Yeo Tong - LiBerry Project Portfolio

Introduction

Purpose

This portfolio is written to document my role as the **User Interface** (UI) designer and my contributions to our project, **LiBerry**. The following sections describe our project and the enhancements that I added in detail, together with relevant documentation from the user guide and developer guide.

About the project

LiBerry is a **free**, **single-user**, and **lightweight** library management system, developed by my team of 5 people for our software engineering module. It is a command line application designed for small community and private libraries in rural areas. Considering the lack of internet access in rural areas, the application is designed to not require any internet connection to function.

The figure below displays the graphical user interface of our application.

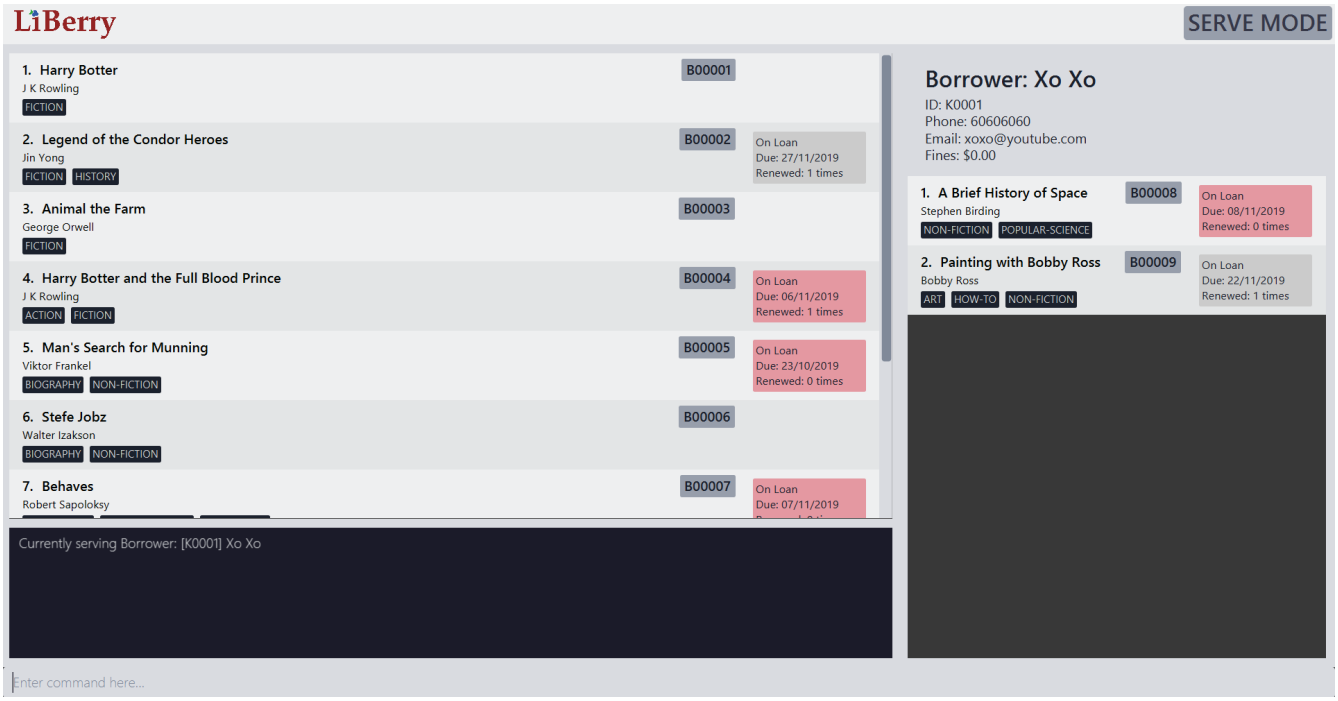

Figure 1. Screenshot of LiBerry user interface

The **Graphical User Interface** (GUI) of **LiBerry**, as shown above, is created with **JavaFX** while the entire software is written in **Java**.

Motivations

LiBerry is created to promote the setting up of private and community libraries, so as to improve literacy rates. One of the main reasons why literacy rates is low due to the lack of accessibility of knowledge. This is more prevalent in rural communities due to the lack of sources of knowledge.

Hence, in order to breach this gap in knowledge, we created **LiBerry**, which is a free, connectionless, easy to use library management system so as to encourage more people to set up their own community libraries.

