ECE651

Final Project Design Decisions

Charlie Prior, Louise Li, Kenan Colak, Qianyi Xue

1 - Evolution 1

Overall Design

Overall, we broke our project up using the Model, View, Controller (MVC) design pattern. This was chosen to allow us to create a text-based UI while remaining flexible for any future UI upgrades requested by the client.

Model

Storage of State

For the storage of information and data, we began by creating classes to represent the professor, students, courses, and lectures, and record the attendance of students. The main implementation for these classes came through how they inherited one another with the choice being to have the Professor class act as the “base” class and have an n-amount of courses. As such, we designed Professor to contain an ArrayList of Course called courses so that the Professor could access each of the different classes they taught. As each of the professors, courses, lectures, and students need identifiers, they were all given a form of identification, commonly using IDs. The Professor class contained variables for the name, ID, email and courses taught by the Professor with methods corresponding to getting and setting these variables as well as adding and removing courses from the Professor’s catalog.

Since the professor will need an n-amount of courses to manage, we created a Course class next. In this class, we had the identifier variables such as the courseID, courseName, the professor who taught it, and the times the course took place. Most importantly, the class had two ArrayList variables that contain the students within the course as well as a list of lectures for the Course class. The normal getter-setter methods were implemented as before for the Course class. As the Course has an n-amount of students and lectures, we had to implement those classes next. The Lecture class contains the variables for identification as before, including the lectureID, courseName, professor who teaches it, the date of the lecture, another list of students for that lecture for attendance purposes. Lecture class also includes an attendanceSession variable of class AttendanceSession for the purpose of storing information regarding the attendance of students, discussed further in the *Manipulation of State* section. The Student class simply had the normal getter-setter methods as well as the identifier variables of legalName, studentID, email and displayName to be able to identify and distinguish each of the students.

As all these classes contain each other within one another, this allows for us to have very well made has-a relationships within our classes as a professor has-a course which has-a lecture which has-a student and so on. Through these classes, we were able to properly implement this portion of the assignment and store the proper variables within the proper locations.

Manipulation of State

To be able to store the Attendance of each student, we divided our attendance classes into two separate ones named AttendanceSession and AttendanceRecord. AttendanceSession was made to record and store the attendance in a specific lecture that is going on at that moment whereas AttendanceRecord keeps track of all previous attendances through the dates, the student associated with the attendance, their status and what lecture that student's attendance is being checked for. One is for modifying and the other is for storage.

To implement the requirement to email students upon change of attendance we used the Observer design pattern. The application (App; the Publisher in the design pattern) has an EventManager to keep track of the listeners to its events. Listeners implement the EventListener interface. In this evolution, we only have one type of Listener, the EmailAlertsListener, which takes a student and attendance status and notifies via the students email that their attendance status has changed.

View

For the UI and viewing portion of the assignment, a class called TextUserView was created that managed all the printing necessities to output the proper display outputs to the user. This was done so by creating methods to print the courses, students, lectures for starters and then having methods for a start screen, a header as well as the status of students to see their attendance status for the current class. By putting all of those methods together, a proper UI was formed to be called within the App.java class for the fully put-together code functionality.

Controller

For the controller portion of our MVC, we created a class to handle the entirety of this portion on its own called TextUserController. This class took care of such methods as removing students from a course, showing attendance from a course, changing students' display names, updating students' attendance records and many more. These methods were used to build the overall program's ability to manipulate the state of the students, classroom, lectures, course and professor through its joint UI managed through the aptly named TextUserView class. By having all of our methods for the controller in one class, we were able to streamline the modification of the different classes by the user and simply call the methods necessary when the manipulation was demanded.

2 - Evolution 2

2.1 - Admin

Overall Design

For the overall design of the admin side, user registration was the forefront of the design as we wanted a way for the students and faculty members to be registered into the system by saving them into the database. We also considered applications such as removing the user, updating passwords, etc. To structure this, there were a set amount of methods divided into it:

- Model → adding students, adding professors, removing students, removing professors, updating students, updating professors.
- View/Controller → display for the methods and the controlling section using user inputs to decide what the user wants to do given the options in the model.

Model

The model was built around the methods mentioned above. The methods made were addStudent, addProfessor, removeStudent, removeProfessor, updateStudent, updateProfessor, getStudentID and getProfessorID. These methods were built by using the DAO classes built for the purpose of updating, retrieving information and removing information from the databases relevant to each of the objects made for the model.

The tests are made in the testing class UserRegistrationTest.java by testing each possible branch and option in the methods. Adding, Removing, Updating users were all tested. Since we need the user's ID to remove or update the user, we need the getID methods for both faculty and students, which is why these methods were written. As tested, the update, remove and adding methods all were successful and passed the testing phase.

