Architecture & Design Specification

Version 1.0
Date: October 31, 2024

In-Class Polling Application

Prepared by: Mark Ospina,
Jared Fuller,
Ohm Shah,
Diana Bayandina

TABLE OF CONTENTS

5. Architecture and Design Patterns

1. Introduction

1.1. Purpose

The purpose of this document is to provide an overview of the current architecture and design patterns employed in the Pollus system. Pollus is a web-based polling application that enhances classroom engagement by enabling real-time interaction between instructors and students.

1.2. Scope

Pollus is designed to facilitate classroom quizzes in a physical environment. Instructors can upload and manage quizzes accessible by students on their devices, allowing for interactive and adaptable learning sessions. Students can respond to quiz questions in real-time, with results being available to both the instructor and the student.

1.3. Definitions, Acronyms, and Abbreviations

- **Poll**: A set of questions to be answered by students.
- **Instructor**: The user who creates and manages polls.
- **Examinee**: The participant who answers poll questions.

1.4 Document Conventions

- **Bold text** highlights key terms and important points.
- *Italicized* text is used to reference diagrams, which are located throughout this document.

2. General Overview and Design

2.1. High-Level Overview

Pollus follows the **Model-View-Controller (MVC)** design pattern, which divides the application into three core components: the **Model**, **View**, and **Controller**. In Pollus, the Model is represented by **Firebase Firestore**, a cloud-hosted NoSQL database where user data, quizzes, and responses are stored. This component is responsible for data consistency and facilitates CRUD (Create, Read, Update, Delete) operations. The View is the web-based User Interface (UI), accessible by both students and instructors, allowing them to interact with Pollus by creating quizzes or responding to questions. The Controller manages communication between the View and the Model, routing requests, processing data, and ensuring real-time synchronization across clients. This MVC structure enhances Pollus's modularity and scalability.

2.2. Architectural Styles

Pollus employs a **client-server architecture**, where user interactions on the client side are processed by the server and stored in the **Firebase database**. The system's database structure is built on a NoSQL format. Additionally, Pollus incorporates the **Observer pattern**, allowing the system to monitor user actions and provide immediate updates across clients. For instance, when an instructor advances to the next question, all connected student devices update accordingly. Pollus also employs the **Repository pattern** to manage version control, ensuring stability and consistency across updates. By maintaining multiple versions, the application can be updated or rolled back as needed, preserving a reliable user experience.

2.3. Key Architectural Decisions

1. **Firebase's tree-like NoSQL structure** is one of the primary architectural decisions in Pollus which is optimal for handling polls that contain multiple questions and answer options.
2. **Observer pattern** is also applied in the Pollus to enable real-time interactions. This allows the system to respond instantaneously to user inputs, such as student responses or instructor commands, providing a smooth, interactive experience for both students and instructors.
3. **Repository-based version control** system based on **GitHub** allows the development team to manage multiple versions of the application, ensuring that stable iterations are available for deployment while development continues on new features.

3. Architectural Diagrams

3.1. Block Diagram

The block diagram of Pollus illustrates the core components of the system and the flow of data among them. These components include the **Instructor UI**, which provides instructors with options to sign up, log in, upload and manage quiz, start the lobby and quiz itself, view student responses and results; the **Student UI**, where students log in, access quiz, submit responses, and view results; the **Server**, which acts as a central point for processing requests and maintaining data integrity; and the **Firebase Database**, which stores quiz data, user information, and response records. This layout reinforces the MVC structure, with clear interactions between the

user interfaces, server, and database. *Figure 1*, labeled "*Data Flow and Navigation of Pollus*" provides a visual representation of these interactions.

