Comparative Observation for Learning or Attention (COLA) Final Report

Rabin Ranabhat, Gaurav Bishnoi, Daniel Malmer
CSCI 5448-Dist
12/8/15

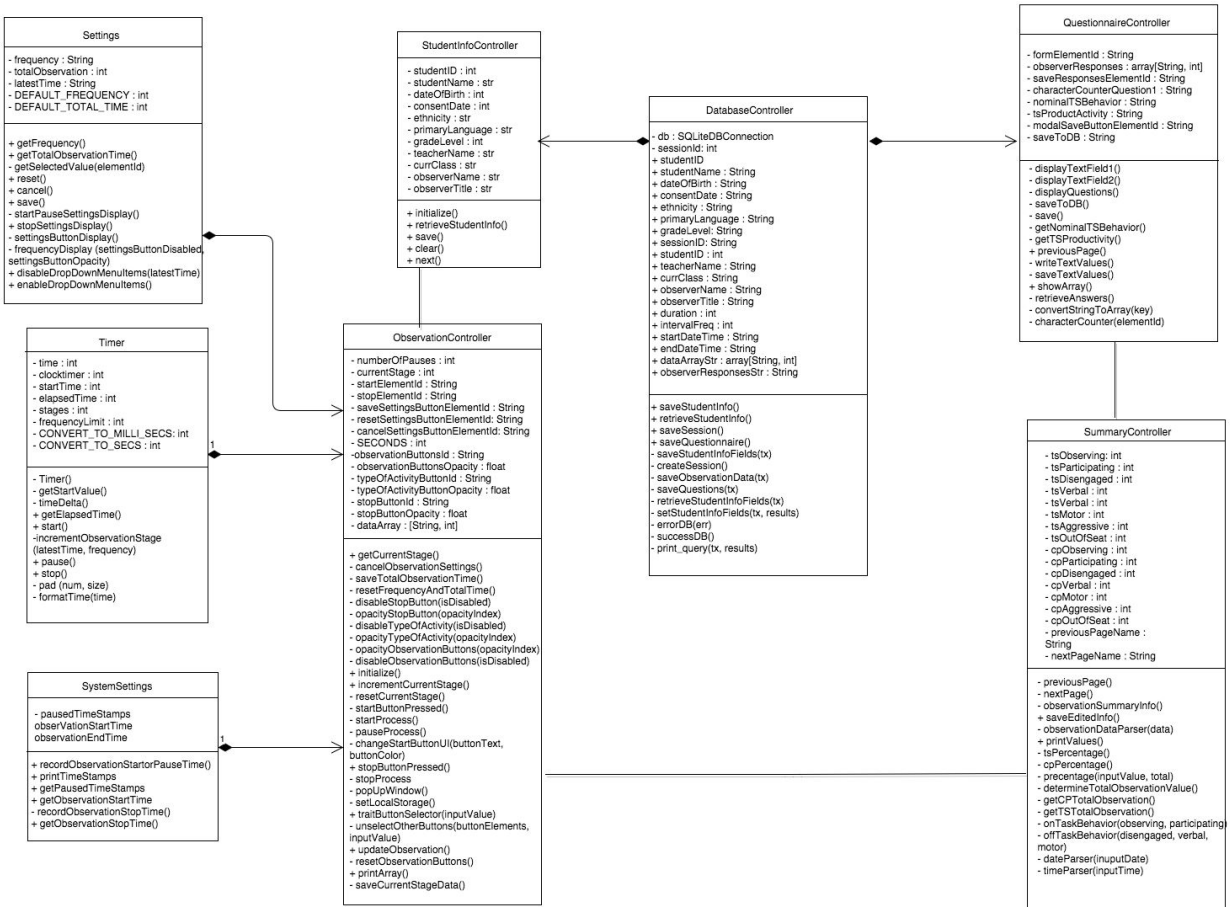
1) Our final product contained most of the features we planning on implementing in parts 1 and 2 of the project. For completeness, we have listed out every feature we proposed and stated how much progress we were able to make on those features:

