Zhang Jian - Project Portfolio

PROJECT: Duke Academy

Overview

Duke Academy is a Java programming platform equipped with a library of programming questions related to the field of Data Structures and Algorithms, as well as an automated judging system. The user interacts with it using a CLI, and it has a GUI created with JavaFX. It is written in Java, and has about 15 kLoC.

This is what our project looks like:

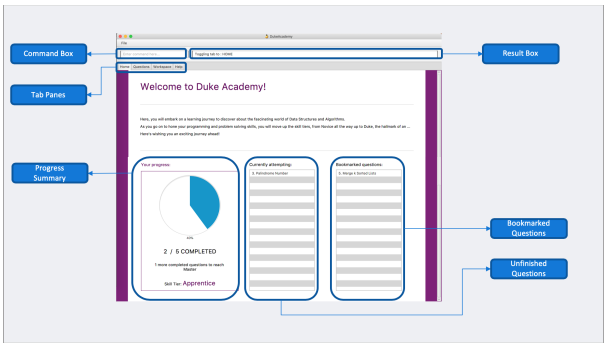


Figure 1. The graphical user interface for Duke Academy.

Summary of contributions

Major enhancement:

Model

1. Created the `Question` object and the corresponding attributes, such as `title`, `description`, `topics`, etc.

Storage

1. Refactored the json serializer and deserializer to target Duke Academy.
    - What it does: Duke Academy is able to save data into json format and load questions from json files.
    - Justification: prevents loss of user data.
    - Highlights: requires interaction between logic and storage. Requires knowledge about json serializer/deserializer's API and its command usage.
2. Implemented the `loadquestions` command.

- What it does: Imports new questions into the application through a text file.
- Justification: User is able to load external questions into this app. As such, professors can distribute practice assignments through this platform despite the fact that the materials may change yearly.
- Highlights: Requires knowledge on reading files from a given directory, create objects based on the string parsed, interaction with the storage model, update the new changes onto the user interface.

Command

1. implemented **the `view` command**
   - What it does: Displays the problem statement of the question.
   - Justification: the user need to view the problem description in order to understand the requirement of the problem context, as well as the specified input/output format.
   - Highlights: requires user interface to be updated based on the data obtained from the storage.
2. added **the `browse` command**
   - What it does: allows the user to search through all questions with the specified keyword(s). A question is listed as a search result as long as it contains one of the keyword(s) in their *title, topics, description, status* or *difficulty*.
   - Justification: This feature improves the product significantly because it would be convenient for a user who want to search questions by a specific category or certain keywords.
   - Highlights: This enhancement involves all the questions in the question bank. It requires interaction between models and storage to actively filter questions by certain predicates. It also requires the UI to reflect the changes in the question list after the filtering conditions have been updated. It required a comprehensive understanding of the code base architecture.
   - Credits: *{part of it is re-used from the `find` command in the original AB3 code base.}*
3. added **the `find` command**
   - What it does: Searches for question of which the title contains **strictly** the keywords entered.
   - Justification: benefits the user if they only want to search by question title.
   - Credits: _{part of it is re-used from the `find` command in the original AB3 code base.}
4. added **the `showall` command**
   - What it does: Navigates to the **Questions** Tab and displays all available questions.
   - Justification: after user searches for questions with certain keywords, they may want to restore to the original question list.
   - Credits: _{part of it is re-used from the `list` command in the original AB3 code base.}

Tests

Implemented tests for the `showall` command, namely `ShowallCommandFactoryTest` and

`ShowallCommandTest`.

Minor enhancement:

Added a Problem Display Panel to the Questions and Workspace Tab.

Code contributed:[Reposense]

Other contributions:

- Project management:
  - Assigned milestones for `v1.2`.
  - Created tags for the issues, such as `Type.Enhancement`, `Severity.High`.
- Documentation:
  - Did cosmetic tweaks to existing contents of the User Guide's quickstart section.
  - Contributed to Developer Guide and User Guide as shown at the end.
- Community:
  - Reported bugs and suggestions for other teams in the class (examples: Typos in UG, Feature not present: 6 Load commands stated in UG not implemented, Budget command not working with only amount and description entered)
- Tools:
  - Integrated 2 third party library (Travis, Codacy) to the project.

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.*

Loading new questions: `loadquestions`

Imports new questions into the application through a text file.

