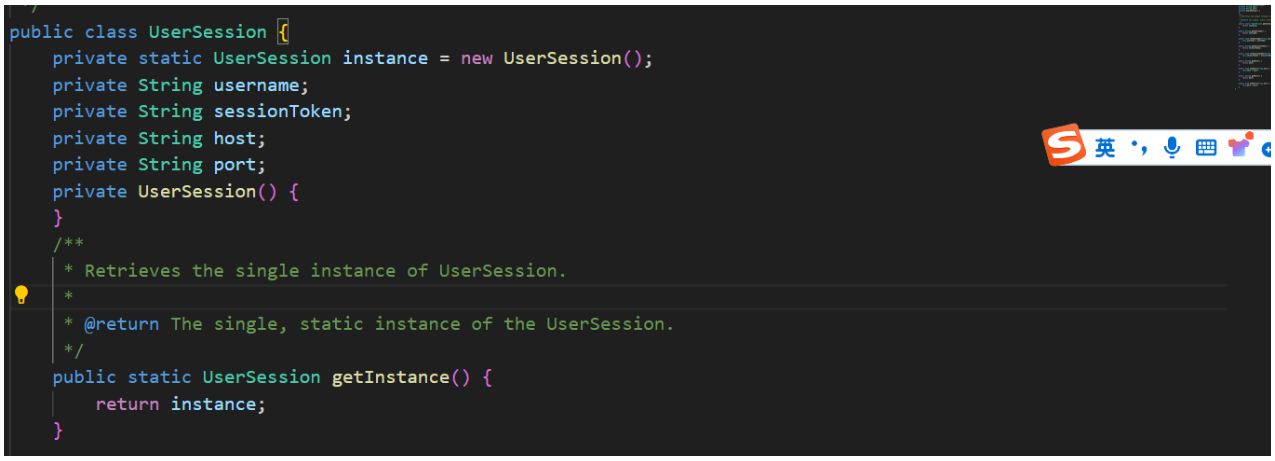
Design Decisions Document

Decision to Use Singleton Pattern

Singleton Pattern for Session Management:

Implementation: To manage user sessions, we implemented the `UserSession` class using the Singleton design pattern. This ensures that only one instance of the `UserSession` class exists throughout the application lifecycle. The class is defined in the `edu.duke.ece651.team1.client.model` package and includes properties for the username, session token, host, and port.



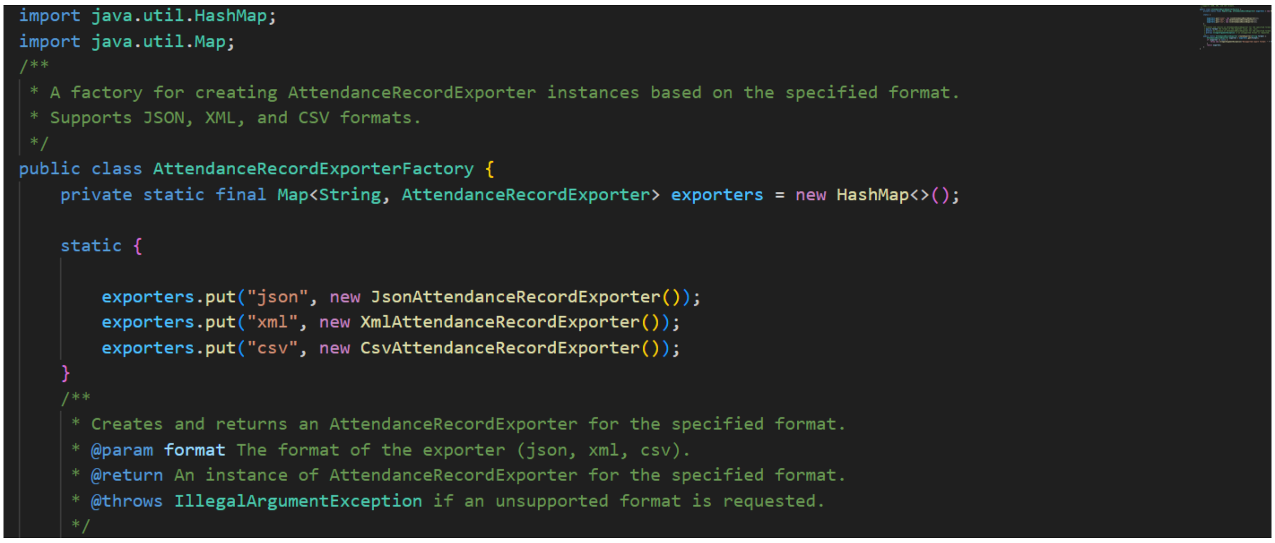
```java
public class UserSession {
    private static UserSession instance = new UserSession();
    private String username;
    private String sessionToken;
    private String host;
    private String port;
    private UserSession() {
    }
    /**
     * Retrieves the single instance of UserSession.
     *
     * @return The single, static instance of the UserSession.
     */
    public static UserSession getInstance() {
        return instance;
    }
```