Main Features

The main features of **LiBerry** are:

- Adding/Removing of books
- Registering/Unregistering of borrowers
- Loaning/Returning of books
- Searching of books

These basic features would allow librarians to manage their libraries effectively.

Key symbols

The following symbols and formatting are used in this document:

TIP	Denotes useful tips.
IMPORTANT	Denotes important details to take note of.

Summary of Contributions

This section describes the contributions that I have made to **LiBerry**.

Here is the link to the code that I contributed to the project: Code contributed: RepoSense Report

Major enhancement - Undo/Redo feature:

Description:

This feature enables users to undo all previous commands one at a time. Undone commands can be reversed one at a time by using the redo command.

Justification:

This feature makes the application much more user friendly. This is because this feature provides a convenient method for users to rectify any immediate mistakes that was made.

Highlights:

This enhancement was implemented in a way that minimises the memory usage required. In addition, this enhancement is extensible and can be applied to future commands with little overheads. This required an in-depth analysis of design alternatives and the consideration of the target audience of our application.

Secondary enhancement - Set User Settings feature:

Description:

This feature enables users to set customisable settings to our application. Some of the settings includes loan period, which is the number of days a book can be loaned out, and renew period, which is the number of the days a loan can be extended. These user settings are then stored locally so that users do not have to set them every time they open the application.

Justification:

As every library have different loan policies, this feature allows libraries to customise their own loan policies which is essential for the operation of the library. This feature also allows users to set different policies for different type of books which makes the application more flexible.

Highlights:

The most challenging part of this enhancement is making it persistent as it requires some knowledge of teh storage system.

Minor enhancement:

Added a help command that opens up a window to show the users a summary of commands available.

Other contributions:

- Project management:
  - Managed release `v1.2.1` on GitHub.
- Enhancements to existing features:
  - Created and Updated our application icon (Pull request: #99).
  - Updated User Interface to fit our application (Pull requests: #114, #149).
- Documentation:

    - Updated UI component of the Developer Guide (Pull request: #158).
- Community:
    - Reviewed teammates' pull requests (with non-trivial review comments) (Pull requests: #98, #125).
    - Contributed to forum discussions as a group by sharing a tip on checking code coverage when running tests (Link to post).
    - Added suggestions for other teams taking the module (Example).

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.

Setting User Settings: `set`

Sets the user settings for loan period (in days), renew period (in days), fine increment (in cents) and maximum renews allowed.

Format: `set [lp/LOAN_PERIOD] [rp/RENEW_PERIOD] [fi/FINE_INCREMENT] [mr/MAX_RENEWS]`

- Updates the user settings with the specified `LOAN_PERIOD`, `RENEW_PERIOD`, `FINE_INCREMENT` and `MAX_RENEW`.
- All the fields that are specified must be a positive integer.
- If none of the fields are specified, the current user settings would be displayed.
- `LOAN_PERIOD` refers to the number of days that a book can be loaned out for.
- `RENEW_PERIOD` refers to the number of days that the loan can be extended for.
- `FINE_INCREMENT` refers to the amount of cents charged per day for each overdue book.
- `MAX_RENEWS` refers to the maximum amount renewals that can be made per loaned out book.

Examples:

- `set`
  Shows the current user settings.

The figure below show the user interface after the `set` command has been added. The yellow box shows the change to the result display as now it shows the current user settings of the application.

