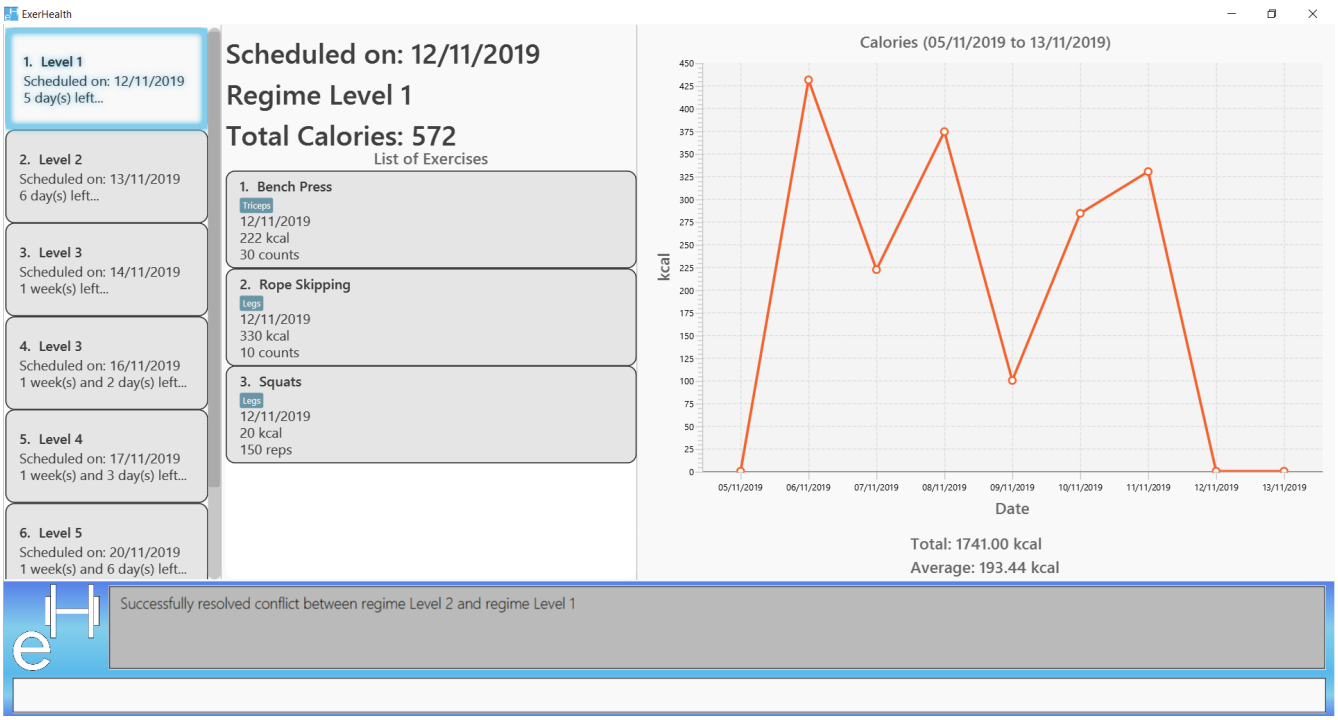
Gary Lim - Project Portfolio

PROJECT: ExerHealth

Overview

Our team modified a basic command line application (AddressBook3) into **ExerHealth**, a desktop application used for tracking and scheduling the user's exercises. The application provides statistical analysis of exercises completed by the users. Additionally, it also acts as a personal trainer by suggesting different exercises which both beginners and advanced users can choose from to incorporate into their exercise regimes. The user interacts with it using a command line interface, and it has a GUI created with JavaFX.

Below is a screenshot of what our desktop application looks like:



Summary of contributions

- **Major enhancement**: added **the ability to undo/redo previous commands**
  - What it does: allows the user to undo previous commands one at a time. Preceding undo commands can be reversed by using the redo command.
  - Justification: This feature improves the product significantly because a user can make mistakes in commands and the app should provide a convenient way to rectify them.
  - Highlights: This enhancement requires an in-depth analysis of design alternatives as well as the interaction between all undoable commands and the backend model. The

implementation too was challenging as it required changes to existing commands while allowing ease of integration with future commands.

- **Minor enhancement 1**: implemented the sorting of exercise list, regime list and schedule list such that entries are listed in a systematic order for users.
- **Minor enhancement 2**: modified list command to improve the navigability across various lists (exercise tracker, exercise suggestions, regime list, schedule list).
- **Code contributed**: RepoSense
- **Other contributions**:
  - Enhancements to existing features:
    - Wrote additional tests for existing features to increase coverage (Pull requests #124, #186)
  - Documentation:
    - Did cosmetic tweaks to contents of the User Guide and Developer Guide: #110, #231
  - Community:
    - PRs reviewed (with non-trivial review comments): #16, #81, #114
    - Reported bugs and suggestions for other teams in the class (examples: 1, 2, 3, 4)

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.

