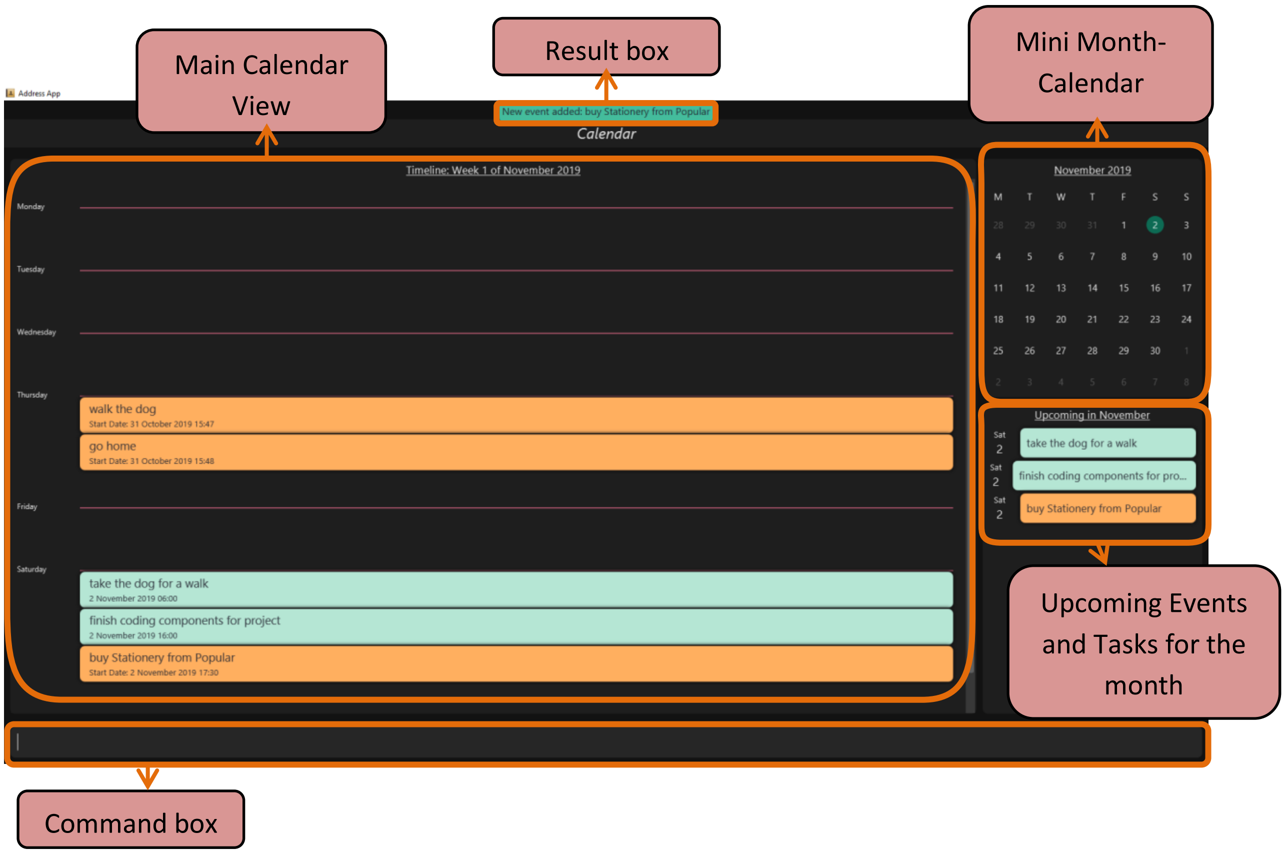
Bruce Ong - Project Portfolio for Horo

About the project

My team of 5 software engineering students were given the task of enhancing a basic command line interface desktop addressbook application for our Software Engineering Project. We decided to morph it into a schedule-manager called Horo. Horo allows one to record down daily events and tasks, all of which can be easily accessed in the form of a visual aid- Horo's built-in calendar. My team designed Horo with university students in mind, to provide them a convenient way to organise their activities better through a modern and minimalistic interface.

This is what our project looks like:



New event added: buy Stationery from Popular

Timeline: Week 1 of November 2019

Monday

Tuesday

Wednesday

Thursday

walk the dog
Start Date: 31 October 2019 15:47

go home
Start Date: 31 October 2019 15:48

Friday

Saturday

take the dog for a walk
2 November 2019 06:00

finish coding components for project
2 November 2019 16:00

buy Stationery from Popular
Start Date: 2 November 2019 17:30

November 2019

M	T	W	T	F	S	S
28	29	30	31	1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	1
2	3	4	5	6	7	8

Upcoming in November

Sat 2 take the dog for a walk

Sat 2 finish coding components for pro...

