Low Ee Ter - Project Portfolio

1. Introduction

For my CS2103T module in NUS, we developed a software project in teams, and this document is about my contributions to the project. My team consisted of five Year 2 students, namely Choong Jin Yao, Ryan Tay, Stanley Yuan, Zhou Tian Yu, and I. We were tasked with enhancing a basic command line interface desktop address book application initially containing about 10 kLoC (10,000 lines of code). We chose to morph it into a food delivery management system called DeliveryMANS. This enhanced application enables delivery call centres to efficiently manage their delivery manpower, and the data of their customers, restaurants, and orders.

A command line interface is a method of controlling a program by sending lines of text, or commands, to the application. This is in contrast to a graphical user interface, which includes menus and sometimes icons. Since we were constrained to keeping the command line interface, we did not remove it. Our application has a command line interface in the sense that manipulation of the data is done via commands typed and sent to the app, but there are still elements of a graphical user interface with the various boxes and panes shown in the image below. It is also a desktop application, meaning that it is made to be run on desktop computers and laptops, and not on smartphones or mobile devices.

This is what our project looks like in the restaurant context:

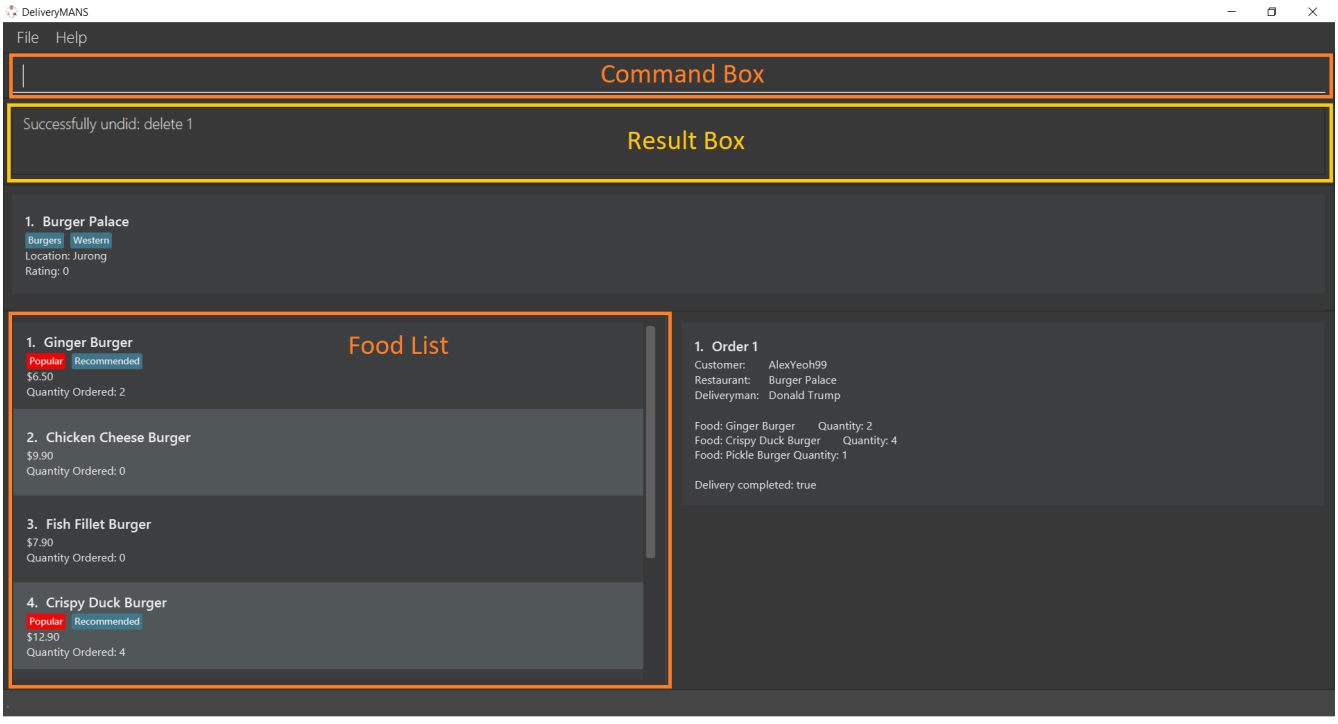


Figure 1. The graphical user interface for DeliveryMANS, consisting of the command box, result box, and various information panes including the food list pane.