Undo previous command: undo

Undo the previous successful command entered.

Supported Undoable Commands: add, delete, edit, clear, schedule, resolve

- `add t/exercise n/Push ups d/1/10/2019 c/123 q/100 u/reps`
- `add t/regime n/Cardio i/1 i/3 i/5`
- `delete t/exercise i/7`
- `edit t/exercise i/3 n/Push Ups c/140 m/Chest`
- `clear`
- `schedule n/Regime Five d/20/11/2019`
- `schedule i/1`
- `resolve n/New Regime i/1 r/2`

TIP If there is no previous command, undo will do nothing.

Format: `undo`

Redo undone command: `redo`

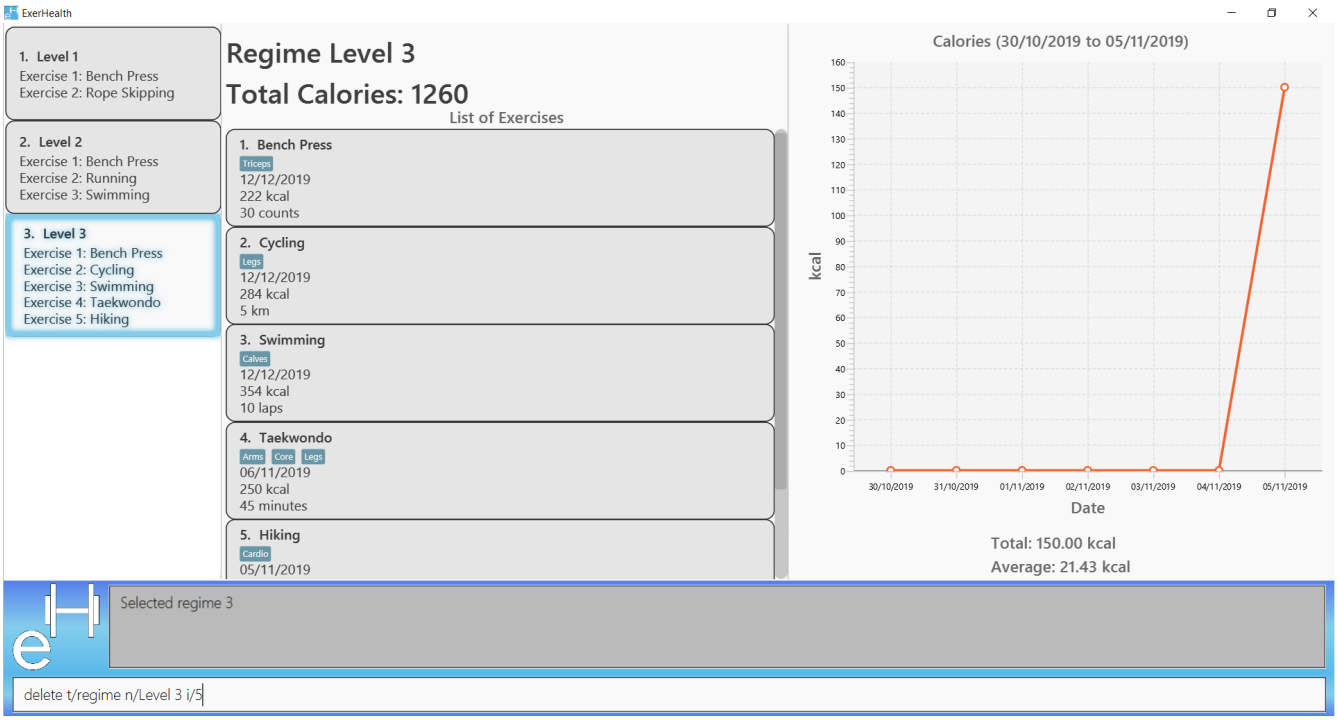
Redo the previous command that was undone by the user. It can only be executed after successful executions of the undo command. For a list of Undoable commands, refer to the Undo Command.

> **TIP** If the `Undo` command has not been executed after the execution of the last Undoable command, there will be no command to redo.

