- `addTutorial tn/Tutorial-01 st/1300 dur/60 d/Mon w/1,2,3 m/CS1010S`

NOTE START_TIME is in the format hhmm.
TUTORIAL_DURATION is in minutes

TIP Other input options for `weeks` field:

- `w/even` - weeks 4, 6, 8, 10, 12
- `w/odd` - weeks 3, 5, 7, 9, 11, 13
- `w/x-y` - weeks x to y inclusive, where x and y are integers from 1 to 13 inclusive

Omit `w/` field for the default tutorial weeks (3-13 inclusive).

Command synonyms: `addtut`, `addtutorial`

== 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.

Undo feature

The undo feature allows the user to undo multiple commands at once.

Implementation

The undo mechanism is facilitated by `JsonStateStorage`. It extends `TARence` with an undo history, stored externally as a `stateXX.json` file where 'XX' represents the sequential state of json files committed.

It relies on the following operations from Storage, exposed via the `Storage` interface.

- `Storage#isValidNumberOfRollbacks(Integer numberOfStatesToUndo)` — Checks if the specified number of undo commands to execute is valid.
- `Storage#getLatestStateIndex()` — Obtains the largest state stored. Used to obtain the state number which is required.
- `Storage#getSpecifiedState()` — Obtains the required state to reset the model to.
- `Storage#saveApplication()` — Saves the application state into `application.json`. This method additionally calls `JsonStateStorage#saveApplication()`, which is explained below.

It relies on the following operation from Model, exposed via the 'Model' interface.

- `Model#setModel(`ReadOnlyApplication`)`` — Resets the model to the supplied application state.

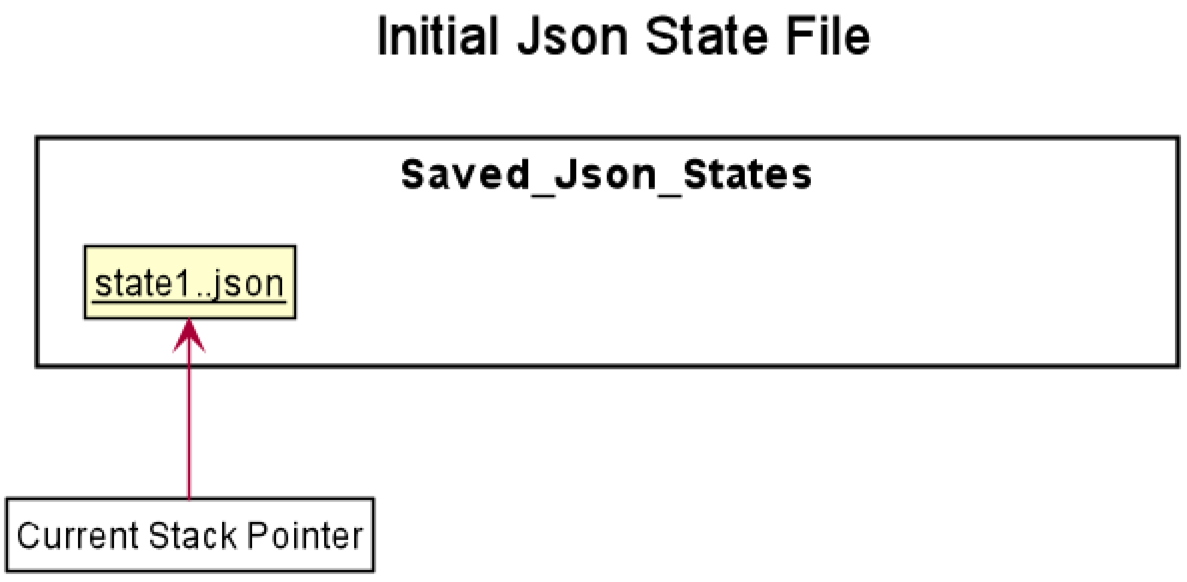
Additionally, it relies on the following operation from JsonStateStorage, not exposed via the Storage interface:

- `JsonStateStorage#saveApplication()` — Saves the current application state to file if there is a change in state. The files are saved as `stateXX.json`, where the file name is chronologically generated.
- `JsonStateStorage#stateStack` — A private attribute, which keeps track of the current and previous states. Its implementation is similar to the `currentStatePointer` used in AB4.

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 mechanism behaves at each step.

