

---

# PROJECT REPORT

for

## JDBC School Management Project

Version 1.0

Prepared by : 1. Rutul Patel (IMT2022021)  
2. Varnit Mittal (IMT2022025)  
3. Aditya Priyadarshi (IMT2022075)  
4. Hemang Seth (IMT2022098)

Submitted to : Sujit Kumar Chakrabarti  
Lecturer

November 26, 2024

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Purpose . . . . .	4
1.2	Document Conventions . . . . .	4
1.3	Intended Audience and Reading Suggestions . . . . .	4
1.4	Project Scope . . . . .	5
1.5	References . . . . .	6
<b>2</b>	<b>Overall Description</b>	<b>7</b>
2.1	Product Perspective . . . . .	7
2.2	User Classes and Characteristics . . . . .	7
2.3	Product Functions . . . . .	7
2.3.1	Actors . . . . .	9
2.3.2	Core Use Cases: . . . . .	9
2.3.3	Entities . . . . .	10
2.3.4	Processes . . . . .	10
2.3.5	Data Stores . . . . .	10
2.3.6	Data Flow . . . . .	10
2.4	Operating Environment . . . . .	11
2.5	Design . . . . .	11
<b>3</b>	<b>Folder Structuring</b>	<b>15</b>
3.1	Folder Structuring Flowchart . . . . .	15
3.2	Folder Structure . . . . .	15
3.3	Detailed Explanation of Each Component . . . . .	16
3.3.1	Main Entry Point . . . . .	16
3.3.2	Service Layer . . . . .	16
3.3.3	DAO Layer . . . . .	16
3.3.4	Models . . . . .	17
3.3.5	Database Management . . . . .	17
3.3.6	SQL Scripts . . . . .	18
3.3.7	Testing Layer . . . . .	18
3.4	Folder Relationships and Dependencies . . . . .	19
<b>4</b>	<b>Code Overview</b>	<b>21</b>
4.1	Code Architectural Description . . . . .	21
4.1.1	SQL Scripts . . . . .	21
4.1.2	Main Application Files . . . . .	21

4.1.3	Service Layer . . . . .	21
4.1.4	DAO Layer . . . . .	22
4.1.5	Database Management . . . . .	23
4.1.6	Entity Models . . . . .	23
4.1.7	Factory Pattern Implementation . . . . .	24
4.1.8	Testing Layer . . . . .	24
4.2	Conclusion . . . . .	24
<b>5</b>	<b>Conclusion</b>	<b>26</b>
5.1	System Maintenance . . . . .	26
5.2	Performance and Scalability . . . . .	26
5.3	Flexibility and Adaptability . . . . .	26
5.4	Code Conclusion and Folder Structuring . . . . .	26
5.4.1	Layered Architecture . . . . .	26
5.4.2	File Organization . . . . .	27
5.4.3	Conclusion on Code . . . . .	27

# 1 Introduction

## 1.1 Purpose

The purpose of this **Project** is to define the comprehensive requirements for the JDBC-based School Management Project. This project aims to develop a **robust**, **efficient**, and **scalable** school management system that leverages *Java Database Connectivity (JDBC)* to seamlessly integrate with a relational database. The proposed system will streamline administrative tasks, enhance data integrity, and improve the overall management of school operations, including **student information**, **staff management**, **class schedules**, and the **library management system** for the school. By defining clear *functional* and *non-functional* requirements, this document serves as a foundation for development, ensuring that the final system aligns with the school's operational needs and enhances the **user experience** for both administrators and faculty.

## 1.2 Document Conventions

Headings are in bold, and requirements are numbered sequentially for easy reference. High-level requirement priorities are inherited by detailed requirements, with each major requirement statement retaining its own priority level. Important in-between terms are also in bold.

## 1.3 Intended Audience and Reading Suggestions

This SRS document is intended for all stakeholders involved in the JDBC School Management Project, including:

- **Project Managers:** To understand the scope and objectives for effective planning.
- **Developers and Database Administrators:** To review system requirements and database configurations.
- **Quality Assurance Teams:** To design test cases based on outlined functionalities.
- **School Administrators and Users:** To understand system features for training and usage.