Sat 2 buy Stationery from Popular

*Figure 1: The Graphical User Interface (GUI) for **Horo**.*

My role was to design and implement the undo and redo features. The sections that follow highlight my approach to implementing these enhancements, as well as the contributions I have made to the user and developer guides.

Note the following symbols and formatting used in this document:

❗	This symbol indicates crucial information.

undo	A grey highlight indicates that this is one of the commands that Horo accepts.
ModelManager	Blue text with grey highlight indicates that this is a component, class or object in Horo’s architecture.

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 is able to undo any of the previous commands that have been executed; it restores the state of Horo as though the previous command had not been executed at all. The redo command does the exact opposite. It redoes a previously undone command.

- Justification: Suppose the user accidentally deletes the wrong task from Horo, or adds a task only to find that there was a misspelling of the task description and the wrong event time provided. Of course, the user can simply add the task back or edit the task using other Horo commands. However, this can be rather tedious if such mistakes happen often. As such, providing an undo command allows the user to conveniently “undo” mistakes. A redo command is also provided as a natural counterpart to the undo command.

- Highlights: Since my team was morphing the addressbook, as opposed to starting from scratch, I was not fully aware of how the data was being conveyed from the addressbook to the GUI. As such, my first implementation correctly manipulated the data, but all changes were not reflected on the GUI. I had to change my approach slightly in order to fix this issue.

Code contributed:

RepoSense Link

Function Code: 1 2 3 4 5 6 7
Test Code: 1 2

Other contributions: 1 2 3 (Helped to create add, delete and edit task commands)

Contributions to the User Guide

My team had to update the original addressbook User Guide, as many of our enhancements comprised either new instructions or existing ones that have been tailored to Horo. The following is an excerpt from our Horo User Guide, showing the additions that I have made in relation to the undo and redo features.

Undoing a previous command: `undo`

This command restores Horo to the state before the previous command was executed.

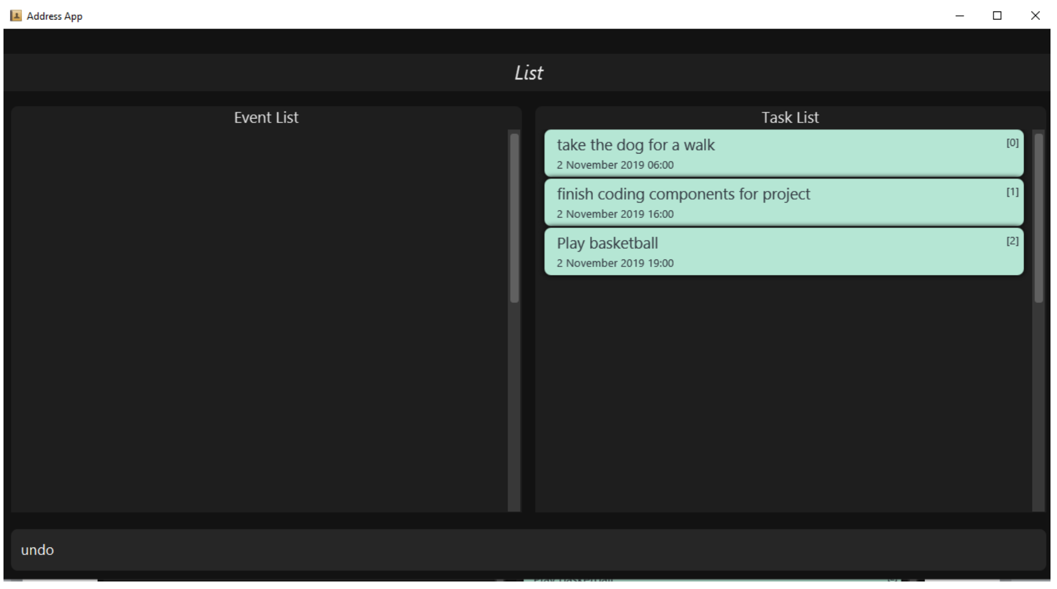
Example of why it is useful:
Suppose that you have deleted the wrong task from Horo. Rather than having to type the `add_task` command along with the description of the deleted task (to add back the wrongly deleted task), you can simply type in `undo`, which will revert Horo to the state before the deletion of the wrong task was executed.