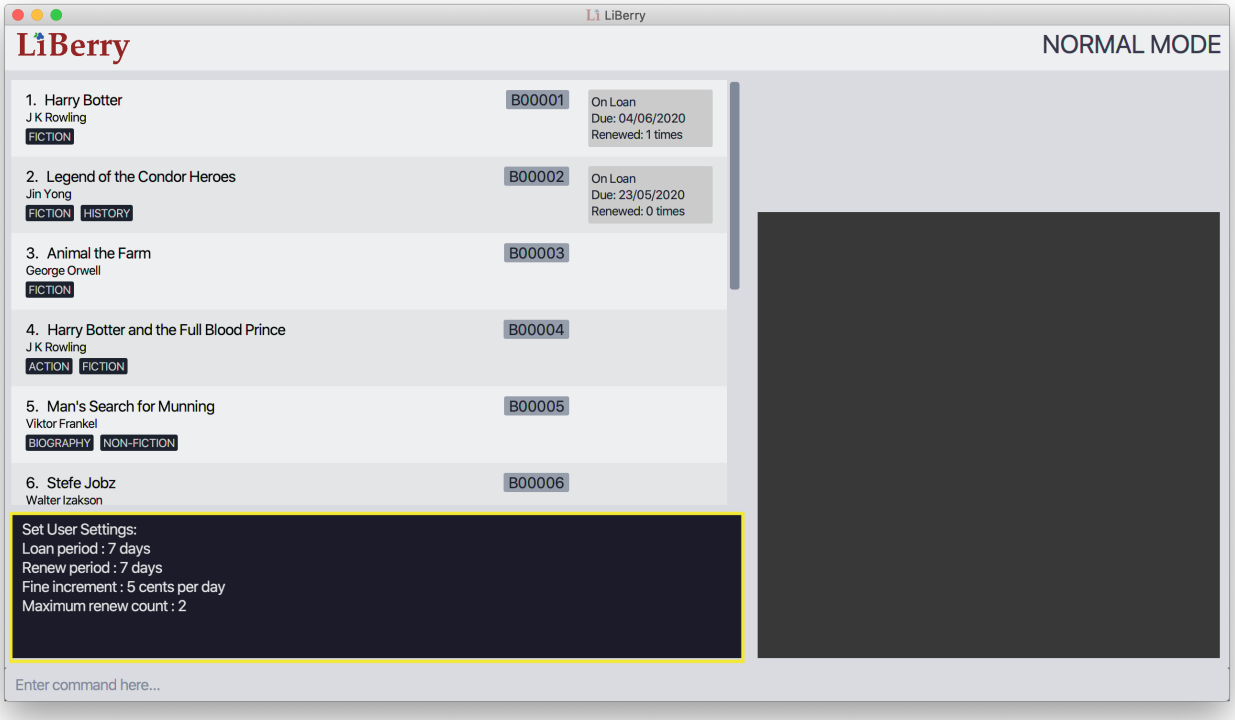

Figure 2. User interface after set to used to view current user sittings

- `set lp/7 rp/7 fi/5 mr/2`
  Sets the loan period to 7 days, renew period to 7 days, fine increment to 5 cents per day and maximum renews allowed to 2.

