# CSIA
# Solution Overview

**Hakobyan Aram**

March 12, 2018

## Contents



## List of Figures

## List of Tables

# 1 Use Cases

Use cases describe the functionality of the system that yields an observable value to the user or other actors involved. They omit any technical details and present the interaction between the system, the primary actor, i.e. the user and secondary actors, e.g. the remote database.

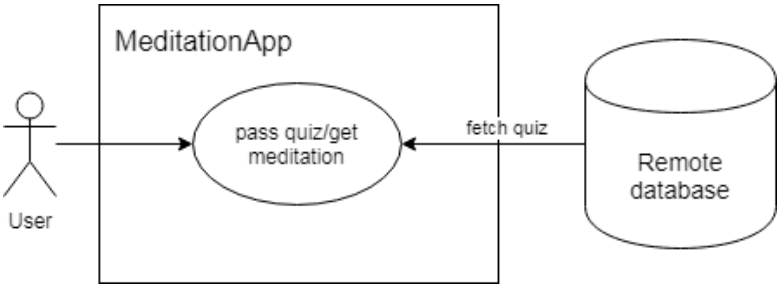


Figure 1: Use Case Diagram

Table 1: Use Case 1 - Complete the quiz

| **Action** | **Complete the quiz** |
|---|---|
| Summary | User provides answers to quiz questions and receives a link to a meditation video based on the answers to the quiz. |
| Description | This use case describes how a user can receive a highly specific meditation session based on the responses to the quiz questions provided by the application. |
| Actors | User, Remote database |
| Pre-Conditions | User has opened the application and the application is connected to the internet. |
| Success Guarantee | The user receives a link to a meditation video based on the answers provided to the quiz questions. |
| Main Success Flow | 1. System fetches quiz questions from the remote database.<br>2. User answers to all questions provided in the quiz.<br>3. System identifies the most suitable meditation session for the user based on the quiz outcome and presents the link to the user |
| Exceptions | Quiz questions are not presented if the application is not connected to the internet |
| Post-Conditions | The application will present a view with the meditation description and a web link to the video. |

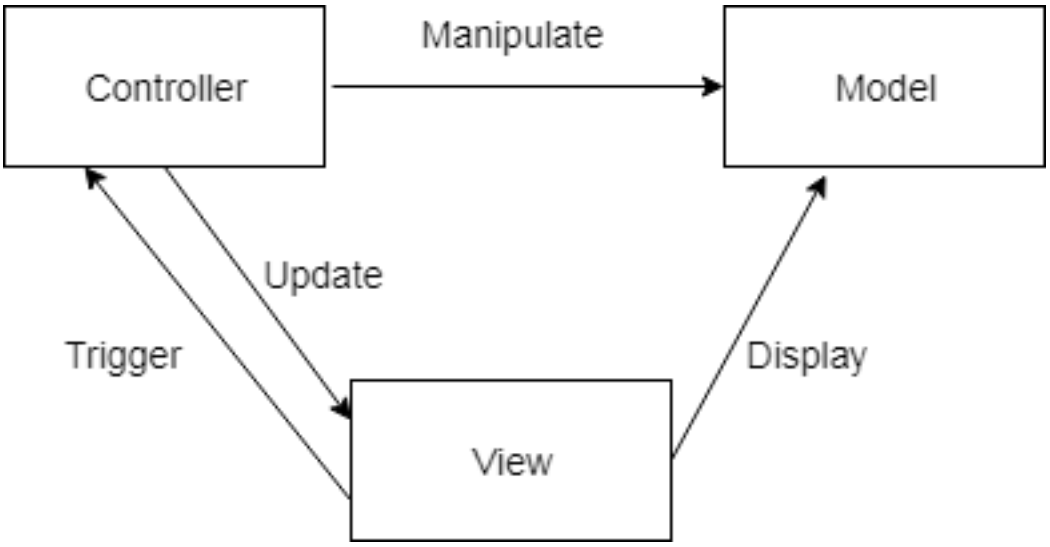


Figure 2: Architecture Diagram

## 2 Architecture

The application utilizes the Model-View-Controller (MVC) architectural pattern. This approach allows separation of the representation of information from its display and allows efficient code reuse. The model component represent the information used in the application, e.g. the quiz or the meditation. The controller classes manipulate the models and populate corresponding view components, which then render or display the model components. The controllers also handle user input and transform them into appropriate software calls that manipulate update the views or manipulate the models through a specific service component.

