SIEMENS Internship

Contents

[Problem 1: Library Management System 2](#_Toc154051017)

[Task 1: Class diagram 2](#_Toc154051018)

[Task 2: Database diagram: 3](#_Toc154051019)

[Problem 2: Online Quiz System 4](#_Toc154051020)

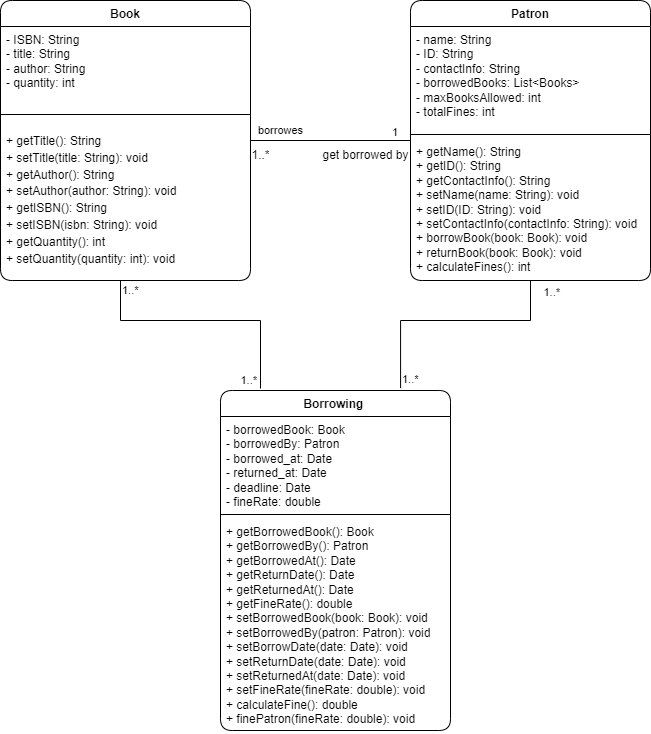
[Logical design 4](#_Toc154051021)

[Algorithm implementation 5](#_Toc154051022)

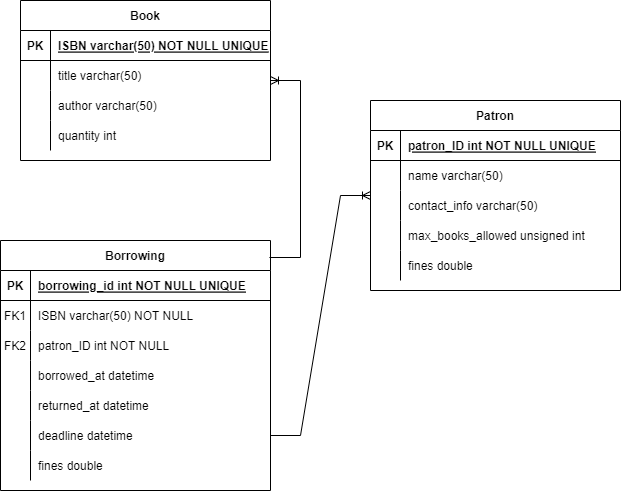
[Class and Database Representation 6](#_Toc154051023)

# Problem 1: Library Management System

## Task 1: Class diagram



## Task 2: Database diagram:



# Problem 2: Online Quiz System

## Logical design

There are two approaches in the matter of this problem, depending on the data storage solution: using a SQL solution or a NoSQL solution. I will approach both solutions below.

Using a SQL solution, I would implement a database containing the questions and their attributes in a table and additionally a table of users and their scores. Creating a table for the users could be useful in the future implementation of an authentication system and a history of scores achieved on this quiz for each user.

In a NoSQL context, the implementation would be similar, only that a service like MongoDB will be used and all the data will be found in a JSON file.

Every question entry will have a unique ID, the question, the answers and the correct one.

The user progress will be measured by the number of questions they got right out of the fifty questions.

The system will generate the questions without repetition by always “having in mind” the questions it already asked. When a random number gets generated, the program checks if it already asked the question with the id equal to the number generated. If the question is new, the system will display the question and mark that number as being already used. If the random number has been generated before, the program will generate a new one.

