CS2103/T Revision Tool - Developer Guide

By: `Team F10-3` Since: `Aug 2019` Licence: `MIT`

1. Setting up

Refer to the guide here[1].

2. Design

2.1. Architecture

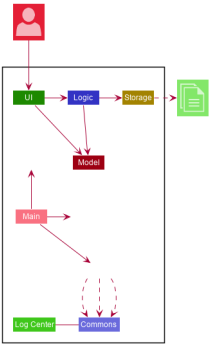


Figure 1. Architecture Diagram

The **Architecture Diagram** given above explains the high-level design of the App. Given below is a quick overview of each component.

The `.puml` files used to create diagrams in this document can be found in the diagrams[2] folder. Refer to the Using PlantUML guide[3] to learn how to create and edit diagrams.

`Main` has two classes called `Main`[4] and `MainApp`[5]. It is responsible for,

[1] SettingUp.xml
[2] https://github.com/AY1920S1-CS2103-F10-3/main/docs/diagrams/
[3] UsingPlantUml.xml
[4] https://github.com/AY1920S1-CS2103-F10-3/main/src/main/java/seedu/address/Main.java

- At app launch: Initializes the components in the correct sequence, and connects them up with each other.
- At shut down: Shuts down the components and invokes cleanup method where necessary.

`Commons` represents a collection of classes used by multiple other components. The following class plays an important role at the architecture level:

- `LogsCenter` : Used by many classes to write log messages to the App's log file.

The rest of the App consists of four components.

- `UI`: The UI of the App.
- `Logic`: The command executor.
- `Model`: Holds the data of the App in-memory.
- `Storage`: Reads data from, and writes data to, the hard disk.

Each of the four components

- Defines its *API* in an `interface` with the same name as the Component.
- Exposes its functionality using a `{Component Name}Manager` class.

For example, the `Logic` component (see the class diagram given below) defines it's API in the `Logic.java` interface and exposes its functionality using the `LogicManager.java` class.

Figure 2. Class Diagram of the Logic Component

How the architecture components interact with each other

The *Sequence Diagram* below shows how the components interact with each other for the scenario where the user issues the command `delete 1`.

[5] https://github.com/AY1920S1-CS2103-F10-3/main/src/main/java/seedu/address/MainApp.java

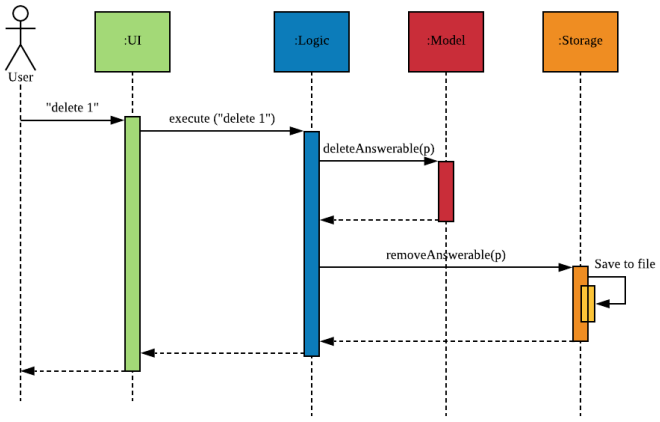


Figure 3. Component interactions for `delete 1` command

The sections below give more details of each component.

2.2. UI component

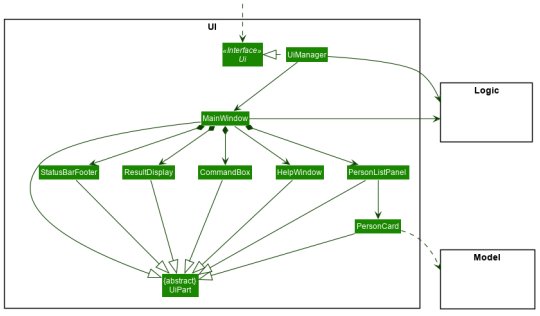


Figure 4. Structure of the UI Component

API : `Ui.java`[6]

The UI consists of a `MainWindow` that is made up of parts e.g.`CommandBox`, `ResultDisplay`, `PersonListPanel`, `StatusBarFooter` etc. All these, including the `MainWindow`, inherit from the abstract `UiPart` class.

[6] https://github.com/AY1920S1-CS2103-F10-3/main/src/main/java/seedu/address/ui/Ui.java

The `UI` component uses JavaFx UI framework. The layout of these UI parts are defined in matching `.fxml` files that are in the `src/main/resources/view` folder. For example, the layout of the `MainWindow`[7] is specified in `MainWindow.fxml`[8]

The `UI` component,

- Executes user commands using the `Logic` component.
- Listens for changes to `Model` data so that the UI can be updated with the modified data.