## 1.4 Project Scope

The JDBC School Management Project aims to develop a comprehensive software system for streamlining school operations, including student information management, staff records, scheduling, and library management system. Leveraging Java Database Connectivity (JDBC), the project will integrate a user-friendly Java application with a relational database, enabling secure, real-time access to data. The system will address the limitations of manual record-keeping by automating core administrative tasks, reducing errors, and ensuring data consistency. Key stakeholders—school administrators, teachers, and support staff—will benefit from the system’s efficiency, as it will facilitate smoother communication, data accessibility, and decision-making.

The scope of the project includes the development of user roles for different levels of access, including administrators, teachers, and support staff, each tailored to their responsibilities. Administrators will have comprehensive access to manage all school operations, while teachers will focus on student performance. Role-based access ensures data security, preventing unauthorized access and maintaining user-specific functionalities.

Key components of the project include:

- **Data Integrity and Consistency:** Built-in database constraints and validation checks will ensure data accuracy, preventing duplication and maintaining up-to-date records.
- **Real-Time Reporting:** Customizable reporting tools for generating real-time insights on student performance, faculty workload, financials, and overall school metrics.
- **Role-Based Security:** A secure authentication and authorization framework to provide role-specific access and maintain sensitive data privacy.
- **Adaptability for Future Growth:** The system is designed with scalability in mind, allowing for future upgrades, additional modules, or integration with other educational tools or databases.

Figure 1.1 (Entire work-flow) is the overview of the project.

- **Student Management:** Handles the addition, update, removal, and retrieval of student details.
- **Course Management:** Facilitates course creation, modification, deletion, and listing of courses by teacher.
- **Library Management:** Manages books, links them to courses, and allows removal and viewing of course materials.
- **Teacher Management:** Manages teacher records, including addition, update, removal, and retrieval.