Here is a list of the main features of our application.

- Management of restaurant, food, customer, and deliveryman data

- Viewing of customer orders
- Adding of orders and assigning them to deliverymen
- Keeping track of deliverymen
- Undo/redo of changes to data

My role was to design and write the code for the undo/redo feature and food data. The following sections illustrate these enhancements in more detail, as well as the relevant documentation I have added to the user and developer guides in relation to these enhancements.

Note the following symbols and formatting used in this document:

NOTE	Important details to take note of.
`undo`	This formatting indicates that this is either a command that can be input into the command line and executed by the application, or a component, class or object in the architecture of the application.

2. 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. The user can also undo and redo multiple commands at once.
- 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, as it only deals with the state of the program. An in-depth analysis of design alternatives was necessary to ensure that the application runs smoothly. The implementation of the undo mechanism itself was somewhat straightforward. However, I had to ensure that it works with all the restaurant, food, customer, deliveryman, and order data. To do so, I read and understood most of the code for the entire application, so that I could make the various components work with the undo feature (making the classes immutable as described in the Developer Guide). Afterwards, I had to find the source of bugs which occurred after making those changes and fix them.
- Credits: The address book developer guide for suggesting the idea.

Enhancement added: I added the code to store and display food data.

- What it does: The feature is used by the restaurant feature to store its menu and display it when required.
- Justification: A food item has many attributes such as its name and price, and this data needs to

be stored and displayed.

- Highlights: The feature was designed so it worked well with the undo feature and reduces the possibility of bugs (the `Food` class was immutable). In addition, price is stored as a `BigDecimal`, which means each price is stored as an exact value in the application. This eliminates the rounding errors that have caused serious consequences in stock market applications and other businesses.

Code contributed: Please click these links to see a sample of my code:

Initial implementation of undo command [Functional and test code]

Other contributions

- Enhancements to existing features: Adapted some of the original test utility classes to the new application structure.
- Documentation: Added details of undo and redo features to the user and developer guide.
- Community: Reviewed pull requests, pointed out possible bugs and suggested alternative methods of implementing certain components of the application.

3. Contributions to the User Guide

We updated the original address book User Guide with instructions for the enhancements that we added. The following is an excerpt from our DeliveryMANS User Guide, showing additions that I have made for the undo listing feature. Due to space constraints, I have left out the undo and redo feature, which can be found in the User Guide.

(begin contribution to User Guide)

3.1. Listing of undo/redo actions: `-undo_list`

This command lists the actions that can be undone and redone so that multiple actions can be redone at once with the `-undo_till` command.

Format: `-undo_list`

Upon executing the command, the result box will show something similar to the following:

Here are the actions that can be undone or redone, use -undo_till INDEX to undo or redo till that action.

