Victor Varian - Project Portfolio

Overview

Modulo - a java application that enable students to keep track of their studies by enabling them to create modules, plans, quizzes and managing their finances. The code is written in Java and the user interacts with it using a CLI, and it has a GUI created with JavaFX.

Summary of contributions

- **Major enhancement**: added **the ability to switch application for different features**
  - What it does: allows user to switch logic, model, storage, UI from one feature to another.
  - Justification: This feature is the first step for the user to switch and use several features that Modulo offers. Switching from one feature to another is maintained to smooth.
  - Highlights: This enhancement can be extended further if the next developer decides to add a big feature to Modulo. It required an in-depth analysis of design alternatives. The implementation too was challenging as it required a depth understanding on how to make sure each feature care for their own implementation without others interfering it. Furthermore it will also reduce coupling in our code.
- **Major enhancement**: added **the ability to compromise for typo and categorical search**
  - What it does: allows the user to search for questions in multiple ways based on the result of the filter that the user keys in. It also substantially helps the user who often type carelessly to capture for typo if the user enable it.
  - Justification: This feature helps the user search for related questions faster and flexible. The feature can help users save more time from linear searching and retyping the keyword they want to search.
  - Highlights: This enhancement affects existing commands and commands to be added in future. It required an in-depth analysis of design alternatives. The implementation too was challenging as it required changes to existing commands.
- **Major enhancement**: added **the ability to undo/redo previous commands**
  - What it does: allows the user to undo all 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 affects existing commands and commands to be added in future. It required an in-depth analysis of design alternatives. The implementation too was challenging as it required changes to existing commands.
  - Credits: *addressbook3*
- **Minor enhancement**: added **the ability to hide the answer when the user is in the studyMode and vice versa**

- **Minor enhancement**: added **the ability to split screen to reveal the full details of the question and add explanation to the question**
- **Code contributed**: RepoSense Report
- **Other contributions**:
  - Project management:
    Managed releases `v1.1` - `v1.2.1` (3 releases) on GitHub
  - Community:
    - PRs reviewed (with non-trivial review comments): #104
    - Parts of the switch feature I added was adopted by several other class mates (63)
  - Tools:
    - Setting up the GitHub, Travis, Coveralls.
    - Integrated a new Github plugin (Travis and Coveralls) to the team repo
    - Maintaining the issue tracker every week

Contributions to the User Guide

Quiz

To enter the Quiz section please enter the command: `switch quiz`

Note on pre-populated data: The pre-populated data are used to help you to get the picture of how quiz show the data. You can use the command `clear` to clear the given data.

Note on UI: You can adjust the size of the split pane by sliding the window separator between the two windows to fully see the whole question. Go to `3.4.6` section to fully see the question properly.

Add the details of the quiz question:

Users can add new quiz questions with the details in it. The details of the question should be added with the following requirements.

The keyword instructions of the question include:
- `<qns>` Indicate that the next several phrases will be the name of the question.
- `<ans>` Indicate that the next several phrases will be the answer to the question.
- `<cat>` Indicate that the next several phrases will be the category of the question.
- `<type>` Indicate that the next several phrases will be the priority/importance level of the question.
- `<tag>` Indicate that the next several phrases will be the customized tag of the question.

The details instructions after the keyword of the question:
- `<qns>` are required. The word limit is 200 and minimum 3 letters.
- `<ans>` are required. The word limit is 125.
- `<cat>` are required. The word limit is 50.
- `<type>` are required. The format after this command should only be: `high`, `normal`, `low`.
- `<tag>` are optional. To add multiple tags, prepend <tag> keyword to differentiate one from another.

Note: Two questions are the same if they have the same **<qns>**, **<ans>** and **<cat>**
Format: `add <qns> your_question <ans> your_answer <cat> your_category <type> your_priority <tag> your_tag`

Example 1:
add **<qns>** *What is always coming, but never arrives?* **<ans>** *Tomorrow* **<cat>** *CS2131* **<type>** *high* **<tag>** *lecture* **<tag>** *tutorial*

Example 2:
add **<qns>** *What can one catch that is not thrown?* **<type>** *normal* **<ans>** *A cold* **<cat>** CS2131

- Remark (please avoid this):
  add **<qns>** *What is <qns>always comi<qns>ng, but never arrives?* **<ans>** *Tom<ans>orrow*

Delete question from record:

Delete a specific quiz question from the group questions.