Functionality	**Progress**
Record and save student information to a local database	Completed
Record and save observer, teacher and class information for each session to a local database	Completed
Set duration and interval frequency settings for each session	Completed
Record behavior of target student and comparison peer during an observation session and save to a local database	Completed
Pause and restart an observation session after the session has started	Completed
Present a summary screen with statistics after the session has finished	Completed
Have the observer answer a questionnaire after the session has finished, and save the answers to a local database	Completed
Can add notes to the marked instances after the session	Incomplete
Can simultaneously access previously recorded data in real time	Incomplete
Stretch: Simultaneously records video	Incomplete

Stretch: Uploads the data into a remote database	Incomplete

The last two functionalities (recording video and using a remote database) were stretch functionalities, so we are not concerned that we weren't able to implement them. The two functionalities before them (adding notes and simultaneously access of previously recorded information) were also lower priority. We feel the current state of the app provides a very useful tool for observing troubled students in schools.

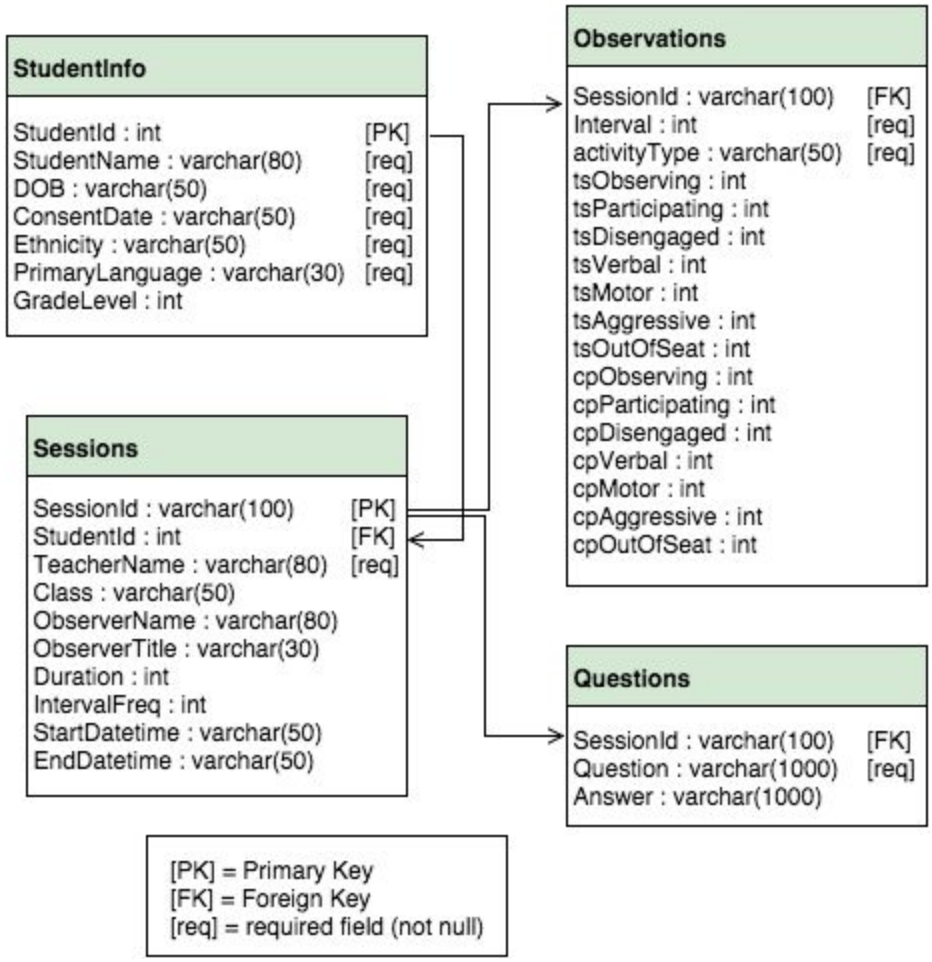
The final class diagram is below:



We found designing before coding to be very useful to our completion of the project. While the final class diagram changed from how we initially set out to complete the project (see section 3 of this report), we kept the basic structure and were able to think about how each page would interact to avoid conflicts in the code. For instance, while designing the software system, we were able to identify that the Observation page was

composed of the timer, the settings screens and the system clock. Therefore, from the get-go, we knew that the observation class used composition. Similarly, we were able to identify different domains of the software architecture meaning, differentiate what needed to be done in student information screen, observation screen, summary screen and the questionnaire screen. Design and analysis gave us confidence every step of the way during code development that we were headed down the right path.