Rationale: The Singleton pattern was selected for its ability to ensure that a single user session is maintained across the client application. It prevents the creation of multiple instances of the session object, which could lead to inconsistencies and errors in session management.

Decision to Utilize Interface and Factory Pattern for Export Functionality

Interface-Driven Design for Flexible Exporting:

Implementation: To support the flexible exporting of attendance records into various formats, we employed an interface-driven design complemented by the Factory Pattern. The core of this design is the `Exporter` interface, which defines the contract for exporting records. Concrete implementations of this interface— `XmlAttendanceRecordExporter` , `JsonAttendanceRecordExporter` , and `CsvAttendanceRecordExporter` —provide the specific logic required to export data into XML, JSON, and CSV formats, respectively. This design is encapsulated within the `edu.duke.ece651.team1.export` package.

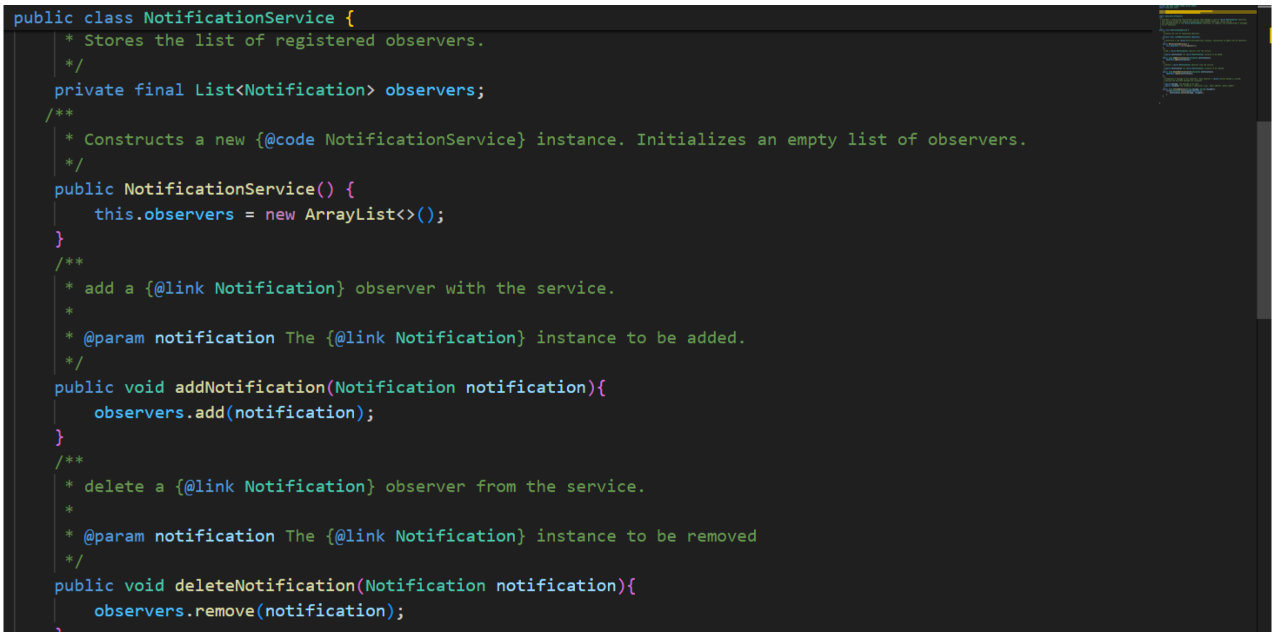

```java
import java.util.HashMap;
import java.util.Map;
/**
 * A factory for creating AttendanceRecordExporter instances based on the specified format.
 * Supports JSON, XML, and CSV formats.
 */
public class AttendanceRecordExporterFactory {
    private static final Map<String, AttendanceRecordExporter> exporters = new HashMap<>();

    static {

        exporters.put("json", new JsonAttendanceRecordExporter());
        exporters.put("xml", new XmlAttendanceRecordExporter());
        exporters.put("csv", new CsvAttendanceRecordExporter());
    }
    /**
     * Creates and returns an AttendanceRecordExporter for the specified format.
     * @param format The format of the exporter (json, xml, csv).
     * @return An instance of AttendanceRecordExporter for the specified format.
     * @throws IllegalArgumentException if an unsupported format is requested.
     */
```