## 3 System software diagrams

The diagram of the main sequence is depicted below. As soon as the application runs, the MediationApp component, which is responsible for coordination of controllers, fetches the Quiz object using the QuizSerivce component. Then, it proceeds to update the QuizController with the fetched object.
Now, the QuizController is displaying the questions and the user can proceed to answer them. The user can switch back and forth between questions, however all the questions must be answered. Once all the questions are answered, the user will be able to submit the quiz. The MeditationApp passes the results of the quiz to the MeditationService component, which, based on the answers to the questions, determines the most appropriate meditation and returns it to the MeditationApp component.
Finally, the MeditationApp component passes the Meditation object to the QuizController, which displays it for the user.

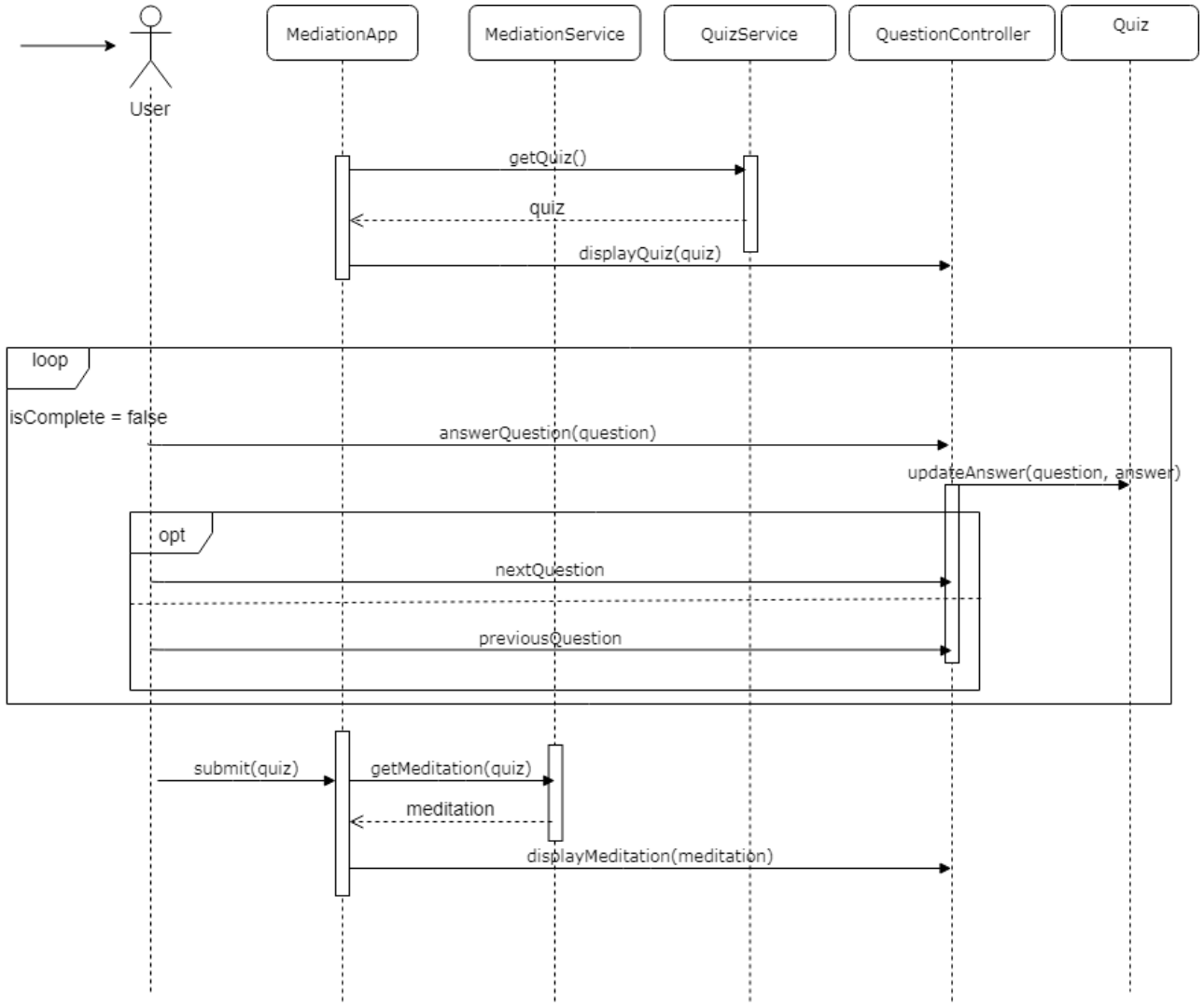


Figure 3: Software sequence diagram

Below is a flow-chart representation of the described sequence.