This means Horo is now restored to its desired state, as if you did not commit the deletion mistake at all! You can now proceed to delete the right task.

Illustration:
Suppose that we wanted to delete task 2 in the list, but deleted task 3 instead.
(Horo also has a list view on top of the calendar view; I will illustrate the feature with the list view)

To undo:

1. Type undo in the command box, and press the Enter key to execute it.



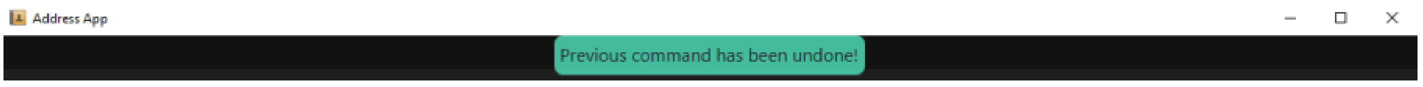
take the dog for a walk [0]
2 November 2019 06:00

finish coding components for project [1]
2 November 2019 16:00

Play basketball [2]
2 November 2019 19:00

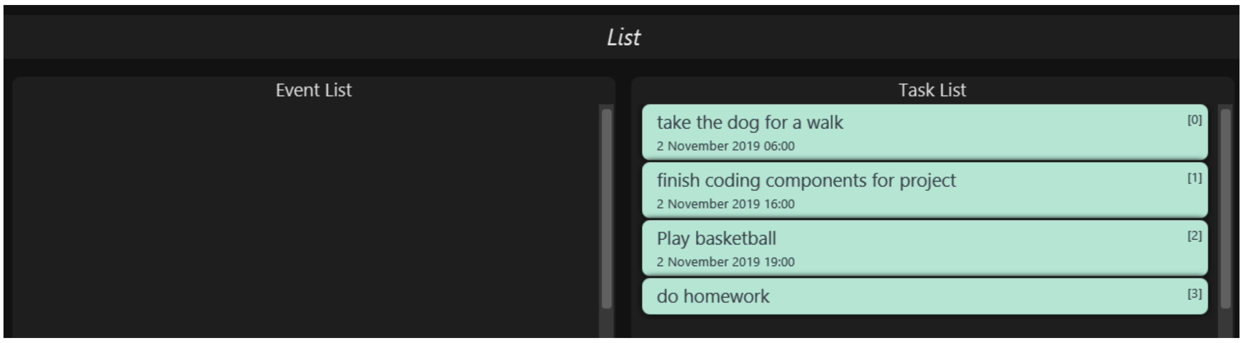
undo

2. The result box will display the message: “Previous command has been undone!”



Previous command has been undone!

3. You can see that the wrongly deleted task (task 3) is visible in the list once again.



take the dog for a walk [0]
2 November 2019 06:00

finish coding components for project [1]
2 November 2019 16:00

Play basketball [2]
2 November 2019 19:00

do homework [3]

The undo command undoes previous commands in reverse chronological order.
Suppose that you have executed the following commands in this order:

1. Adding a task
2. Editing a task

Now, if you execute the undo command, you will first revert Horo to the state before a task was edited.
Then, if you execute undo again, you will revert Horo to the state before a task was added.

The undo command only works on state-changing commands.
State-changing commands are those that manipulate task and event data stored in Horo. Examples include add_task, delete_task and edit_task.
Undo commands only work on these types of commands because there is an actual change in the state of Horo that can be undone.

On the other hand, non-state-changing commands include find and help. These commands are only concerned with producing user output for the user in the GUI, but do not modify any of the data stored in Horo. As such, these types of commands are ignored by the undo operation since there is nothing to undo.

As such, if we first add a task to Horo, then we call the help command, calling undo will ignore the help command and proceed to revert Horo to before a task was added.

The undo command only executes if there are previous states to revert back to.
If no command has been previously executed, or if Horo has already been reverted to the earliest possible state by multiple undos, then calling undo further will amount to no effect.

