Marc Fong Yung Kit — Project Portfolio for Jarvis

About Jarvis

Jarvis was developed by the se-edu team. We are a team based in the School of Computing, National University of Singapore.

My team was tasked with contributing to a Command Line Interface (CLI) desktop address book application for our Software Engineering project. We decided morph the application into a productivity tool for students called Jarvis.

Jarvis enables students to track their calendar, CCAs, finances and modules. The application will display all these information in their respective tabs to allow students to manage all these aspects of their life in this single application.
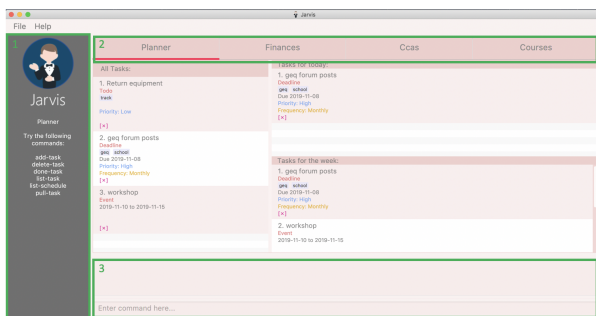
This is a layout of how our application looks like:

Figure 1. Application Page Layout

In Figure 1, there are three boxes labelled with numbers. These are the shared GUI sections across all the features:

1. Help Desk — Unique to each feature's tab, it will display a list of commands you can use so you do not need to remember them all!

2. Feature Tabs — The four main features of Jarvis have been split into distinct tabs. The tab that you are currently on would be underlined in red, as seen in Figure 1.

3. Command Result (top) & Command Entry (bottom) Boxes — Keying commands into the Command Entry Box will cause the respective Command Results to show in the Command Result Box. An invalid command will result in a red border around the Command Entry Box and an appropriate error message will be displayed in the Command Result Box.

My role

My role was to implement the undo and redo feature of Jarvis and implement storage components to save the information stored in Jarvis to local storage.

Summary of Contributions

- Features added:
  - Undo and redo feature. #7 #9 #10 #18 #19 #20 #21 #96 #211 #334 #363
- Project Management
  - Managed development progress with the team for the application for the several releases, from version 1.1 to 1.5, that are available on Github. List of versions
- Enhancement to existing features
  - Updated the Storage component to extend to all new features. #98 #102 #127 #145 #146 #168 #170 #174 #182 #185 #186 #190 #191 #190 #200 #190 #332 #345
- Documentation
  - Documented how `Model` component works, which handles interactions with Jarvis' data is managed. #369
  - Documented how `Storage` component works, which handles writing and reading Jarvis' data from local storage. #342
  - Documented design considerations faced when implementing the undo and redo feature.
  - Documented implementation details on undo and redo feature. #150 #153 #155 #160 #161
- Community
  - Reviewed other PRs, with comments on improving code quality, defensive coding, testing, code design and code practices. Here are the list of reviews.
- Code contributed
  - This is a link to my contributions displayed on RepoSense.
  - Here is a list of all my pull requests.

Undo and redo feature

The following sections covers brief versions from the **Developer Guide** and **User Guide** explaining the following points:

- Design Considerations (Developer Guide)
- Feature Details (Developer Guide)
- Implementation (Developer Guide)
- Commands (User Guide)
- Use case for `undo`, `redo` and `list-history` (User Guide)

Design Considerations

There were several available behavioral design patterns that we were considering to adopt to implement our undo/redo feature in the application.

- Command Pattern
- Memento Pattern

These patterns are common useful patterns to enable undo/redo functions. These are also viable options as our application design allows both of these approaches to be integrated easily.

Command Pattern

The application already makes use of the command pattern to decouple the internal state of the application and the user action.

Characteristics

- Space efficient due to storing commands instead of states of the entire application. Efficient usage of RAM and local storage for the application.

- Commands logic will be more complex as they must know how to undo its execution. Commands contain more data to retain information needed to undo its execution.

- Complex inverse executions may be unnecessarily convoluted compared other approaches (such as the memento pattern). This involves more planning and support on the classes that commands execute on.

