SIMPLE QUESTION ANSWERING SYSTEM

SYSTEM DESIGN AND IMPLEMENTATION

An Truong
truan004@mymail.unisa.edu.au
110313636

SIMPLE QUESTION ANSWERING SYSTEM

Contents

CLASS IMPLEMENATION DESIGN

Abstract

This assignment is a simple design development of a question answering system stimulating a basic conversation using natural language/human language (in English). The system uses a JSON dataset of pair of questions and corresponding answers to receive question input from user application, compute the matching score using Similarity Calculation (*Jaccard Similarity Score* formula), then identify the appropriate output to response and display into the interface.

The Use Cases of the system are simple as the design which ensures the basic functionalities and meets the non-functional requirements of a question answering system.

Functional Requirements

- UC01 Welcome:
  The system should start and displays a welcome message in the interface
- UC02 Ask Question:
  The system enables user to enter a single sentence question in plain English into the system. The system should log the question of user in the Asked Question Store, search for the answer based on matching engine and determine the most appropriate message to display in the application (either corresponding answers or relevant notifications).
- UC03 Quit:
  The system should quit/ terminate the execution with the goodbye message when user enter a blank line.

Other Requirements

- Text Interactive Console Client (Terminal on MacOS/Linux and Command Prompt on MS Windows).
- 'asked_questions_log.txt' stores logged text questions.
- 'faq.json' stores the catalogue of questions bank and its corresponding answers
- Use Similarity Calculation for matching engine

Assumptions

This decomposition has been created based on the volatility assumptions:

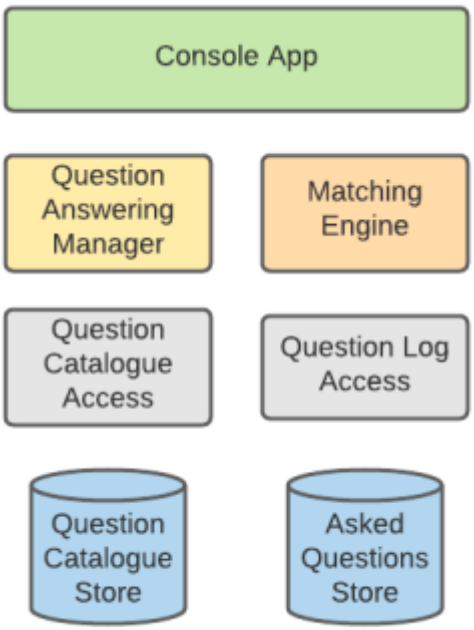
- More sophisticated matching engines may be developed
- Question catalogue store and asked questions store might use upgrade or different implementation approach to database technologies.
- The system is plain English text base with updatable question catalogue bank set in English
- The data store can be classified into different branchs of language (might be availble for update functionalities in future)

Component Decomposition

According to the case study of the system, below is the fundamental design of the system, component decomposition, that outlined the skeleton of the required functionalities and provided a layered architechture of the system. The design consists of 7 componentd, divided into 4 layers which are specified in the table.