Rationale: The decision to adopt this interface and factory design was driven by the need for extensibility and maintainability in the exporting functionality. By defining a common interface, we established a contract that all exporters must adhere to, allowing for the easy addition of new formats without modifying the core logic. The Factory Pattern further encapsulates the instantiation logic, providing a straightforward mechanism for obtaining exporter instances based on format requirements.

Decision to Utilizing the Observer Pattern for Notification Services

Observer Pattern for Notification System:

Implementation: To establish a robust notification system, we embraced the Observer pattern, encapsulating this design within the `NotificationService` class. This service manages observers, each conforming to the `Notification` interface that mandates a `send` method. Concrete implementations of this interface, such as `EmailNotification`, enable the delivery of messages across different mediums.


```java
public class NotificationService {
     * Stores the list of registered observers.
     */
    private final List<Notification> observers;
   /**
     * Constructs a new {@code NotificationService} instance. Initializes an empty list of observers.
     */
    public NotificationService() {
        this.observers = new ArrayList<>();
    }
    /**
     * add a {@link Notification} observer with the service.
     *
     * @param notification The {@link Notification} instance to be added.
     */
    public void addNotification(Notification notification){
        observers.add(notification);
    }
    /**
     * delete a {@link Notification} observer from the service.
     *
     * @param notification The {@link Notification} instance to be removed
     */
    public void deleteNotification(Notification notification){
        observers.remove(notification);
```

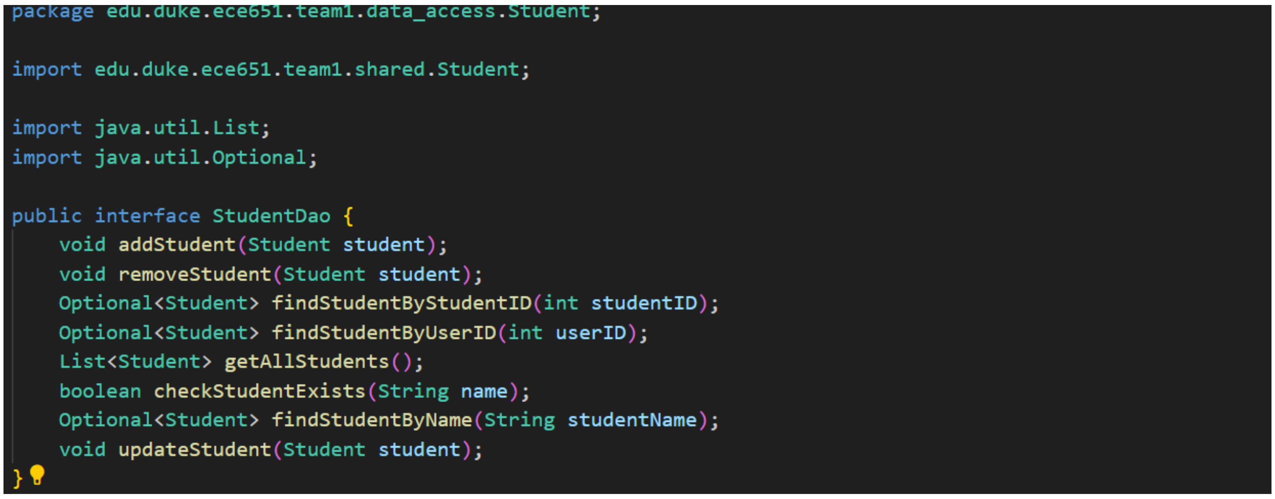
Rationale: The decision to implement the notification system using the Observer pattern was driven by the need for a flexible and extendable architecture that can support multiple notification mechanisms (e.g., email, SMS, push notifications) without tightly coupling the notification logic to the consumer services. This pattern allows the `NotificationService` to broadcast messages to all registered observers, which can then handle the message delivery according to their specific protocol.