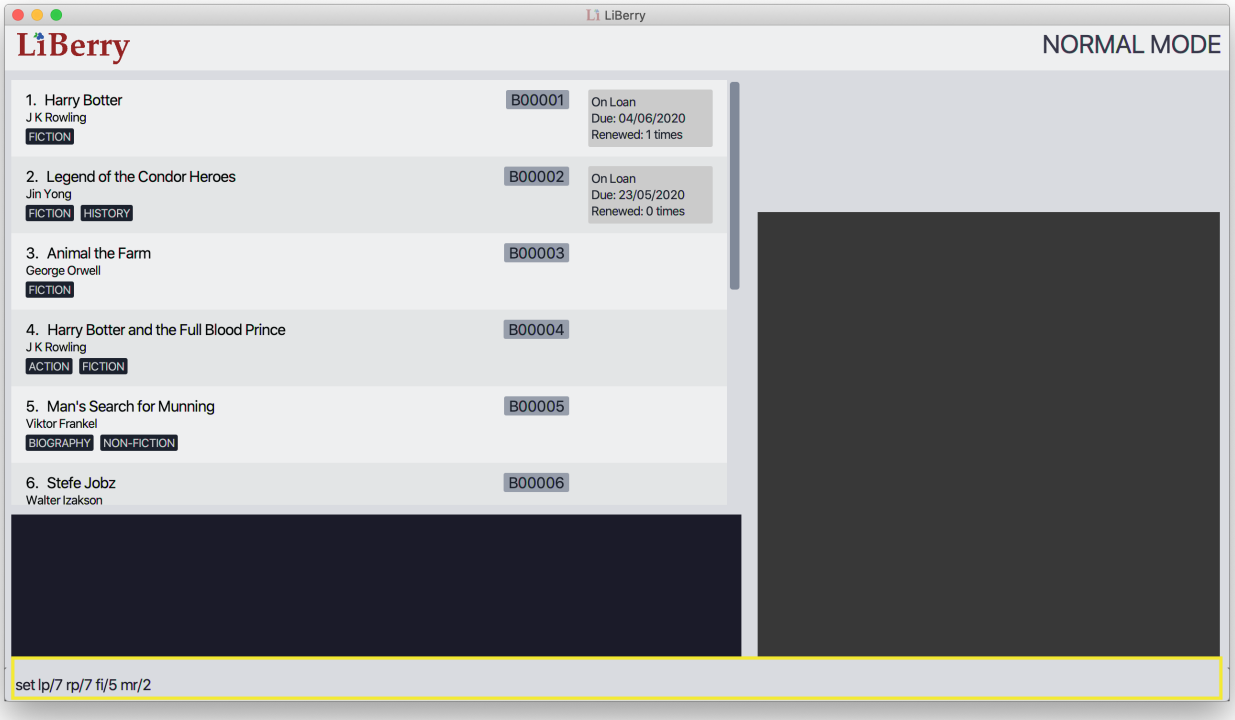

Figure 3. User interface before set command is executed.

The figure above shows the user interface before the set command is executed. The yellow box shows the set command that is being entered.

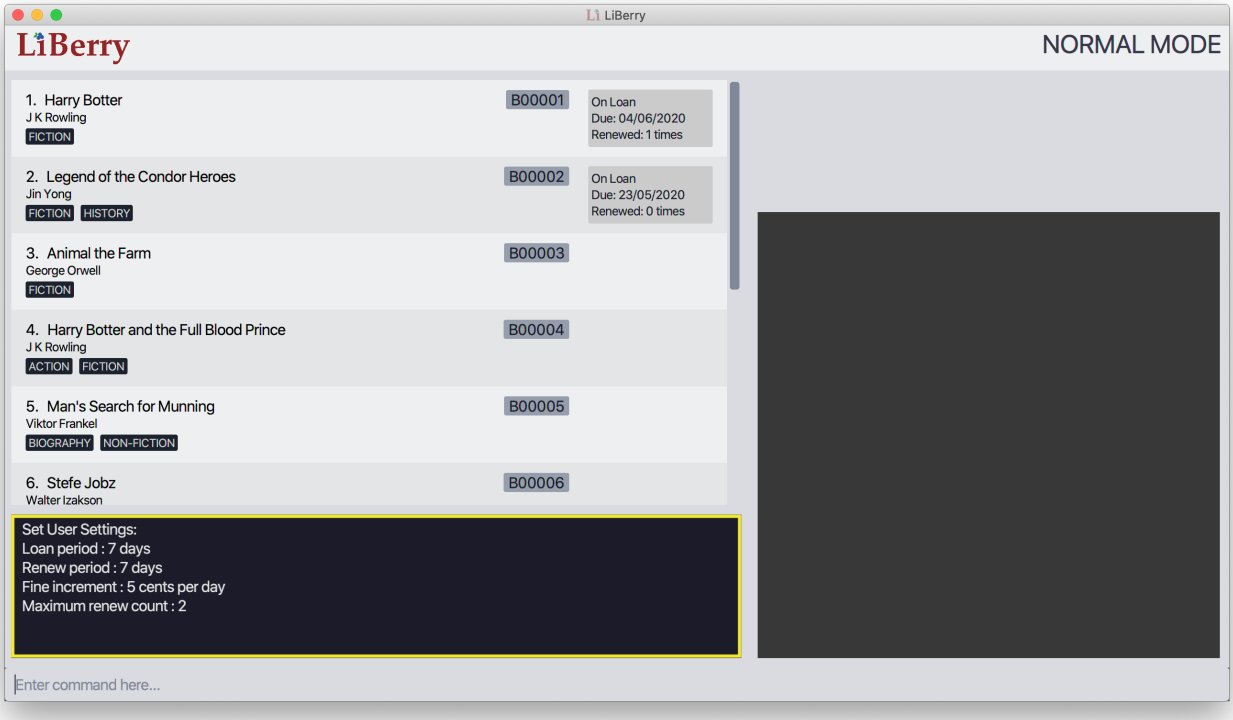

Figure 4. User interface after set command is executed.

After entering the set command, the user settings would be updated. The updated user settings will then be displayed in the result display as shown in the yellow box in the figure above.

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/Redo feature

Details of Implementation

The undo/redo mechanism is facilitated by `CommandHistory`. It contains a undo/redo command history, stored internally as an `commandHistoryList` and `currentCommandPointer`.