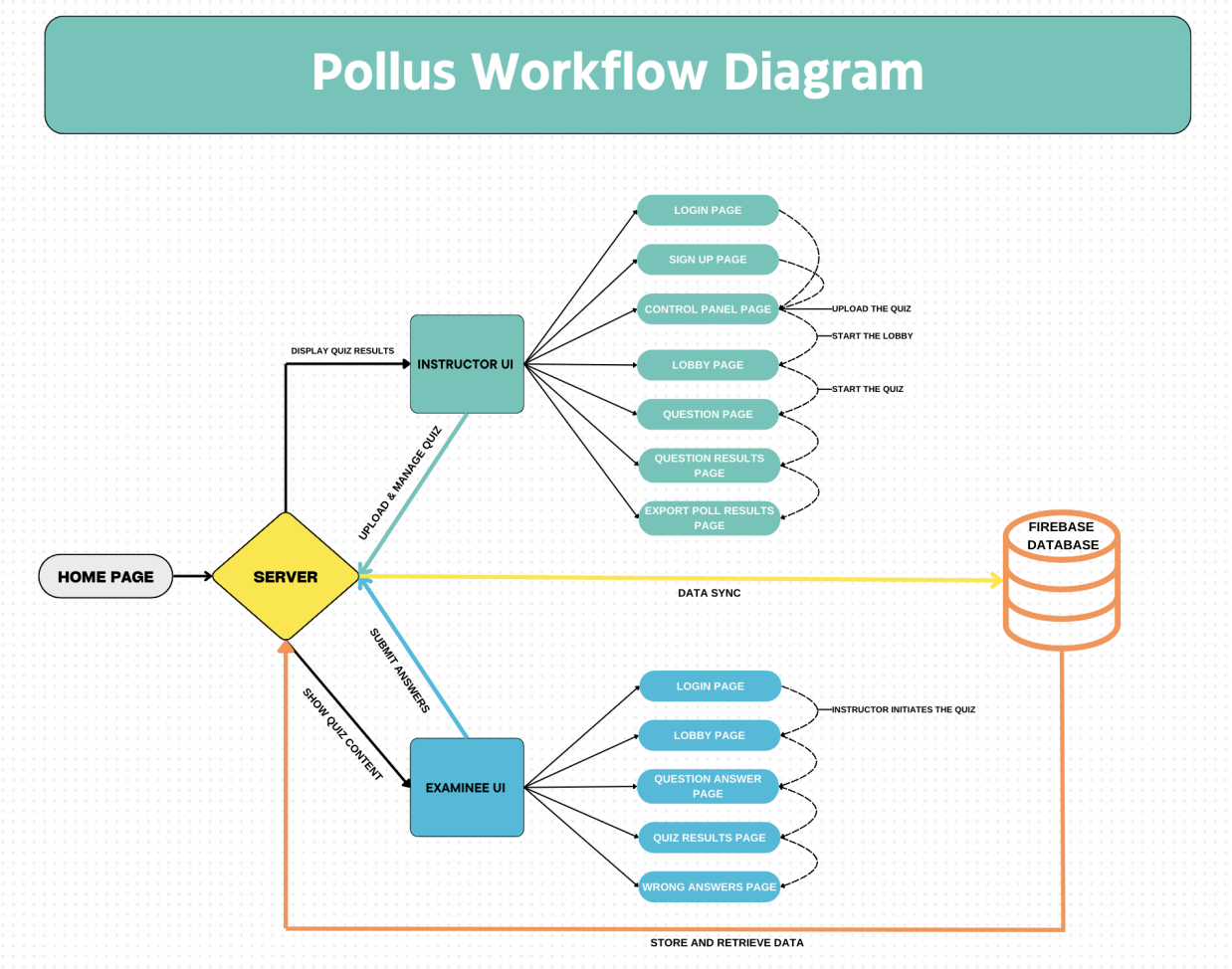

Figure 1. Data Flow and Navigation of Pollus

3.2. Component Diagram

The component diagram expands on the block diagram by illustrating how Pollus's modules interact in a **client-server structure**. The diagram displays how each module - the **Instructor UI**, **Student UI**, **Server**, and **Firebase Database** - operates within the MVC framework, with the Server acting as the Controller that links the View and Model. Each module plays a distinct

role, with the server coordinating all data flow and processing, ensuring that updates made in the database are reflected in real-time across clients. *Figure 2*. "*Client-Server Architecture of Pollus*" below visually demonstrates the separation of user roles and how the Server Component manages requests and responses between Instructor UI and Student UI and the Firebase Database.