2.3. Logic component

Overview of Logic Component

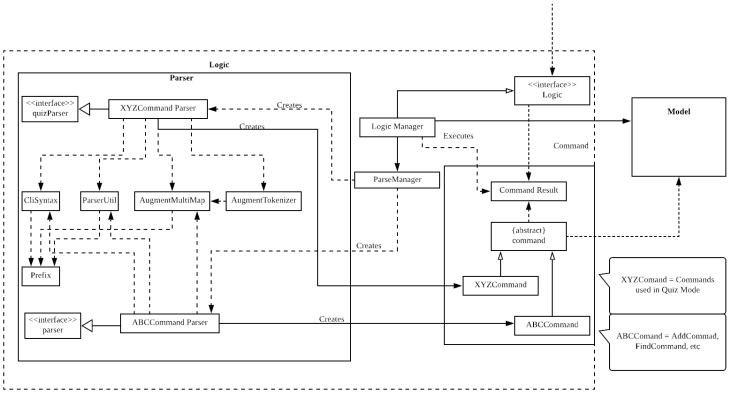


Figure 5. Structure of the Logic Component

API : `Logic.java`[9]

1. `Logic` uses the `ParserManager` class to parse the user command.
2. This results in a `Command` object which is executed by the `LogicManager`.
3. The command execution can affect the `Model` (e.g. adding a question).
4. The result of the command execution is encapsulated as a `CommandResult` object which is passed back to the `Ui`.

[7] https://github.com/AY1920S1-CS2103-F10-3/main/src/main/java/seedu/address/ui/MainWindow.java
[8] https://github.com/AY1920S1-CS2103-F10-3/main/src/main/resources/view/MainWindow.fxml
[9] https://github.com/AY1920S1-CS2103-F10-3/main/src/main/java/seedu/address/logic/Logic.java

5. In addition, the `CommandResult` object can also instruct the `Ui` to perform certain actions, such as displaying help to the user.
6. In quiz mode, the `CommandResult` object is also used to determine whether the user's answer is correct.

Managing parsing in Configuration and Quiz Mode

The Revision Tool uses two Parser interfaces (`Parser` and `QuizParser`) to parse different sets of commands (i.e. in Configuration Mode and in Quiz Mode).

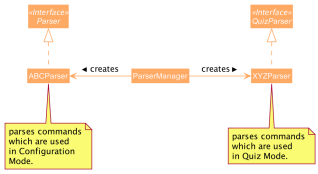


Figure 6. Class Diagram of ParserManager to display how parsers are created in both modes.

As shown in the figure above, the `ParserManager` class is responsible for creating the respective parsers for Configuration and Quiz Mode. This was designed while taking into consideration that the Quiz Mode Parsers (i.e. XYZParsers) will require an extra `Answerable` argument on top of the user input String in order to carry out commands such as determining whether the user's input is the correct answer. (E.g. to call methods such as Answerable#isCorrect(Answer))

As different commands are accepted in Configuration and Quiz Mode, the `ParserManager` class uses overloaded methods (`parseCommand(String)` and `parseCommand(String, Answerable)` to determine the valid commands in each mode. If a Configuration Mode command such as `add` were to be used in Quiz Mode, the ParserManager would deem the the command as invalid.

With reference to Figure 6, The following are the parsers used in each mode:

- ABCParser (Configuration Mode):
    - AddCommandParser
    - DeleteCommandParser
    - EditCommandParser
    - FindCommandParser

    - ListCommandParser
    - StartCommandParser
- XYZParser (Quiz Mode):
    - McqInputCommandParser
    - TfInputCommandParser
    - SaqInputCommandParser

A more detailed description of the implementation of parsing in Configuration and Quiz Mode and its design considerations can be found in Section 3.2, “Configuration and Quiz Mode”.

2.4. Model component

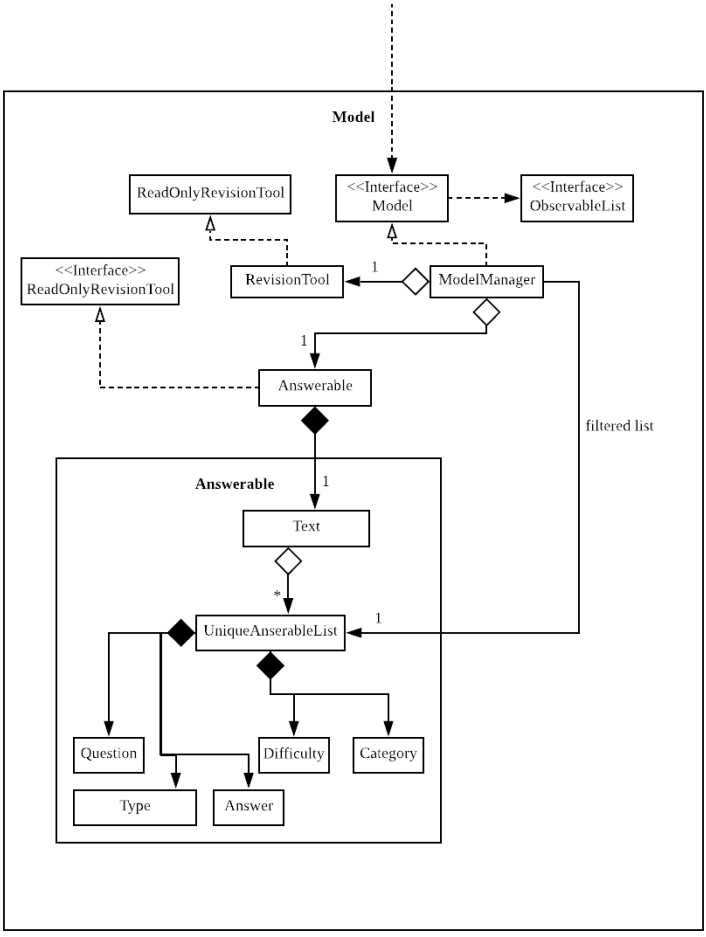


Figure 7. Structure of the Model Component

API : `Model.java`[10]

The `Model`,

- stores a `QuestionBank` object that represents the Question Bank.
- stores the Question Bank data.