Format: `delete [NUMBER]`
Usage: `[NUMBER]` is the index/row from that category that you want to delete.

Example 1:
delete 1

Edit details of a question : `edit`

Specify which question you want to edit and modify it from the question list.

The details/instructions of the new questions include:
- `<qns>` Indicate that the next several phrases will be the name of the question.
- `<ans>` Indicate that the next several phrases will be the answer to the question.
- `<cat>` Indicate that the next several phrases will be the category of the question.
- `<type>` Indicate that the next several phrases will be the priority/importance level of the question.
- `<tag>` Indicate that the next several phrases will be the customized tag of the question.

Note that at least one of the following must be modified and replaced:
- `<qns>` word limit is 200. No minimum letter.
- `<ans>` word limit is 125. No minimum letter.
- `<cat>` word limit is 50.
- `<type>` format after this command should only be: `high`, `normal`, `low`.
- `<tag>` are optional. To add numerous tags, please prepend <tag> keyword to differentiate one tag from another.

Format: **`edit [NUMBER]`** **`<qns>`** `your_new_question` **`<ans>`** `your_new_answer` **`<cat>`** `your_new_category` **`<type>`** `your_new_priority` **`<tag>`** `your_tag`

Usage: `[NUMBER]` is the index/row from that category that you want to edit.

Example 1:
edit 1 <**qns**> *How many mammals are there in the universe?* <**type**> low

- Remark (Please avoid this):
  edit 1 <**qns**> *What is <qns>always comi<qns>ng, but never arrives?* <**ans**> *Tom<ans>orrow*

List all the questions

To list all the questions.

Format: `list`

Find the quiz questions with keyword:

Find the quiz questions from the list of questions.

Format: `find [INSTRUCTION] <key>[KEYWORDS]`
Note: The [KEYWORDS] is the list of keywords that I want to search.

The expected details/instructions for [INSTRUCTION] keyword are:
- `question` - Indicate that I want to include the name of the question in my search.
- `answer` - Indicate that I want to include the answer to the question in my search.
- `category` - Indicate that I want to include the category of the question in my search.
- `type` - Indicate that I want to include the priority/importance level of the question in my search.
- `tag` - Indicate that I want to include the customized tag of the question in my search.

Note:
• To search for everything, leave the [INSTRUCTION] field blank.
• To include multiple instructions to be searchable or to search for multiple keywords, you can split it with a comma (`,`)
• To enable friendlier syntax [KEYWORDS] when searching your desired questions, add `-i` after your find command.

Example 1: **find** <*key*> CS2131, lecture
Explanation: search for *CS2131* and *lecture* keyword from the whole question

Example 2: **find** category, answer <*key*> Tomorrow
Explanation: search from the category and answer portion of each question that matches keyword *Tomorrow*

Example 3: **find -i** answer <*key*> Tomrrow
Explanation: search from the answer portion of each question that matches keyword that is similar to *Tomrrow*

Details of a selected question:

Show the details of a selected question. This details will show the whole questions that is truncated

from the preview window and reveal the comment/explanation from the selected question.

Format: `detail [INDEX]`
Usage: **`[INDEX]`** is the index/row from the question that you want to see.
Example: detail 1

Add Comment of a question:

Users can add a comment/explanation of a particular questions.

Format: `comment [INDEX] <val> your_comment`
Usage: **`[INDEX]`** is the index/row from the question that you want to comment.
Example: comment 1 <val> The explanation is in pg 194 textbook.

Hide answer of a question:

Users can hide all question answers when they want to revise or for other purposes. It also provide the user the hint of the answer. By default the answer is shown.

