Project Architecture and Design

for

BoardCast

Prepared by TheTeamFromNTU

University of Waterloo

CS 446, ECE 452, CS 646

19/03/2018

# Introduction (Metadata)

## Team Members

|  |  |
| --- | --- |
| **Name** | **Quest ID** |
| Elgin Lee Wei Sheng | 20753953 |
| Norman Yeo Guo Kuan | 20755357 |
| Lee Kian Lon | 20755364 |
| Harold Lim Jie Yu | 20755365 |

## Purpose

The proposed system is BoardCast, a mobile application which serves to aid the evolution of smart classrooms using a peer-to-peer infrastructure to reduce the cost of setup and complexity of deployment.

# Architecture

The below diagram shows a physical representation of how our system would be.

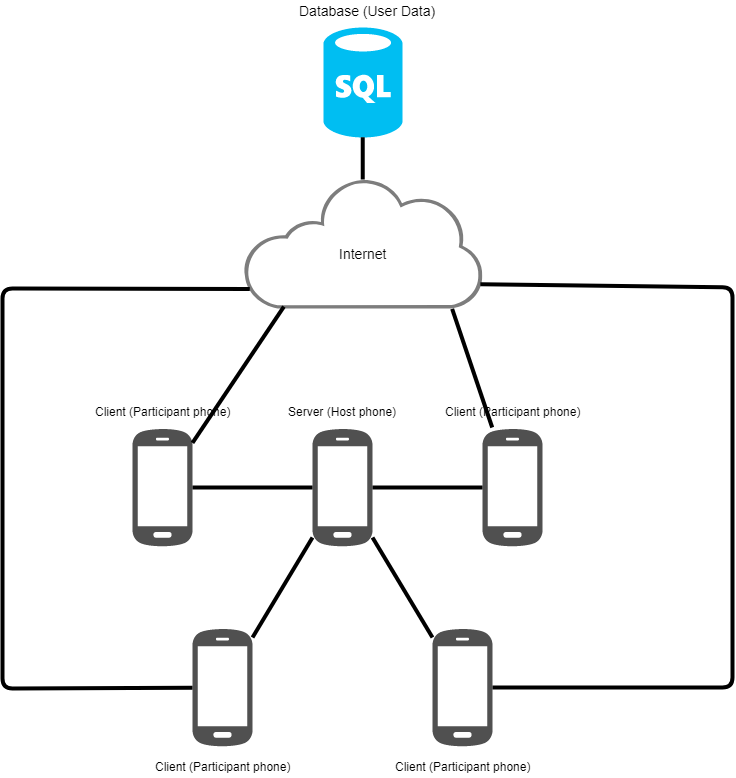


Figure 1 - Physical System Diagram

A few key points to note:

1. The polling and polling server are all local and do not require the database in the cloud to function
2. The database in the cloud is solely used to store user data (past scores and reports) and enhance the user experience but it is not essential to the core application (polling)
3. The database was added due to the pivot given

