Ryo Chandra Putra Armanda - Project Portfolio

PROJECT: MooLah

Overview

MooLah is a Command Line Interface (CLI)-centered expense tracker poised to be hassle-free and tailored for students of National University of Singapore (NUS). This project is developed by me and 4 Computer Science students in NUS. In developing the project, we were initially tasked to enhance or morph an existing project, Address Book Level 3. Throughout the semester, we decided to morph the original codebase into MooLah and developed features for it incrementally. Although MooLah is CLI-centered, it also has a neat Graphical User Interface (GUI) made by JavaFX code, and supports features such as autosuggestion to achieve the intended convenience of using MooLah.

Summary of contributions

Major enhancement

I implemented the functionality to **undo and redo commands** that have been executed by the user. The details are as follows:

- What it does: The functionality enables the user to undo commands that have been executed one-by-one. Commands that have been undone can be reverted back by using the redo command, provided that the user has not executed any new commands at the time.
- Justification: This component improves the application significantly as a user can make mistakes in commands and the component helps the user amend them.
- Highlights: This enhancement affects existing commands and commands to be added in future, as commands can now be either undoable or non-undoable in nature. It required a thoughtful analysis of design approaches and resource trade-offs, namely in computation time versus memory. The implementation too was challenging as it hinges on the immutability of the data structures of the model, and persistence of model states throughout execution.

Minor enhancements

Other than the major enhancement described above, I have also:

- Added a new command that adds an expense from the list of NUS canteens' food and drinks supported by MooLah. This command is a shorthand form of the general add expense command tailored for NUS students.

Code contributed

Refer here for the list of code contributions I have done for the project.

Other contributions

Project management

As the person-in-charge for project integration, my responsibility is to streamline the process of proposing code changes and contributions, and to preserve the integrity of the codebase. The measures I have taken to achieve that, are namely:

- Ensured the master branch is protected from rogue code pushes
- Made passing Travis build a requirement for merging pull requests to ensure the integrity of the master branch.
- Configured Coveralls to post coverage status to pull requests after passing Travis build with a warning if coverage drops by 5%.
- Set the requirement for one teammate approval to be able to merge pull requests to improve codebase quality.

Improvements to existing components

I improved some aspects of the original codebase to better complement my enhancements and to improve the quality of the project as a whole. For the improvements, I have:

- Separated command validation from execution #85

Community

As part of a project community with my teammates, I have done the following as contributions to the group discussion:

- Reviewing pull requests (with in-depth comments): #62, #89

Integration Tools

I have incorporated third-party tools to help automate or manage the development of the project. In summary, I have:

- Integrated Coveralls to the team repo to keep track of the project's line coverage.
- Integrated Trello to manage project issues and backlog throughout the development process.

Contributions to the User Guide

*Below are the **major** sections that I contributed to the User Guide. These showcase my ability to write documentation targeting end-users.*

Undo the previous command : `undo`

Did something wrong? You can undo it.

However, do keep in mind that you can only undo commands that modify the data in MooLah, such as `addexpense`, `deleteexpense-primary`, and so on. Head over to "Command Summary" section for list of commands that are undoable.

Do not worry if you forgot what you did a few steps back (it happens!). MooLah will display a short description of what particular command it undid to help you confirm that you undid the right thing.

Format: `undo`

Example:

Let's say you just added a new expense...

```
addexpense d/bowling p/10 c/entertainment
```