Format: `showAnswer [yes/no]`
Example: showAnswer no

Undo:

Undo some of the previous action.
Action that is undoable include: `add`, `edit`, `delete`, `clear`, `comment`
Format: `undo`

Redo:

Redo some of the previous action.
Action that is redo-able include: `add`, `edit`, `delete`, `clear`, `comment`
Format: `redo`

Contributions to the Developer Guide

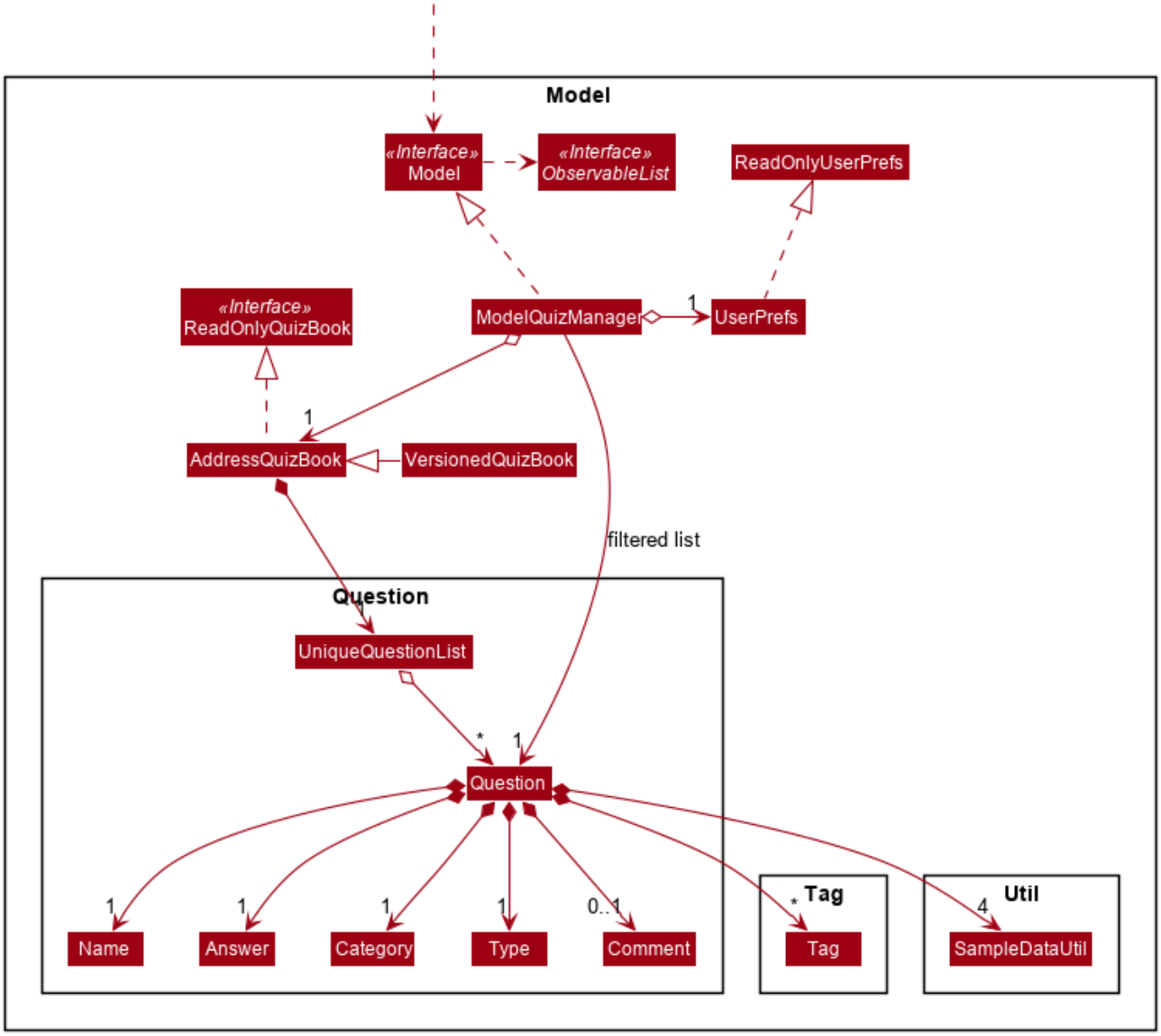
Quiz feature

Create question feature

Implementation

To use this feature, the user will need to switch to the quiz mode and add the question they want with several requirements on the syntax stated in the UserGuide.