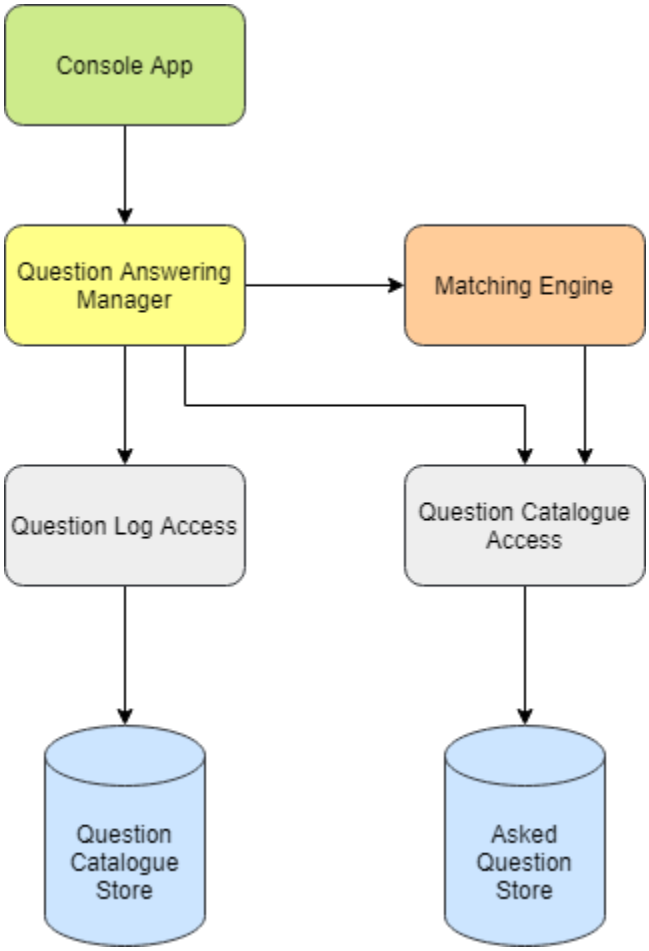
Component Responsibilites

Layer	Component	Responsibilies
Interface	Console App	Interact with user application Prompt for user input Display system response
Manager	Question Answering Manager	Answer the question via the result of matching engine Log user input into asked question (as long as it is not NULL)
Engine	Matching Engine	Compute the similarity among question phrases using *Jaccard Similarity Score* formula
Resource Accessor	Question Catalogue Access	Access and retrieve appropriate response
	Question Log Access	Log asked questions
Database	Question Catalogue Store	Store questions catalogue
	Asked Question Store	Store asked questions



Communication Diagram

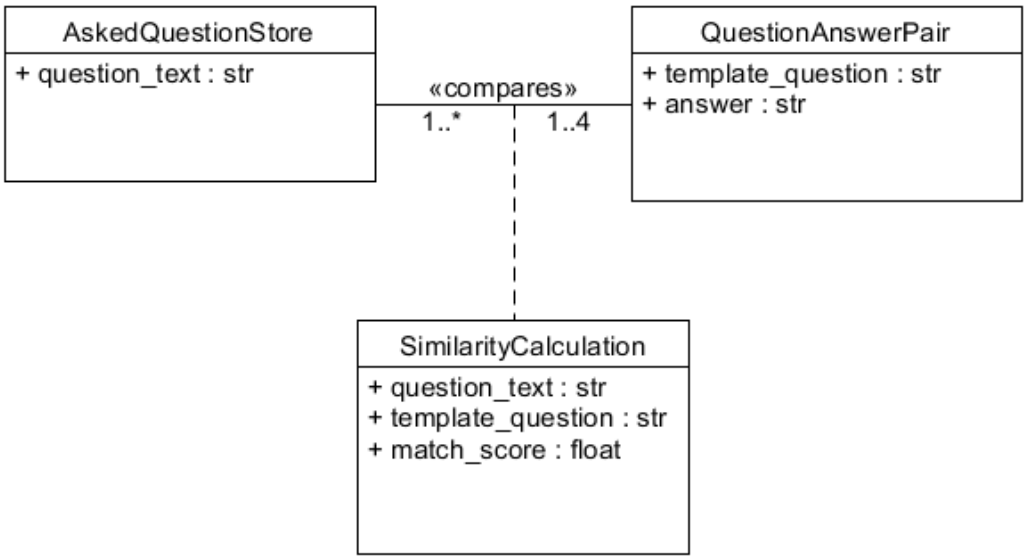
Communication Diagram is an illustration of how each provided component associate with others and their responsibilites within the basis flow of the system. Through the implementation of Question Answering Manager, the Console App is able to use the calculation result of Matching Engine and interact with the data of question-answer pairs to response to the request of clients.



Domain Model

It is clear that the system includes 2 seperated database which are Question Catalogue Store and Asked Question Store. The system stores the database of question-answer catalogue provided by JSON format file in Question Catalogue Store and keep track of the list of question made by user in Asked Question Store. The Similarity Calculation class plays as a association class that provides connection between 2 data stores of the system.

Data class	Attributes	Responsibilities
AskedQuestionStore	+ question_text : str	Store and log user question input
QuestionAnswerPair	+ template_question : str + answer : str	Store and retrieve data of question-answer bank
SimilarityCalculation	+ question_text : str + template_question : str + match_score : float	Calculate the match score between the most similar question and user input



Implementation Design & Reflection

Basic Design

According to the given Component Decomposition Diagram, the Communication Diagram and Domain Model, a simple design covering all of the components is created that are implemented by 7 seperated classes with 7 different purposes. This design stimulates the operation of component decomposition design that demonstrates the responsibility of each component within each corresponding class.