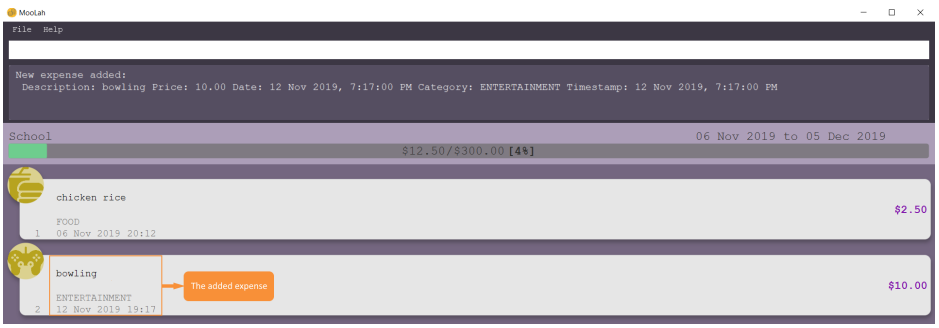


Figure 1. The state of MooLah after adding the "bowling" expense

But you added it to the wrong budget! You wish to undo the mistake now...

```
undo
```

And MooLah will undo it for you and return the data to the way it was before. Also, it will display `Undid "Add expense bowling (10.00)"` in the result display for reference. Phew!

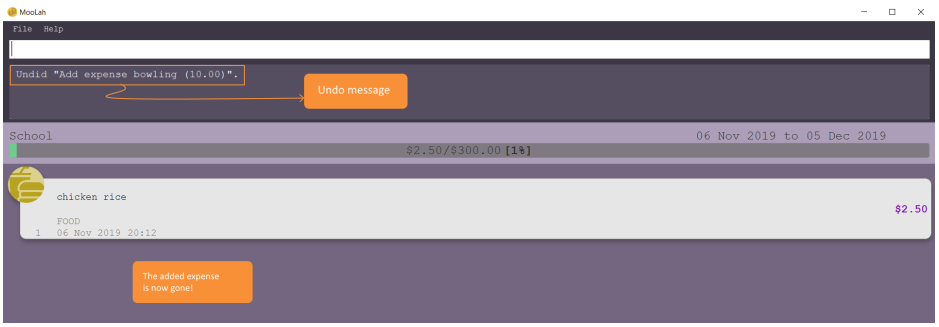


Figure 2. The state of MooLah after undoing the expense addition

Redo an undone command : `redo`

You might hit one undo too many and wish to cancel that one undo. That's okay, redo will help you on that.

Similar to undo, MooLah will display a short description of the command it redid to you.

Format: `redo`

Example: Continuing the example in `undo`... now you think that the "bowling" expense is in the right place after all, and you wish to do the addition again.

```
redo
```

Result: At the end, the "bowling" expense will be re-added back to MooLah. Yay! MooLah will also display a message `Redid "Add expense bowling (10.00)"` for reference.

Adding an expense from NUS canteens' menu : `addmenuexpense`

This is what makes MooLah special for NUS students. Shortcut your way to adding expenses for foods and drinks in NUS!

Currently, MooLah only supports just a handful of menu items, though. Head over to our "Menu Items List" section to see the supported menu items. A more comprehensive menu list is on its way in v2.0!

Format:

```
addmenuexpense m/<MENU_ITEM> [t/TIMESTAMP]
```

Example:

```
addmenuexpense m/deck chicken rice
```

This will add a new expense that corresponds to The Deck's Chicken Rice and MooLah will automatically fill in the description, price, and category for you.

Contributions to the Developer Guide

*Below are the **major** sections that I contributed to the Developer Guide. These showcase my ability to write technical documentation and the technical depth of my contributions to the project.*

Undo and Redo feature

Implementation

The undo and redo functionality is facilitated by `ModelHistory`, which is available as an instance-level member of `Model`. It keeps track of the model's history by storing the changed fields of the model throughout execution, which will be represented as `ModelChanges`. Internally, `ModelHistory` stores the history by using two stacks of `ModelChanges`, namely, `pastChanges` and `futureChanges`.