Pseudocode:

Initialize an empty array called excludedNumbers

Function generateQuestion()

do{

randomNumber = generateRandomNumber()

If number not in excludedNumbers:

displayQuestionWithID(randomNumber)

excludedNumbers.add(randomNumber)

} while excludedNumbers contains randomNumber

Function generateRandomNumber()

Generates a number in the range of questions ID.

The fact that the number corresponds to a question ID is guaranteed.

The score of the user will be calculated after every question depending on if the answer is right or wrong. The score will be shown after the quiz is done.

-Pseudocode:

Initialize a variable called score with the value 0.

generateRandomQuestion();

Function getAnswer():

returns the choice of the user

variable response = getAnswer();

Function validateAnswer():{

If response is equal to (randomQuestionGeneratedAbove.answer):

score = score + 1

checkDone()

}

Function checkDone():{

If excludedNumbers.length = dataObject.questions.length:

redirectUser()

sendScore()

Else generateQuestion()

}

## Algorithm implementation

You will find the implementation in the attached folder, in the quiz.js file.

Additionally, I implemented the whole application :) with extra functionalities in React and deployed it on GitHub Pages (not mobile-friendly **yet**).

Link to repository: <https://github.com/MoldoveanuAlex/Quiz-App>

Link to deployed app: <https://moldoveanualex.github.io/Quiz-App/>

In the React implementation of the application I used a different approach to the randomization of the questions logic, due to the React re-rendering aspect. When the Quiz component loads, it fetches all the questions, selects a random question and it passes it to the Question component, and then deletes the question from the fetched object. This process repeats until the length of the fetched questions array is zero.

## Class and Database Representation

Class representation:

The first class in this context would be Question. The attributes of a question are questionId(int), the question(String), the choices(array of String) and correct answer(String) or the index of the correct answer(int). Up next is the User class with the attributes userID(int), username(String). And the last class is Quiz whit the attributes questions (array of Questions), the user(User) that is doing the quiz and the score of the quiz. Additionally, we could store two timestamps: startedAt(timestamp) and finishedAt(timestamp) which we could use to create a quiz results history for each user or to analyze the average time of completion for each user or to use it as a metric for the entire application.

The relationships between classes are quite simple. A Quiz has multiple Questions, meaning that the class Quiz aggregates the class Question. A quiz is associated with a User.

Database Schema:

For a relational database I would design the tables as follows:

* Questions Table with the attributes question\_id as Primary Key, question\_text, choices, and correct answer.
* Users Table with the attributes: user\_id as primary key and username.
* Quizzes Table with the attributes: quiz\_id as primary key, user\_id as foreign key referencing the User table), score and startedAt and finishedAt timestamps.
* Quiz\_Question Table to model the many-to-many relationship having the attributes: quiz\_id as foreign key referencing the Quiz table, question\_id as foreign key referencing the Questions table, user\_choice and a Boolean is\_answered indicating if the question has been answered or skipped.

For a non-relational database, such as MongoDB the structure would differ. This approach allows for more flexibility in storing data. In the partial code I used a JSON file. The document structure would contain the following collections:

* Questions: {“\_id:” , “question\_text”: , “choices”:[] , “correctAnswerIndex”: }
* Users: {“\_id”: , “username”: }
* Quizzes: {“\_id”: , “user\_id”: , “questions”: [] , “score”: }

In document-based databases relationships are represented by using references between documents. For example, in the Quizzes collection there is an array of questions with additional information like whether the question is answered or not and the answer given by the user.

Dataflow during the test:

The user starts a new quiz by pressing the “Start quiz” button. A new Quiz is created in the Quizzes collection. The user\_id is set, and the score is zero.

The questions get fetched and added to the “questions” array in the Quiz object.

The first question along with the four possible answers is displayed to the user.

The user selects an option.

System receives the user’s choice and updates the Quiz object: is\_answered = true and the answer given gets registered.

System checks if the answer given is correct and updates the score if necessary.

Other questions get displayed.

When all the questions have been displayed, the quiz ends.

The score will be displayed.

If this feature is implemented, the user’s history or statistics will be modified.

*Summary*

User -> starts -> Quiz

Quiz -> fetches -> Questions

(start loop)

Quiz -> displays -> Question

User -> answers -> Question

Quiz -> validates -> answer

Quiz -> updates -> score

(end loop)

Quiz -> displays -> score

Quiz -> ends