- Requires implementation and testing of each command (and future commands) to enable undo/redo function with respect to that command. Development of the application will involve more overhead when integrating new commands to the application as there are more behaviour to test.

- Development of `HistoryManager` scales along with commands that are added to the application. Even after `HistoryManager` is developed and integrated into the application, additional work is required with each command, such as supporting inverse execution and serializing the command (for local storage). This can affect development schedule and add time constraints when working with tight deadlines.

Memento Pattern

The application follows a structural facade pattern, storing the data in ModelManager which implements the Model, which is an interface for commands to interact with. ModelManager manages classes that wrap their respective data.

Characteristics

- Expensive on space due to storing multiple copies of the application state. This increases the usage of RAM and local storage for the application.

- Simple robust implementation that can be developed quickly, which can be useful for tight

schedules in the development process.

- Protects the encapsulation of private data of the application state (provided local storage data is also encrypted). This prevents violation of encapsulation of classes.

- Development of `HistoryManager` scales with how the information to be remembered changes. Whenever the nature of the information to be remembered changes, the memento class `Version` needs to be updated along with how `VersionedModel` updates and saves its state. Adding new commands also do not require any changed to `HistoryManager` unless there are changes to the data fields to be saved in `Model`.

Our Thoughts

These are the following questions we asked ourselves when deciding between these two approaches

- RAM and storage

- Development process

- Software design principles

Design Choice

We determined that going along with the command pattern. We want to cater this application to students whose laptops may not have generous amounts of RAM. On top of the fact that students typically use their browsers *extensively*, we felt that we should be mindful of RAM usage.

Comparisons between command pattern and memento pattern with respect to the development process were trivial since the `Model` and command sets are already planned, and future changes to `Model` and commands would not have any serious drawback regardless of the approach.

Feature Details

The application should be able to undo and redo changes made by commands to give the user more flexibility in their inputs. Undo and redo operations should also be undo or redo multiple commands in a command. In the event that a undo/redo command that comprises of multiple undo/redo operations fails at any point, all changes made by the command should be rolled back. This is reflected in the Activity Diagrams below.

Therefore there is a need to remember commands that change the state of the `Model`. Commands that just render a view without actually changing the application should not be stored as it does not make sense to undo or redo them. We will distinguish these types of commands into two categories, **invertible commands** and **non-invertible commands**.

- **Invertible commands** — commands that mutate the state of the `Model` and should be stored for undo/redo functions.

- **Non-invertible commands** — commands that do not mutate the state of the `Model` and should not be stored for undo/redo functions.

NOTE	Undo and redo commands will be considered non-invertible commands even though they technically change the state of the `Model`. The reason is that they are commands facilitating the undo and redo operation, thus they should not be stored.

The following activity diagram illustrates how commands are remembered when a user types in a command:
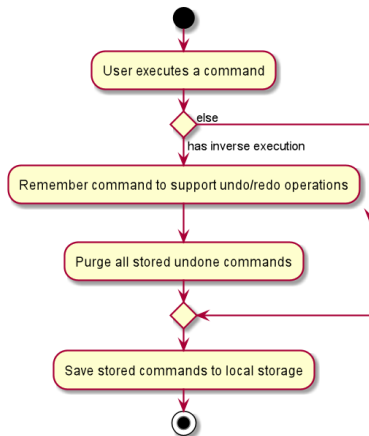

Figure 2. Activity Diagram for how commands are remembered after their successful execution

Implementation

The undo/redo feature mechanism is facilitated by `HistoryManager`. `HistoryManager` remembers **invertible commands**. These commands are stored internally in two `CommandDeque` objects, `executedCommands` and `inverselyExecutedCommands`. `CommandDeque` serve as custom `Deque` data structure, which stores the latest added command to the top.

An undo operation would comprise of taking the latest executed command from `executedCommands`, inversely executing it, and adding it to `inverselyExecutedCommands`. A redo operation would comprise of a taking the latest inversely executed command from `inverselyExecutedCommands`, executing it, and adding it to `executedCommands`.