[10] https://github.com/AY1920S1-CS2103-F10-3/main/src/main/java/seedu/address/model/Model.java

- exposes an unmodifiable `ObservableList<Answerable>` that can be 'observed' e.g. the UI can be bound to this list so that the UI automatically updates when the data in the list change.
- does not depend on any of the other three components.

The `Answerable` Class

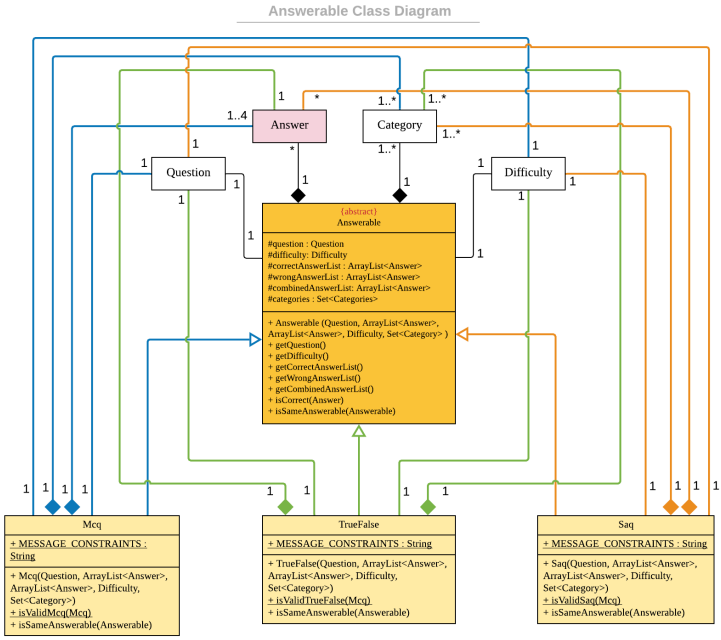


Figure 8. Class Diagram of the Answerable Class

The main class that the Revision Tool operates on is the `Answerable` class.

Each `Answerable` class must have 1 `Question`, 1 `Difficulty` and can have any amount of categories associated with it. The amount of answers that an `Answerable` can have depends on its type.

There are 3 subclasses of the `Answerable` Class which are: `Mcq`, `TrueFalse` and `Saq`. Each class defines it's own rules on the validity of `Answer` s (highlighted in red in the class diagram) provided to it.

The following are the rules of validity for each subclass:

- Mcq: 4 answers in total. 1 correct answer, 3 wrong answers.

- TrueFalse: Either 'true' or 'false' as its answer.
- Saq: Any amount of answers.

For all subclasses, there cannot be any duplicates of answers. For example, if an Mcq class has "option1" as one of its wrong answers, it cannot have "option1" as its correct answer or another wrong answer.

2.5. Storage component

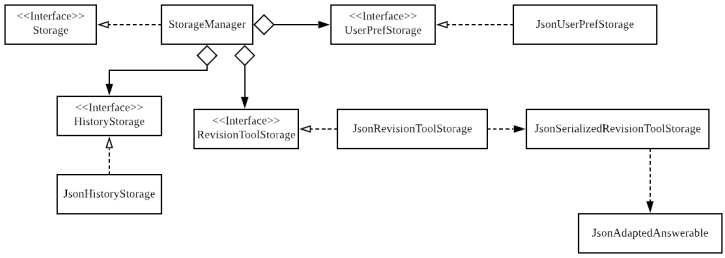


Figure 9. Structure of the Storage Component

API : `Storage.java`[11]

The `Storage` component,

- can save `Question Bank` objects in json format and read it back.
- can save the Test Bank data in json format and read it back.

2.6. Common classes

Classes used by multiple components are in the `seedu.revision.commons` package.

3. Implementation

This section describes some noteworthy details on how certain features are implemented.

[11] https://github.com/AY1920S1-CS2103-F10-3/main/src/main/java/seedu/address/storage/Storage.java

3.1. Add questions feature

Implementation

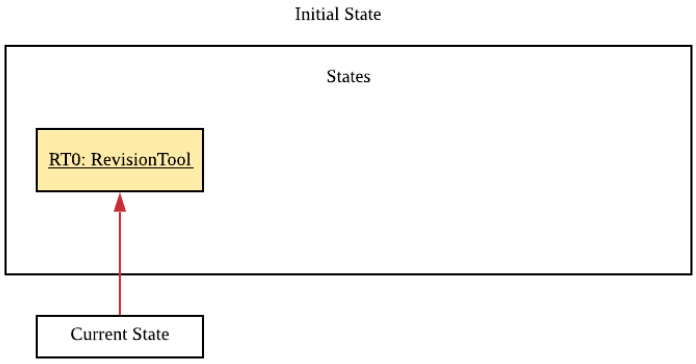
The add questions mechanism is facilitated by `AddCommand`. It extends `Command` that will read a user command and execute the command result. Additionally, it implements the following operations:

- `AddCommand#addMcq()` — Adds a mcq question to the question bank.
- `AddCommand#addShortAns()` — Adds a short answer question to the question bank.
- `AddCommand#addTf()` — Adds a True False answer question to the question bank.