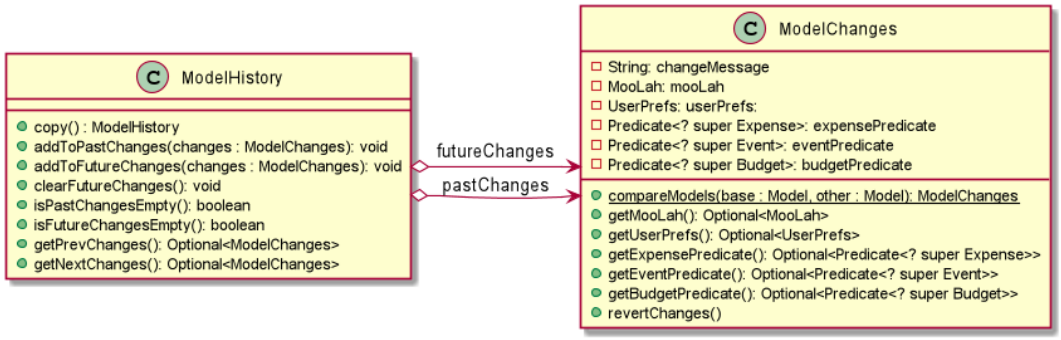
With the addition of model history, `Model` can support these operations:

- `Model#applyChanges(ModelChanges)` — Applies the given changes to the model.
- `Model#commit(String, Model)` — Saves the changes between the current model and the given previous model to the past changes history, keeping the previous model's data should there be any differences. This adds the changes to the past history, and clears the future history.
- `Model#rollback()` — Restores the model one step back by applying the changes described in the immediate previous changes in history.
- `Model#migrate()` — Moves the model one step forward in history by applying the changes described in the immediate next changes.

To support these capabilities, `ModelChanges` offers these methods as well:

- `ModelChanges#compareModels(Model, Model)` — Compares two models' data and creates a new `ModelChanges` object that describes the field data of the first model that is different with the second model.
- `ModelChanges#revertChanges(Model)` — Reverts the current changes with respect to the base model given.

Refer to the class diagram below for comprehensive list of the methods offered and the association of both classes:



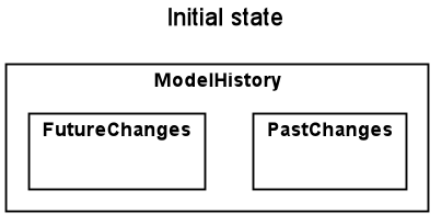
NOTE Typical field accessor and mutator methods are omitted for brevity, except when it returns a different type from the field's type (e.g. `ModelChanges#getMooLah()`).

`ModelHistory` only stores changes of models that were (or are going to be) executed by model-manipulating - or simply, undoable - commands. As some of the commands available are intuitively not undoable (e.g. `help`), every command is configured to extend either `UndoableCommand` or a non-undoable `Command` classes. With the division, `Model#commit(String, Model)` will only be called if the command to be executed is an instance of `UndoableCommand`.

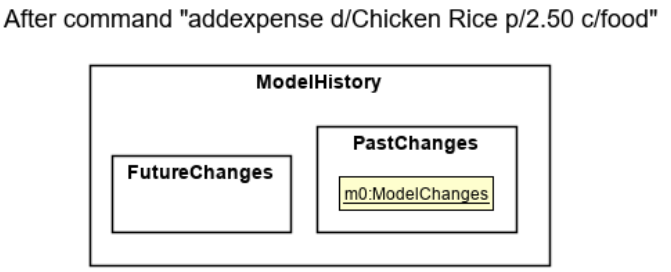
Given below is an example usage scenario and how the undo and redo functionality behaves at

each step.

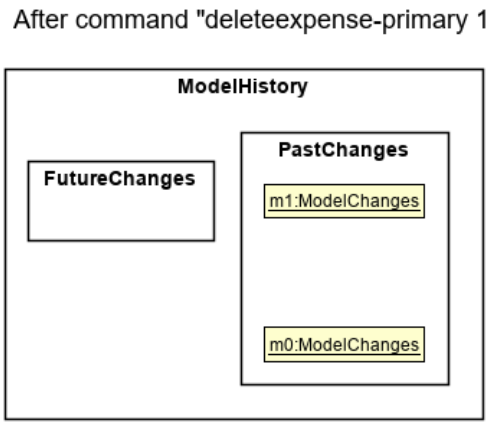
Step 1. The user launches the application for the first time. The current `ModelHistory` is now empty.