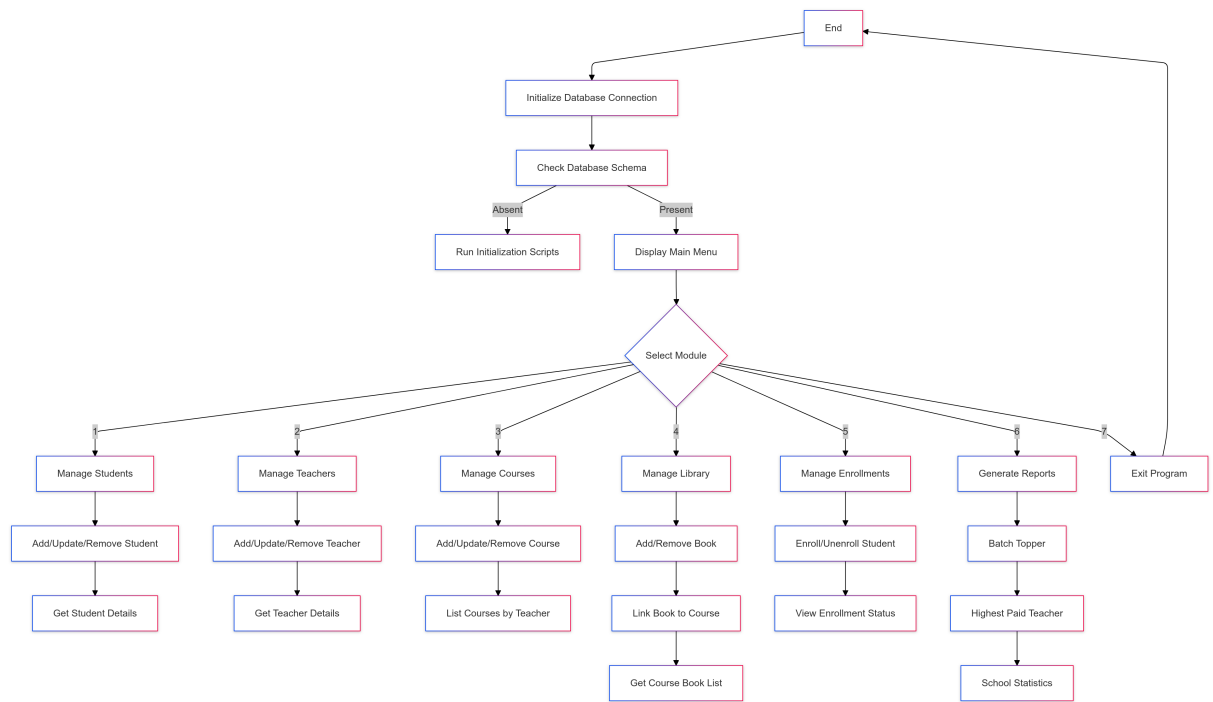


Figure 1.1: Entire work-flow

## 1.5 References

1. IEEE Std 830-1998, "IEEE Recommended Practice for Software Requirements Specifications," IEEE, 1998.
2. R. Pressman, *Software Engineering: A Practitioner's Approach*, 9th ed. McGraw-Hill, 2014.

## 2 Overall Description

### 2.1 Product Perspective

The proposed system is an integrated solution for managing academic and administrative operations in educational institutions. It aims to streamline processes such as student, teacher, course, enrollment, and library management, thereby reducing manual tasks and improving data accuracy. The system will serve administrators, faculty, and students, providing efficient tools for data management, reporting, and course material handling. It will offer a secure, scalable web-based platform with a user-friendly interface, capable of integrating with existing institutional databases. By providing real-time data access and analytics, the system will support informed decision-making and enhance operational efficiency.

### 2.2 User Classes and Characteristics

“JDBC School Management Tool” has basically 4 types of users.

- Teachers
- Students
- Admin
- Librarian

The Teacher manages course content and interacts with students, while the Student accesses materials, tracks progress, and enrolls in courses. The Admin oversees the system, manages users, and generates reports, and the Librarian manages library resources and book loans.

### 2.3 Product Functions

The **Use Case Diagram** for the School Management System provides an overview of the interactions between different actors and the system. It defines the primary functionalities available to each user role, along with the system’s core operations related to student, teacher, course, and library management. The system includes the following key actors and their respective interactions:

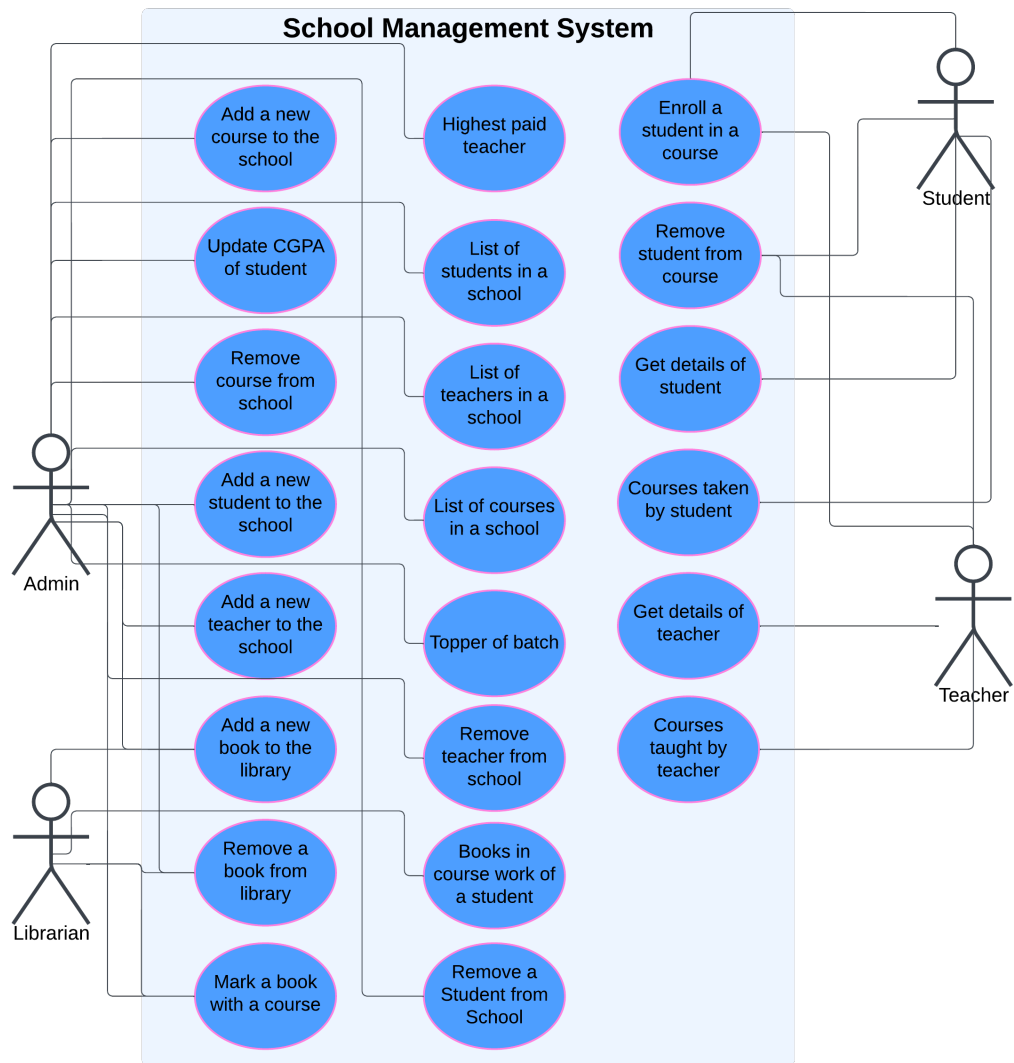


Figure 2.1: Type of Users



### 2.3.1 Actors

- **Admin:** Responsible for high-level management tasks, including adding and removing students, teachers, and courses. The admin also updates critical data such as student CGPA and teacher salaries, retrieves lists of all students, teachers, and courses in the school, and determines the highest-achieving student and the highest-paid teacher.
- **Student:** Enrolls in and removes themselves from courses, checks their details, and views the list of courses in which they are enrolled.
- **Teacher:** Manages student enrollments and removals within courses, checks their personal details, and reviews the courses they teach.
- **Librarian:** Manages the library catalog by adding and removing books, associating books with specific courses, and reviewing books relevant to students' coursework.

### 2.3.2 Core Use Cases:

- **Student Management:** Includes functionalities to add or remove students, view student details, and update a student's CGPA. Admin handles student additions/removals and CGPA updates, while students can enroll in or withdraw from courses.
- **Teacher Management:** Involves adding and removing teachers, adjusting teacher salaries, and obtaining teacher details. Teachers can enroll or remove students from courses, view their own information, and see the courses they teach.
- **Course Management:** The admin can add or remove courses, view all available courses, and link books to courses. Both students and teachers can interact with courses based on enrollment.
- **Library Management:** The admin and librarian handle adding/removing books and associating books with courses. Librarians also retrieve information about books in students' course materials.

Each use case is designed to facilitate seamless communication between actors and system functionalities, supporting efficient school administration and resource management.

This **Data Flow Diagram (DFD)** illustrates the high-level architecture of a school management system, outlining key entities, processes, and data stores for an SRS document.

### 2.3.3 Entities

- **Admin:** Central authority responsible for overseeing processes related to student, teacher, and course management.
- **Teacher, Student, Librarian:** System users interacting with various modules—teachers manage grades and courses, students handle enrollment and grade requests, and librarians oversee library inventory.

### 2.3.4 Processes

- **Information Management Modules:** Separate modules for managing Student, Teacher, and Course information, processing requests and maintaining their respective records.
- **Enrollment, Grade, and Library Management:** Processes focused on student enrollment, grade submission, and library inventory, interfacing with relevant databases.

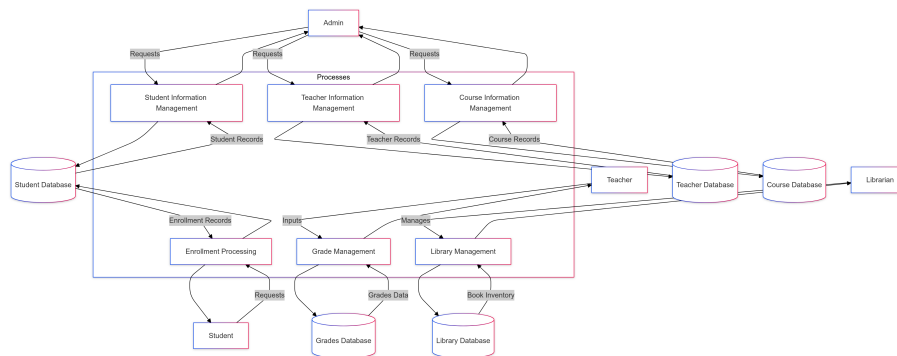


Figure 2.2: Data Flow Diagram

### 2.3.5 Data Stores

- **Student, Teacher, and Course Databases:** Hold core records for students, teachers, and courses, updated by respective processes.
- **Grades and Library Databases:** Specialized storage for grades and library inventory, supporting academic and resource management.