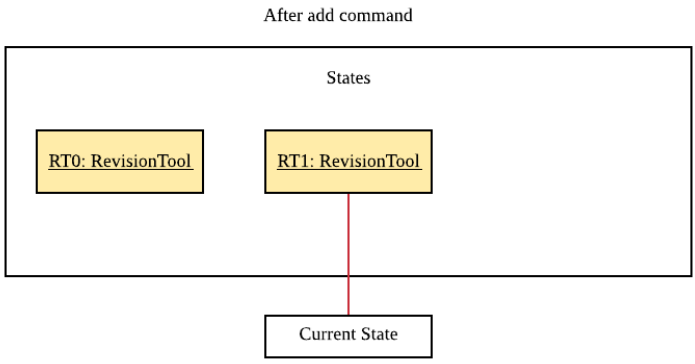
These operations are exposed in the `Model` interface as `Model#addMcqCommand()`, `Model#addTfCommand()` and `Model#addShortAnsCommand()` respectively.

Given below is an example usage scenario and how the add questions mechanism behaves at each step.

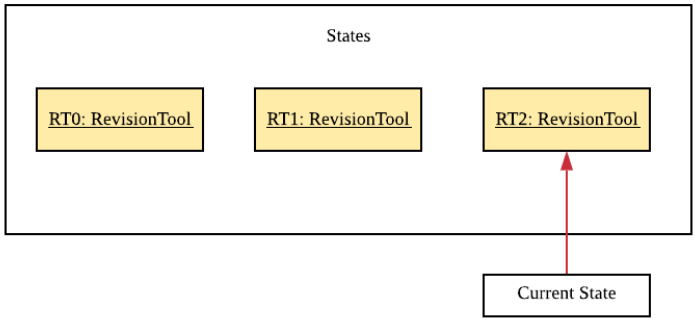
Step 1. The user types `add type/mcq q/"string of question" x/option1 x/option2 y/option3 x/option4 cat/[UML] diff/[easy]`, this command adds a easy difficulty mcq question about UML with 4 options and option3 being the correct answer.



Step 2. The command is being parse into the parser and the AddCommand object of type Command will be created.



Step 3. The AddCommand object will call its `addMcq()` method, this method will read the command and store the question with the answers into the `test bank` accordingly.



If a command fails its execution, it will not save the question into the `revision tool`. It will however throw an invalid command exception.

Design Considerations

Aspect: How add executes

- User enters the command "add …".
- Command is taken in and parsed to validate if it is a valid command.
- Add command is executed.
- New question is saved in the question bank.

Aspect: Format of the add command

- Use a single line containing all the information of the question.
- Pros: Faster for user to add a question.
- Cons: Hard for user to memorize the long sequence which may cause invalid command input.
    - **Other alternative we considered:** Use multiple steps to guide user on adding the question.
- Pros: User do not have to memorize the correct format and less chance for an invalid command input.
- Cons: Slow to add questions, requiring multiple steps to fully complete a question.

3.2. Configuration and Quiz Mode

As different commands are available for Configuration and Quiz Mode, we have to determine which commands are valid based on the state of the application. To do so, we had two main design considerations: The structure of the parser component and how to determine which parser to use. We will discuss these considerations in the following segment.

Design Considerations

Aspect	Alternative 1	Alternative 2	Conclusion and Explanation

Structure of the Parser Component	Command parsers for both modes implement the same interface (i.e. implement both parse(String) and parse(String, Answerable) methods) but for the method which is not used, throw an error if a client calls it.	Command parsers belonging to each mode implement different interfaces (i.e. a `Parser` or `QuizParser` interface) which dictates the parameters of their parse() methods. (i.e. parse(String) for Configuration Mode vs parse(String, Answerable) for Quiz Mode)	Alternative 2 was implemented. The main reason for this choice was to adhere to the interface-segregation principle. If alternative 1 were to be implemented, a Configuration Mode command may have to implement a parse(String, Answerable) dummy method which it will not use. This is bad design as a client might be able to call the dummy method and receive unexpected results. Thus, by separating the interfaces, clients will only need to know about the methods that they need.
Determining which parser to use	Create two parser manager classes (i.e.	Use a single `ParserManager` class which	Alternative 2 was implemented. By doing so,

	`QuizParserManager` and `ParserManager`) with accompanying `QuizLogicManager` and `LogicManager` classes whose methods will be called in `MainWindow` for Configuration Mode and `StartQuizWindow` for Quiz Mode respectively.	has overloaded methods of parse(String) and parse(String, Answerable). A single `LogicManager` will also implement `execute(String)` and `execute(String, Answerable)`. In Configuration Mode, the `LogicManager` will call `execute(String)` as there is no need to take in an Answerable and in Quiz Mode, the `LogicManager` will call `execute(String, Answerable) to initiate quiz-related commands.	we were able to adopt a facade design pattern. The main benefit would be that the client doesn't need to know the logic involved in selecting which type of parser and logic to use. This hides the internal complexity of the `ParserManager` class which will be responsible for determining which type of parser to use.