`Model` supports operations to facilitate undo and redo capabilities by extending the `HistoryModel`.

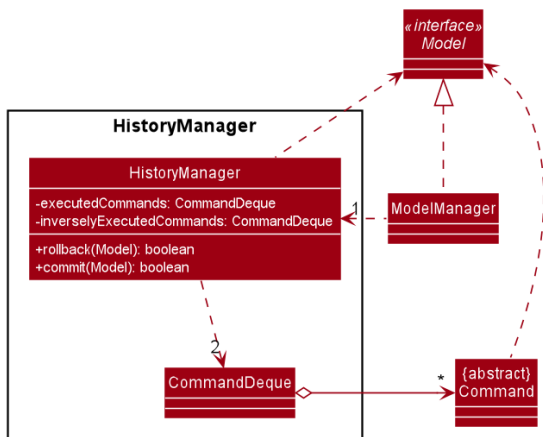Below is a class diagram between `Model`, `ModelManager`, `HistoryManager`, `CommandDeque` and `Command`.

Figure 3. Class Diagram for `Model`, `ModelManager`, `HistoryManager`, `CommandDeque` *and* `Command`

Undo and redo operations are executed with `UndoCommand` and `RedoCommand` These commands store an integer value referencing the number of commands to undo or redo, represented by `UndoCommand#numberOfTimes` and `RedoCommand#numberOfTimes`. The Class Diagram below shows details about `UndoCommand` and `RedoCommand`.
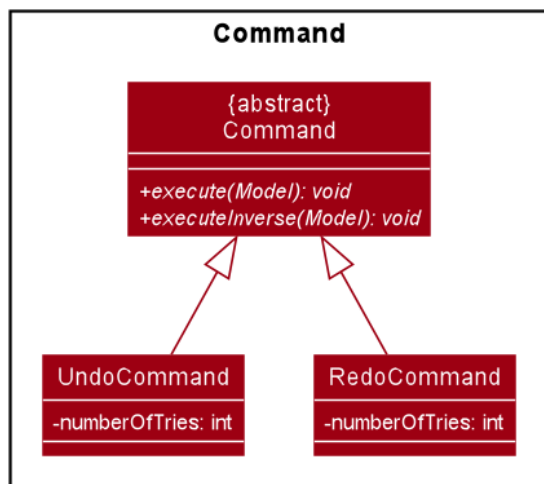