In addition, we designed the following SQL schema to organize our database:



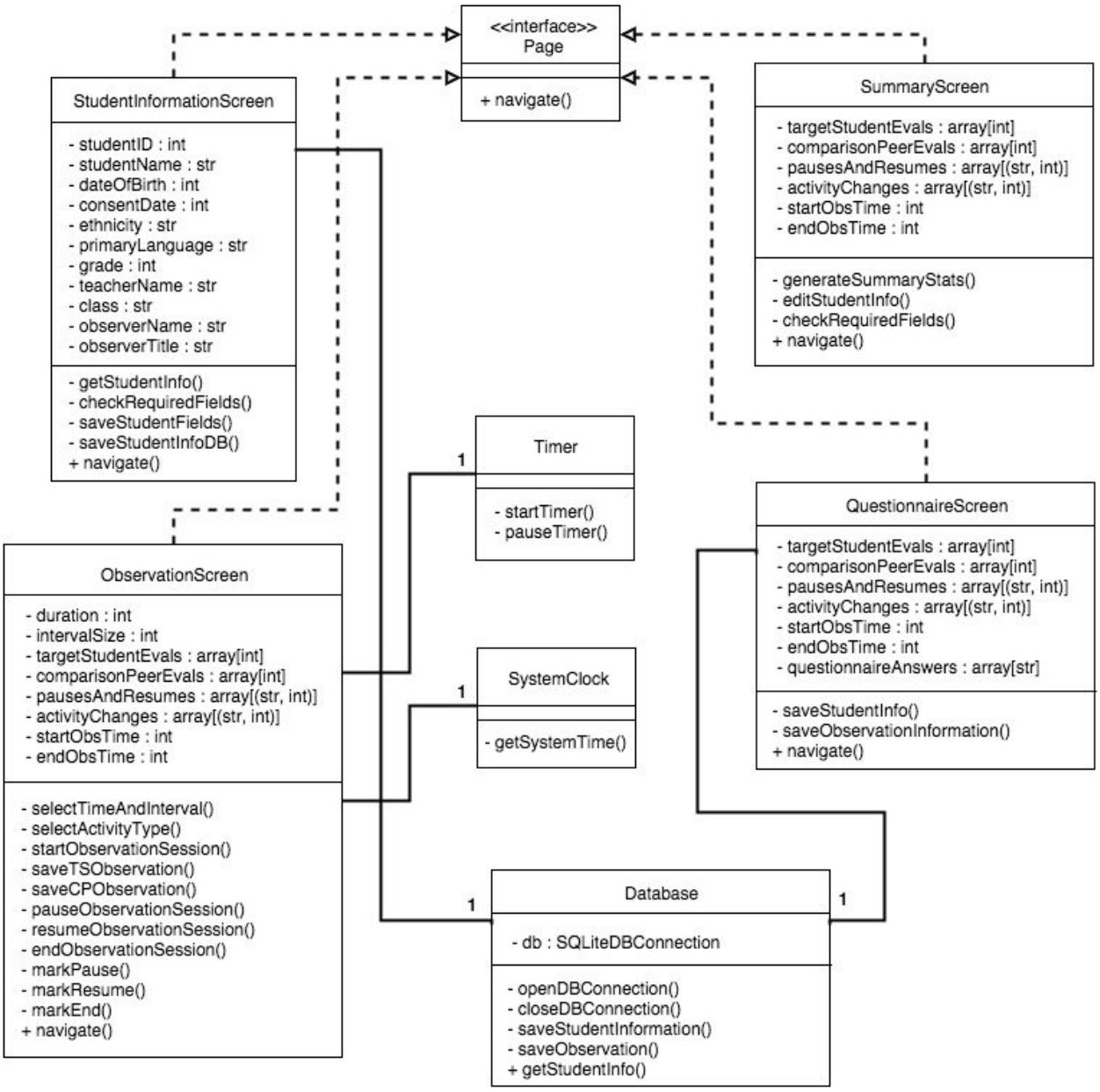
This schema allowed us to store all of the required information with minimal duplicate data. The StudentId fields connect the students to each observation session in the Sessions table. Then the unique SessionId values connect each session with the observation data (Observations table) and the post-observation questions (Questions table). The use of foreign keys allows us to explicitly define these connections.