Redoing an undone command: redo

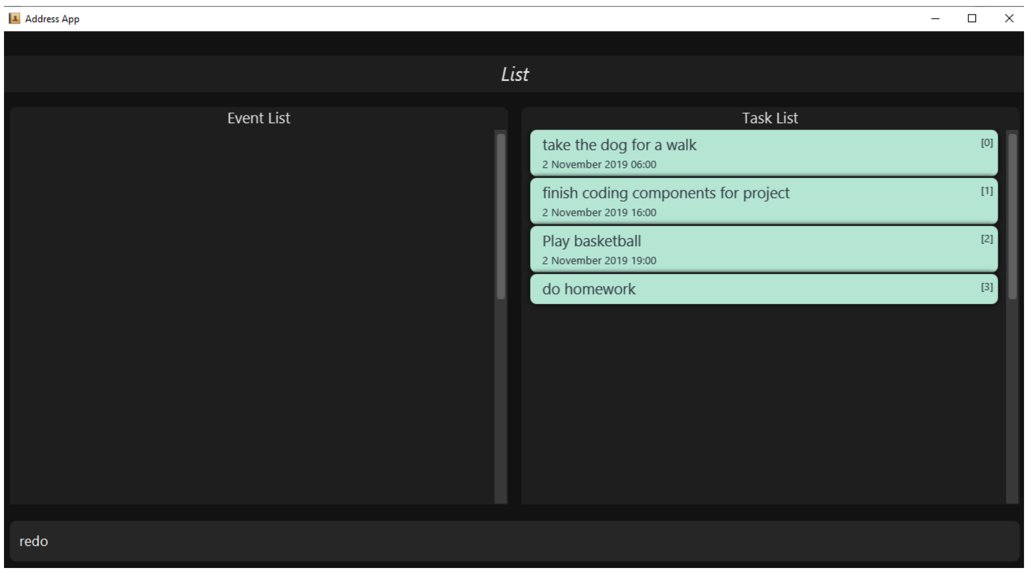
This command re-executes the most recent command that was undone.

Example:
Suppose that you wrongly deleted task 3 from the list, but actually wanted to delete task 2 instead. As a result, you type in the undo command, and Horo is restored to the previous state where task 3 still exists.

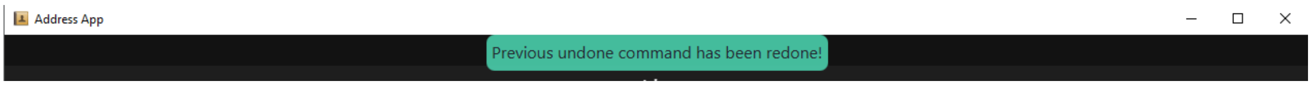
However, now you decide that you want to remove task 3 from the list after all. Without having to key in the delete_task command, you can simply type in redo and the most recent command that was undone (the deletion of task 3) will be re-executed. This results in a list where task 3 is deleted.

To redo:

1. Type redo in the command box, and press the Enter key to execute it.

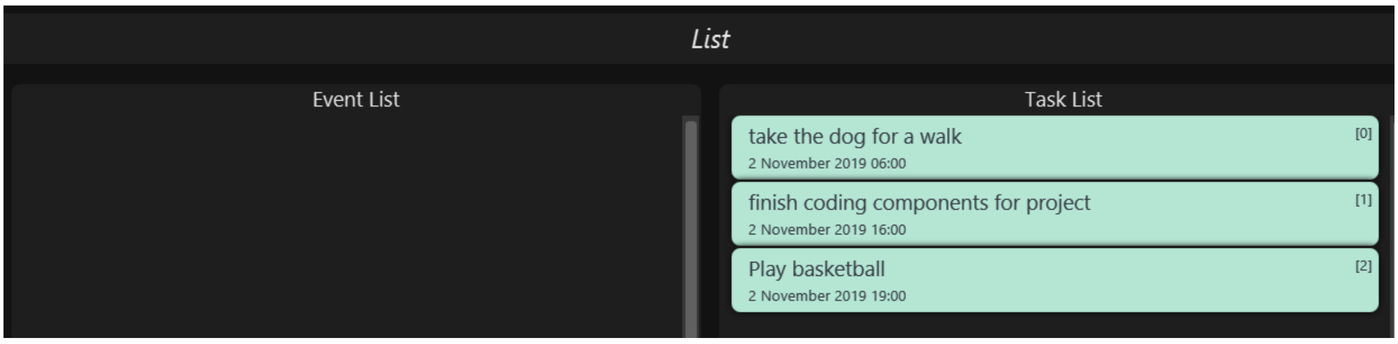


2. The result box will display the message “Previous undone command has been redone!”



Previous undone command has been redone!