Format: `loadquestions [filename]`

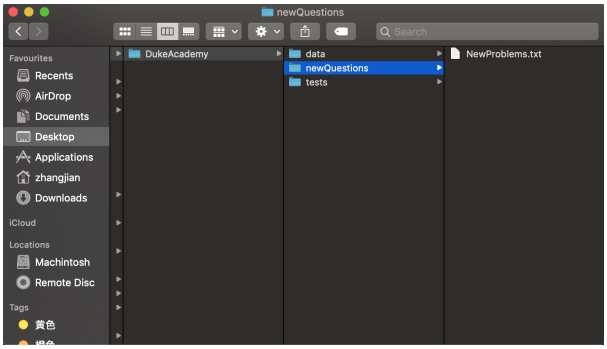
> - Your text file should be located in the `../DukeAcademy/newQuestions/` directory. The `DukeAcademy` folder is located in the same directory as where you put the jar file.
> - Your text file should follow the format specified at Format for Custom Questions.

Examples:

- `loadquestions my_problem_set.txt`

Loads the questions from the file "my_problem_set.txt" located in the `../DukeAcademy/newQuestions/` directory.

We have prepared a dummy problem set for you to try out as shown in diagram below.



Type `loadquestions NewProblems.txt` (case sensitive), and two new questions titled `Apple` and `Banana` will be loaded onto Duke Academy.

FAQ

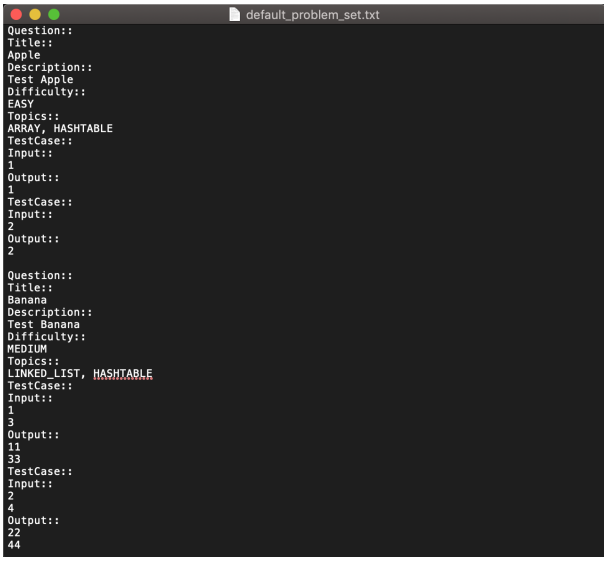
Q: How do I transfer my own problem sets to another computer?

A: Copy the problem set text file into DukeAcademy's home folder on the other computer and repeat the loadquestions command.

Q: How do I transfer data to another computer?

A: Install the app in the other computer and overwrite the empty data file it creates with the file that contains the data of your previous DukeAcademy folder.

Format for Custom Questions



- Create a .txt file.
- The format of a question goes like follows:

```
Question::

Title::

Description::

Difficulty::

Topics::

TestCase::

Input::

Output::
```

- All inputs must be in the order stated above.
- Title, Description can be any non-empty string.
- Difficulty can only be `EASY`, `MEDIUM` or `HARD`. (Must be capitalized)
- Topics can only be `ARRAY`, `LINKED_LIST`, `HASHTABLE`, `TREE`, `GRAPH`, `RECURSION`, `DIVIDE_AND_CONQUER`, `DYNAMIC_PROGRAMMING`, `SORTING`, or `OTHERS`. (Must be capitalized)
- One question can only have one title, description and difficulty. It can have multiple topics separated by `,`. It can have multiple test cases, each begin with a `TestCase::` identifier.
- For sample questions, refer to the `../DukeAcademy/newQuestions/NewProblems.txt` file.

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.