Decision to Use Data Access Object (DAO) Pattern for Data Layer Abstraction

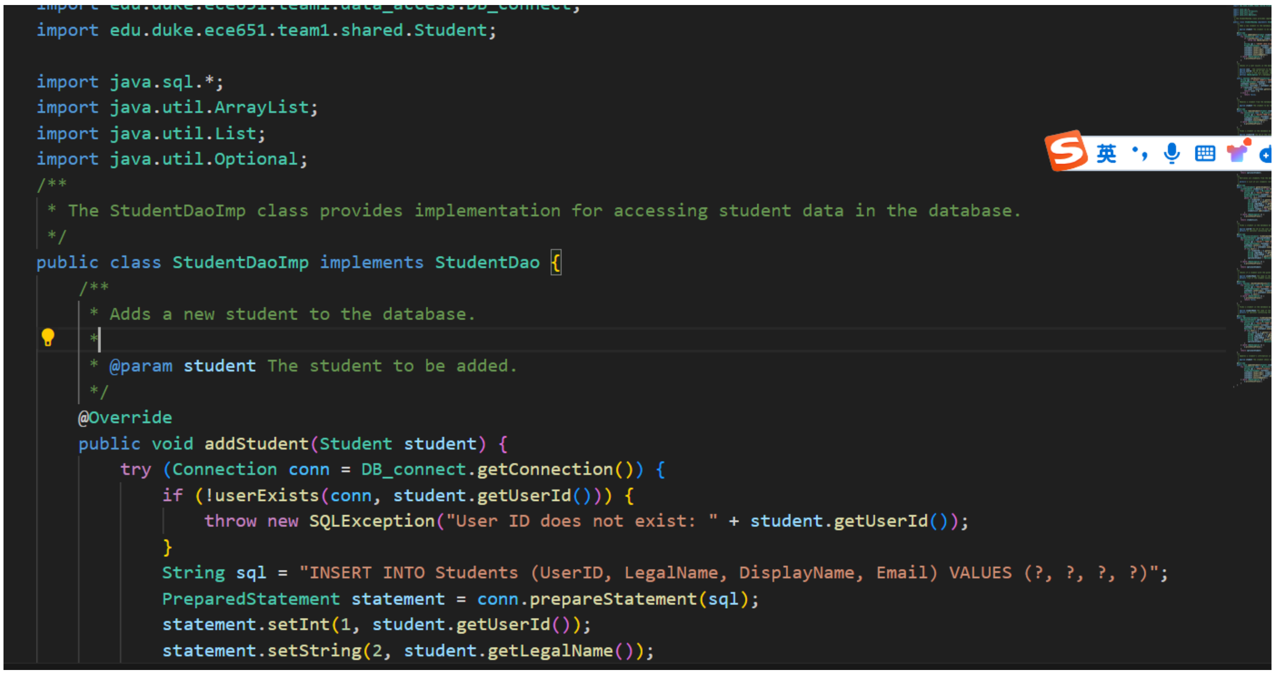
Eg： DAO Pattern for Student Data Management