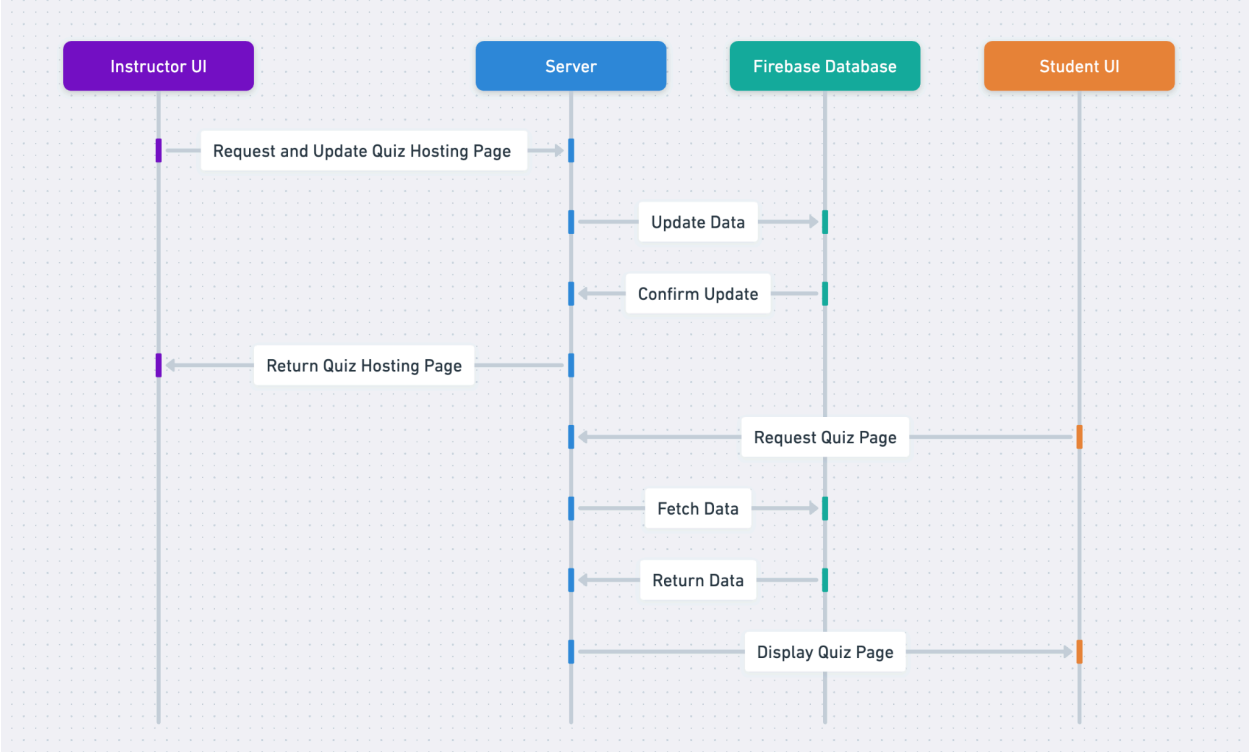

Figure 2. Client-Server Architecture of Pollus

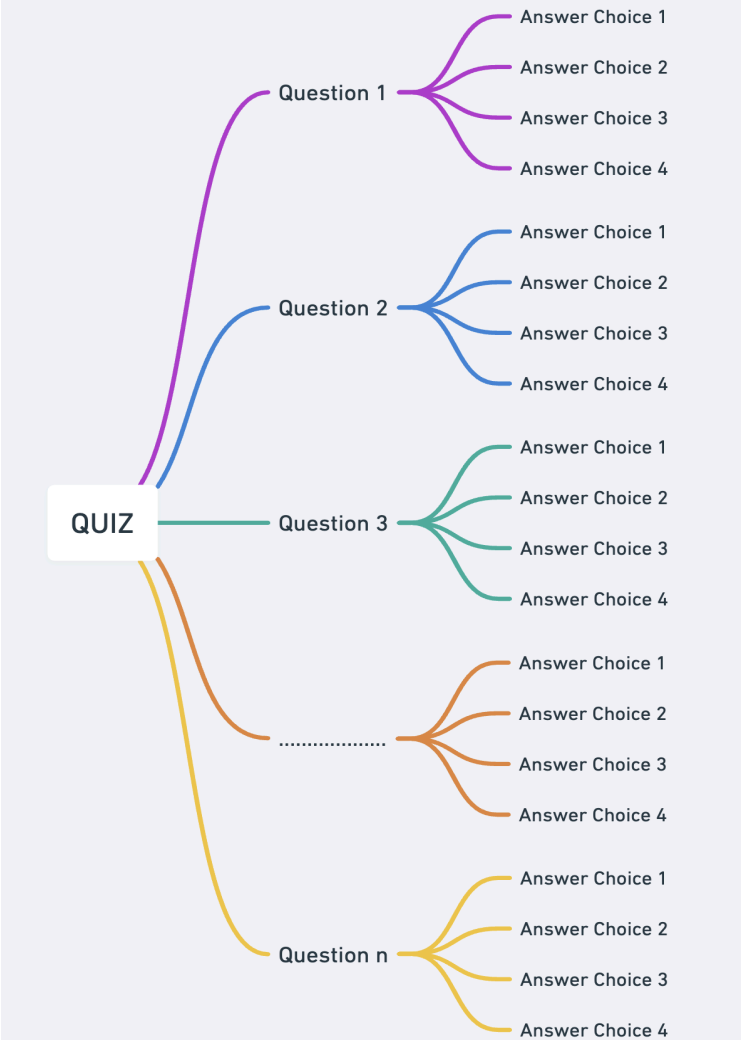

Figure 3. Firestore Object-Oriented Architecture for Pollus

3.3. Use Case Diagrams

The Use Case Diagram for the Instructor UI (*Figure 4. Use-Case Diagram for Instructor UI)* outlines the following actions an instructor can perform within the Pollus application:

Home Page:

- The instructor starts at the Home Page, where they select the "Instructor Role." This action redirects them to the **Log In Page**.

Log In Page:

- Instructors can enter their credentials to log in. If the login is successful, they are directed to the **Control Panel**.
- If there's an incorrect username or password, an **Error Message** is displayed.
- Alternatively, instructors can choose to **Sign Up** if it's their first time, leading them to the **Sign Up Page**.

Sign Up Page:

- New instructors can create an account here. After successful sign-up, they are directed back to the **Log In Page** for authentication.

Control Panel:

- This is the main hub for instructors, where they can manage various poll settings:
    - **Upload Poll**: Allows the instructor to upload questions for the poll.
    - **Set Time Per Question**: Instructors can set the duration for each question.
    - **Start Lobby**: Initiates the lobby where students can join in real time.
    - **Export Results**: Enables downloading the poll results after the session.
    - **Sign Out**: Logs the instructor out of the application.
- Once the **Start Lobby** action is chosen, instructors proceed to the **Lobby** page.

Lobby:

- The Lobby page displays the list of examinees who have joined the session. Once the instructor is ready, they can **Start Quiz**, leading to the **Quiz Session**.

Quiz Session:

- This is the active polling session where questions are presented to examinees in real-time.

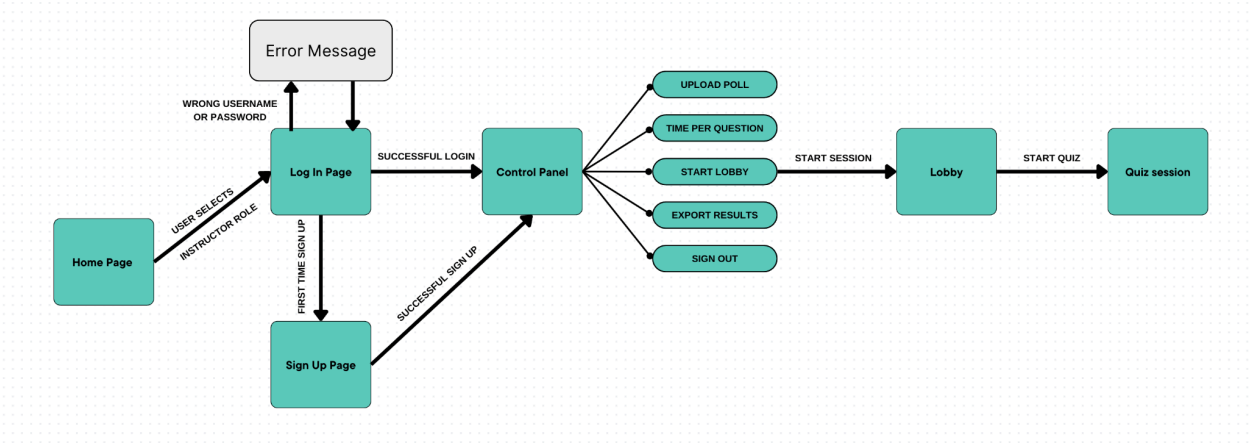

Figure 4. Use-Case Diagram for Instructor UI

The Use Case Diagram for the Examinee UI (*Figure 5. Use-Case Diagram for Examinee UI)* outlines the following actions an instructor can perform within the Pollus application:

Home Page:

- The examinee begins at the Home Page. They can proceed to the **Log In Page** to enter their details and join the session.