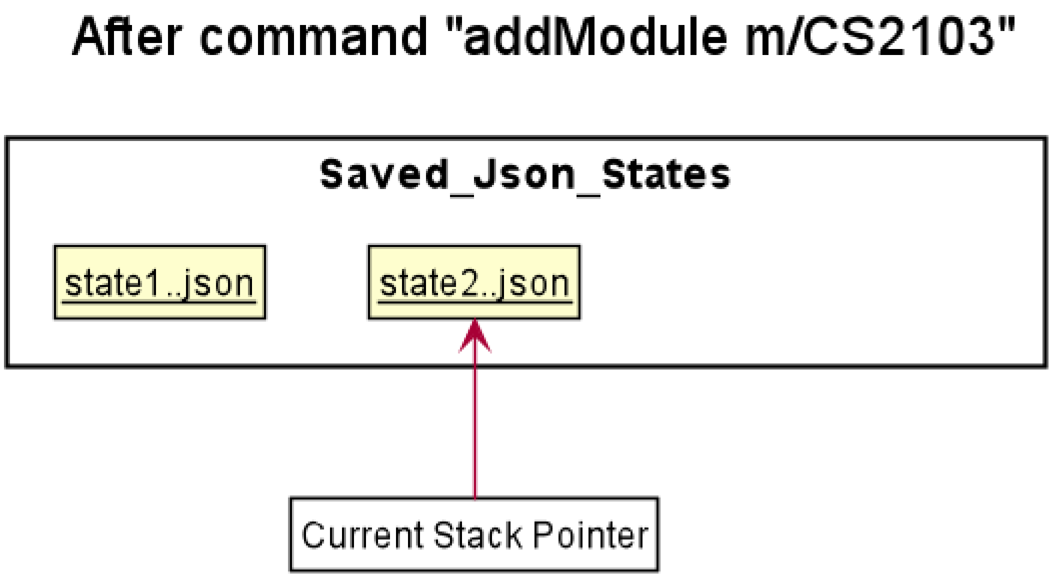
Step 1. The user launches the application for the first time. The `JsonStateStorage` will be initialized with the initial json file, `state1.json` and the `stateStack` will contain the value `1`.



Step 2. The user executes `addModule m/CS2103` command to add the module `CS2103` in the application.

The `addModule` command calls `Storage#saveApplication()`, which calls `JsonStateStorage#saveApplication()`, causing the modified application state, after the `addModule` `m/CS2103` command executes, to be saved externally in the `data/states` folder.

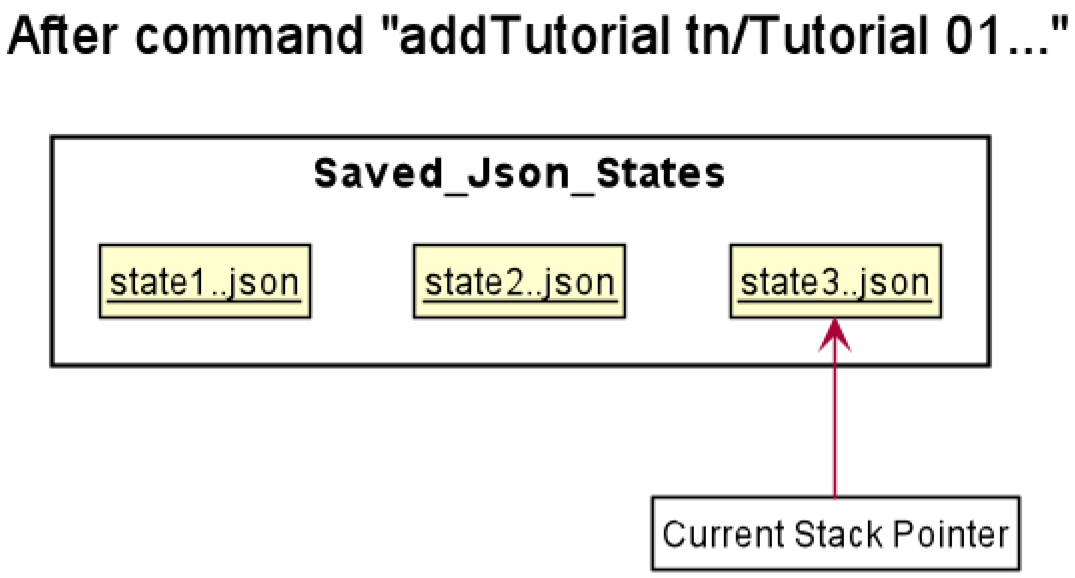
The latest state number `(stateStack.peek() + 1)` is pushed onto the stack, which corresponds with the json state file saved.



Step 3. The user executes `addTutorial tn/Tutorial 01 …` to add a new Tutorial. The `addTutorial` command also calls `JsonStateStorage#saveApplication()`.