View + Controller

Since the UserRegistration needed a UI for the user to be able to navigate and update their status through, the controller and view were combined into one class and methods were made for each of the different option selections. One was made for the studentOptions, professorOptions, addStudent, removeStudent, updateStudent, updateProfessor, removeStudent, removeProfessor and finally, a method for the main menu which gives the option to use the class for a student, faculty or just quit. The first menuOptions() method gives the user the options of choosing to manage a student, faculty or quit, navigated by typing 1-3 respectively to correspond with each of the options. If a number that isn't 1, 2 or 3, the user would be told that they entered an invalid option and be prompted to enter again.

Once a student or faculty is chosen, both options take you to a menu corresponding to a student or faculty member with the options to add that user, remove that user or update that user, once again prompting 1-3 with the error message for an invalid option still present. Since for the student object, the email, name, displayName, studentID and password are all needed, a user input is used to take these from the terminal and place them into their corresponding variable values, whether they be a string or an integer, and place them into the corresponding method (addStudent or addProfessor). The methods will then create an ID to correspond with the newly made student or faculty and store the data of the person in the database corresponding to the person made. The remove and update methods both take the ID of the person being removed or having their password updated. When prompted with an invalid ID, the selection states “there is no user with this ID…” and exits out to the main menu for the user to be able to try another option. If a correct ID that is in the database is entered, the user and their information including their password is removed for the removal method and for the update method, a prompt is given asking the user for the new password that they want to set and the password given by the user replaces the previously set password when the user was initially created.

2.2 - Client

Our client application is built using a straightforward and effective design, Model-View-Controller (MVC) architecture. This design helps organize the app into three main parts, each with a clear role, making it easier to manage and update.

App

It initializes the client-side controller and view, handles network connections, and orchestrates the interaction between the user input/output and the server-side processing. This part handles all the data and the rules of the application. It's where user information is managed and where we talk to the server. We use a tool called ObjectMapper to help convert data to efficient JSON format, ensuring that all communications are smooth and secure in data serialization.

View

The ClientSideView is the part of the application that users interact with. It's designed to be user-friendly, making sure that everything from logging in to accessing reports is easy for both students and faculty.

Controller

The ClientSideController is essentially the middleman that takes user inputs, makes logical decisions, and communicates these decisions to the Model or updates the View. This setup helps keep the application flexible, ensuring that the application remains agile.

Network Communication

For this part, we utilize Socket, ObjectInputStream, and ObjectOutputStream to handle real-time data exchange with the server. This setup enables sending requests and receiving responses from the server.

Workflow and Logic

Upon starting the app, it attempts to connect to the server using the connectToServer() method. This method continuously tries to establish a connection using the provided server address and port until it succeeds, or an error occurs.
We initialize the ObjectInputStream and ObjectOutputStream wrapped around the socket's streams to facilitate serialized communication.

Functionality and User Type

- **Login/Logout**

For Students: Allows setting of email preferences and retrieving course reports.

- **Setting Preferences:** Involves selecting course sections and specifying how and when to receive notifications.
- **Retrieving Reports:** Involves requesting detailed academic or attendance reports for specific courses.

For Faculty: Allows recording, updating, exporting attendance, and selecting courses to teach.

- **Recording Attendance:** Facilitates the entry of attendance data for a selected lecture.
- **Updating Attendance:** Allows modifications to previously recorded attendance data.
- **Exporting Data:** Enables downloading or exporting formatted attendance data for administrative use.
- **Course Selection:** Allows faculty to pick unassigned courses to add to their teaching schedule.

Error Handling and User Feedback

Our application robustly handles errors, particularly in network communication, by catching exceptions and providing feedback through the client-side view. This feedback informs users of the current state of their requests, whether successful, pending, or failed due to errors.

2.3 - Server

Overall Architecture

Model: Data management and persistence are handled by DAO classes not detailed in the script but assumed to be part of the system (e.g., StudentDAO, SectionDAO). These classes interact with a database to retrieve and store data.
View: The server-side view focuses mainly on outputting logs and essential information to the server console.
Controller: ServerSideController acts as the central hub for processing business logic, encapsulating the decisions and interactions needed for each type of request.

App

This class initiates the server and listens for incoming client connections. It continuously accepts client sockets, creates a controller for each connection, and starts a new thread (ClientHandler) to manage interactions with each client. Our system supports multithreading. The server side handles each client in a separate thread, it can manage multiple clients simultaneously, which is critical for high user engagement times. Upon accepting a new client connection in the App class's connectToClients() method, a new ClientHandler object is created. This object takes the client's socket as a parameter along with instances of ServerSideView and ServerSideController. A new thread is then started with this ClientHandler.

ServerSideController

This component handles all the business logic for the server. It processes requests such as login/logout, attendance tracking, and email preference settings. It performs authentication, manages session states, handles CRUD operations related to attendance and courses, and responds to client requests with appropriate actions.