2) The main design pattern used in this project was the observer pattern. In the observer pattern, objects follow a "publish-subscribe" structure where when an object changes state, all of its dependant objects are automatically updated from this state change. In the context of this project, the publishers are the page controller objects, and the subscribers are the global HTML "window" object that controls the page view (this is how PhoneGap displays apps) as well as the database object. We implemented the pattern by leveraging event key listeners on all of the pages' buttons. When a button is pressed, it notifies the database object that something needs to be created/saved/retrieved in the SQL database and/or notifies the window object that we need to navigate to a different page. This is of course more efficient than have the window or database *ask* if something in the pages has changed or having the functions of the database/window hard coded into each button.

In addition, we think the Database object would have been a perfect use-case for the singleton pattern. The first and last pages in the app both need access to the SQL database, but there is no reason to create more than one instance of it. However, due to the way that PhoneGap navigates through pages, there is no way (that we could find at least) to declare a global instance of an object that can be accessed from multiple pages. Instead, we had to create a new instance of the Database object at each page and simply have the objects load the same SQL database in each instance.

In addition to observer and singleton, we also considered using the adapter design pattern when interacting with the database. The database used by PhoneGap uses a specific SQL language (SQLite), so our create/save/retrieve functions have SQL code specific to that language. However, if we want to try a different SQL database used on the backend (eg. switch the Postgres or MySQL), we don't want to change the way the pages interact with the DatabaseController object and we don't want to completely remove the existing SQLite code in case we need to switch back to it. By using the adapter pattern, we could have the individual pages interact with a database adapter, so it is calling the same function, regardless of the database backend. The we could write multiple backends (SQLite, Postgres, etc.) and be able to easily switch between them using the adaptor interface. However, we didn't end up implementing the adaptor interface due to time concerns.

3) The first class diagram we submitted for part 2 of the project is below:



Between the originally planned class diagram and the final class diagram, the number of methods and the private, protected and public variables are a lot more. The reason for it, our design was inspired by the classical MVC architecture. Like classical MVC architecture, the Controller contains the logic for the View elements too while for the most part, the View is dumb. It doesn't contain much logic. The View elements are changed by the Controller. This has made our implementation of the Controller classes a lot bigger than initially planned in the class diagrams.

The other major difference is that we no longer used the Page interface in our final design. This mostly has to do with the fact that we used PhoneGap as a framework for our app development, which uses Javascript as its programming language. Javascript is a dynamically-typed object-based language rather than a class-based language. As such, it has a no concept of an interface in the sense that you can't require objects to contain a specific set of functions. Because of this, we removed the Page interface from our class structure, but do not feel this hurt the structure of our code too much as the Page interface only handled the navigation between pages.

4) The class was tremendously helpful in understanding why planning and envisioning the program before write even a first line of code is important. For our project, we did a PhoneGap application called 'Comparative Observation for Learning and Attention (COLA), however, we did not start writing the code until we were done through writing requirements, creating activity diagram and writing class diagram. We talked to the end user of our application, Brian Gustman, school psychologist at Malone Elementary School in Denver, when creating the business requirement. We started off the process knowing very little about the application and how to approach it. However, the more we talked to Brian, the more we understood his vision of what needed to be included in the application. The process was iterative and completely worth it. Coming out of the requirements formulation stage, we had a good picture of what the application needed to do.

Similarly, the session we had as a team to come up with state diagram, use cases and class diagrams were also very helpful. In a way, they helped create a great view of how to design the system even before we started writing the code. The process of coming up with the architecture document was key in helping understand the software architecture, how different components worked together and how the final application would look like. Going through the process of analysis and design gave us more confidence that the application development will be completed with rather ease.

The other thing we would like to point out was how learning about the different design patterns has changed our programming paradigm. Previously, where the approach was to "get the code working", now the approach has come to writing "good code". The only downside of learning different design patterns have been the immense urge to refactor the code every time a class module is revisited. We have refrained from continually refactoring the code with the final goal in mind that the project has to be completed and there is a balance between too much refactoring and achieving the goal of the project.