## Communication and Component Diagram

Figure 3 - Client Component Diagram

Figure 2 - Communication Diagram

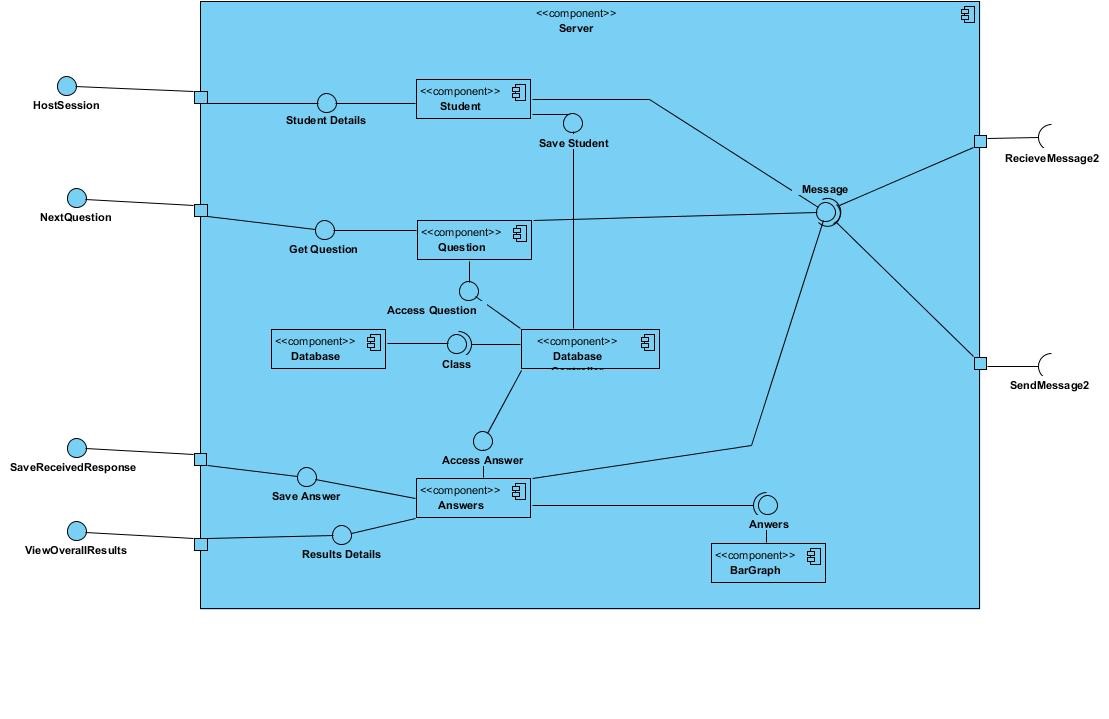
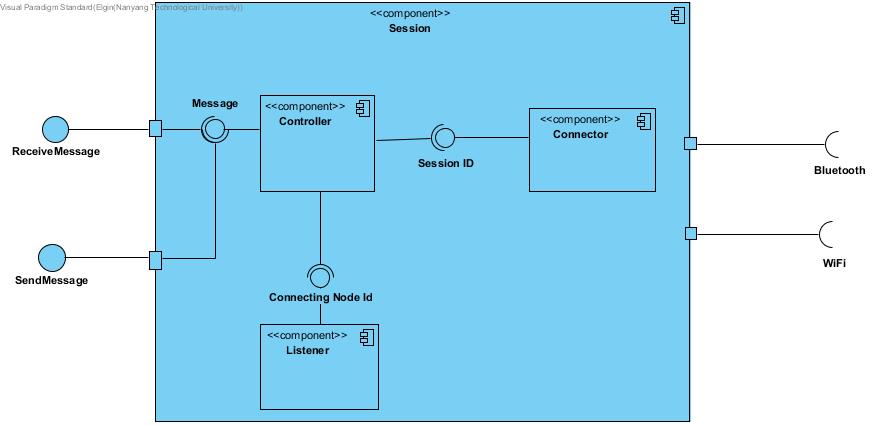


Figure 4 - Server Component Diagram

Figure 5 - Shared Component Diagram

## Function Requirements

The following section would describe how a client-server architecture would support our functional requirements.

Core Features

1. Hosting and Connection
   * The hosting device would create a session and broadcast it to all clients in the proximity
   * The hosting device would have most of the processing logic and this is in line with how the client server model works where the server processes requests (answers to questions in this case) and returns a result (correct or wrong)
   * The client devices are unable to see each other’s connection and data that is being sent to the server. Each client must be isolated so that answers cannot be eavesdropped on or altered. This is in line with how the clients do not communicate with each other in a client-server architecture.
   * The clients are duplicates of each other and all want a common function (process their answers and get the result) which the server would provide.
   * We acknowledge that the hosting device is a single point of failure, but this is a tradeoff we must make due to the nature of polling. We have taken mitigations such as session restoration in event of crashes.
2. Polling
   * The application requires evolvability where the server can be updated to change the way it processes answers (e.g. data analytics of answers) without affecting clients and client operations
   * The application requires scalability in the sense that many new clients can be added to the system by simply cloning the previous client. Though there is a physical limitation of resources on the hosting device, we have considered that the Nearby API we are using can only handle 100 client connections and there are sufficient resources on modern mobile phones to handle 100 clients.
3. Results Export
   * The remote database server would handle all export requests and save it on persistent storage so that the client can retrieve the same data even if he is logging in from a different device. This is in line with the usefulness of a client-server architecture.

## Non-Functional Requirements

The following section would describe how a client-server architecture would support our non-functional requirements.