Log In Page:

- After logging in, the examinee is directed to the **Lobby Page**, where they wait for the instructor to start the quiz.

Lobby Page:

- In the Lobby Page, the Examinee is shown a message or status indicating that they need to wait for the Instructor to start the quiz. Once the instructor initiates the session, the examinee can proceed.

Question Page:

- The quiz begins on the Question Page, where the examinee answers the current question. After answering, they are directed to the **Question Result Page**.

Question Result Page:

- This page shows the Examinee feedback on their answer (correct or incorrect) for each question. The examinee continues through this loop until reaching the last question.

Last Question Page:

- When the Examinee reaches the final question, they submit their last response and proceed to the final **Question Result Page**.

Quiz Result Page:

- After completing all questions, the Examinee is directed to the Quiz Result Page, where they can view a summary of their overall performance, including scores and feedback.

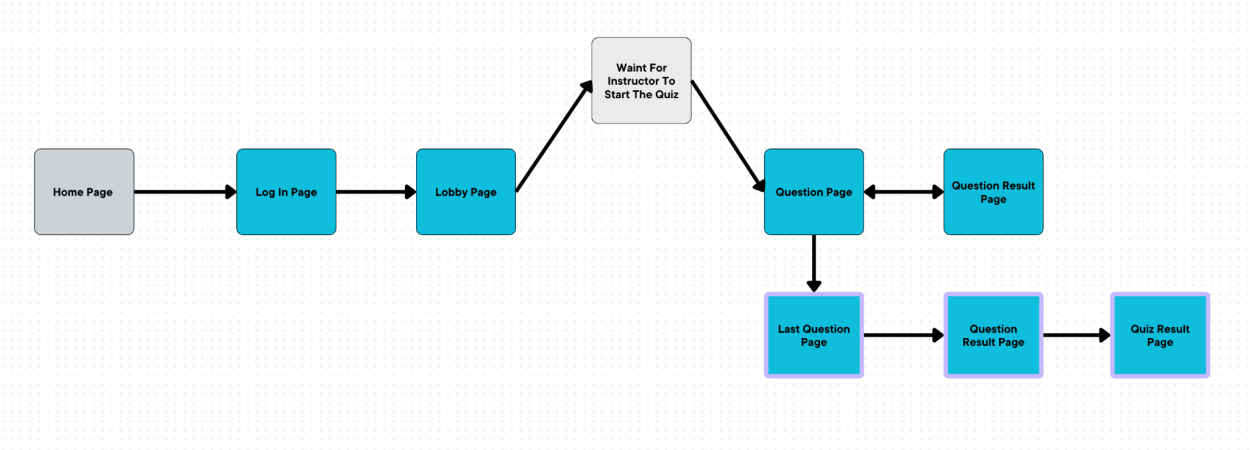

Figure 5. Use-Case Diagram for Examinee UI

4. Component Descriptions

4.1. User Interface Component

The user interface of Pollus is composed of coding languages such as HTML, CSS, and Javascript which help give the website functionality and design. The pages in Pollus have different input controls, buttons, and navigation based on what the user is signing in as. The presenter can control how the quiz functions with what file they uploaded and can download the results and the graphs compiled from them. The examinee's functions are logging into the poll's waitroom using a code and answering the questions to continue the poll once the presenter enables access for examinees.

4.2. Database Component

For the database component, Pollus is supported by Firestore which is a part of the Firebase Realtime Database. Firestore helps control authentication usage and gathers all quizzes/polls data

while the user is accessing Pollus. It allows the user to store collections of data and documents needed for the polls to function correctly. Firestore records the email and data of all users who use Pollus for security purposes and future updates. Firestore is also a NoSQL database that provides great performance and development for Pollus.

4.3. Server Component

Firebase is used for the Pollus backend component. It supports code requests made by the user while using data from the Frontend. Javascript is used to help the backend function properly. The backend has data protection for users and helps detect errors or bugs when processing data from Pollus and Firestore. Firebase has security features such as rules that allow users to enforce user permissions and control access to counter malicious users. Firebase also has cloud functions that will enable users to run code for the backend to respond to events and require zero maintenance because Firebase automatically scales to computing resources to match the patterns of users.