Below are the quiz model class diagram:



In quiz feature, a `Question` has 6 attributes namely: `Name`, `Answer`, `Category`, `Type`, `Comment`, `Tag`, with the first three attributes differentiate one `Question` from the others. When the users first time launch the app or there are no data yet, Modulo will automatically populate the four questions from `SampleDataUtil`. The users are able to use the `clear` command if they want to delete the given data entirely. UI will then pull and updates the data from the `ModelQuizManager` which represents the in-memory model of the `AddressQuizBook` data, and show them to the users.

Given below is an example usage scenario on how to add a question properly and the mechanism that behaves at each step.

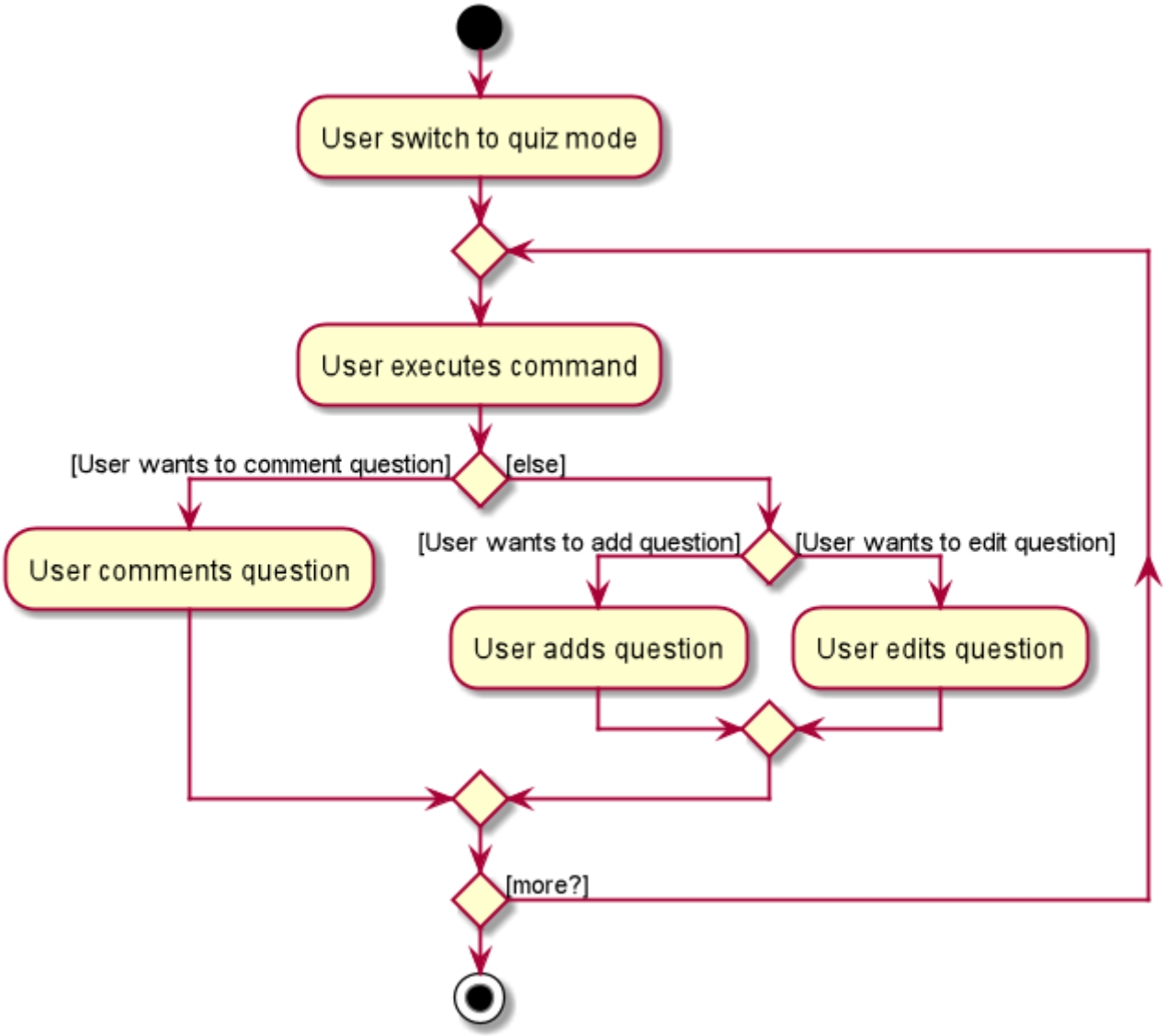
Step 1. The user launches the application and switch to the **quiz** mode by executing `switch quiz`.