Commands in Configuration Mode

In Configuration Mode, a single string is passed as an argument to the Logic#execute method (i.e. execute(String)).

Given below is the Sequence Diagram for interactions within the `Logic` component for the `execute("delete 1")` API call.

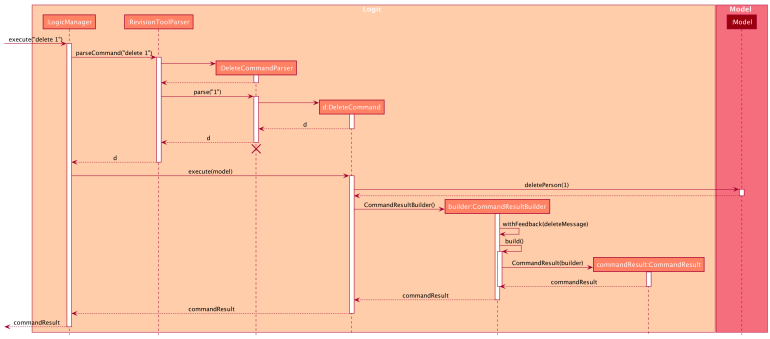


Figure 10. Interactions Inside the Logic Component for the `delete 1` Command

The lifeline for `DeleteCommandParser` should end at the destroy marker (X) but due to a limitation of PlantUML, the lifeline reaches the end of diagram.

Commands in Quiz Mode

In Quiz Mode, a string and the current `Answerable` object are passed as arguments to the Logic#execute method. (i.e. execute(String, Answerable)) Given below is the Sequence Diagram for interactions within the `Logic` component for the `execute("c", answerable)` API call.

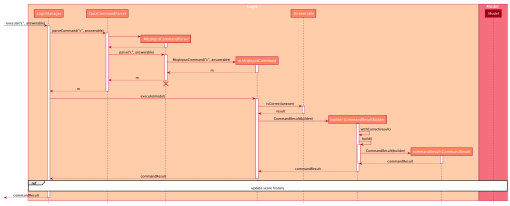


Figure 11. Interactions Inside the Logic Component for the `c` input command.

Key differences between Configuration Mode and Quiz Mode:

Configuration Mode	Quiz Mode
Logic#execute takes in a single string.	Logic#execute takes in a String and an Answerable.
No methods of answerables are called.	Answerable#isCorrect(Answer)) is called from the AnswerableInputCommands such as

	`McqInputCommand` to check whether the user's answer is correct.
CommandResult is used to display feedback to the user	CommandResult is used to display feedback to the user and inform `LogicManager` whether the selected answer is correct.
Model is used to save actions such as the addition/deletion of answerables.	Model is used to update the score history statistics.

Implementation of CommandResult (Builder Design Pattern)

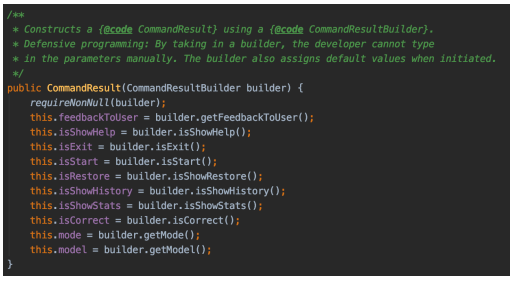
The `CommandResult` class is designed using a builder pattern to allow flexibility of values returned to the `LogicManager`. To guard against null values, default values are provided to every field in the `CommandResult` class upon construction. Objects that call `CommandResult` can choose to customise `CommandResult` according to their needs.

Below is a code snippet of the `CommandResultBuilder` and `CommandResult` class:

```
/**
 * Adds feedback to the {@code CommandResultBuilder} that will be returned.
 * @param feedbackToUser feedback that will be provided to the user.
 * @return CommandResultBuilder with the updated feedback.
 */
public CommandResultBuilder withFeedBack(String feedbackToUser) {
    this.feedbackToUser = feedbackToUser;
    return this;
}

/**
 * Adds a boolean to the {@code CommandResultBuilder} to indicate whether the command is a {@code HelpCommand}.
 * @param isHelp input boolean to determine result.
 * @return {@code CommandResultBuilder} with the withHelp boolean updated according to the input.
 */
public CommandResultBuilder withHelp(boolean isHelp) {
    this.showHelp = isHelp;
    return this;
}
```

```
/**
 * Builds a new instance of the {@code CommandResult} using the builder design pattern.
 * @return {@code CommandResult}
 */
public CommandResult build() {
    return new CommandResult( builder: this);
}
```



```
/**
 * Constructs a {@code CommandResult} using a {@code CommandResultBuilder}.
 * Defensive programming: By taking in a builder, the developer cannot type
 * in the parameters manually. The builder also assigns default values when initiated.
 */
public CommandResult(CommandResultBuilder builder) {
    requireNonNull(builder);
    this.feedbackToUser = builder.getFeedbackToUser();
    this.isShowHelp = builder.isShowHelp();
    this.isExit = builder.isExit();
    this.isStart = builder.isStart();
    this.isRestore = builder.isShowRestore();
    this.isShowHistory = builder.isShowHistory();
    this.isShowStats = builder.isShowStats();
    this.isCorrect = builder.isCorrect();
    this.mode = builder.getMode();
    this.model = builder.getModel();
}
```

Examples of how to build a CommandResult:

```
CommandResult c = new
 CommandResultBuilder().withFeedback(message).withExit(true).build();
CommandResult c = new CommandResultBuilder().isCorrect(true).build();
```

3.3. Quiz Mode

How the quiz works

After the user has `start` ed the quiz, the application enters Quiz Mode. The following is the flow of events after a quiz session has started.