3. You can see that task 3 has been removed from the list.



The redo command redoes previously undone commands in reverse chronological order.
Suppose that you have executed the following commands in this order:

1. Adding a task
2. Editing a task

As discussed in **Undoing a previous command**, if we run undo twice, we will revert Horo to before a task was edited, and then revert Horo to before a task was added.
Our sequence of undo commands is in this order:

1. Undo editing of a task
2. Undo adding of a task

Now, if we run the redo command, Horo will be restored to the state after the task was added. If we execute redo again, Horo will be restored to the state after the task was edited.

The redo command only executes if the most recent state-changing command(s) are undo commands.
If no undo command has been executed since the starting up of Horo, or undo commands have been executed but other state-changing commands were executed after those undos, then executing the redo command amounts to no effect.
For example, let's say I deleted a task from the list, undid that deletion, and then added another task to the list. Executing the redo command here will not do anything because add_task was executed after the undo.

Contributions to the Developer Guide

The following section shows my contributions to the developer guide in relation to the undo and redo features.

Undo/Redo feature

The undo/redo mechanism is facilitated by UndoRedoManager, which contains undoStateList - a history of ModelData states.

Each ModelData object contains two lists: one to store EventSources and the other to store TaskSources, together representing the state of all event and task data at that point in time. UndoRedoManager also contains an undoIndex, which keeps track of the index of the model data being used presently, as well as a ModelManager object.

ModelManager contains a ModelData object.
Horo's StorageManager, UiManager and UndoRedoManager components implement the ModelDataListener interface which listens for any changes to this model data so that they can be updated accordingly. Every time a state-changing command (that is not undo or redo) is executed, a new model data representing the modified version will

replace the old one and this new version will then be deep-copied and added to undoStateList. Should there be a need to revert back to a past or future state (if undo or redo is called), ModelManager#ModelData will retrieve their data from the appropriate copy of ModelData in the list of duplicates.

UndoRedoManager also implements the following operations:

- UndoRedoManager#undo() — Restore ModelManager#ModelData to their previous versions from the appropriate duplicate in undoStateList
- UndoRedoManager#redo() — Restore ModelManager#ModelData to their future versions from the appropriate duplicate in undoStateList
- UndoRedoManager#clearFutureHistory() -- Delete all ModelData states that occur in undoStateList after the index given by the undoIndex

The undo and redo commands will interact directly with UndoRedoManager while other state-changing commands (such as adding or deleting tasks) will interact only with ModelManager.

The ModelDataListener interface helps us achieve the desired undo-redo functionality:

This listener interface contains a single method, onModelDataChange(ModelData modelData).

The UndoRedoManager implements the ModelDataListener interface's method onModelDataChange(ModelData modelData) to "listen" for any changes to ModelManager#ModelData (e.g. when an event or task is added or deleted)

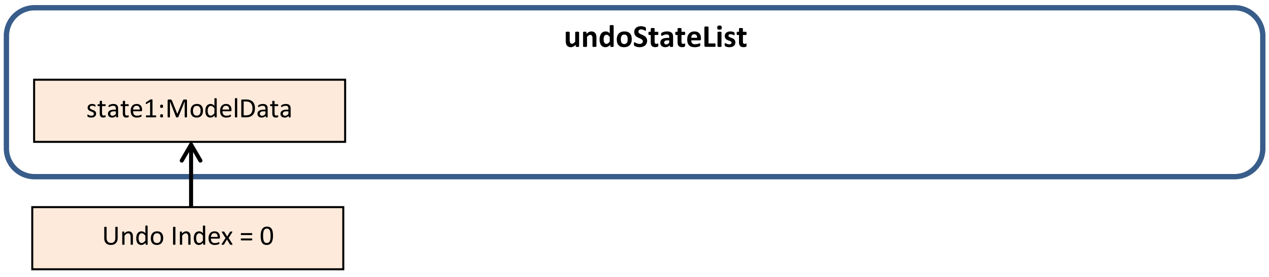
If such a change exists, it will be handled by first instantiating a ModelData with a deep-copied version of the taskList and the modified eventList, calling UndoRedoManager#clearFutureHistory(), and finally, committing the state to undoStateList.

On the other hand, whenever an undo or redo is executed, ModelManager#modelData is updated to match the data of the model data with index undoIndex in undoStateList so that the correct version of model data is being reflected in the GUI.