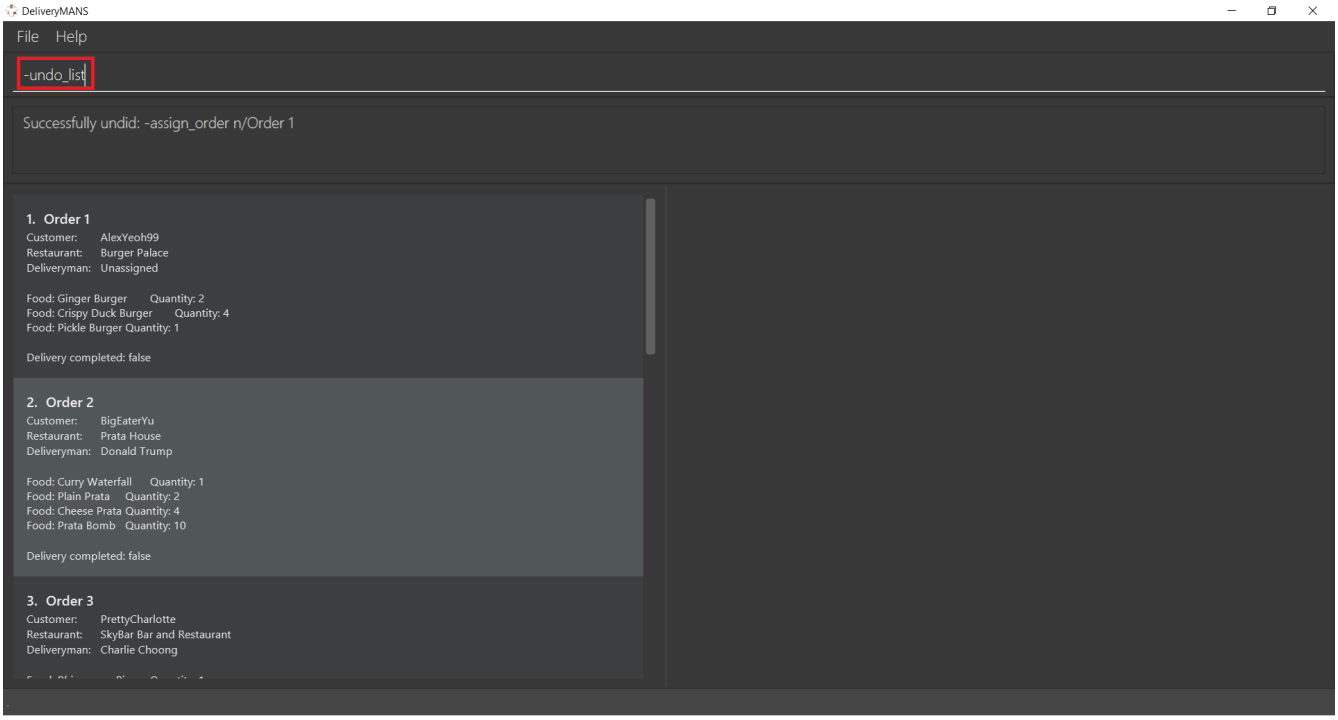
1: (Undo) (command you previously executed here)

3: (Redo) (command you previously undid here)

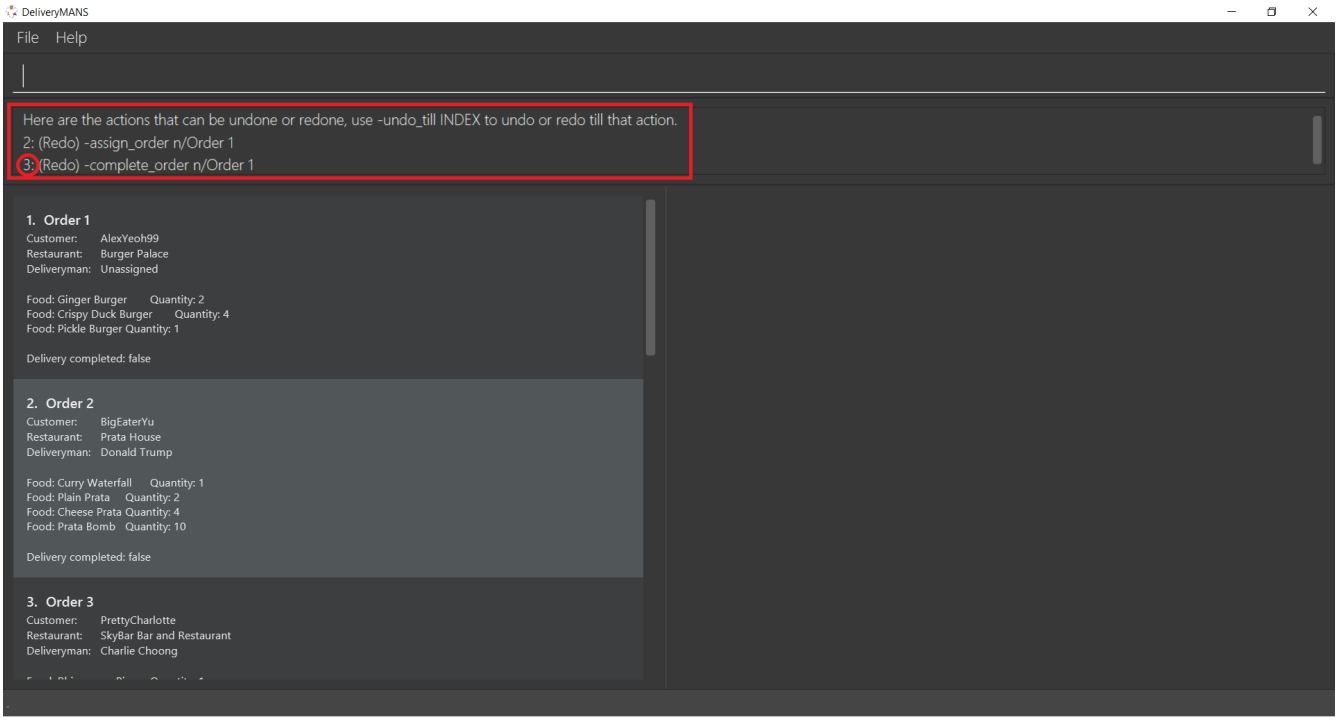
(Undo) indicates that the command was performed, and its effects can be undone. (Redo) indicates that the command has previously been undone, and can now be redone. After using this command, take note of the index of the command that you want to undo (or redo) until, and use the command `-undo_till INDEX` to perform the undo or redo.