1. Efficiency
   * The application should respond to user requests instantaneously. This is technically a problem as the server is usually the performance bottleneck. We have taken note of this and tried to reduce the amount of work the server must do and outsourced more work to the client. For example, the server would only send the most minimal question content, the client would then parse the question into the correct format.
   * The client server architecture allows the client and server to focus on its respective tasks. This allows the application to streamline the processes and increase efficiency overall.
2. Reliability
   * The application should be able to recover in the case of intermittent connections or dropped connections to the host device. As previously mentioned, the server is a single point of failure. We have solved the reliability issue by saving the state of the quiz every time the host goes to the next question. The session can be recreated in the event of a crash.
3. Portability
   * The student can view his results and reports from different devices. This is supported by the client server architecture as the client can be on any type of device and still retrieve the same information from the server, if he is authenticated.
4. Security
   * Other students are unable to eavesdrop on answers and it must seem like they are the only one connected to the server. This is a constraint of the client server architecture that fits very well into our use case. The clients do not communicate with one another and all communication goes through the server.

# Design

## STRATEGY PATTERN

By using the strategy pattern in question creation, I was able achieve modularity, evolvability, extensibility, and scalability.

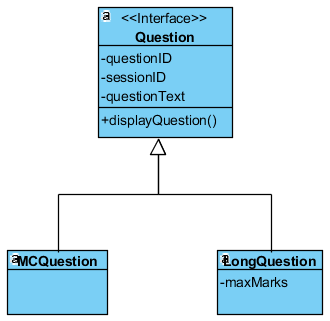


Figure 6 - Strategy Pattern Class Diagram

In the figure above, we can see that the Question interface is implemented by the concrete question types MCQuestion (multiple-choice question) and LongQuestion (long answer question). By having an abstract function displayQuestion(), our calling class only has to call the displayQuestion() function from the abstract object to use its implementation in the concrete class.

For example, when the UI wants to display all questions, it does not need to know what type of question is being displayed. It simply iterates through the list of abstract questions and calls the displayQuestion() function on each of them.

This also ensures that in future, if other types of questions need to be implemented, then it is a trivial matter to create a new class and implement the Question interface.

For example, the current questions only display text. In future, to implement functionality to display pictures or videos, a new class could be created, and its displayQuestion() function could parse the question text and render pictures or videos from URLs within the text.

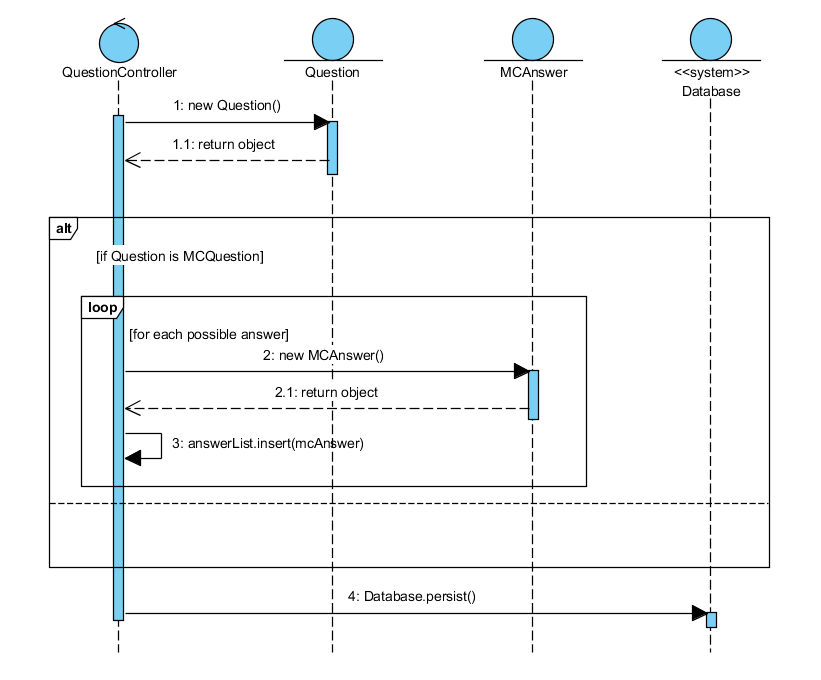


Figure 7 - Strategy Pattern Sequence Diagram

## Observer Pattern

The push mechanism of the Observer pattern is used to push data to its subscribers. It is implemented such that when the teacher (host) moves to the next question, the question will be broadcasted to all students (client). This allows the data to be broadcasted to all subscribers in an efficient manner. Subjects are also able to add or remove subscribers without much modification. This is in line with our objectives to push out updates (e.g. new questions, end of quiz) to clients that want them. Observer pattern is used instead of Command pattern as the host only need to broadcast question and receive the answer from the clients. Command pattern is also redundant as the information does not to be encapsulated.