Given below is an example usage scenario depicting the behaviour of the undo/redo mechanism:

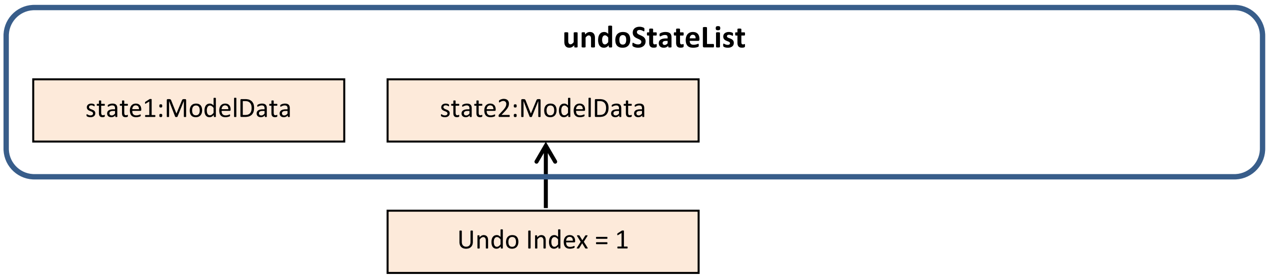
Step 1. The user runs the program for the first time. The UndoRedoManager will be initialized with the initial undoStateList. A model data object will be added to undoStateList and the undoIndex will point to that single model data in the list.

Initial state



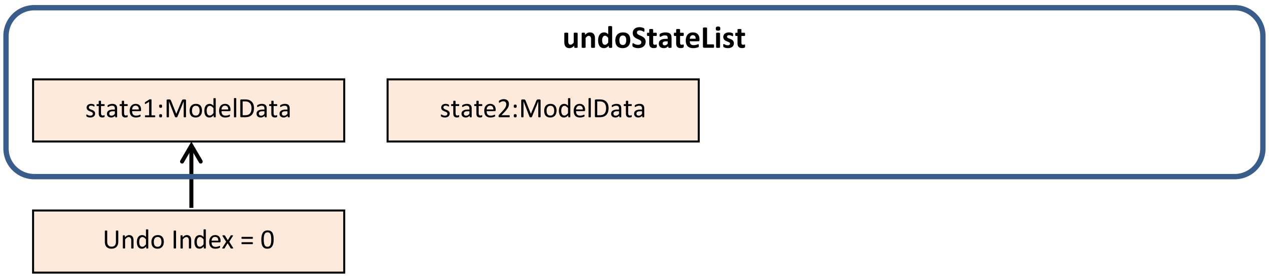
Step 2. The user executes add_event “Suntec City Computer Fair” --at “17/11/2019 12:00”. ModelManager#ModelData will be reset to a new model data object with the added event. Then, UndoRedoManager#onModelDataChange(ModelData modelData) will be called (as there has been a change to the model data), deep-copying the modified model data. All future states beyond the undoIndex will be cleared as they are no longer useful. In this particular case, there are no future states to be cleared. Finally, the deep-copy of the new model data state will be committed; added to undoStateList. The undoIndex is incremented by one to contain the index of the newly inserted model data state.

After command “add_event...”