NOTE	The undo list can be seen as a timeline of commands; it is not possible to undo only certain commands and not the other intervening commands.

Say you have undone two commands and decided that you actually want both of the commands redone. First type `-undo_list` in the area indicated below by the red rectangle and press **Enter**.



The list of commands that can be undone or redone is then listed. In the image below, there are two commands that can be redone. To redo *till* the last command, take note of the index of the last command, which is 3.



Now execute `-undo_till 3`. This command will be explained in more detail in the following section.

3.2. Undoing or redoing commands till a specific command: `-undo_till`

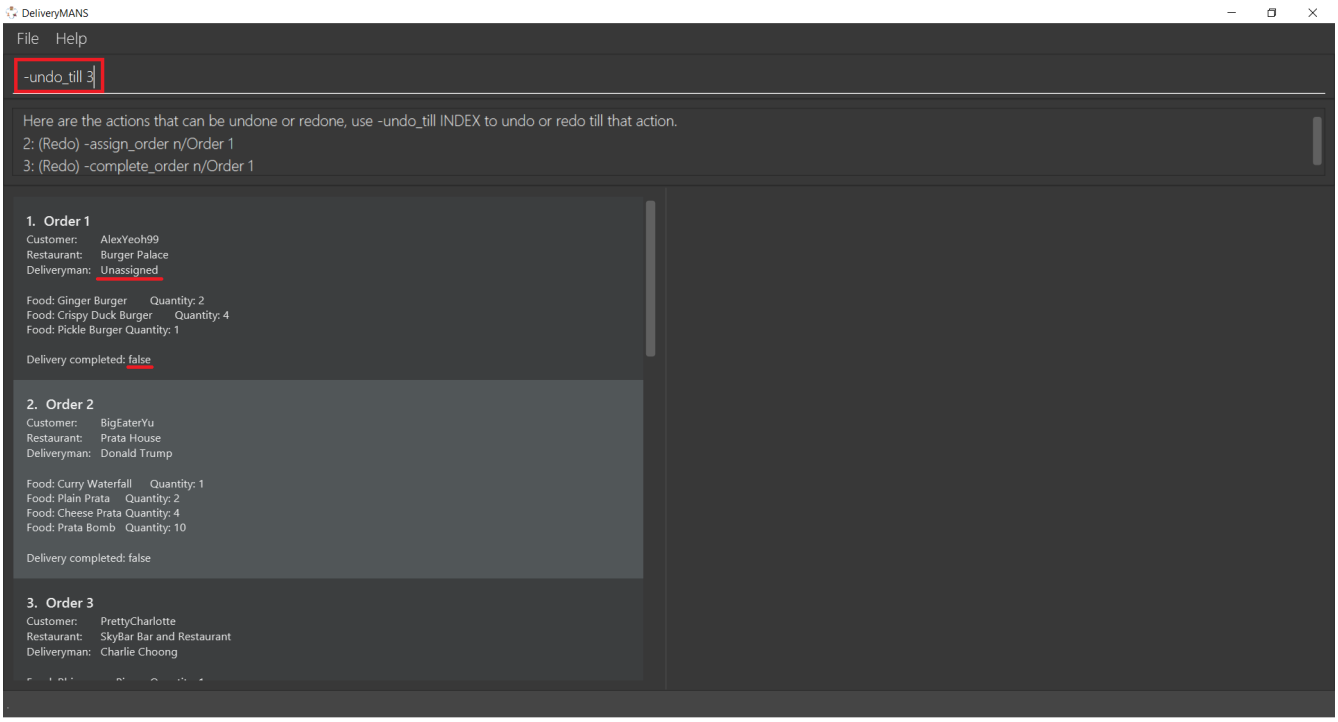
This command undoes or redoes all commands from the current state until after a specific command. It takes in one parameter, INDEX, which is the index of the specific command. This index is obtained from the command described in the previous section, `-undo_list`.