Step 2. The user executes `addexpense d/chicken rice p/2.50 c/food` command to add an expense. The `addexpense` command, being an `UndoableCommand`, calls `Model#commit(String, Model)`, which saves the state of the model just before the command executes to `pastChanges`, and `futureChanges` is cleared.

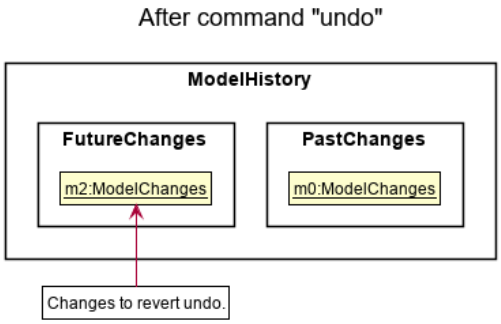


Step 3. The user executes `deleteexpense-primary 1` to delete the first expense on the list. The `deleteexpense-primary` command, also an `UndoableCommand`, calls `Model#commit(String, Model)`, inserting another entry to the `pastChanges` and clearing `futureChanges` again.



NOTE If a command fails its execution, it will not call `Model#commit(String, Model)`, so the model will not be saved into `ModelHistory`.

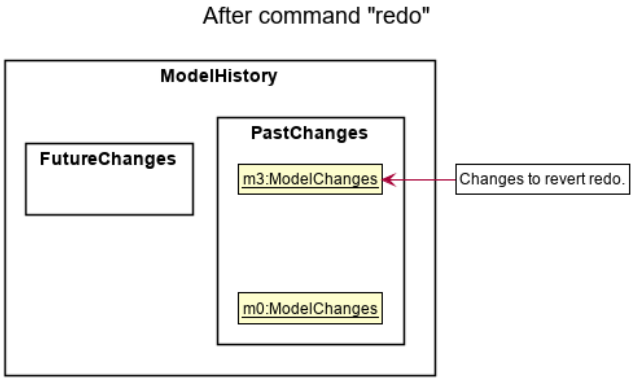
Step 4. The user now decides that deleting the expense was a mistake, and decides to undo that action by executing the `undo` command. The `undo` command will call `Model#rollback()`, which will retrieve the immediate previous change in history, adding the reverting change to the future history of the model, and applies the change to the model.



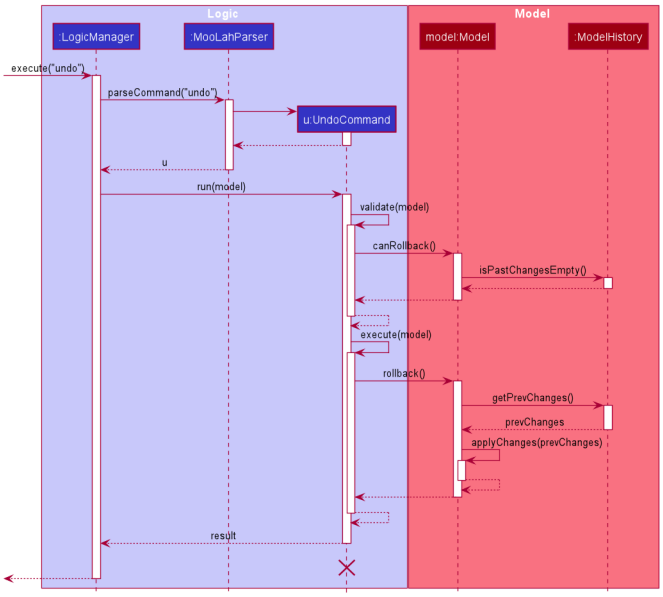
NOTE

If `pastChanges` is empty, then there are no previous changes to roll back. The `undo` command uses `Model#canRollback()` to check if this is the case. If so, it will return an error to the user rather than attempting to perform the undo.