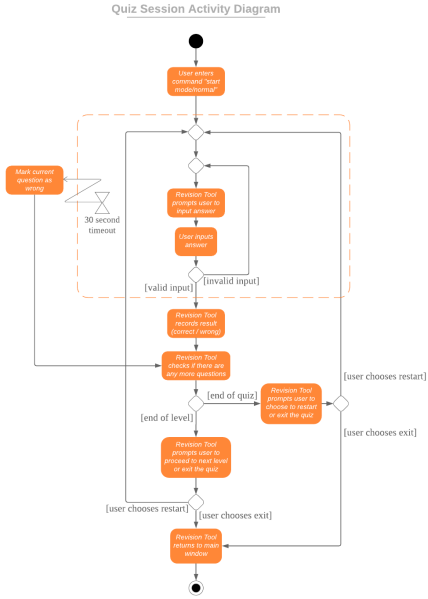


Figure 12. Activity Diagram of a Quiz Session in `NormalMode`

Elaboration of Steps:

1. After the user has started either a normal / custom mode quiz. He/she will be prompted to key in their answer.
2. If the input is valid, the revision tool will display the subsequent question until the level / entire quiz has ended.
3. If the input is invalid, the revision tool will prompt the user to key in their input again with guidance provided on the commands accepted.

4. If the time limit is exceeded (e.g. 30 seconds in Normal Mode), the revision tool will mark the question as wrong and move on to the next question.
5. Once a level has ended, the user will be given the choice to move on to the next level or exit the quiz.
6. Once the entire quiz has ended, the user will be given the choice to restart or exit the quiz.

For Arcade Mode, when a users enters a wrong answer, the quiz will end.

3.4. Restore feature

Implementation

The restore mechanism is facilitated by `RestoreCommand`. It extends `Command` that will read a user command and execute the command result. Additionally, it implements the following operations:

- `#handleRestore()` — Prompts the user with an alert box if he really wishes to execute the restore function.
- `#setRevisionTool()` — Clears the current question bank and reset it with our own default questions.

These operations are exposed in the `Model` interface as `Model#setRevisionTool()` and from `MainWindow` as `#handleRestore()` respectively.

Design Considerations

- When implementing the restore feature, we didn't want users to face a problem if they entered the command accidentally hence the alert popup was implemented, to prompt users if they really want to carry out the command before executing it.
- With this popup, users will now be more cautious when trying to restore and only do so when they really want to reset their revision tool.

- Furthermore, the questions that we included in the default revision tool question bank are questions taken from the lecture quiz and weekly quiz which are most probably deemed important by the professor himself.

Aspect: How Restore executes

- User enters the command "restore".
- Command is taken in and a popup is shown to reconfirm if the user would like to carry out the restore command.
- Upon clicking yes, restore command will be handled.
- Current questions will be deleted and default questions will reset to the revision tool.

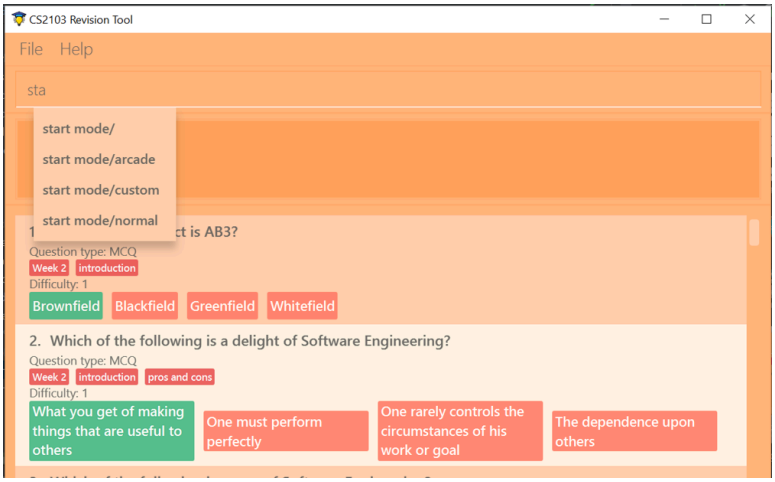
3.5. AutoComplete feature

Implementation

- A set of commands and auto completed text are saved in a set.
- When users type a command on the text box, method `#populatePopup` will be called where the user's command will be matched against our SortedSet.
- If there is a match, a contextMenu showing all possible auto complete text will show up.
- This method is implemented such that the results in the contextMenu will change and show as the user is typing and this would make it more intuitive for users.