Format: `-undo_till INDEX`

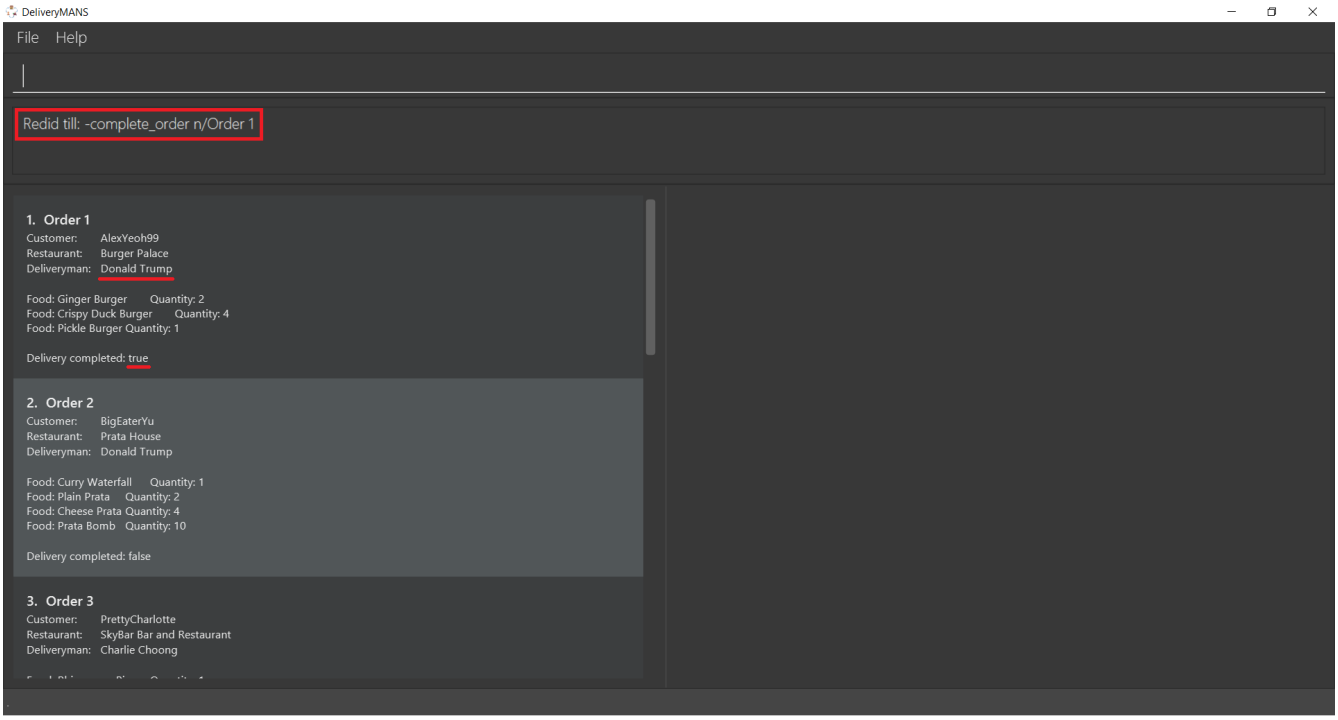
Example: `-undo_till 3`

NOTE	*Requirement* The index provided must be a valid index obtained from `-undo_list`.

From the previous section, the index of the command to redo till is 3. Type `-undo_till 3` in the app and press **Enter**. From the result box in the image below, it can be deduced that the first order was assigned a deliveryman and marked as completed, but these two commands were then undone. The image also shows that the deliveryman is now unassigned and the order is now shown as not delivered.