Step 2. The user executes **`add <qns>`** `What is always coming, but never arrives?` **`<ans>`** `Today` **`<cat>`**

`CS2131 <type> high` command to add a question with the question name: *What is always coming, but never arrives?* and answer: *Tomorrow*, category: CS2131, type: *high* in the quiz book. The `add` command calls `Model#addQuestion()`, causing the modified state of the quiz book, after the command executes, to be saved in the `quizBookStateList` and shown in the UI.

Step 3. If the user realized that they have typed the wrong answer for a particular question, then the user can executes **`edit 1 <ans>`** `Tomorrow` command to replace the previous answer with the new answer with the given index prepend behind. The `edit` command calls `Model#setQuestion()`, causing the modified state of the quiz book, after the command executes, to be saved in the `quizBookStateList` and updated in the UI.

Step 4. *[Addtional]* The user can execute **`comment 1 <val>`** `The explanation is in pg 194 textbook` if he/she now decide to add a comment or explanation of a quiz question at index 1. The `comment` command calls `Model#setQuestion()` to parse in an additional parameter of comment into the entity of the question. It will then be saved in the `quizBookStateList` and shown in the UI. The following activity diagram summarizes the basic question creation process:



Design Considerations

Aspect: How to add question

- **Alternative 1 (current choice):** Users add question to the last of the list.
  - Pros: Easy to implement and fast (Insertion O(1)).
  - Cons: User may not be able to position the question from the list of questions.
- **Alternative 2:** Users insert the question to the desired position.
  - Pros: User can order the list as they desired and remember things better.
  - Cons: Not as fast as the insertion at the back of the list from alternative 1.

Undo/Redo feature

Implementation

The undo/redo mechanism is facilitated by `VersionedQuizBook`. It extends `QuizBook` with an undo/redo history, stored internally as an `quizBookStateList` and `currentStatePointer`. Additionally, it implements the following operations:

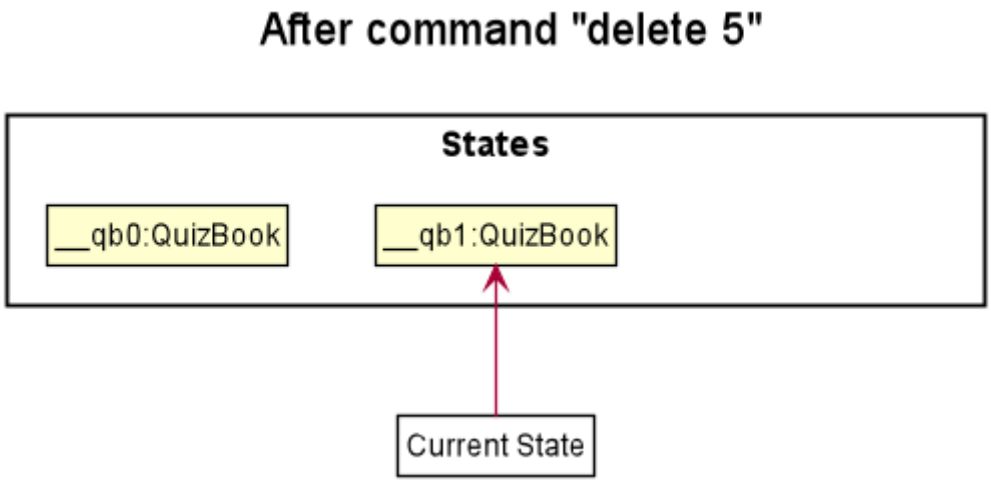
- `VersionedQuizBook#commit()` — Saves the current quiz book state in its history.
- `VersionedQuizBook#undo()` — Restores the previous quiz book state from its history.
- `VersionedQuizBook#redo()` — Restores a previously undone quiz book state from its history.