Design Considerations

- The main design consideration here would be to have value added auto complete list to pop up.
- How we managed that is to show:
    - The basic command
    - Basic command + possible parse commands where they can easily fill in.



Aspect: How AutoComplete works

- Users wishes to enter an "Add" command `add type/mcq q/What is 1 + 1 y/2 x/1 x/3 x/4 cat/easy diff/1`
- Upon typing either "a", "ad" or even "add", the auto complete context menu will pop up showing possible auto complete list, mainly:
  - add
  - add type/ q/ y/ x/ cat/ diff/
- Upon seeing that, users will be able to select those options or use those as a guideline to complete his commands more intuitively.

3.6. Logging

We are using `java.util.logging` package for logging. The `LogsCenter` class is used to manage the logging levels and logging destinations.

- The logging level can be controlled using the `logLevel` setting in the configuration file (See Section 3.7, “Configuration”)
- The `Logger` for a class can be obtained using `LogsCenter.getLogger(Class)` which will log messages according to the specified logging level
- Currently log messages are output through: `Console` and to a `.log` file.

Logging Levels

- `SEVERE` : Critical problem detected which may possibly cause the termination of the application
- `WARNING` : Can continue, but with caution
- `INFO` : Information showing the noteworthy actions by the App
- `FINE` : Details that is not usually noteworthy but may be useful in debugging e.g. print the actual list instead of just its size

3.7. Configuration

Certain properties of the application can be controlled (e.g user prefs file location, logging level) through the configuration file (default: `config.json`).

4. Documentation

Refer to the guide here[12].

5. Testing

Refer to the guide here[13].

6. Dev Ops

Refer to the guide here[14].

A. Product Scope

Target user profile:

- is a CS2103/T student
- prefer to use an app to help them to revise
- can type fast
- prefers typing over mouse input
- is reasonably comfortable using CLI apps

[12] Documentation.xml
[13] Testing.xml
[14] DevOps.xml

Value proposition: helps student to ace CS2103/T

B. User Stories

Priorities: High (must have) - * * *, Medium (nice to have) - * *, Low (unlikely to have) - *

Priority	As a …	I want to …	So that I can…
* * *	lazy CS2103 student	refer to the revision tool solely for my consolidated module revision	do not have to refer to Luminus
* * *	CS2103 student	have a personalised application to store all my questions and answers in one place	refer to it conveniently for revision.
* * *	CS2103 student	have a revision tool to test my coding skills and concepts through	test myself on coding proficiency too.

Priority	As a …	I want to …	So that I can…
		writing short codes	
* * *	CS2103 student	keep track and see how much progress I have made in completing the questions	gauge my level of progress in completing the syllabus.
* * *	vim-using CS2103/T student	use the keyboard-based commands	further increase my efficiency
* * *	busy CS2103/T	use quick revision tools	learn using spaced-retrieval
* * *	busy CS2103 student	mark certain concepts as easy	will not have to spend as much time studying the easy concepts.
* * *	CS2103 student with a lot of things	mark certain questions that I am unsure of	refer back to the question when I am free.

Priority	As a …	I want to …	So that I can…
	on my mind		
* * *	CS2103 student	import questions from my peers	study on my own.
* * *	conscientious CS2103 student	export the questions I am unsure of	raise them up during tutorials.
* * *	indecisive student	be recommended questions instead of me having to plan my own study plan	go directly to studying
* *	competitive CS2103 student	at least know where I stand among my cohort	look at who is the next person I can beat.
* *	gamer CS2103/T student	accomplish tasks that give me a sense of achievement, preferably	I feel good.

Priority	As a …	I want to …	So that I can…
		through in application rewards	
* *	A+ CS2103 student	review and give suggestions to improve the application	benefit more CS2103 students.
* *	CS2103 student	port this application over to my other modules	revise for my other modules using this application as well.
* *	unorganized CS2103 student	get reminders about my quiz deadlines	complete my quizzes on time
* *	organized CS2103 student	schedule reminders to remind me when I should use the application to do revision	will not forget to do revision.

Priority	As a …	I want to …	So that I can…
* *	user of the application	get an estimate of my final grade for CS2103	know what to expect on result release day.
* *	CS2103 peer tutor	use this as a form of teaching tool	teach better
* *	CAP 5.0 CS2103 student	show off my IQ by perfecting my test scores	motivate other students.
* *	CS2103 student	view the questions/ topics that most students answered wrongly	revise for those topics.
* *	visual oriented student	the app to have different colours as compared to the regular black and white	learn better

Priority	As a …	I want to …	So that I can…
* *	non-motivated CS2103 student	use the application to remind me to study	I will study
* *	student that wants shortcuts	type a partial command and have it be auto-completed	I can save time.
* *	CS2103 student new to Git	have a help function which lists all the commonly used Git commands	become more proficient with Git.
* *	master software engineer taking CS2103	be able to access the source code	to make the application better and customise it for myself.
* *	CS2103 student	get recommended a list of questions that I frequently get wrong	learn from my mistakes

Priority	As a …	I want to …	So that I can…
*	lonely CS2103 student	have someone to talk to, even if it's a computer	I won't feel lonely
*	CS2103 student who keeps having stomach ache	the application to tell me where the nearest toilet is	go and shit

C. Use Cases

(For all use cases below, the **System** is the `RevisionTool` and the **Actor** is the `user`, unless specified otherwise)

Use case: Delete answerable

MSS

1. User requests to list answerables
2. RevisionTool shows a list of answerables
3. User requests to delete a specific answerable in the list
4. RevisionTool deletes the answerable

   Use case ends.