Figure 4. Class Diagram for `UndoCommand`, `RedoCommand` *and* `Command`

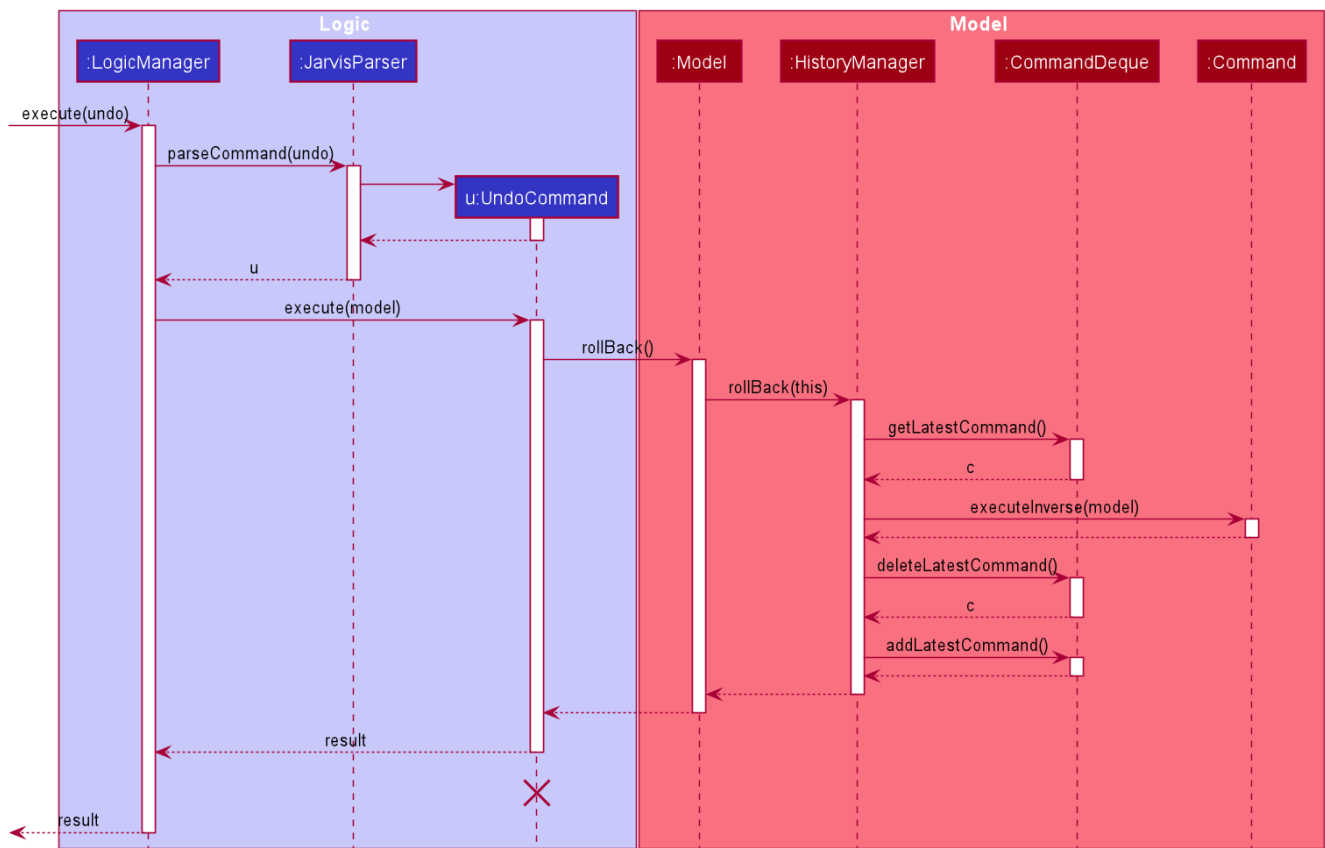Below is a sequence diagram for how an `UndoCommand` executes in the program.

Figure 5. Sequence Diagram for `UndoCommand` *(undo a single command)*

Commands

You can see how to undo and redo commands in JARVIS in the following sections below, which will cover two new commands:

- `undo [NUMBER]`
- `redo [NUMBER]`
- `list-history`

NOTE

`[NUMBER]` is an optional argument indicating the number of commands that you want to undo/redo.

`[NUMBER]` should be a positive integer that is not larger than the available number of commands that can be undone/redone (Don't worry, JARVIS will let you know if `[NUMBER]` is out of range).

If `[NUMBER]` parameter is not supplied, then `[NUMBER]` is taken to be the value **1**.

Undo command(s): `undo`

Format: `undo [NUMBER]`

where `[NUMBER]` is an optional argument indicating the number of commands that you want to `undo`.

NOTE	[NUMBER] should be a positive integer, no larger than the available number of commands that can be undone. (Don't worry, JARVIS will let you know if [NUMBER] is out of range). If [NUMBER] is not supplied, then it is taken to be the default value of **1**.

Example

- undo — This command will undo the latest command entered by the user. It is equivalent to the command undo 1
- undo 5 — This command will undo the 5 latest commands entered by the user, starting from the latest command.

Redo command(s): redo

Format: redo [NUMBER]

where [NUMBER] is an optional argument indicating the number of commands that you want to redo.

NOTE	[NUMBER] should be a positive integer, no larger than the available number of commands that can be undone. (Don't worry, JARVIS will let you know if [NUMBER] is out of range). If [NUMBER] is not supplied, then it is taken to be the default value of **1**.

Example

redo — This command will redo the latest command undone by the user. It is equivalent to the command redo 1

redo 5 — This command will redo the 5 latest commands undone by the user, starting from the latest command that was undone.

NOTE	If you have entered new commands after undoing previous commands, you will not be able to redo the old commands.