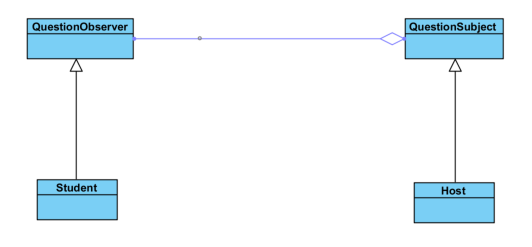


Figure 8 - Observer Class Diagram

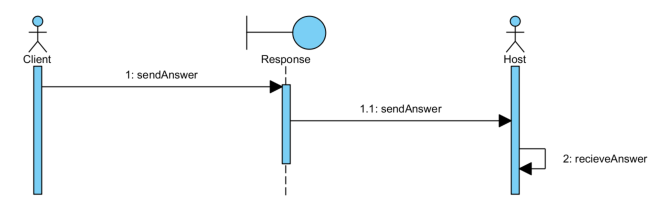


Figure 9 - Observer Sequence Diagram

## Data Access Object Pattern

The pivot requires the system to have a login system and storage for students’ past grades. As a result, a relational database needs to be created. We have chosen MySQL as the relational database implementation. To enable scalability and provide options for users in the future, the DatabaseController class acts as a wrapper for all database controls. It implements the Data Access Object (DAO) pattern so that the system can easily be expanded to support other databases such as PostgreSQL and Oracle Database. Classes that access the database does not have to know the implementation of the database connections and commands because it will be handled by the DatabaseController class.

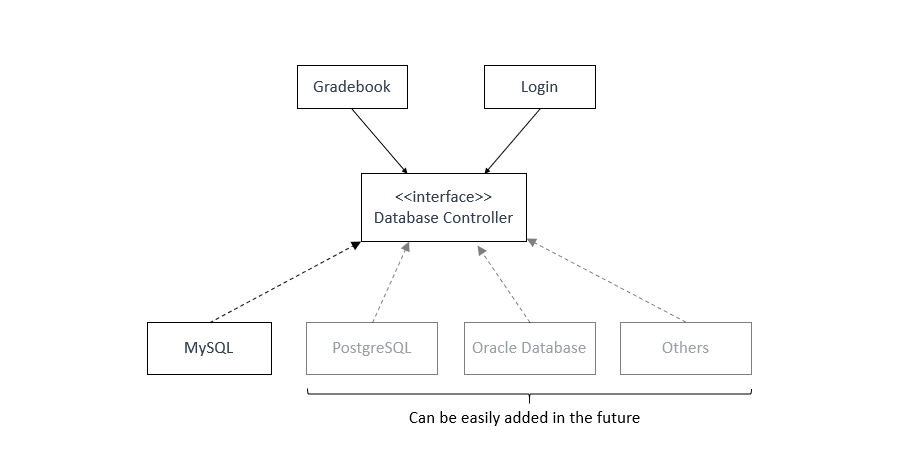


Figure 10 - DAO Class Diagram

## Coupling and Cohesion

Our design has reusability and adaptability in mind. For example, if we wanted to change the actual database being used, the DAO pattern would support this. If we wanted to add new question types, the strategy pattern would enable this. If we wanted to add a new type of observer (maybe a participant who does not answer questions but just sees the results), we can simply register the new class to the subject without changing the current implementation.

The cohesion between classes is minimized as we have tried to follow the SOLID principles, especially Liskov’s substitution principle. We envision that the main change in the future would be adding more question types to the system to engage the student’s better. This can be easily achieved, as mentioned before, with the strategy pattern.

## Overall Class Diagram

Figure 11 - Overall Class Diagram