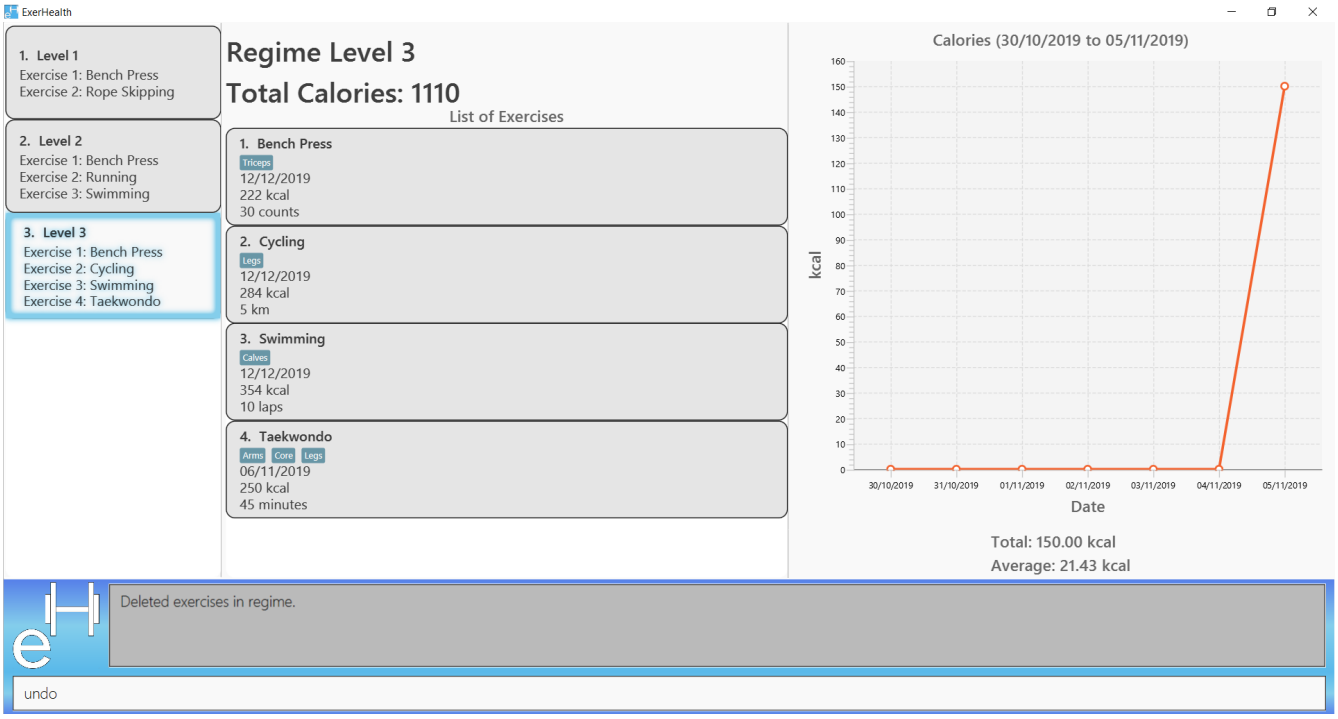
Format: `redo`

Example:

- Undoing a `delete t/regime [i/INDEX]⋯` command that deletes exercise from an existing regime.
  Initial:

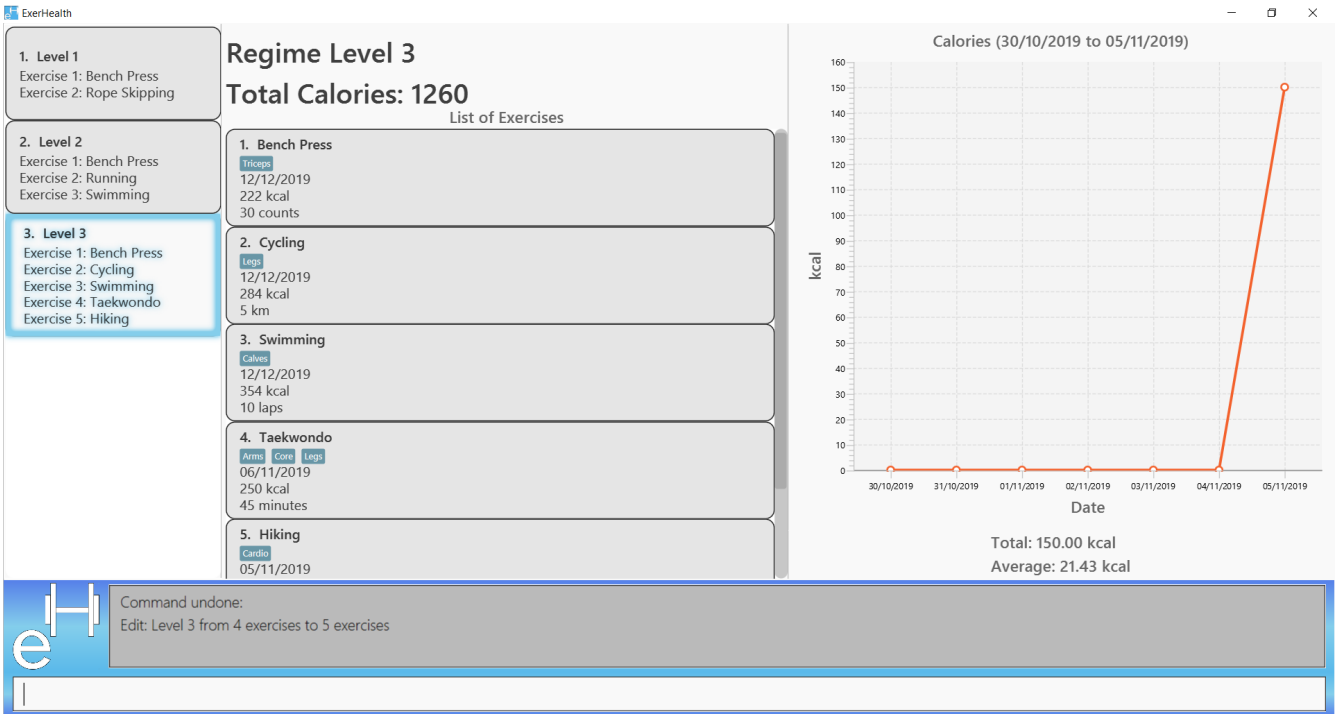


Step 1: `delete t/regime n/Level 3 i/5`
Deletes the fifth exercise `Hiking` from the regime `Level 3`



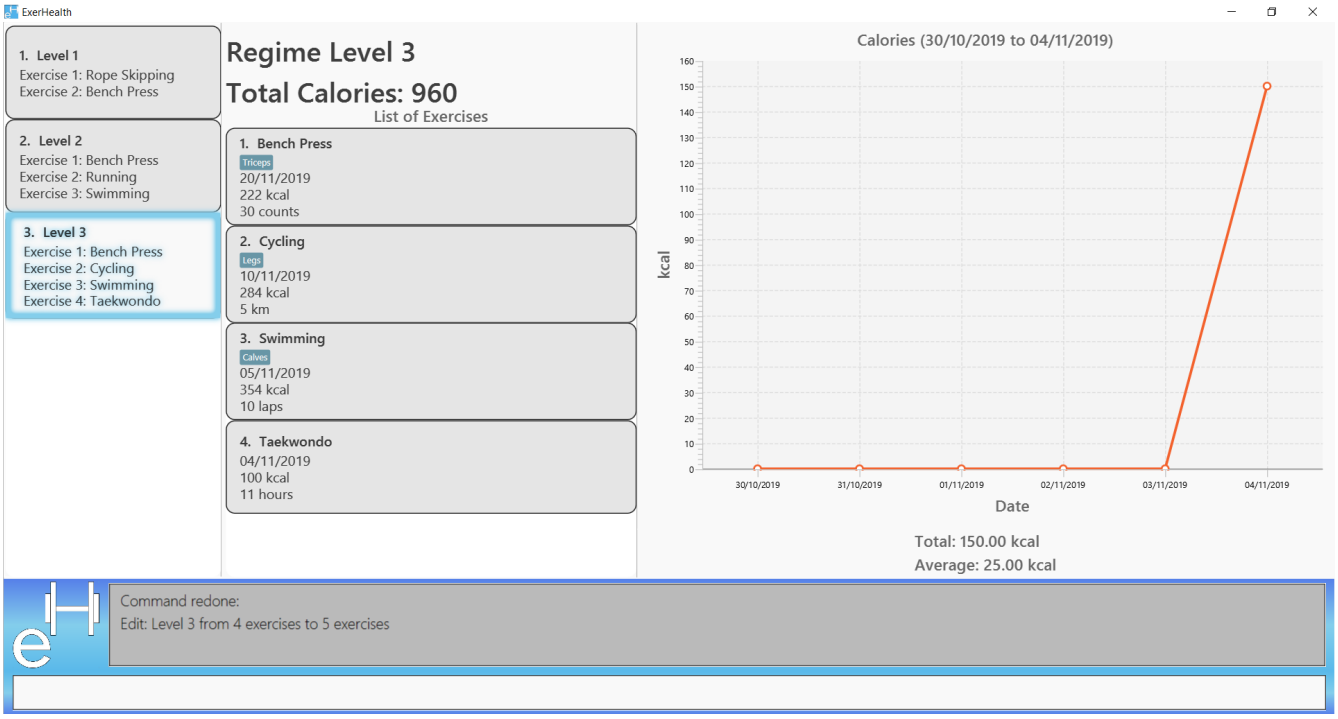
Step 2: `undo`

Adds the exercise `Hiking` back to the regime `Level 3`



Step 3: `redo`

Redoes the deletion of `Hiking` is deleted from the regime `Level 3`



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

Implementation

The undo/redo mechanism is facilitated by the `events` package consisting of `EventHistory`, `EventFactory`, `EventPayload` and the various `Event` classes.

The `EventHistory` is a singleton class used to store a history of successfully executed commands as `Event` objects. Instances of `Event` are stored in either the `undoStack` or the `redoStack` depending on the user's course of action.