Additionally, it implements the following operations:

- `CommandHistory#commit()` — Saves the current reversible command in its command history.
- `CommandHistory#getUndoCommand()` — Returns the undo command for the most recent reversible command.
- `CommandHistory#getRedoCommand()` — Returns the redo command for the most recent undone command.

These operations are exposed in the `Model` interface as `Model#commitCommand()`, `Model#getUndoCommand()` and `Model#getRedoCommand()` respectively.

The undo/redo mechanism only works for commands that implements the `ReversibleCommand` interface. Currently, this mechanism only works for commands that modifies the catalog, loan records, borrower records or user settings.

Below, is the list of commands that can be undone/redone:

Undoable/Redoable Commands:

- `add`, `delete`, `edit`, `loan`, `register`, `renew`, `return`, `set`, `toggleui` and `unregister`.

NOTE	After every `serve`, `done` and `pay` command, the command history is cleared. This means that you will not be able to undo after entering one of the commands above. This is done to ensure that the user do not accidentally modify books that have been loaned out and that payments are not refundable.

The `ReversibleCommand` interface specifies that each command contains these three operations:

- `ReversibleCommand#getUndoCommand()` — Returns a command that undoes the `ReversibleCommand`.
- `ReversibleCommand#getRedoCommand()` — Returns a command that redoes the `ReversibleCommand`.
- `ReversibleCommand#getCommandResult()` — Returns the command result of the `ReversibleCommand`.

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 `CommandHistory` will be initialized with an empty `commandHistoryList` as shown in the figure below.

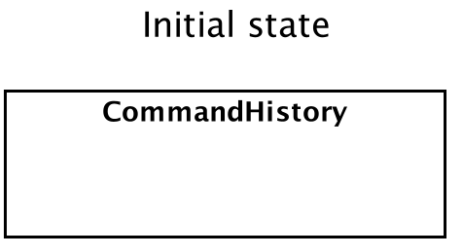

Figure 5. Initial state of CommandHistory

Step 2. The user executes `delete 5` command to delete the 5th book in the catalog. The `delete` command calls `Model#commitCommand()`, causing the `delete 5` command to be saved in the `commandHistoryList`, and the `currentCommandPointer` is pointed to the newly inserted command as shown in the figure below.

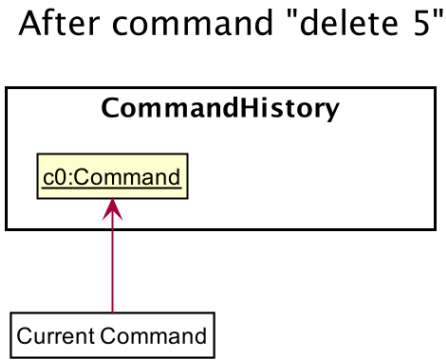

Figure 6. State of CommandHistory after `delete 5`

Step 3. The user executes `add t/Animal Farm …` to add a new book. The `add` command also calls `Model#commitCommand()`, causing the add command to be saved into the `commandHistoryList` as shown in the figure below.

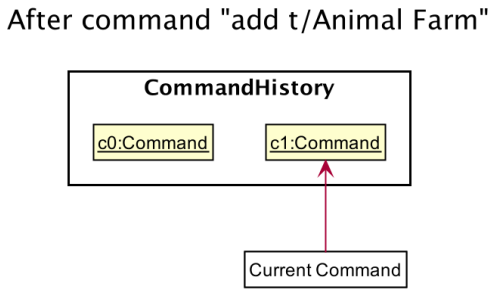

Figure 7. State of CommandHistory after `add t/Animal Farm`

NOTE	If a command fails its execution, it will not call `Model#commitCommand()`, so the command will not be saved into the `commandHistoryList`.

Step 4. The user now decides that adding the book was a mistake, and decides to undo that action by executing the `UndoCommand`. During the execution of the `UndoCommand`, `Model#getUndoCommand()` will be called. This would call `CommandHistory#getUndoCommand()`, which will retrieve the most recent `ReversibleCommand` that was executed, which is the `add` command. `ReversibleCommand#getUndoCommand()` would then be called and the `Command` returned would be executed, undoing the add command. This will then shift the `currentCommandPointer` once to the left, pointing it to the previous `ReversibleCommand` in the `commandListHistory` as shown in the figure below.