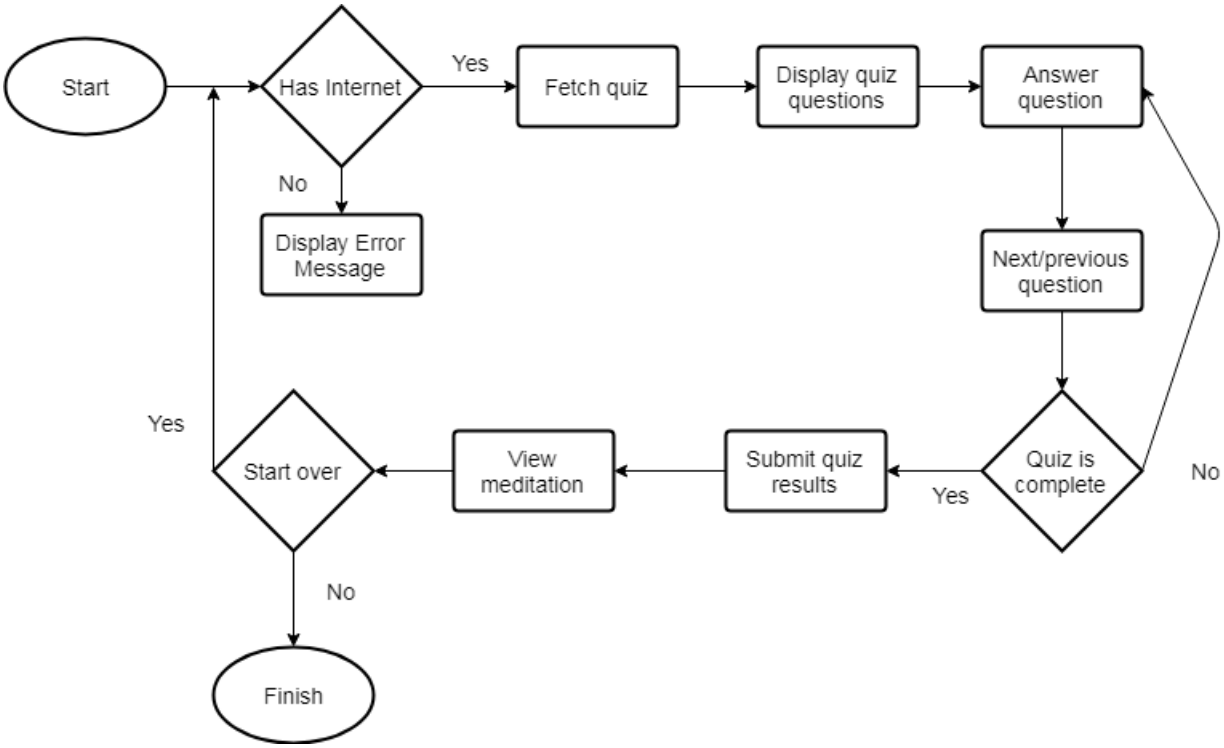


Figure 4: Process flowchart

# 4 Test plan

| Test plan | |
|---|---|
| Action test | Testing approach and result |
| The application runs successfully | Run the application, make sure the main window is displayed properly |
| The window has dynamic layout | Resize the window, make sure the GUI components dynamically change their size and position |
| The application works correctly without the internet connection | Run the application while not connected to the internet. Make sure the app displays an error message indicating the absence of connection |
| The application fetches the quiz questions | Run the application while connected to the internet. Make sure the window displays the first quiz question |
| The application displays the number of quiz questions | Run the application while connected to the internet, make sure the window displays the index of the current question and the total number of quiz questions next to it |
| The "Previous" button is disabled when the first question is displayed | Run the application, when the first question is displayed, verify that the "Previous" button is disabled. |
| The user cannot navigate to the next question while the current one is not answered | Verify that the "Next" button is disabled if the current question is unanswered. Answer the current question and check that the "Next" button is now active |
| The "Submit" button is disabled while there are still unanswered questions | Run the application, check that the "Submit" button is disabled. Answer the questions one by one, making sure the button is still disabled. When answering the last question, check that the button becomes active |
| The "Next" button is disabled when on the last question | Answer all questions and reach the last one. Verify that the "Next" button is not active |
| The app saves the answers to previous questions | Answer the questions and navigate back and forth, verifying that the answers to the questions are saved |
| The app displays a meditation description and a web url when submitting the quiz | Complete all the questions, click the submit button, make sure that a window with meditation information and a clickable web link is displayed |
| The app should allow restarting the quiz when finished | Complete all the questions and submit the quiz. Check that the window with the meditation contains a button that allows the user to restart the quiz |

# 5 Reference


- Craig Larman - Applying UML and Patterns
- Amazon DynamoDB developer guide
- Dagger2 developer guide
- JavaFX developer guide