The `EventHistory` class has two primary methods namely `undo(Model model)` and `redo(Model model)`:

- `eventHistory.undo(model)` — Undoes the `Event` at the top of the `undoStack`, executes it, and pushes it to the top of the `redoStack`
- `eventHistory.redo(model)` — Redoes the `Event` at the top of the `redoStack`, executes it, and pushes it to the top of the `undoStack`

These operations are utilised in the `UndoCommand` and `RedoCommand` respectively.

The following steps will describe the steps taken in the execution of an UndoableCommand, and subsequently the UndoCommand and RedoCommand.

Step 1: When an `UndoableCommand` is executed, key information used during the command will be added into a newly initialized `EventPayload`.

NOTE	The `EventPayload` is a wrapper class to store key information about the particular command. For instance, if an `EditCommand` has been executed, the `EventPayload` will store the `originalExercise` as well as the `editedExercise`.

Step 2: The `EventFactory` takes in the `UndoableCommand` and generates an `Event` using the `EventPayload` stored in the `UndoableCommand`. The `Event` is then added to the undo stack of the `EventHistory`.

NOTE	The `EventFactory` checks for the command word of the `UndoableCommand` to decide which specific `Event` object to generate. It will then obtain the `EventPayload` from the `UndoableCommand` and pass it into the constructor of the `Event` so that the `Event` captures the key information of the `UndoableCommand`.

Step 3: To undo the latest `UndoableCommand` the user executes the `UndoCommand` by entering `undo` into the command box.

Step 4: The `UndoCommand` executes `eventHistory.undo(model)`, which prompts the `EventHistory` instance to pop the next `Event` to undo from the undo stack. Once the `Event` is undone, it will be pushed to the top of the redo stack.

Step 5: To redo the command that has been undone, the user executes the `RedoCommand`. This execution behaves similarly to step 4, except that the next `Event` is taken from the top of the redo stack and pushed to the undo stack instead.

NOTE	In steps 4 and 5, if any of the respective stack is empty when undo or redo is called, a `CommandException` will be thrown and an error message will be displayed to indicate there is no undoable or redoable commands.

The following two Sequence Diagrams show a sample flow of the execution when an `EditCommand`, which is an `UndoableCommand`, has been executed and subsequently undone.

The first diagram (Figure 9) describes the process of storing an `EditEvent` to `EventHistory` during the execution of the `EditCommand`. The `EventPayload` is only initialized when the `EditCommand` is executed. The `EventPayload` is subsequently used for the initialization of the `EditEvent`.