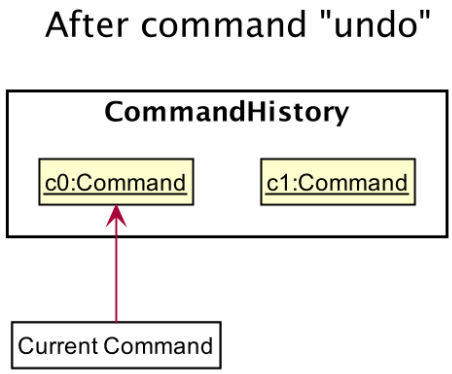

Figure 8. State of CommandHistory after `undo`

NOTE

If the `currentCommandPointer` is at index -1, pointing to no command, then there are no previous command to undo. The `undo` command uses `Model#canUndoCommand()` 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:

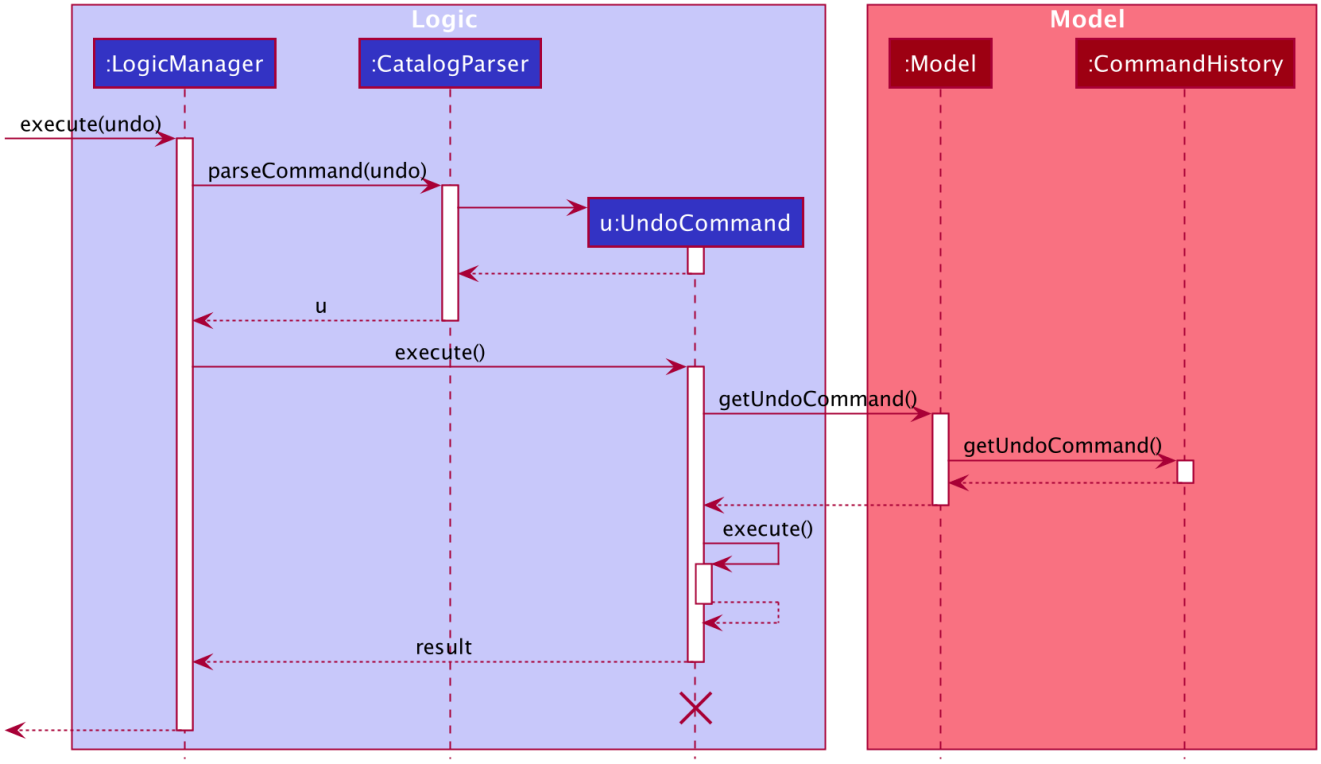

Figure 9. Sequence diagram for undo command

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.

The `redo` command does the opposite — it calls `Model#getRedoCommand()`, which shifts the `currentCommandPointer` once to the right, pointing to the previously undone Command, and executes the redo command from `ReversibleCommand#getRedoCommand()`.