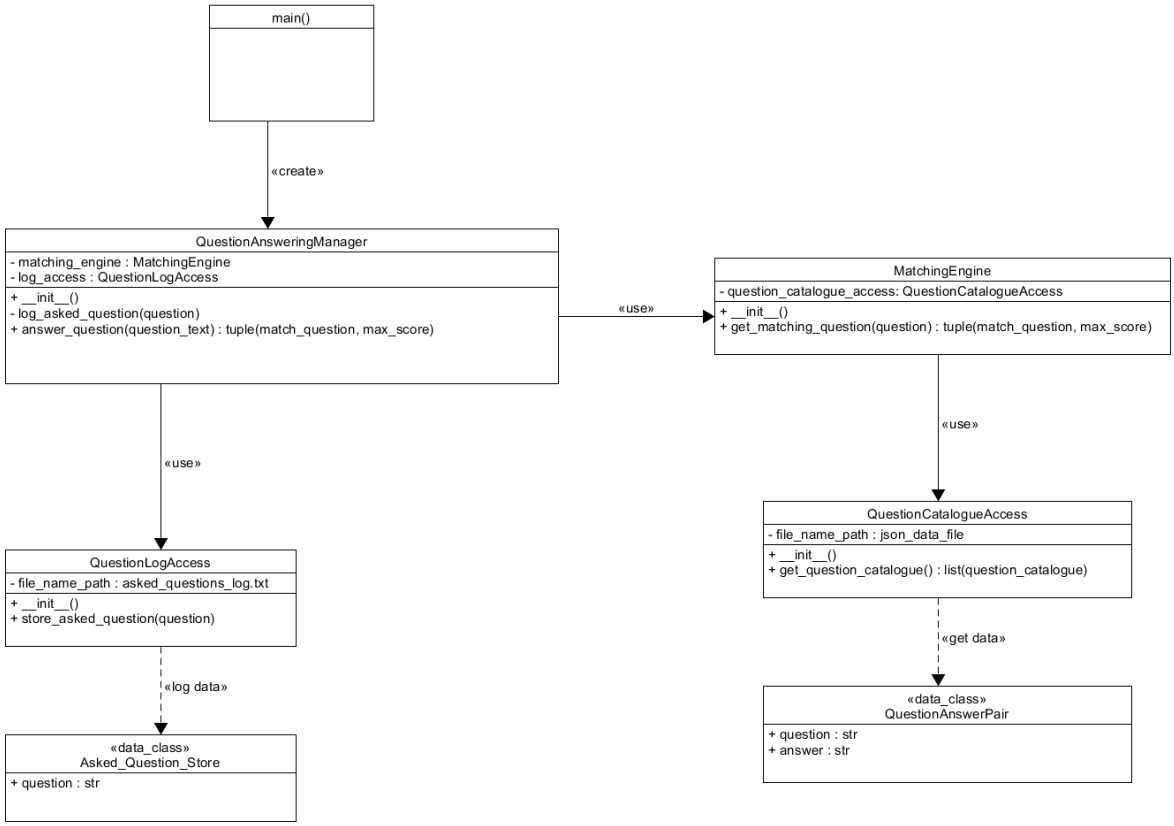
Interaction between classes and components

Class	Cover Components	Functionalities
main()	Console App	Call manager and activate all classes of system Interacr to the interface application
QuestionAnsweringManager	Question Answering Manager	Call MatchingEngine and QuestionLogAccess Get answer/response and log data
MatchingEngine	Matching Engine	Implement Similarity Calculation Engine Get the Similarity Calculation (match score)
QuestionCatalogueAccess	Question Catalogue Access	Get and retrieve relevant question catalogue
QuestionLogAccess	Question Log Access	Log user input
QuestionAnswerPair	Question Catalogue Store	Store question catalogue
AskedQuestionStore	Asked Question Store	Store asked questions

Association among classes (information exchange)

Class	Information needed	Class have information
main()	question_catalogue_access matching_engine question_log_access manager	QuestionCatalogueAccess MatchingEngine QuestionLogAccess QuestionAnsweringManager
QuestionAnsweringManager	match_question, max_score	MatchingEngine
MatchingEngine	question_catalogue_access	QuestionCatalogueAccess
QuestionCatalogueAccess	file_name_path	main()
QuestionLogAccess	file_name_path question	main()
QuestionAnswerPair		QuestionAnswerPair(question, answer)
AskedQuestionStore		AskedQuestionStore(question)