### 2.3.6 Data Flow

Data flows between entities, processes, and databases, with arrows indicating updates, requests, and record exchanges. This DFD provides an overview of system interactions and data dependencies at a high level, suitable for specifying functional requirements in the SRS.

## 2.4 Operating Environment

The website will be operate in any Operating Environment - Mac, Windows, Linux etc.

## 2.5 Design

- **School Class:** Represents the school itself. It aggregates Students, Teachers, and Courses (indicating that a school can manage multiple students, teachers, and courses). It has methods for adding and removing these entities.
- **Person Class:** This is a base class inherited by Student and Teacher. It contains common properties like name, date of birth, and address.
- **Student Class:** Inherits from Person and has attributes specific to students (e.g., roll number, CGPA). Students can enroll in courses and have their CGPA updated.
- **Teacher Class:** Also inherits from Person and has attributes specific to teachers (e.g., employee ID, salary). Teachers can teach courses and have their salary incremented.
- **Course Class:** Represents a course and can contain a list of books. Courses can be taken by students and taught by teachers.
- **Book Class:** Represents books in the library and can be marked with specific courses.
- **Library Class:** Represents the school's library and holds books.
- **Associations:**
  - **Aggregation:** A school aggregates students, teachers, and courses. This means that the school can exist independently of the students, teachers, and courses, but these entities are part of the school.
  - **Composition:** A course is composed of a teacher, meaning that a course cannot exist without a teacher.

Students are enrolled in courses, teachers teach courses, and courses can have books.

- **Inheritance:** Both Student and Teacher inherit from Person, sharing common attributes.