These operations are exposed in the `Model` interface as `Model#commitQuizBook()`, `Model#undoQuizBook()` and `Model#redoQuizBook()` respectively.

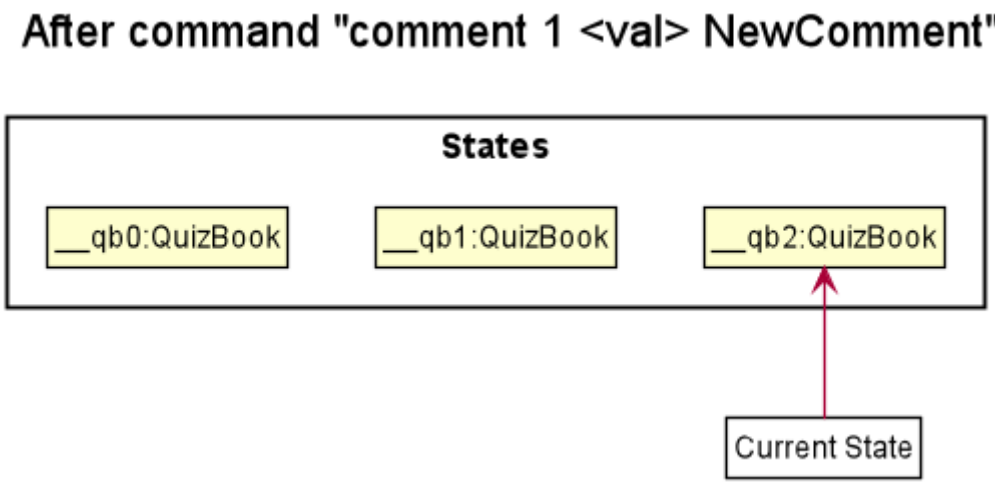
Below is an example usage scenario on how the undo/redo mechanism behaves at each step.

Step 1. The user launches the application for the first time. The `VersionedQuizBook` will be initialized with the initial quiz book state, and the `currentStatePointer` pointing to that single quiz book state.

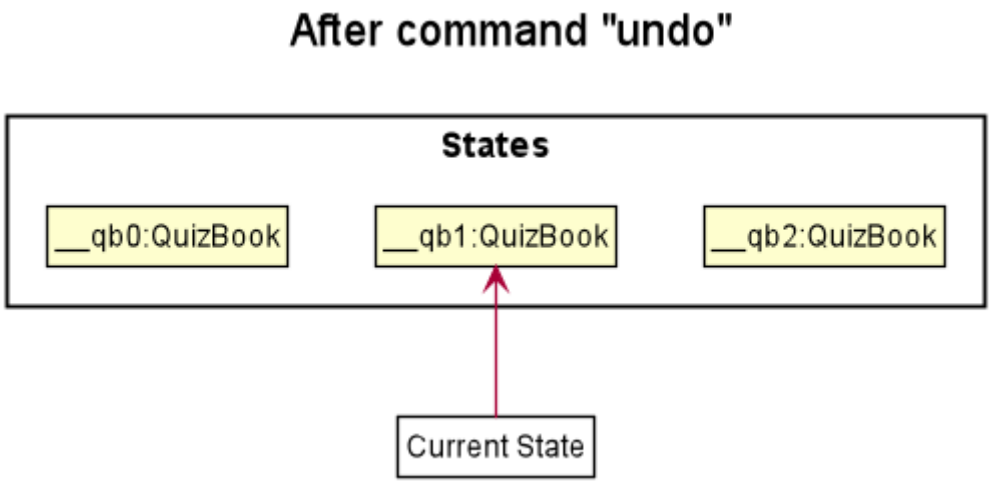
Step 2. The user executes `delete 5` command to delete the 5th question in the quiz book. The `delete` command calls `Model#commitQuizBook()`, causing the modified state of the quiz book after the `delete 5` command executes to be saved in the `quizBookStateList`, and the `currentStatePointer` is shifted to the newly inserted quiz book state.