Implementation Design



Design Evaluation

Advantages

Overall, based on the UML implementation design, here are some positive sides of this design:

- The naming is consistant and clear (snake_case for all attributes, variables and functions: camelCase for classes name)

- Clear and reasonable data visibility (Data hiding): take advantage of public, protected and private data in Python to prevent information leakage incidents.
- Cover all of the components and their responsibilities
- All variables concerning on calling clasess and data access are set to be private to avoid information leakage
- AskedQuestionStore and QuestionAnswerPair are separated into 2 data stores to allow changes in the implementation of database technologies

Disadvantages

According to 2 tables of this design analysis, there are some issues that can be indentified through this design.

- It seems like each of the components owns a seperated class. However, as each component is allocated with more than one tasks, therfore, each of the class within this design will suffer from the possibility of multi-tasks or hard to extend, which results to a bad design. For instance, MatchingEngine is in charge of providing formula for manager to use. However, within this design, MatchingEngine has to change significantly the logic implementation of the get() functions if there is a development of engine algorithms.
- It should not be in the situation that the main and the user interface share the same functionality as the main class should just call and activate the system only, which caused the sophiticated dependency among classes.

Improve Design by applying design principles

Solution

To improve and optimise the effiency of the design, it is essential to apply the principles of system design named

+ Loose coupling high cohesion: It is best to have the high cohesive class as high cohesion means all elements within a class are supporting the same purpose that reduces complexity and increases maintainability and reusability of the system. Therefore, there should be a high cohesion maintained throughout a system design.

+ SOLID Principles

In order to apply these mentioned principles into the system, the best way is to modify the draft and add some needed classes to divide the responsibilites for each class equally.

To improve the outlined problems, it is essential to validate and upgrade the design by applying the principles of design patterns. According to the issue we are having in the recent design, this update design has been applied:

- *Strategy pattern*
  The InteractiveConsoleClient has been added to the design as the solution to specify the activation functionality of main class and create the scope of code that is responsible for user interface interaction, receiving and responding to user request.
- *Abstract Factory & Factory pattern*
  The design has been updated with the abstract factory as interface that handles the creation and the corresponding factories that manages the method operations and the various types of objects. The updating is applied to the engine of the system, MatchingEngine, which might need to be expand or improved its logical implementation in the future. The Abstract Factory is also responsible for connecting the initializing class and execution class (get result data from the implementation).

- There is a low dependency among classes and each class is only responsible for one task/function. (*Single responsibility principle*)
- Code dependencies rely on stable abstractions, avoid concrete elements, replacable or changeable without breaking the program (*Dependency inversion principle and Liskov substitution principle*)

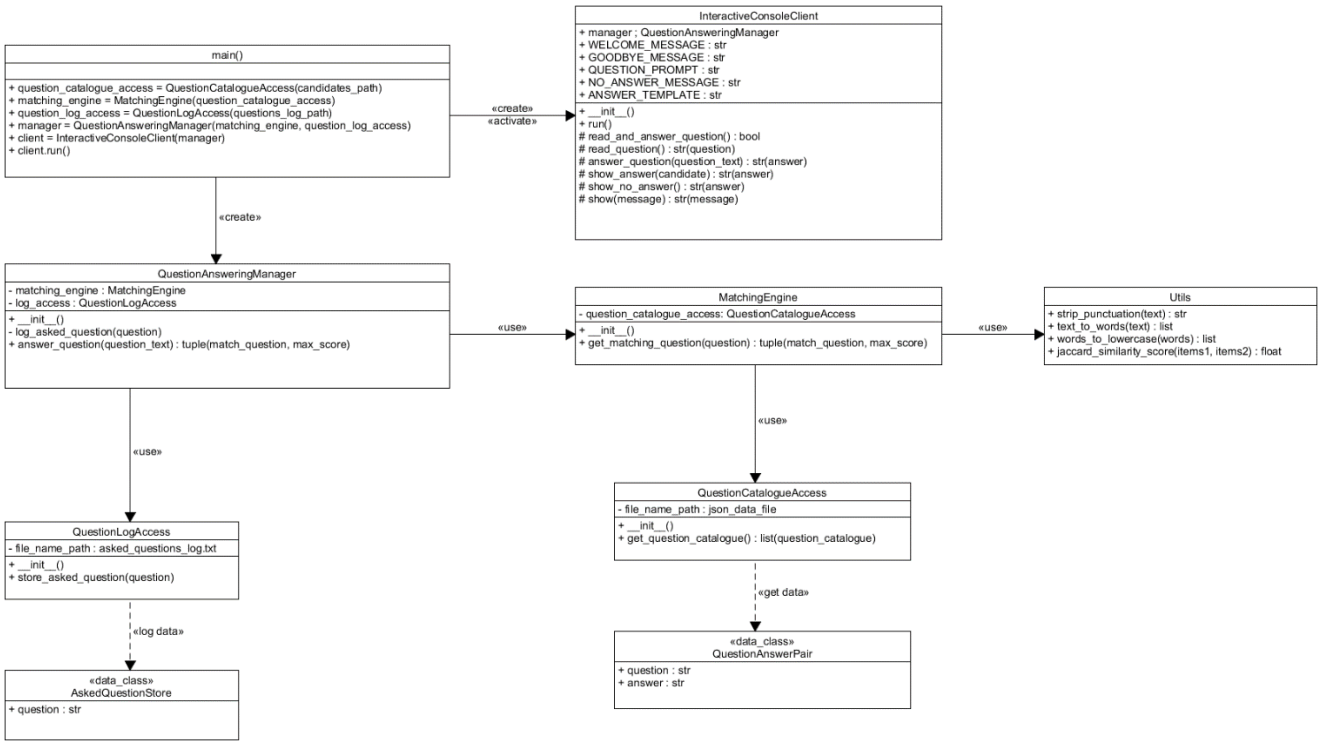
Interaction between classes and components