Design

Architecture

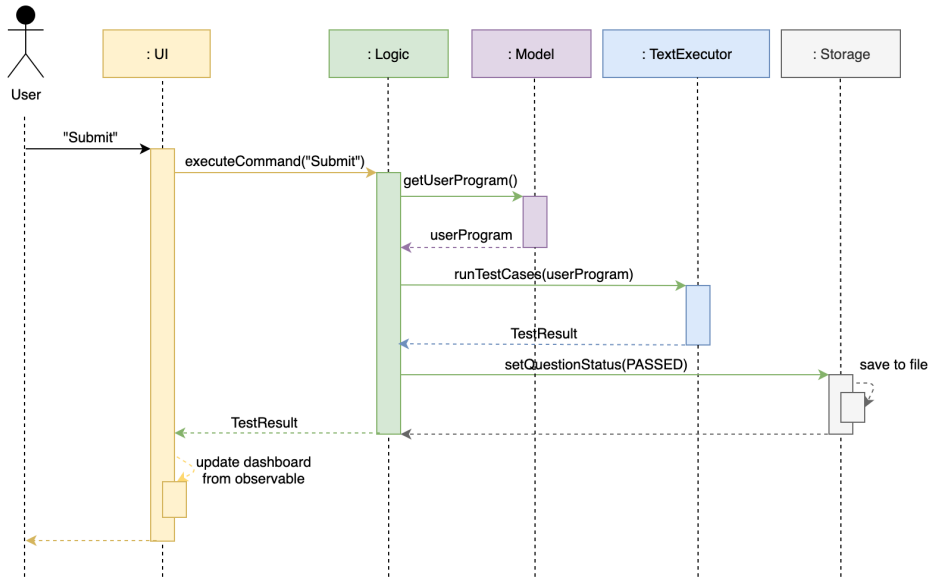


Figure 1. Architecture Diagram

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

`Main` has two classes called `Main` and `MainApp`. It is responsible for,

- 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. `Observable` is responsible for updates on User Interface if internal data changes. `LogsCenter`is used by many classes to write log messages to the App's log file.

The following five components plays an important role at the architecture level:

- `UI`: The User Interface of the App.
- `Logic`: Includes 3 types of executors: the Command Executor, the Program Submission Executor, and the Question Builder Executor,.
- `Model`: Holds the data of the App in-memory.
- `Storage`: Reads data from, and writes data to, the hard disk.
- `TextExecutor`: Compile the user program and run it against test cases. Output result.

Each of the six components:

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

Storage component

Overview

There are 2 main storage components found in the architecture of Duke Academy. They are `QuestionBankStorage` and `NoteBankStorage`. Each storage component serves as a **facade** for the basic operations by the application with regards to **Commands** and **Notes**.

The `Storage` component,

- can save `UserPref` objects in json format and read it back.
- can save the Duke Academy question bank in json format and read it back.
- can save the notes and read it back.

Implementation

The storage components are interfaces so their implementation can be changed easily.

`JsonAdaptedQuestion` serves as a good starting point to understand the implementation.

JsonAdaptedQuestion:

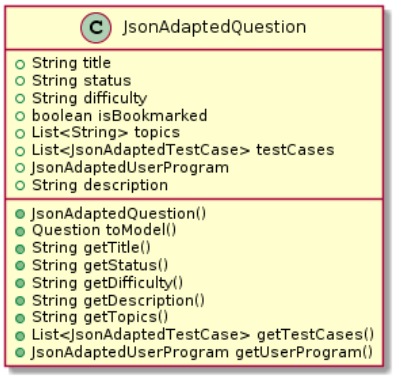


Figure 2. Class diagram of the JsonAdaptedQuestion class

- It contains all the necessary attributes for a question, including title, completion status, difficulty level, isBookmarked, topics belonged to, testCases, userProgram attempted and question description.
- Compared with a standard `question` object, this `JsonAdaptedQuestion` object has processed its attributes to be compatible with json format. That is, this object can be directly serialized to and de-serialized from json files.
- The constructor is used to serialize the `Question` object using the `@JsonProperty` notation.
- The `@JsonProperty` is also able to deserialize strings obtained from json files. The `toModel()` function is then used to construct and return a new `Question` object using attributes it obtained using the getter methods.

Structure

We would hereby use `QuestionBankStorage` to illustrate the implementation.

The standard implementation of the `QuestionBankStorage` is the `JsonSerializableStandardQuestionBank` class.

Overview:

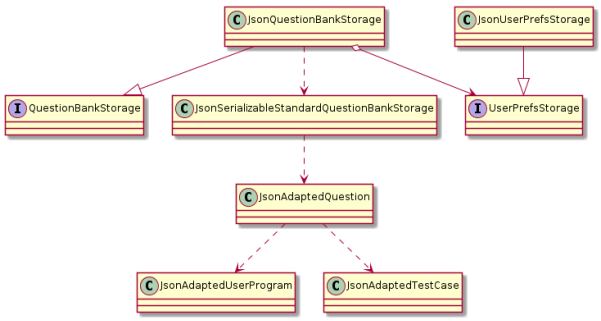


Figure 3. Overall Structure of the Storage Component for Questions

The `QuestionBankStorage` manages tasks such as `saveQuestionBank()` or `readQuestionBank()`.