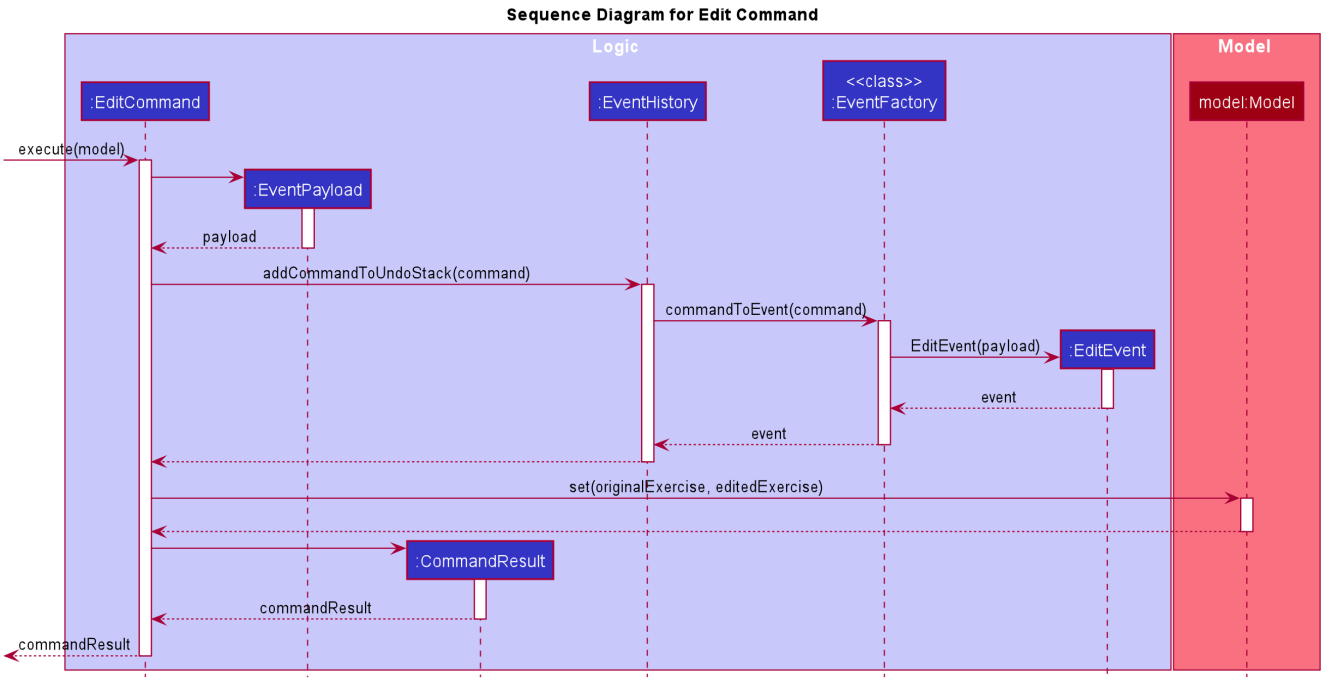


Figure 1. The process of storing an EditEvent to EventHistory

The second diagram (Figure 10) describes the process of undoing the executed `EditCommand` using the `UndoCommand`. When the `UndoCommand` is executed, the `EventHistory` calls the `undo` method of the next `Event` in the undo stack (i.e. the `EditEvent`).

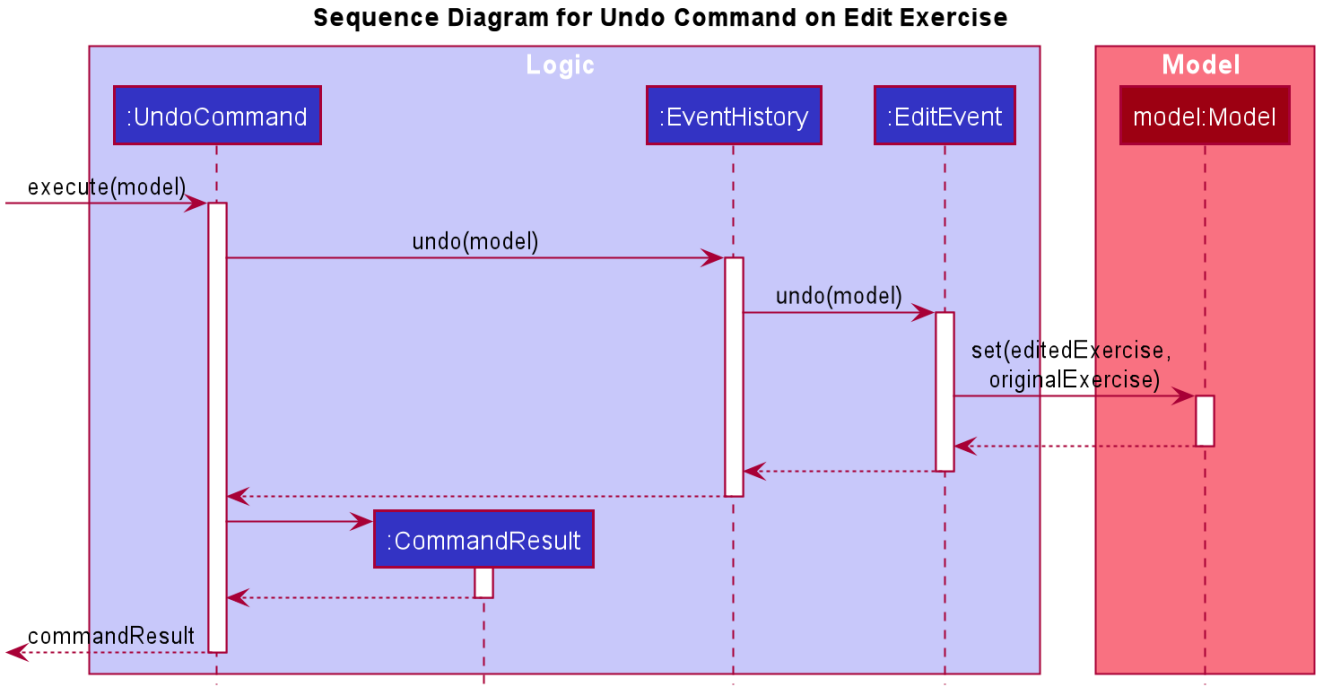


Figure 2. The process of undoing the EditCommand

Given below is a Class Diagram (Figure 11) to show the associations between Event, Command and Model. It is specifically designed such that only objects that implement the `Event` and `Command` interface will need to handle the `model` class.

NOTE

The only commands that implements the `UndoableCommand` are `AddCommand`, `DeleteCommand`, `EditCommand`, `ClearCommand`, `ScheduleCommand` and `ResolveCommand`. They each stores an `EventPayload` instance.