Upon successful execution, the result box shows whether commands were undone or redone, and the specific command that has been undone or redone until. Also, the first order now has the deliveryman reassigned and delivery marked as completed!



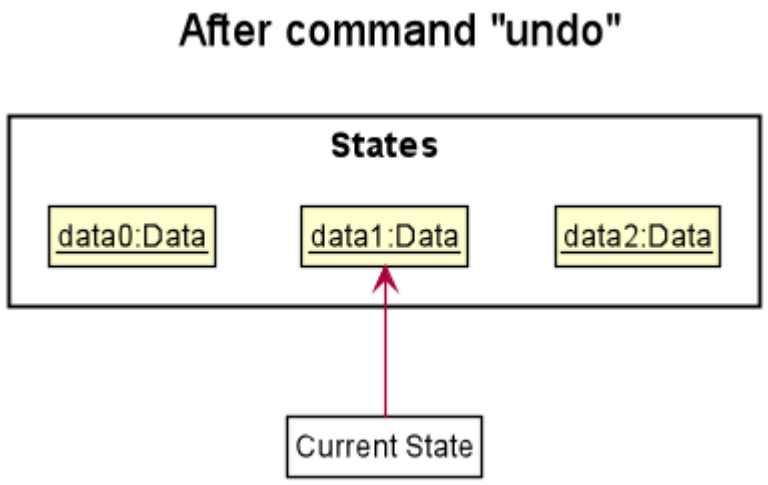
(end contribution to User Guide)

4. Contributions to the Developer Guide

We also replaced the address book implementation details in the Developer Guide with the implementation details of our own features. The following is a portion of our Developer Guide which shows my adaptation of the original guide to fit my implementation of the undo/redo function which I came up with from scratch. The extract starts at Step 4 due to space constraints.

(begin contribution to Developer Guide)

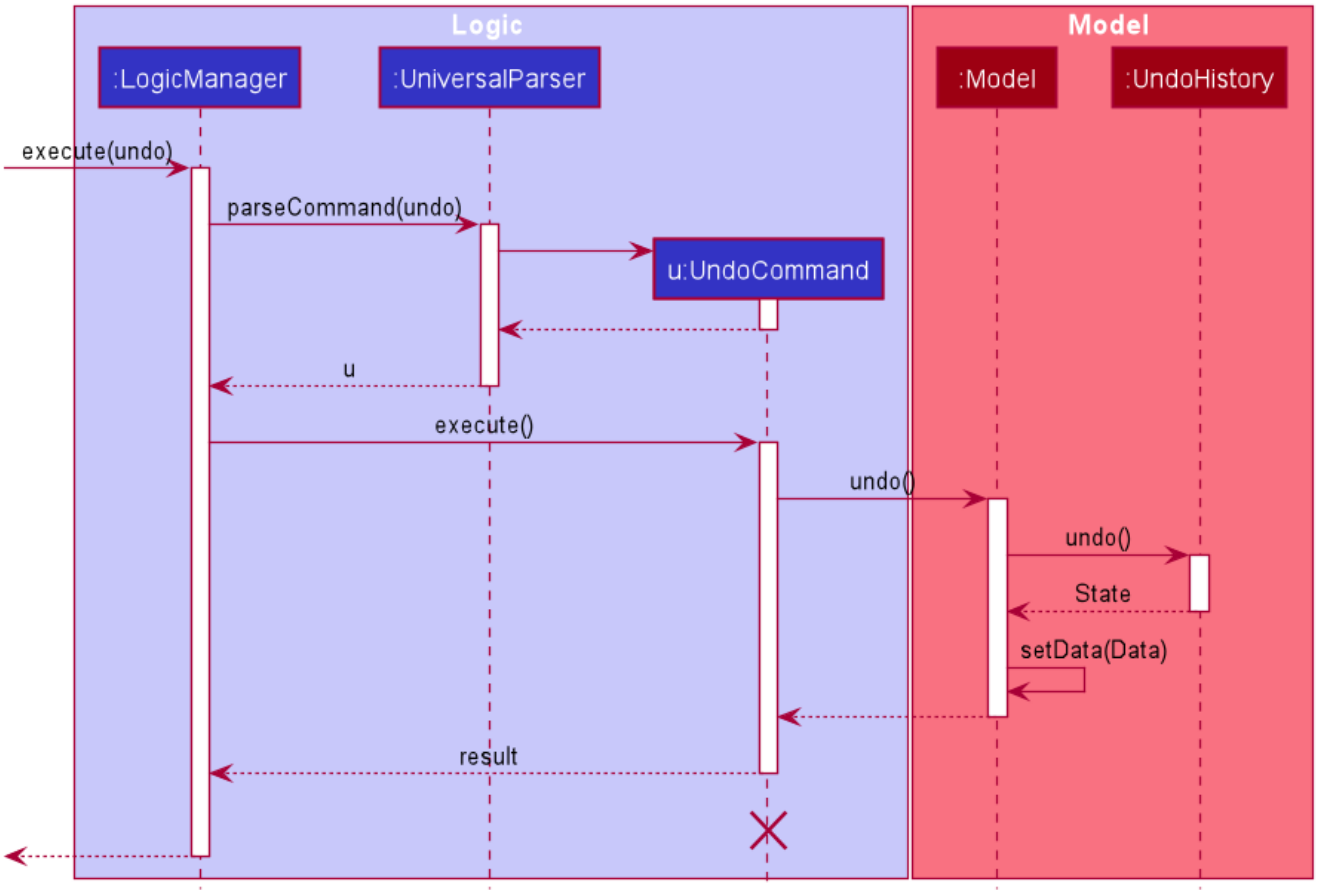
Step 4. The user now decides that adding the person was a mistake, and undoes that action by executing the `undo` command. The `undo` command will call `Model#undo()`, which will shift the `current` state pointer once to the left, pointing it to the previous state, and restores the data to that state.