The `Question` object is associated to the rest of the files in the same package as follows:

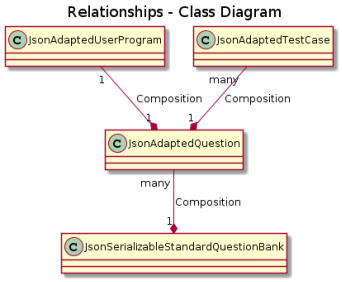


Figure 4. Class Diagram: Compositions that make up the QuestionBank being stored

Problem Statement Panel

Since the problem description cannot be viewed fully from the question list, we introduced a new problem description panel. As shown on Figure 5, when type `view [id]`, the panel updates to display all the additional information a question has to provide.

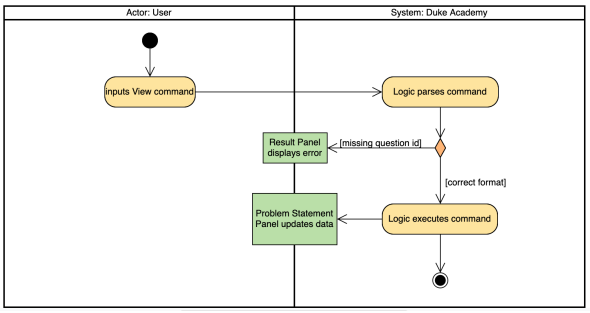


Figure 5. Activity Diagram on View command

General Procedure of Command execution:

- User types `view [id]` in the command box. The `MainApp` class receives the input, calls the `commandLogic` class to executes the command and returns an `CommandResult` object.

Implementation Details

The implementation details are narrated following user cases as follows:

1. When the `view` command is executed, it switches the pane to "Question" by calling the `applicationState` object's `setCurrentActivity(Activity pane)` method.
2. It then updates the `questionLogic` object of the current question being viewed by calling its `selectQuestion(int id)` method.
3. The `ProblemStatementPanel` UI utilizes a JavaFx `@FXML` property called `TextArea` to display information.
4. Every time when the `QuestionPage` pane or the `Workspace` pane is displayed, their respective UI controller checks whether `questionLogic` refers to a question that is currently of interest by the user. If positive, they will call the `ProblemStatementPanel` controller's `setProblemStatement(String problemStatement)` to display data.

As such, the functionality required by problem display panel is well covered.

Future Improvement

In version 2.0, we aim to achieve rich text display of problem description. It can be in MarkDown format, containing LaTeX formulas, images, URL links, coloured text, formatted code snippet, etc.

Design Considerations

This is my design consideration on how to update the problem statement panel when a `view` command is entered.

- Alternative 1 (current choice): Use `questionLogic` to track the current `Question` being viewed by the user. `UI` components can access attributes in `Logic` components and display them.
    - Pros: More OOP. It is clear that `UI` does not interfere with the tasks responsible by the `Logic` component. There is less coupling, making the code easier to understand and undertake testing.
    - Cons: Complicates the code base by abstracting another attribute onto the `QuestionLogic` class.
- Alternative 2 : Stores the `Problem Description` content as a String temporarily. Use `MainWindow` controller to check whether the command generated is a `view` command. If yes, force the `ProblemStatementPanel` to update.
    - Pros: Easy to implement based on the existing code base.
    - Cons: It breaks OOP's open and close principle. It mixed up `UI` class with `Logic` class.

Appendix A: Product Scope

Target user profile:

- has a need to practice a lot of algorithm / data structure problems with the following conditions satisfied:

    instant assessment of answers submitted

    practices under timed conditions

automatic progress checker

personal tutor to recommend problems with suitable difficulties and topics

fun in learning with achievement badges to unlock

no WiFi needed

Appendix B: Use Cases

(For all use cases below, the **System** is the `Duke Academy` and the **Actor** is the `user`).

Use case: UC01 Set questions

MSS

1. User requests to input problem sets.
2. Duke Academy requires a file path.
3. User select file path.
4. Duke Academy imports the problem sets and prompts success message.

   Use case ends.

Extensions

4a. The input format is incorrect.

Duke Academy reports wrong format error. Duke Academy resumes at step 3.

Appendix C: Non Functional Requirements

1. Time taken to assess the submitted programmes should not exceed 3 minutes.
2. Data not intended for disclosure should be encrypted with minimum needs so that it's protected from direct access.
3. Should not take more than 5 seconds to load the initial screen.
4. If interrupted, the program should provide an auto-saved version and prompt for restore when the app opens next time.