List the number of commands you can undo or redo: list-history

Jarvis can provide you with the maximum number of commands you can undo and redo - which will be shown in the Command Result Box.

Format: list-history

Lists the number of available commands to be undone, number of available commands to be redone, and the limit of the maximum number of commands Jarvis can remember for you to support undo and redo commands.

Use case for `undo`, `redo` and `list-history`

Let's say you would like to delete the first task in the `Planner`, highlighted by the green box in Figure 19 below.
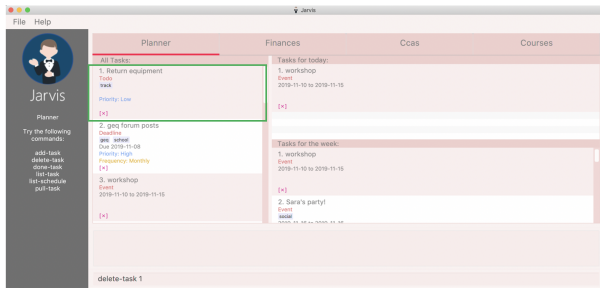

Figure 6. Before deleting the task highlighted in the green box

Deleting the task in Jarvis would then trigger the appropriate command result as seen in the figure below.
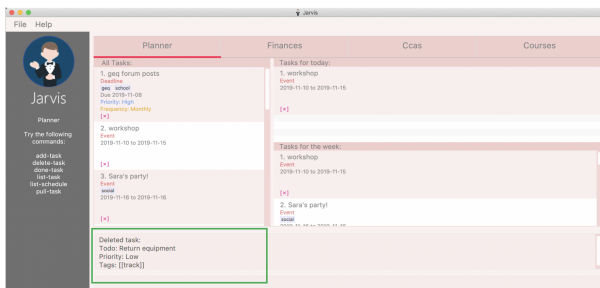

Figure 7. You have just deleted a task as shown in the command result display

As shown in the picture below, entering the command `undo`, would revert the delete command that you had entered earlier.
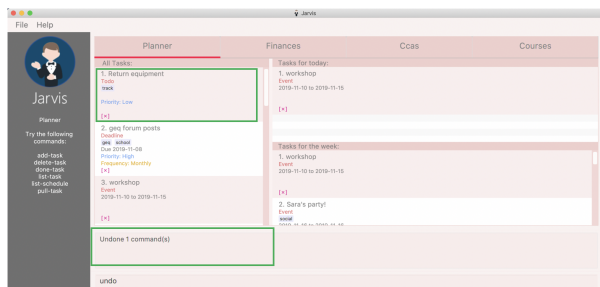

Figure 8. The deleted task is back in Jarvis

You can enter the command `redo`, re-applying the delete command on Jarvis as shown in the picture below.
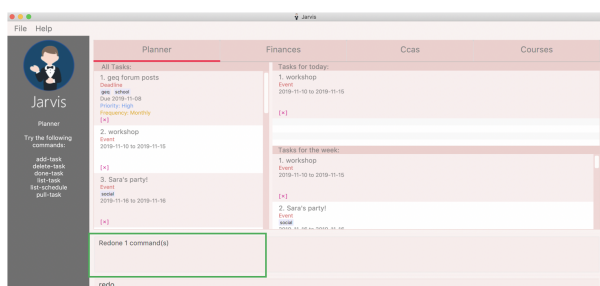

Figure 9. You have just redone that latest undone command

You can enter the command `list-history`, displaying the number of commands that can be undone and redone.
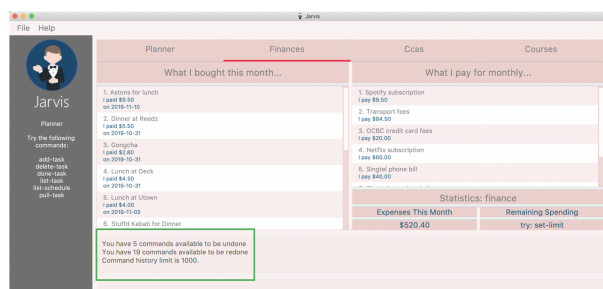

Figure 10. You can see the number of commands that you can undo and redo