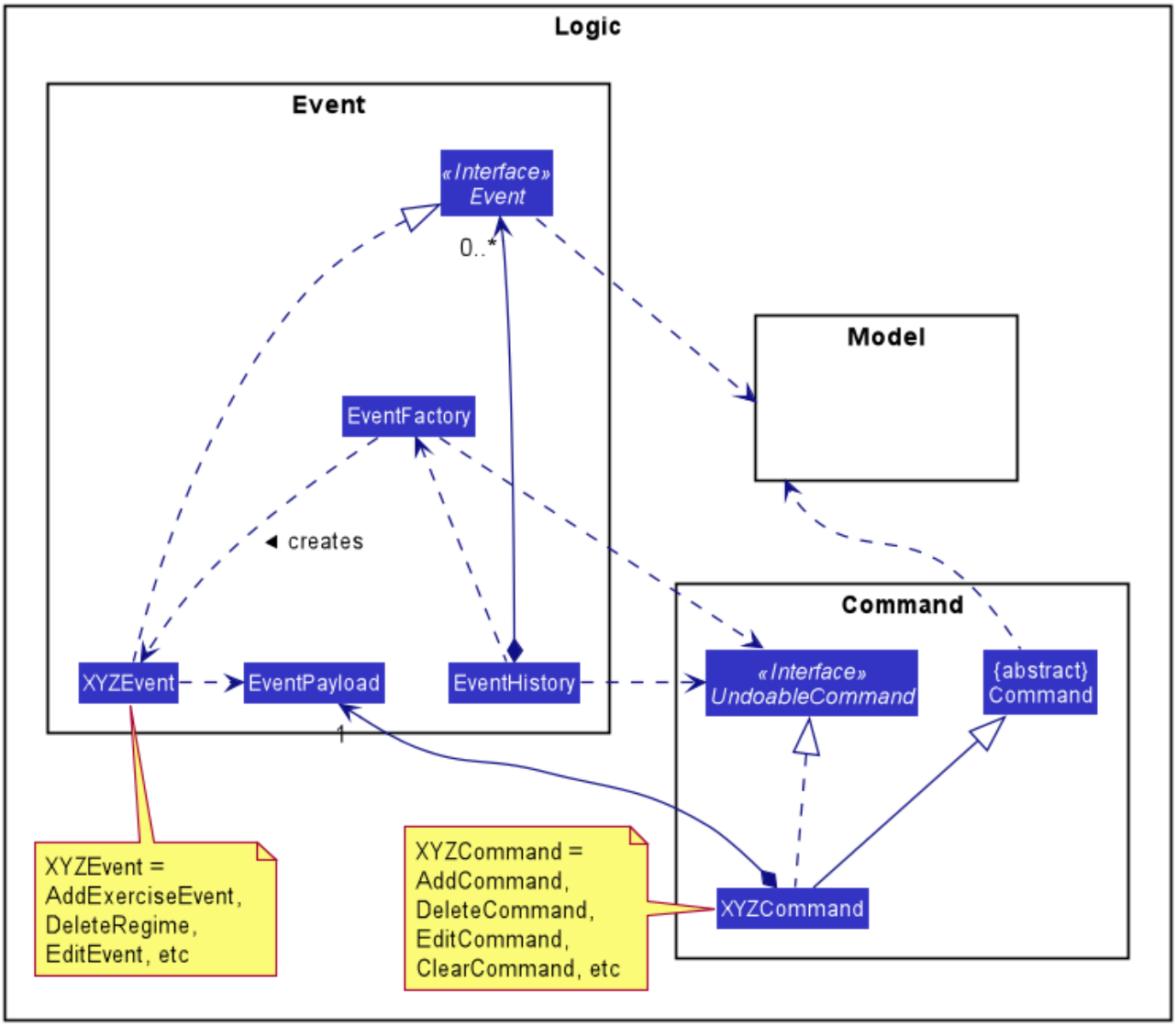


Figure 3. The associations between Event, Command and Model

The following Activity Diagram (Figure 12) summarizes what happens when a user enters undoable commands, the undo command and the redo command.