ServerSideView

Primarily for logging and providing server-side feedback.

ClientHandler

Each client connection gets its instance of handler in a new thread, enabling concurrent transactions of multiple clients. It manages user sessions, processes serialized requests and directs the flow of data between the network stream and the controller. ClientHandler handles all operations for a single client session.

Error Handling

Thread-Level Exception Handling: Each ClientHandler thread is wrapped in a try-catch block to catch any runtime exception.
I/O-Level Exception Handling: All network and I/O (sockets, streams) are enclosed in try-finally blocks and are managed to ensure they are closed properly.
Network-Level Exception Handling: The server side includes error handling for network issues, and catching IOExceptions to handle network interruptions. When an error is detected, it attempts to re-establish or close the error cleanly.

2.4 - Database (DAO)

To implement the database in our project, we utilized the Data Access Object (DAO) design pattern. This separates the application logic from the implementation details of the database. For the database itself, we chose MySQL as this was preferred by the client.

The DAO starts with an abstract generic class that implements some common methods related to building and executing queries. Then, each data class has a corresponding DAO class that is a specialization of DAO.

2.5 - Course Management

Overall Design

CourseManagement classes were made as a way to manage the courses in the universities within our databases while also being able to multitask and multifunction through methods for CSV access and reading, as well as managing lectures, courses and sections within the database as well. There are four main classes within the courseManagement portion of the evolution being: CourseManagement, CourseManagementController, CSVLoader and SelectUniversity.

Model

The CourseManagement.java class specializes in managing courses with getters for all portions of the universities being managed from the university itself, the courses, and the sections. There are also methods to add and remove these objects from their corresponding databases, add students to specific sections of courses, update sections, list professors of sections and list professors in a university. All methods of management for the university's courses and their contents are within this class.

The other class within the Model parameters of the Course Management section would be the CSVLoader.java class, which specializes in methods to read CSV files and be able to pull students from files by taking the variables that specify which index the name, email, display name and university IDs of the students are, and returning the list of students form the CSV file. The class also has a method to simply read a CSV file.

View + Controller

The methods are controlled within the CourseManagementController.java class, which takes the user input to create a controller for the methods in the Model class. This class allows for input and output to be given feedback based on what the user is putting into the terminal. For example, if a course is added successfully into the course database, a message will be given indicating that. Otherwise, the method will return null. When entering sections or courses, the name of the components that need to be placed in the objects being created are asked through prompts, saved into the objects, and then saved into the database corresponding to the object made. This can be from section name to course name, and even asking the user if they are sure they want to remove an object from the database through yes or no prompts on request for removal. The biggest method in this class is through the readStudentsFromCSV method, which reads a CSV file and asks the specifics regarding what the delimiter is, where the columns for each variable of the student object is, and so on. This relies on the CSVLoader class as well (mentioned in the *Model* section). The other View + Controller class would be the SelectUniversity.java class, which specializes in listing, selecting and reading the universities from the corresponding database.

3 - Evolution 3

Summary

Evolution 3 of the project encompasses significant updates and enhancements across various aspects of the system. We have maintained all the previous functions in Evolution 2, improving the code quality by more test coverages and abstraction, adding missing functions that are required in Evolution 2, getting better interactions between different modules, and adding GUI to perform the option selections. We will introduce our updates and new designs in detail below.

3.1 - Course Management

Updates to Model

Updates to the model were made to address concerns from the user raised in evolution 2. Now, students are only added as users to the system in the Admin app and are only enrolled into classes in the course management app. Additionally, when a student is unenrolled from a section, they are not removed from the system entirely.

GUI

The GUI launches with the scene set to a login screen showing the names of all universities that can be selected. Once a university is selected, the main menu is loaded and a model is created for that university. Each of the action screens extends from an abstract class SuperController which extends AnchorPane. The supercontroller contains the logic to load the scene in its constructor and also return to the main menu through the cancel button. Related subclasses (for example enroll and unenroll student) inherit from the same superclass.

3.2 - Admin & User Registration

Updates to Controller and Model

The main updates to the model in the User Registration implementation was adding controller methods for the GUI so that the methods can work with the FXML implementation compared to the text-based UI that was made for evolution 2. New FXML files were made for the different pages from selecting the user, faculty vs student, to adding, updating or removing said user. Each of the three option pages also had additional subsections with all having the option to return to the previous page and finally, exit the program if they wanted to. The methods implemented in the newly made UserRegistrationController.java class implements the new methods made in the UserRegistrationView.java class, that take the text-based UI implementation of the model's methods from evolution 2, and adjusts them to work with the GUI made for the new evolution requirements. This was made by having each of the different methods take in a set of strings that would correspond with the inputs being selected by the users and passing them into the corresponding methods depending on what they user was selecting.