Extensions

2a. The list is empty.

Use case ends.

3a. The given index is invalid.

3a1. RevisionTool shows an error message.

Use case resumes at step 2.

{More to be added}

D. Non Functional Requirements

1. RevisionTool should work on any mainstream OS as long as it has Java 11 or above installed.
2. RevisionTool be able to hold up to 1000 questions without any significant reduction in performance for typical usage.
3. A user with above slow typing speed for regular English text (i.e. not code, not system admin commands) should be able to accomplish most of the tasks faster using commands than using the mouse.
4. RevisionTool should be able to run without any internet connectivity.
5. RevisionTool does not require any further installation upon downloading the jar file.

E. Glossary

Mainstream OS
Windows, Linux, Unix, OS-X

Answerables
A set of question answers, which includes :

- Type: MCQ, True False, Short Answered Question
- Question
- Correct Answers (Can contain multiple answers)
- Wrong Answers (Can contain multiple answers)
- Category
- Difficulty

F. Instructions for Manual Testing

Given below are instructions to test the app manually.

These instructions only provide a starting point for testers to work on; testers are expected to do more *exploratory* testing.

F.1. Launch and Shutdown

1. Initial launch

    a. Download the jar file and copy into an empty folder

    b. Double-click the jar file
    Expected: Shows the GUI with a set of sample answerables. The window size may not be optimum.

2. Saving window preferences

    a. Resize the window to an optimum size. Move the window to a different location. Close the window by using the `x` button at the top right hand corner or by using the command `exit`.

    b. Re-launch the app by double clicking the jar file.
    Expected: The most recent window size and location is retained.

F.2. Adding an answerable

1. Adding a MCQ to the current list

    a. Test case: `add type/mcq q/What is 1 + 1 y/2 x/1 x/3 x/4 cat/easy diff/1`
    Expected: new MCQ answerable will be created and appended at the bottom of the list. Details of the the added answerable will be shown at the bottom of the list, and the correct answer will be highlighted in green.

2. Adding a True False to the current list

    a. Test case: `add type/tf q/What is 1 + 1 = 2 y/true cat/easy diff/1`
    Expected: new True False answerable will be created and appended at the bottom of the list. Details of the added answerable will be shown at

the bottom of the list, and only the correct answer will be shown and highlighted in green.

3. Adding a Short Answer Question (SAQ) to the current list

    a. Test case: `add type/saq q/what is smaller than 10 but bigger than 7? y/8 y/9 cat/easy diff/1`
    Expected: new SAQ answerable will be created and appended at the bottom of the list. Details of the added answerable will be shown at the bottom of the list and all the correct answers state will be highlighted in green.

4. Adding an Answerable that already exist in the Revision Tool

    a. Test case: `add type/mcq q/What is 1 + 1 y/2 x/1 x/3 x/4 cat/easy diff/1`
    Expected: No new answerable will be added as the question already exist in the Revision Tool. An error message will be thrown, informing users that the answerable already exist in the Revision Tool.

F.3. Deleting an answerable

1. Deleting an answerable while all answerables are listed

    a. Prerequisites: List all answerables using the `list` command. Multiple answerables in the list.

    b. Test case: `delete 1`
    Expected: First answerable is deleted from the list. Details of the deleted answerable shown in the status message. Timestamp in the status bar is updated.

    c. Test case: `delete 0`
    Expected: No answerable is deleted. Error details shown in the status message. Status bar remains the same.

    d. Other incorrect delete commands to try: `delete`, `delete x` (where x is larger than the list size) *{give more}*
    Expected: Similar to previous.

F.4. Starting quiz

1. Starting a quiz

    a. Test case: `start mode/normal`
    Expected: Start quiz window will pop up with a question showing under the command box and the answers in the result box further below. Answer the questions using the CLI accordingly to see the progress bar move till quiz completion. Users will be prompted if he wishes to proceed to level 2.

    b. Test case: `start mode/arcade`
    Expected: Start quiz window will pop up similar to previous test case. The only difference would be that once the quiz proceeds and an incorrect answer is input, the quiz ends and the score will be shown. This is the "hard mode" of our quiz mode.

    c. Test case `start mode/custom timer/3`
    Expected: Start quiz window will pop up similar to previous test case. The difference here will be the timer. Instead of the 30 seconds timer per question, the new timer (seen at the bottom right beside the status progress bar) will be at 3 seconds as set by the user.

F.5. Saving data

1. Dealing with missing/corrupted data files

    a. If there is a missing answerables data file, the RevisionTool will automatically create a default data file with all the default answerables inside.

    b. If there is a corrupted answerable data file, the RevisionTool will automatically start with a list of empty file. Users will then be able to use the `#restoreCommand` here to get a list of default answerables or alternatively, create a new set of answerables manually.

2. To identify missing/corrupted data files:

    a. `.\data\revisiontool.json`

    b. `.\data\history.json`