As `JsonStateStorage#saveApplication()` detects that there has been a state change with the latest saved state (`state0.json`), the modified application state is saved externally to the `data/states` folder.

The `stateStack` is correspondingly updated (The latest state number `(stateStack.peek() + 1)` is pushed onto the stack, which corresponds with the json state file saved.)



NOTE	If a command fails its execution, it will not call `storage#saveApplication()`, so the application state will not be saved.

Step 4. The user now decides that adding the tutorial, `Tutorial 01` was a mistake, and decides to

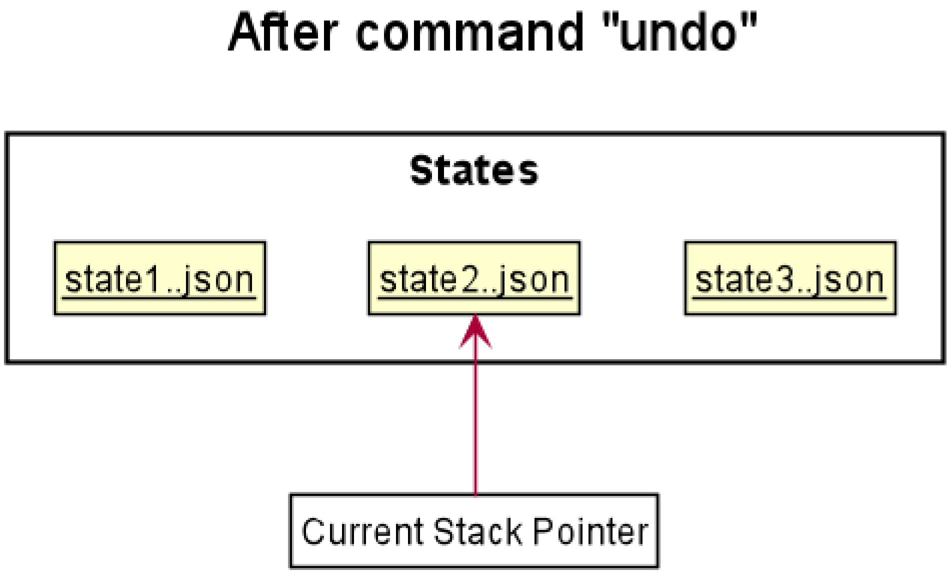
undo that action by executing the `undo` command.

The `undo` command will first verify that the number of commands is valid by calling `storage#isValidNumberOfRollbacks`.

If the number of undo commands is valid, it will call `storage#getLatestStateIndex` to get the position of the current state pointer (which is the number on the top of stateStack).