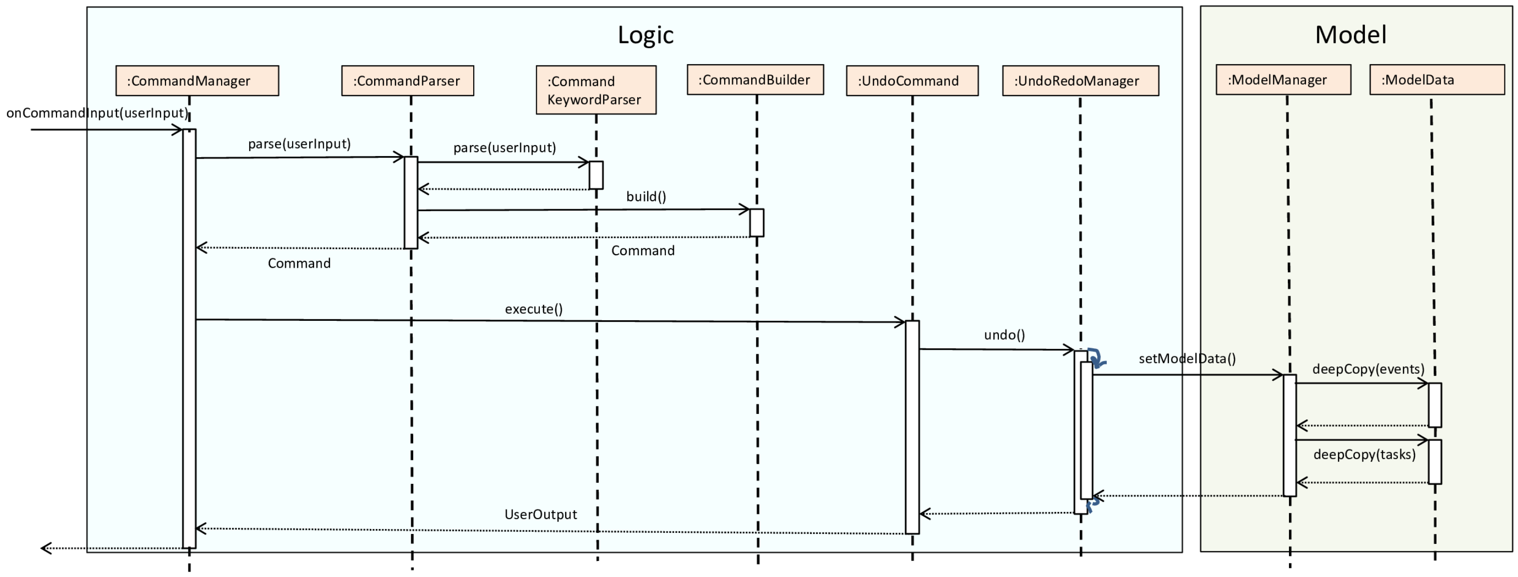


Step 3. Suppose the user decides that adding the task was a mistake. He/she then executes the undo command to rectify the error. The undo command will decrement the undoIndex by one to contain the index of the previous undo redo state, thereafter triggering the UndoRedoManager#notifyModelResetListeners method. This method updates ModelManager#ModelData to match the data of the model data with index undoIndex in undoStateList.

After command “undo”



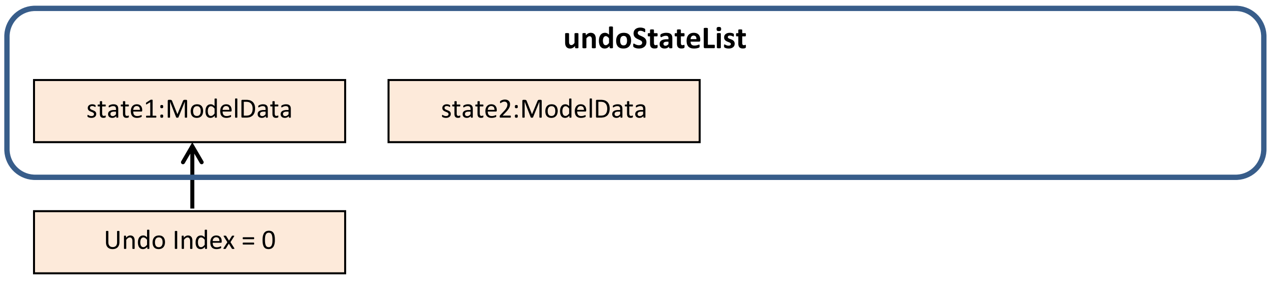
The following sequence diagram shows how the Undo operation works.



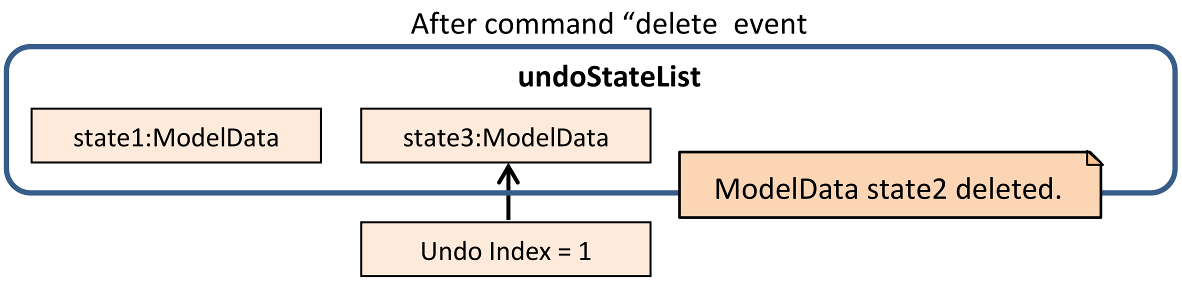
The redo command does the opposite — it calls UndoRedoManager#redo(), which increments the undoIndex by one to contain the index of the previously undone state. The UndoRedoManager#notifyModelResetListeners method then causes ModelManager#ModelData to be reset to this state's data.