NOTE

If the `currentCommandPointer` is at index `commandHistoryList.size() - 1`, pointing to the latest command, then there are no undone command to redo. The `redo` command uses `Model#canRedoCommand()` 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 `help`. Commands that do not modify the model, such as `help`, will usually not call `Model#commitCommand()`,`Model#getUndoCommand()` or `Model#getRedoCommand()`. Thus, the `commandHistoryList` remains unchanged as shown in the figure below.

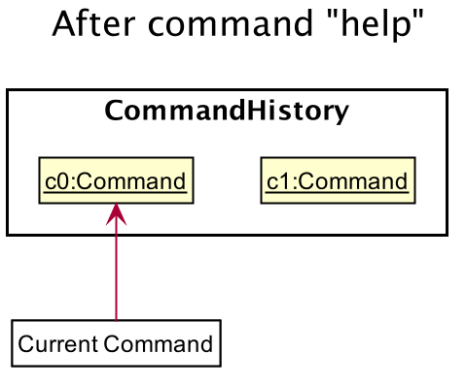

Figure 10. State of CommandHistory after `help`

Step 6. The user executes `set lp/7`, which calls `Model#commitCommand()`. Since the `currentCommandPointer` is not pointing at the end of the `commandHistoryList`, all commands after the `currentCommandPointer` will be purged. We designed it this way because it no longer makes sense to redo the `add t/Animal Farm ⋯` command. This is the behavior that most modern desktop applications follow.

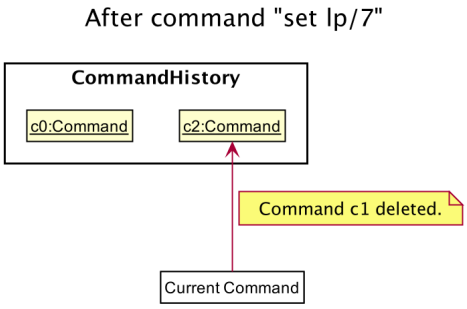

Figure 11. State of CommandHistory after `set lp/7`

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

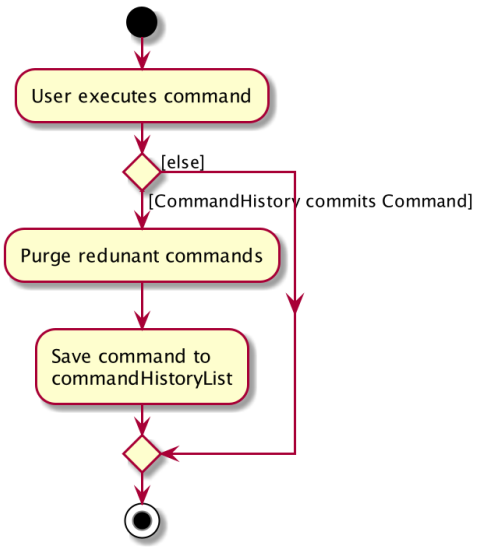

Figure 12. Activity diagram for committing Command

Design Considerations

Aspect: How undo & redo executes

- **Alternative 1 (current choice):** Individual command knows how to undo/redo by itself.
  - Pros: Will use less memory (e.g. for `delete`, just save the book being deleted).
  - Cons: We must ensure that the implementation of each individual command are correct.
- **Alternative 2:** Saves the entire catalog.
  - Pros: Easy to implement.
  - Cons: May have performance issues in terms of memory usage.

Considering our target audience, community libraries, which may be poor. They might be not able to afford a large amount of data storage. As a library may contain many books, borrowers and loans, storing a state of application for each command can be memory intensive. Hence, we chose to implement Alternative 1 so as to reduce the amount of memory usage.

Aspect: Data structure to support the undo/redo commands

- **Alternative 1 (current choice):** Use a list to store the commands for undo and redo.
  - Pros: Only need to maintain one data structure.
  - Cons: Harder for new developers to understand the mechanism for undo and redo.
- **Alternative 2:** Use two stacks to store a list of undoable and redoable commands.
  - Pros: Easy for future developers to understand as there are two separate stacks to keep track of the command to undo and redo.
  - Cons: Additional time required to add and pop from the stack.

We chose alternative 1 as it is easier to maintain a single data structure and it is faster compared to alternative 2.