GUI

The GUI launches with an option select page that prompts the user to choose their user, faculty or student. When either is chosen, the user is given three options between adding a new user, removing a pre-existing user or updating a pre-existing user. If a new user is selected to be added, a new page is prompted where text boxes with instructions on what to fill them with prompt the user to fill in information such as their full name, email address, choice of display name, and choice of password with a dropbox to choose their university from the University.csv loaded into the database. The removal page prompts the user to enter the university ID of the student or faculty they want to remove, checks if the ID is in the database and returns with a pop-up letting them know if they were able to successfully remove the prompted person from the database or not. The update page prompts the user to enter the university ID of the student or faculty they want to update the information of and enter the new password for the faculty or the new password and new display name for the student. If the university the student belongs to has the option to update the display names, a pop-up will appear notifying the user that their display name and password have been successfully changed. If the university does not allow for display name changes, the pop-up instead lets them know the password has been successfully changed and that the university administration that the student belongs to does not allow for display name alterations. Each of the options typed in for the different pop-ups were made through text boxes, and the university select drop-down was made with a combo box.

3.3 - Server

Updates to DAO and ServerSide Controller Functions

As for the Server side, we have added several functions to DAO to make the data retrieval more convenient and fulfill the further requirements, including several deleteAll functions for different tables in datasets, which will make the dataset clean and easier to test. Besides, we also made some abstractions to the previous functions in

ServerSideController.java so that it will be easier to implement the unit test and the codes' quality should be improved.

Tests on Server Side

In Evolution 2, we did not reach a high test coverage due to the lack of understanding of mock and spy, especially for testing the ObjectInputStream and ObjectOutputStream which received objects from the client. In E3 implementation, we have solved the issue by manually adding ByteInputStream to mock the object transfers between different sides. We have added multiple tests this time to test as many functions and lines as possible.

3.4 - Client

Updates to Controller and Model

In previous versions, we always receive requests from text, which will lead to undesired errors which come from the incorrect format input. Therefore, in Evolution 3, we have implemented the Graphical interface which will automatically give clients the options through buttons so mistakes should be increasingly reduced. Moreover, we have added several Fxml files for different pages so that each request can be handled correctly. Clients can log in to the system if the passwords and user id are matched in the database. If not, they will receive an alert prompt as they cannot log in correctly. Moreover, the system will automatically identify whether the user is a student or a professor if the inputs are accepted by the system, they will be led to different pages. The options will be consistent as Evolution 2 but replacing them with buttons which will trigger by clicking them. Multiple functions of these action events are added to the ButtonController.java so that each function should be easy to call and using abstraction will be easier for loading different sections files and lecture files for different options. In addition, we have added several tests for the GeneralController.java which is the ClientSideController.java in Evolution 2. We have made better abstraction and are trying to mock the behaviors of sending objects between server and client.

GUI

GUI background we choose to display is mostly TitlePane which gives a title for the page and describes the behaviors clients should do in general. We also use ComboBox for the choice of Lectures which are way more convenient for choices of sections and lectures which are flexible due to different clients. Besides, it is also worth mentioning that we keep the GeneralController the same through page transitions because the client and the connection to the server should be the same and consistent.

3.5 - Extra Credit

Create an automatic attendance recording system that enables students to self-check-in via an application. It is intended to minimize manual efforts by professors in tracking attendance and to ensure that students are physically present in the class without any possibility of fraudulent check-ins.

User Roles

1. Professors: Can initiate the attendance process for a specific lecture by generating a dynamic code that is displayed in the classroom (e.g., via a projector).
2. Students: Can log into their system to view and submit the displayed code to mark their attendance.

Professor Interface

- Professors can select a specific lecture and start the attendance process by generating a dynamic, time-sensitive code.

Student Interface

- Students can log in to their system to receive and submit the attendance code within its valid time frame.
- If a student submits an expired code, the server will notify them to resubmit.

Server-Side Validation

The server performs several checks to validate the authenticity of the check-in:

- IP Address: Each attendance must correspond to a unique IP address to prevent multiple submissions from the same computer or proxy submissions.
- ISP Verification: Students must be connected to the campus WiFi, and their ISP must match the professor's ISP to ensure physical presence on campus.
- Geolocation: The system checks the city, state, country, and approximates the latitude and longitude (rounded to four decimal places) of the students against the professor’s location data.

Security and Data Integrity

- Dynamic codes are refreshed every 10 seconds to prevent misuse. Stored in map structure in cache <timestamp, code>.
- The server maintains a cache of professor-specific data (e.g., geolocation, IP, ISP, organization) paired with each timestamped code to validate submissions.

Concurrency Management

- The system utilizes synchronized data structures and concurrent maps to manage cache on the server side effectively.
- Write operations are protected with locks to ensure data integrity and prevent race conditions.
- Simulation of class session end is implemented with a sleep timer set to 5 minutes post-class to allow for delayed check-ins and to handle edge cases smoothly