5. Architecture and Design Patterns

5.1. Model View Controller

In our architecture, we utilize the Model-View-Controller architecture pattern, otherwise known as MVC. The model encompasses the data flow, like uploading polls from the Pollus website, reading the questions, and converting them to question objects. It also includes handling the backend through Firebase and ensuring the current question matches with every user. The view consists of dynamic page generation for each user and easy poll uploads for the presenter. The controller comprises updates to each dynamic page for each user. This MVC architecture consolidates functions within the model and encourages code scalability and maintainability.

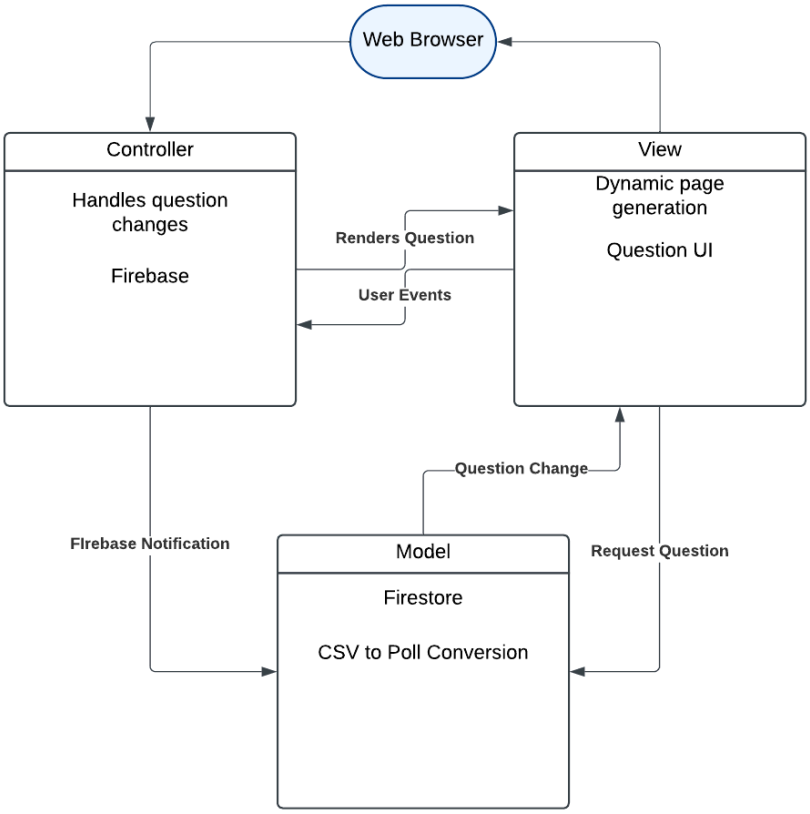

Figure 6. MVC diagram of Pollus

5.2. Repository

Firebase will act as our repository for all our poll-related and user data. Firestore is the database that stores all this data. Firestore stores each question in a collection of questions, known as a poll, and user data separately. The data can interact with each other, but only when called upon by the Pollus web application. Real-time polling is possible because of the interactions between poll and user data. For example, each user will have unique data, allowing for real-time polling across multiple users. Users cannot access each other's data, only their own in a limited fashion. Examinees cannot directly access what questions they got wrong until the end of the quiz, which will display alongside their poll results.

5.3. Client-Server

Firestore inside Firebase acts as our NoSQL database and is used throughout our application. Firestore acts as a hub for our stored data and organizes the user's data. The client side requests the questions from the polls the presenter submitted through the server hosted by Firebase.

5.4. Observer Pattern

The observer design pattern allows data to be synchronized in real time across multiple users. Both of our user classes, Presenter and Examinee, will have different views of Pollus' data. Presenters have a different view of the poll than Examinees. Presenters have the full question along with the full answers, while Examinees will just have the answers organized into A through D. After the question has been answered the Examinee will wait until everyone has answered the question or time has run out. Then they will be told if the answer they choose was correct. The Presenter will have a bar graph of how many people chose each answer after each question.

5.5. Iterator Pattern

The poll itself encompasses the Iterator design pattern. This is because the poll traverses from question to question always asking for the next one while not looking at what is ahead. The Iterator design pattern helps us quickly traverse the poll structures and run operations on each question.