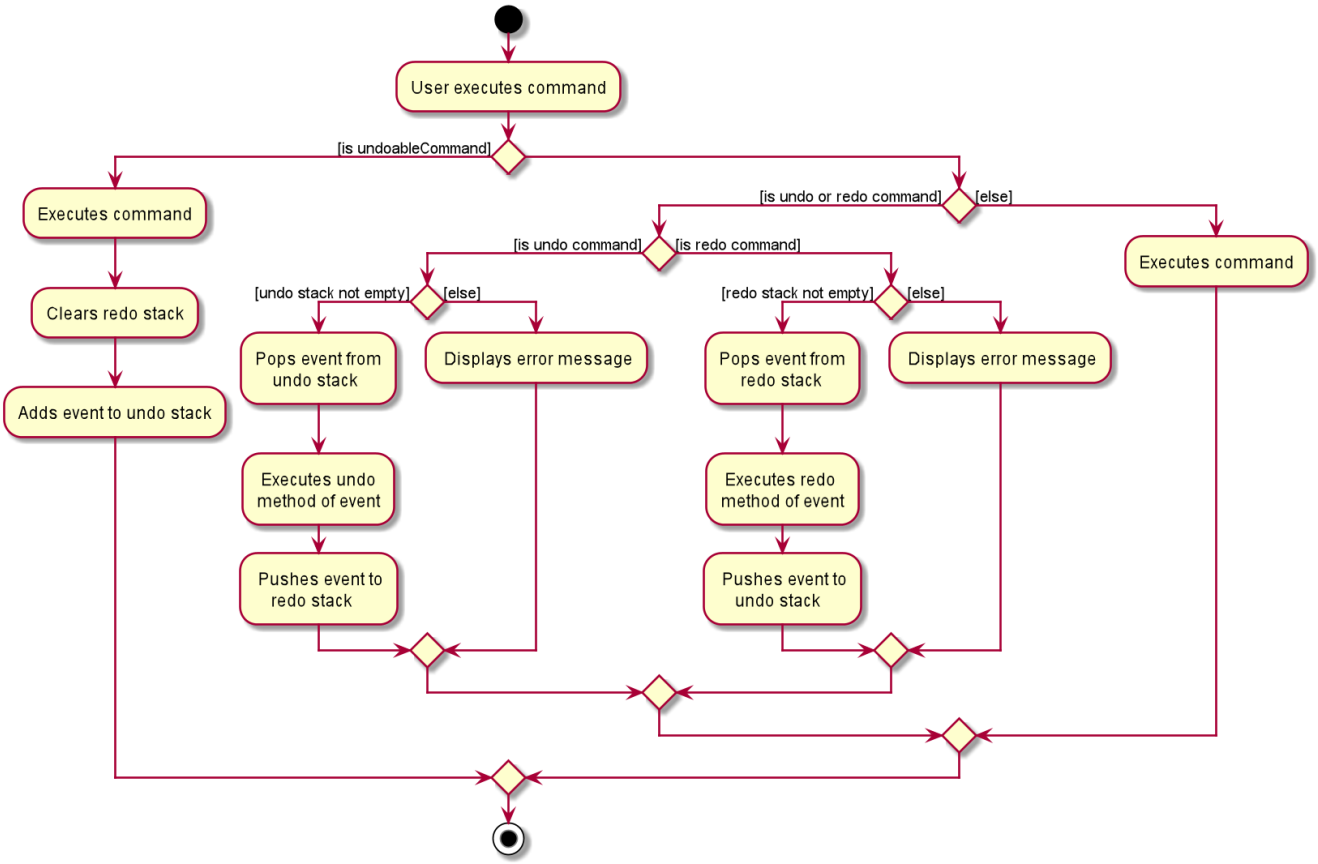


Figure 4. The workflow when a user enters an undoable command and performs undo/redo

Design Considerations

Aspect: How undo & redo executes

- **Choice 1: (current choice)** Implements undo and redo of each Command in a separate Event object stored in the EventHistory
  - Pros:
    - Uses less memory to store Event objects and payloads as compared to entire copies of the Model object.
    - Open for extensions and close to modifications as the Event interface only contains undo and redo methods, and can be easily implemented when new Undoable commands are introduced.
  - Cons:
    - UndoableCommand objects are forced to depend on EventPayloads when it does not actually use it directly. (e.g. `DeleteCommand` has to store the exercise being deleted despite using it only once).
- **Choice 2:** Individual command knows how to undo/redo by itself.
  - Pros:
    - Uses less memory to store each command as compared to entire copies of the Model object.
  - Cons:

        - Violates Single Responsibility Principle as Commands need to contain specific implementation of the inverse action of itself and also stores data such as the exercise being deleted in a local field.
- **Choice 3:** Saves the entire model consisting of the exercise, regime, schedule and suggestion lists.
    - Pros:
        - Easy to implement.
    - Cons:
        - May have performance issues in terms of memory usage as multiple lists need to be stored (i.e. Exercise list, Regime list, Schedule list)
        - Unnecessary storage of irrelevant details such as suggestion list.

Aspect: Data structure to support the undo/redo commands

- **Choice 1 (current choice):** Use a singleton EventHistory to store stacks of Events generated by a EventFactory.
    - Pros:
        - Ensures only one instance of EventHistory exists
        - The EventFactory relies on the Factory pattern that helps to reduce coupling between EventHistory and each individual Event.
    - Cons:
        - The Singleton pattern may have a chance of breaking if multiple threads initialize the singleton class at the same time, creating multiple instances of EventHistory. However, if this problem arises, the instantiation method can be made "synchronized" to circumvent this issue.
- **Choice 2:** Use a list to store the history of model objects.
    - Pros:
        - Very simple to implement as each step simply requires a deep copy of the model to be created and stored.
    - Cons:
        - Difficult to monitor multiple resource books (e.g. Regime books and Exercise books) as they all manage different types of resources that can be altered by commands.