NOTE

If the `current` state pointer is at index 0, i.e. pointing to the initial state, then there are no previous states to restore. The `undo` command uses `Model#hasUndo()` 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

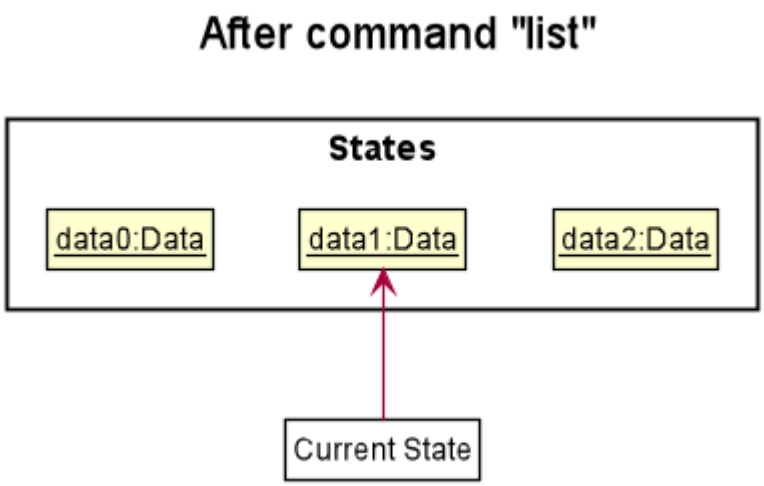
The lifeline for `UndoCommand` should end at the destroy marker (X) but due to a limitation of PlantUML, the lifeline reaches the end of diagram.