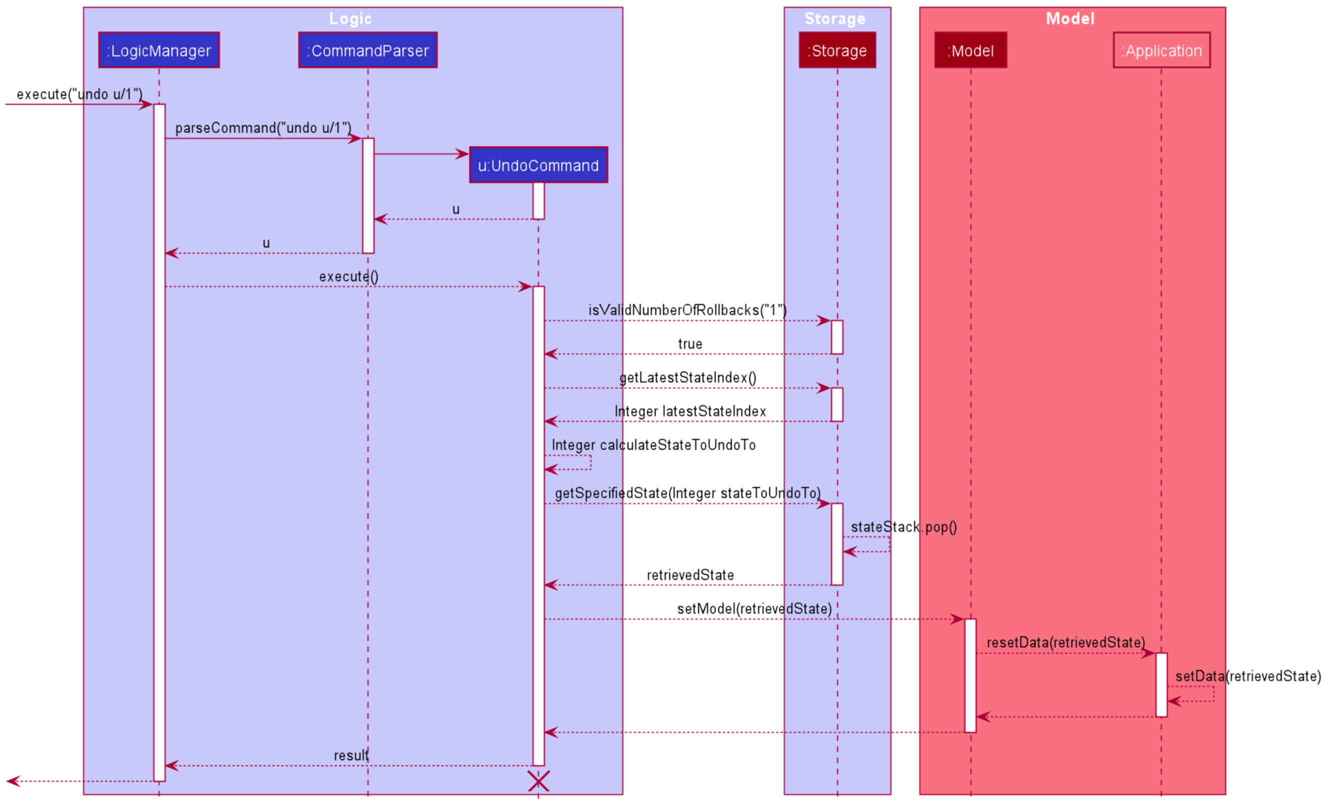
The `undo` command will then call `storage#getSpecifiedState()` to retrieve the desired state to undo from. In `storage#getSpecifiedState()`, the internal `stateStack` will pop the numbers until it gets to the required state number, to ensure that it the "pointer" is at the correct state.

With the desired state retrieved, `undo` command will call `model#setModel(application)` to reset the application to the retrieved state.



NOTE If the `stateStack.peek()` is at index / integer 0, pointing to the initial application state, then there are no previous application states to restore. The `undo` command uses `storage#isValidNumberOfRollbacks(number of states to undo)` to check if this is the case. If so, it will return an error to the user rather than attempting to perform the undo.

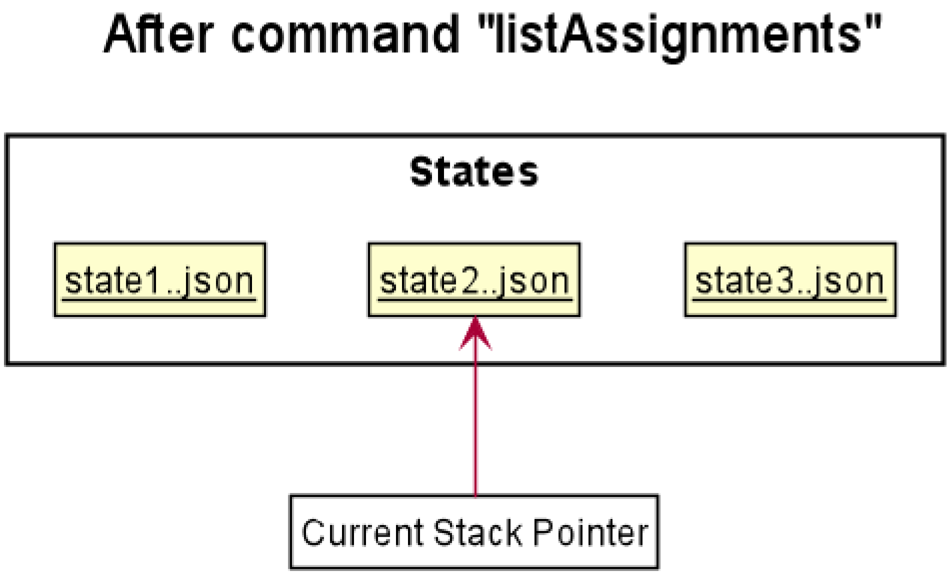
The following sequence diagram shows how the undo operation works:



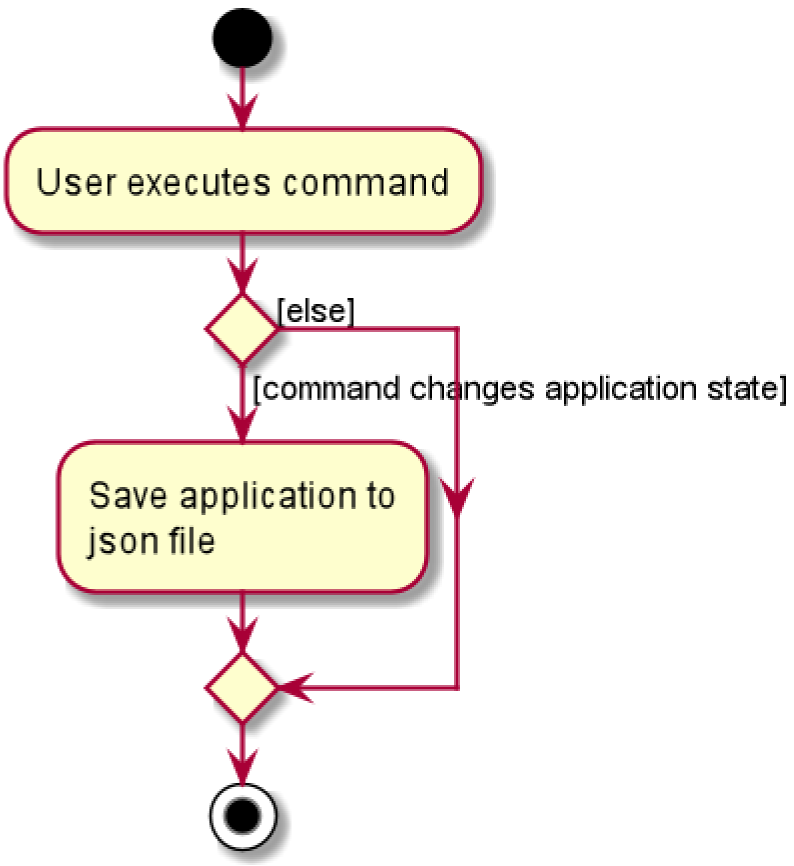
NOTE The lifeline for `UndoCommand` should end at the destroy marker (X) but due to a limitation of PlantUML, the lifeline reaches the end of diagram.

Step 5. The user then decides to execute the command `listAssignments`. Commands that do not modify the address book, such as `listAssignments`, will not result in a saved state.

`JsonStateStorage#saveApplication()` is still called. However, since there is no change with the previous state, `JsonStateStorage#saveApplication()` will not save the current application state.



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


User executes command
[else]
[command changes application state]
Save application to json file

Design Considerations

Aspect: How undo & redo executes

- **Alternative 1 (current choice):** Saves the entire application state into Hard Drive.
  - Pros: Won't result in performance issues if there are many states as it won't use up valuable RAM.
  - Cons: May have performance issues when reading the file from Hard Drive if the json file is large.
  - Cons: Suppose that "undo u/50" is executed, from state 70. State 20 will be loaded into memory. Subsequent states (ie state 21 onwards) will only overwrite the already present state 21 file.

The redundant states are not deleted from the hard disk. Only references to those states ,which is in the stateStack, are removed. However this is a minor problem as once the application is closed or started, it will automatically clear any state files.

- **Alternative 2:** AB3 suggested implementation: Saves the entire application state into RAM.
  - Pros: Easy to implememnt.
  - Cons: May have performance issues in terms of memory usage.
- **Alternative 3:** Individual command knows how to undo/redo by itself.
  - Pros: Will use less memory (e.g. for `deleteTutorial`, just save the tutorial being deleted).
  - Cons: We must ensure that the implementation of each individual command are correct.

Aspect: Data structure to support the undo/redo commands

- **Alternative 1 (current choice):** Use a stack to store the history of application states.
  - Pros: Eliminates redundancy as previous states not needed will be overwritten.
  - Cons: Leftover files of undone states are present in the hard drive.
- **Alternative 2 (better choice):** Use a stack as in Alternative 1. When `undo` command is executed, in addition to popping the stack, the application will delete the corresponding json files. This reduces the likelihood of accidentally reading redundant data.
- **Alternative 3 (AB3 choice):** Use a list to store the history of address book states.
  - Pros: Easy for new Computer Science student undergraduates to understand, who are likely to be the new incoming developers of our project.
  - Cons: Logic is duplicated twice. For example, when a new command is executed, we must remember to update both `HistoryManager` and `VersionedAddressBook`.
- **Alternative 4:** Use `HistoryManager` for undo.
  - Pros: We do not need to maintain a separate list, and just reuse what is already in the codebase.
  - Cons: Requires dealing with commands that have already been 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.
- **Code contributed**:
- [Commits] [Pull Requests] [RepoSense Code Contribution Dashboard]