Step 4. The user decides to execute the command log. Non-state-changing commands such as log do not manipulate task and event data. Since no changes to model data have been made, the listener method will not be triggered and no model data will be saved to undoStateList. Thus, the undoStateList remains unchanged.

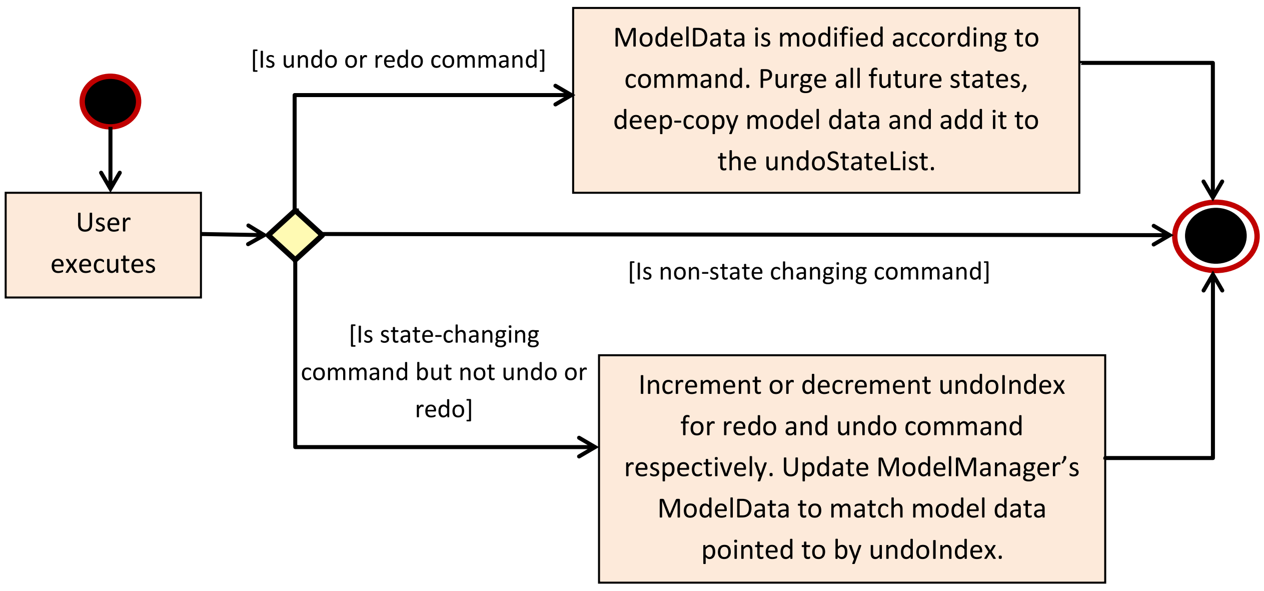
After command "log"



Step 5. The user executes delete_event 1, removing the event from the eventList in ModelManager#ModelData. UndoRedoManager#onModelDataChange(ModelData modelData) will be called (as there has been a change to the model data), purging all future states beyond the undoIndex as they are no longer useful. The modified model data will be deep-copied and a new model data containing the deep-copies will also be added to undoStateList. The undoIndex is incremented by one to contain the index of the newly inserted model data state.



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


User executes
[Is undo or redo command]
ModelData is modified according to command. Purge all future states, deep-copy model data and add it to the undoStateList.
[Is non-state changing command]
[Is state-changing command but not undo or redo]
Increment or decrement undoIndex for redo and undo command respectively. Update ModelManager’s ModelData to match model data pointed to by undoIndex.

Design Considerations

Aspect: How undo and redo executes		
	Alternative 1 **(current choice)**	Alternative 2
	Saves the ModelData state every time a change has been made	Individual command knows how to undo/redo by itself; inverse functions have to be implemented
Pros	Easy to understand and implement	Uses less memory as we only need to keep track of what commands have been executed and their parameters, as opposed to storing all task and event data between every change.
Cons	Performance issues may arise due to relatively larger memory usage required.	Every command will have to be implemented twice, since their inverse operations will all be different. This is compounded by the fact that we have to ensure the correctness of every inverse operation individually as well.