Implementation

For managing student-related data, we implemented the `StudentDao` using the Data Access Object (DAO) pattern. The `StudentDao` interface defines the contract for operations that can be performed on student data, such as adding, removing, updating, and finding students. The `StudentDaoImp` class provides the concrete implementation of this interface and handles the actual database interactions.



```java
package edu.duke.ece651.team1.data_access.Student;

import edu.duke.ece651.team1.shared.Student;

import java.util.List;
import java.util.Optional;

public interface StudentDao {
    void addStudent(Student student);
    void removeStudent(Student student);
    Optional<Student> findStudentByStudentID(int studentID);
    Optional<Student> findStudentByUserID(int userID);
    List<Student> getAllStudents();
    boolean checkStudentExists(String name);
    Optional<Student> findStudentByName(String studentName);
    void updateStudent(Student student);
}
```



```java
import edu.duke.ece651.team1.data_access.DB_connect;
import edu.duke.ece651.team1.shared.Student;

import java.sql.*;
import java.util.ArrayList;
import java.util.List;
import java.util.Optional;
/**
 * The StudentDaoImp class provides implementation for accessing student data in the database.
 */
public class StudentDaoImp implements StudentDao {
    /**
     * Adds a new student to the database.
     *
     * @param student The student to be added.
     */
    @Override
    public void addStudent(Student student) {
        try (Connection conn = DB_connect.getConnection()) {
            if (!userExists(conn, student.getUserId())) {
                throw new SQLException("User ID does not exist: " + student.getUserId());
            }
            String sql = "INSERT INTO Students (UserID, LegalName, DisplayName, Email) VALUES (?, ?, ?, ?)";
            PreparedStatement statement = conn.prepareStatement(sql);
            statement.setInt(1, student.getUserId());
            statement.setString(2, student.getLegalName());
```

Rationale

The DAO pattern was selected for its ability to separate the application's data persistence and retrieval operations from its business logic. By decoupling the two, we gained the flexibility to change the underlying data storage mechanism without affecting the rest of the application. This pattern also promotes code reusability and better testability, as the DAOs can be easily mocked or stubbed in unit tests.

Adoption of Model-View-Controller (MVC) Architecture Across Applications

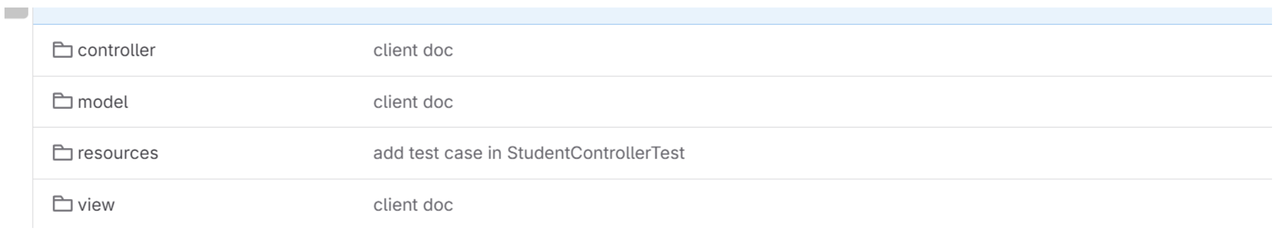
MVC Architecture in Application Design

- **Model** represents the application data and business rules.
- **View** is responsible for printing the data in terminal and handle user Input
- **Controller** acts as an intermediary between the model and the view

Consistent MVC Implementation Across Applications

Implementation

We ensured a consistent implementation of the MVC architecture across all applications within our suite. This includes consistent naming conventions, directory structures, and architectural patterns.



controller	client doc
model	client doc
resources	add test case in StudentControllerTest
view	client doc

Rationale

Maintaining consistency in the MVC implementation across different applications simplifies the development process, as developers can easily switch between projects without needing to relearn the application structure. It also streamlines the onboarding process for new team members and enhances the maintainability of the codebase. By adhering to a consistent architectural approach, we ensure that our applications are robust, scalable, and easy to extend with new features or integrate with other systems.