Step 5. The user then realizes the expense should be deleted after all, and wishes to redo the deletion by entering the `redo` command. The command will call `Model#migrate()`, which will get the immediate next change in history, adding the reverting change to the past history, and applies the change to the model.



The following sequence diagram shows how the undo command works:



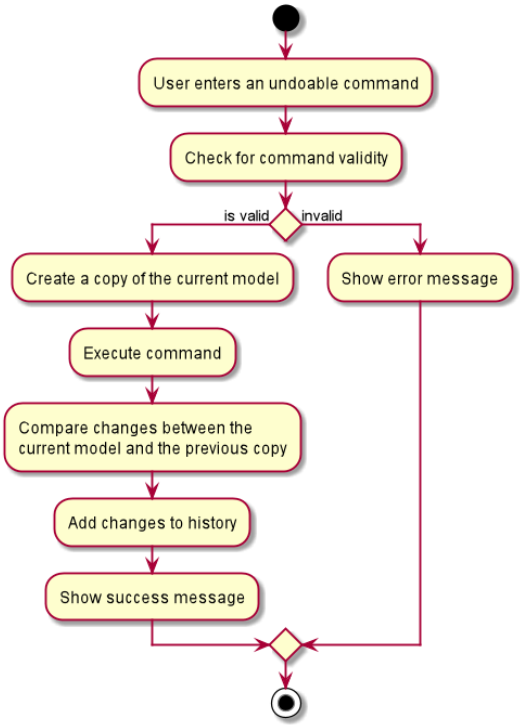
NOTE

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.

Inversely, the `redo` command calls `Model#migrate()`, which retrieves the immediate next changes in history, adds the reverting change to the past history, and applies the changes to the model.

NOTE	If `futureChanges` is empty, then there are no snapshots to be redone. The `redo` command uses `Model#canMigrate()` to check if this is the case. If so, it will return an error to the user rather than attempting to perform the redo.

As explained earlier, additions of entry to `ModelHistory` will only be performed when the command being executed is an instance of `UndoableCommand`. The following diagram briefly describes how the execution of undoable commands will do so:



Design Considerations

In implementing the undo and redo functionality, some design approaches and trade-offs have to be considered to account for feature efficiency in computation time and memory usage.

Aspect: How undo and redo executes

- **Alternative 1:** Saves the model data.
  - Pros: It is easier to implement.
  - Cons: The approach introduces a component that will take up memory.
- **Alternative 2:** Individual commands have their own counter-command that negate the effects.
  - Pros: This approach is quite intuitive (e.g. for `addmenuexpense`, we can do `deleteexpense-primary` to counter it).
  - Cons: It has to be ensured that the implementation of each individual command is correct.
- **Current implementation (Alternative 1)**: We choose this approach as it will be less likely to cause problems specific to restoring the state precisely to the state before the execution of a

command, as some commands will create a problem specific on their own which might not be supported by the proposed counter-command. As an example, to undo `delete 3`, we must re-add that expense to that specific position (third from beginning), which at the moment is not supported by the `addexpense` command.

Aspect: How to store the model data

- **Alternative 1:** Saves the entire model.
  - Pros: It is easier to implement.
  - Cons: The approach introduces a lot of memory usage and some data might not be necessary.
- **Alternative 2:** Saves the members of the model that were (or are going to be) changed.
  - Pros: This approach is more conservative in memory usage, only saving fields that are changed means every data is necessary.
  - Cons: Every field must be immutable to preserve changes, which requires a rework on accessing and manipulating fields.
- **Current implementation (Alternative 2)**: We choose this approach as the time and memory resource trade-off is less significant compared to Alternative 1. As Alternative 2 stores only the data needed to correctly reflect the change, it requires less memory and also executes in less time as Alternative 1.