Step 3. The user executes `comment 1 <val> NewComment` to add a new question. The `add` command also calls `Model#commitQuizBook()`, causing another modified quiz book state to be saved into the `quizBookStateList`.

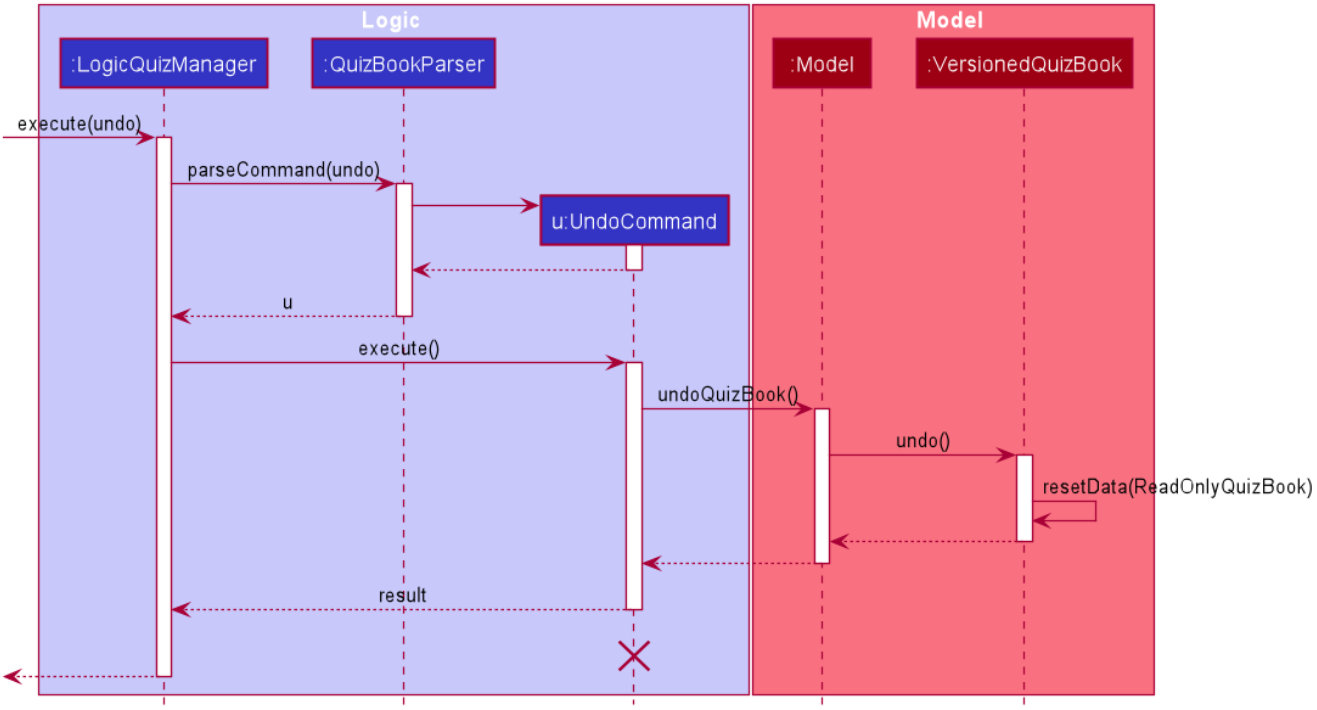


Step 4. The user now decides that adding the question was a mistake, and decides to undo that action by executing the `undo` command. The `undo` command will call `Model#undoQuizBook()`, which will shift the `currentStatePointer` once to the left, pointing it to the previous quiz book state, and restores the quiz book to that state.



NOTE	If the `currentStatePointer` is at index 0, pointing to the initial quiz book state, then there are no previous quiz book states to restore. The `undo` command uses `Model#canUndoQuizBook()` 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:



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

Design Considerations

Aspect: How undo & redo executes

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