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

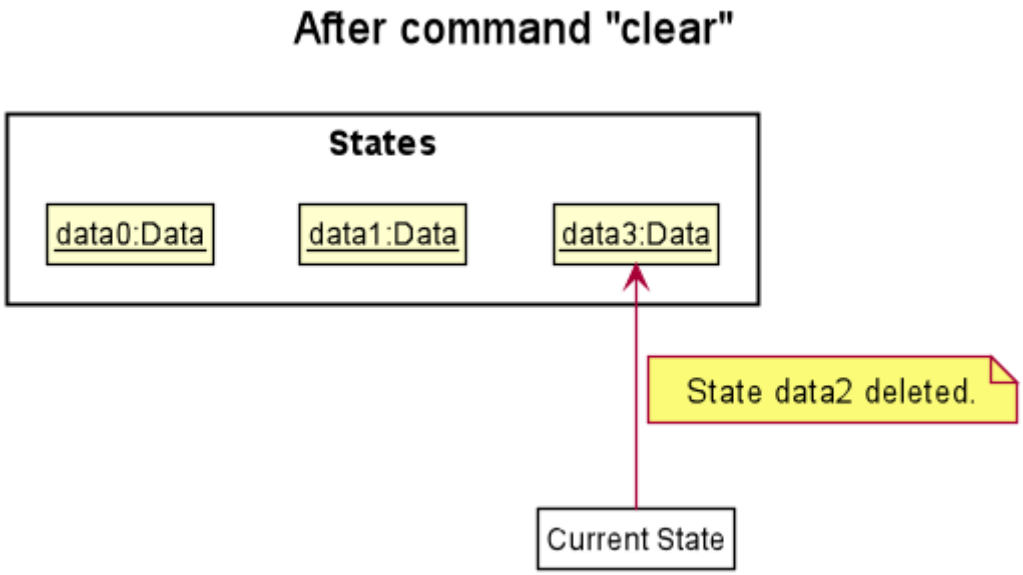
NOTE

If the `current` state pointer is at index `history.size() - 1`, i.e. pointing to the latest model data state, then there are no undone states to restore. The `redo` command uses `Model#hasRedo()` to check if this is the case. If so, it will return an error to the user rather than attempting to perform the redo.

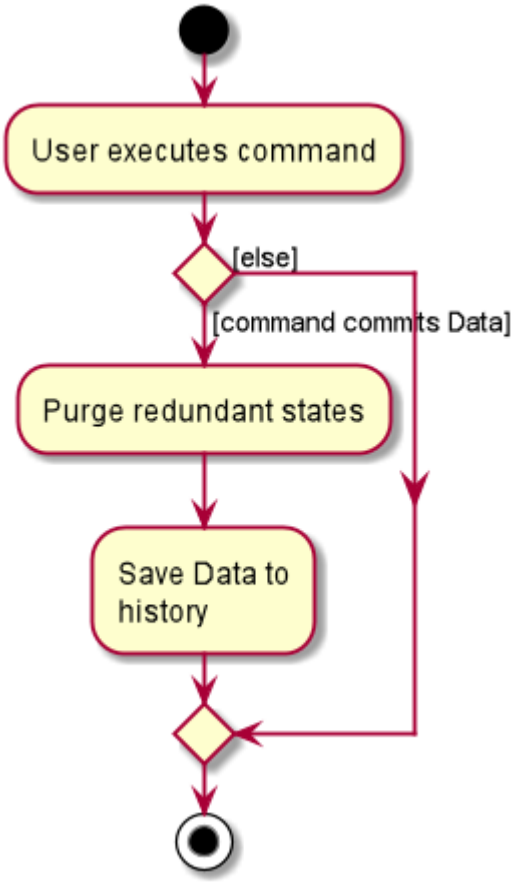
Step 5. The user then decides to execute the command `list`. If a command does not modify the data, it will not be stored in the undo history as `UndoHistory` checks for equality with the previous state. Thus, the `history` list remains unchanged.



Step 6. The user executes `clear`, which calls `Model#notifyChange()`. Since the `current` state pointer is not pointing at the end of the `history` list, all model data states after the `current` state pointer will be purged. We designed it this way because it no longer makes sense to redo the `add u/David …` command. This is the behavior that most modern desktop applications follow.



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



4.1. Design Considerations

This section discusses current and possible alternative methods to design the undo/redo function.

Aspect: How undo & redo executes

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

Saving the entire address book was chosen because it is more straightforward. Adding logic to each command to undo itself would be too much work given the time constraints.

Aspect: How undo states are copied

- **Alternative 1 (current choice):** Make all objects in the databases immutable.
  - Pros: Easy to undo and redo, and objects which are not modified can be shared, saving memory.
  - Cons: Slightly more complex to understand, and every minor change requires updating the database with a new object.

- **Alternative 2:** Add a clone method.
    - Pros: Slightly simpler to reason about.
    - Cons: Uses a lot of memory as each command executed requires cloning of all objects.

Making all objects immutable was chosen as it was the original design of AB3 and saves memory.

(end contribution to Developer Guide)