Class	Cover Components	Functionalities
main()	Console App	Call manager and activate all classes of system
InteractiveConsoleClient	Console App	Interacr to the interface application
QuestionAnsweringManager	Question Answering Manager	Call MatchingEngine and QuestionLogAccess Get answer/response and log data
MatchingEngine	Matching Engine	Get the Similarity Calculation (match score)
Utils	Matching Engine	Implement Similarity Calculation Engine
QuestionCatalogueAccess	Question Catalogue Access	Get and retrieve relevant question catalogue
QuestionLogAccess	Question Log Access	Log user input
QuestionAnswerPair	Question Catalogue Store	Store question catalogue
AskedQuestionStore	Asked Question Store	Store asked questions

Association among classes (information exchange)

Class	Information needed	Class have information
main()	question_catalogue_access matching_engine question_log_access manager	QuestionCatalogueAccess MatchingEngine QuestionLogAccess QuestionAnsweringManager
QuestionAnsweringManager	match_question, max_score	MatchingEngine
MatchingEngine	question_catalogue_access	QuestionCatalogueAccess
QuestionCatalogueAccess	file_name_path	main()
QuestionLogAccess	file_name_path question	main()
InteractiveConsoleClient	manager	QuestionAnsweringManager
Utils		
QuestionAnswerPair		QuestionAnswerPair(question, answer)
AskedQuestionStore		AskedQuestionStore(question)

Final Design



Quality Assurance

A good code is a human readable and easy to change or maintain during the development process. After significant improvement within the design by apllying the design principles, there are characteristics of the design that makes it become the good one.

- Coding conventions: good naming (meaningful, intention revealing names and consistant), divided into small scopes of code (simple small functions) and no duplication
- Simplifize the design with one responsibility each scope of function within a class
- Enable code and design to be expanded or changable in the future: Matching engine is responsible for code execution e.g computing matching score, while Utils is in charge of code logic implementation and formular/algolrithms initialization. This allows component to develop without cracking the whole system
- Meaningful distinctions
- Add reasonable comment and self-explanatory code

Testing

There is one test for each of the classes that both pytest and uniitest are used in this assignment to veryfy and validate the efficency of the code. This testing used test fixtures to test the manager behaviour, Mock test for the calling functions validation of manager class, 3 tests (Test Stub) for the cases at a unit of functionalitity (per class and function) and a general test for the module.The test is named by the testing purpose of them (intention revealing names), for instance, test_MatchingEngine_get_matching_question() is a test for checking the result of matching question and matching score within MatchingEngine class.

Refactoring

- Using pylint to check for the naming conventions, formatting (spaces, indent, line breaks, line length), parenthesis (brace) placement and comment formatting.
- Obtaining clean code and improve programming performance.