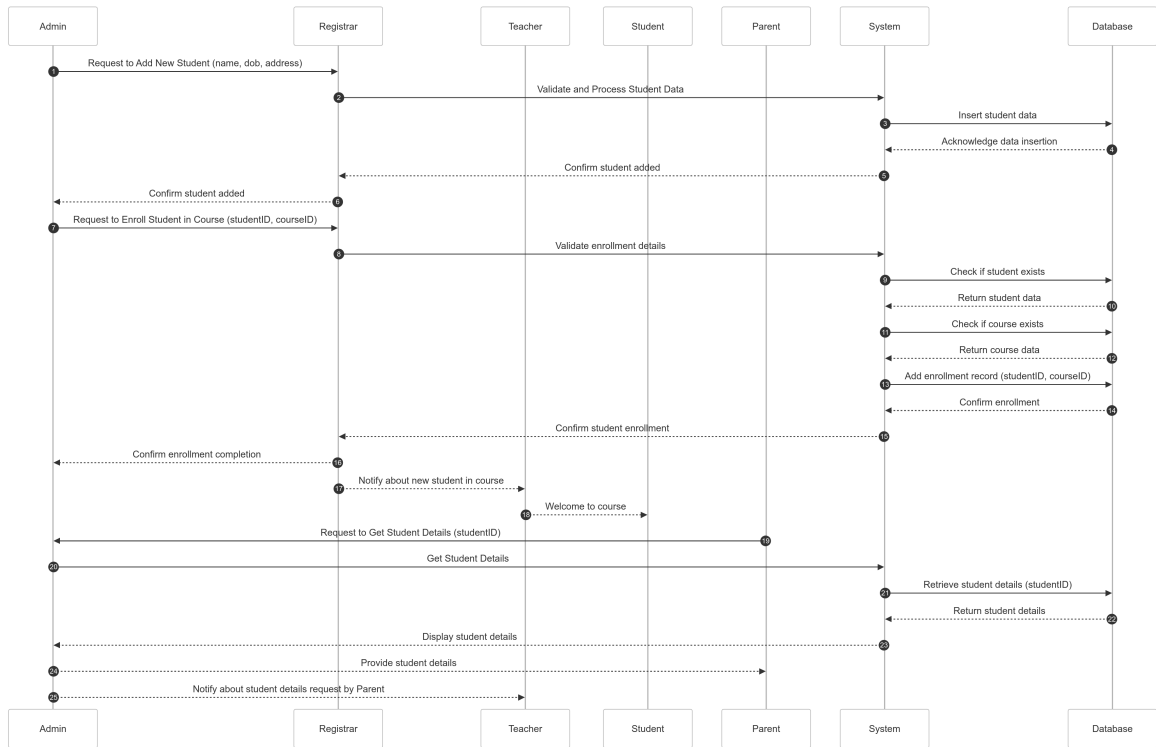


Figure 2.3: Student Activities

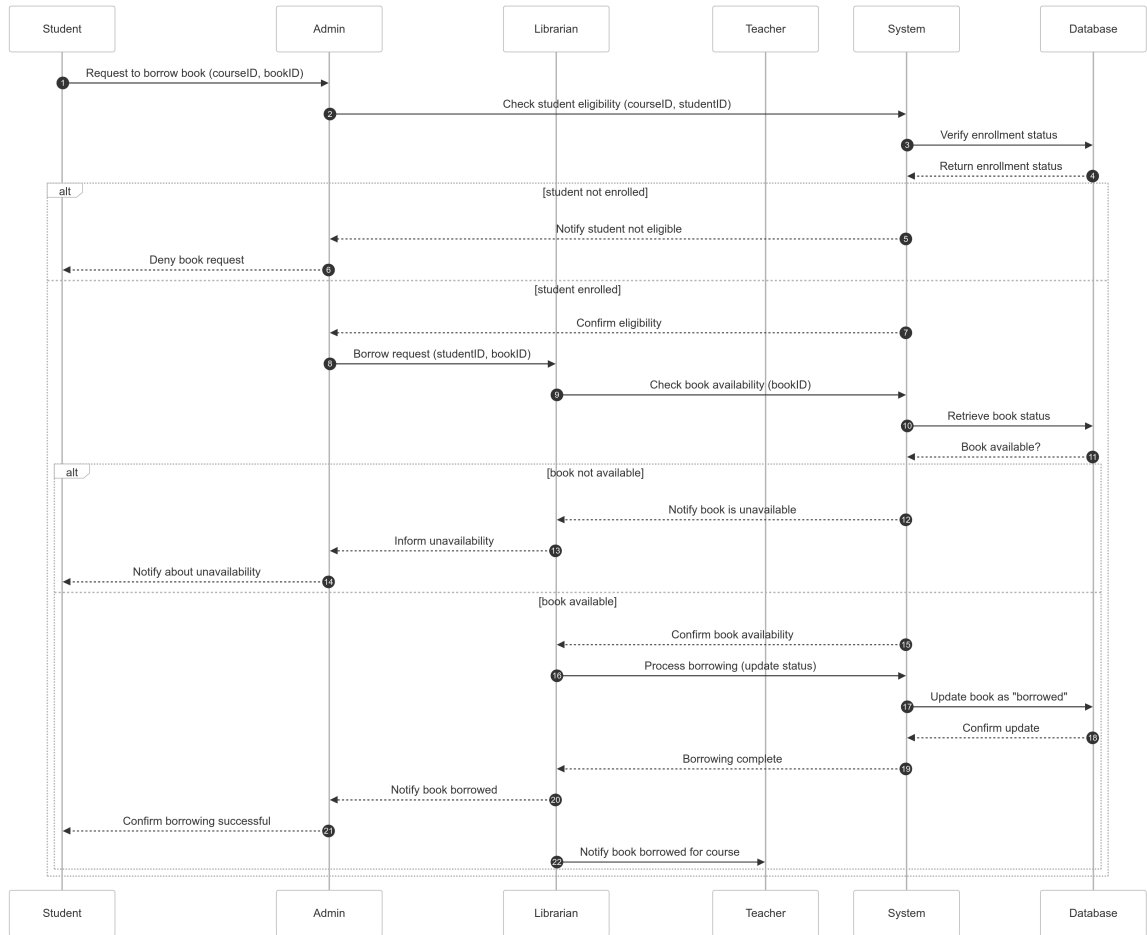


Figure 2.4: Teacher Activities

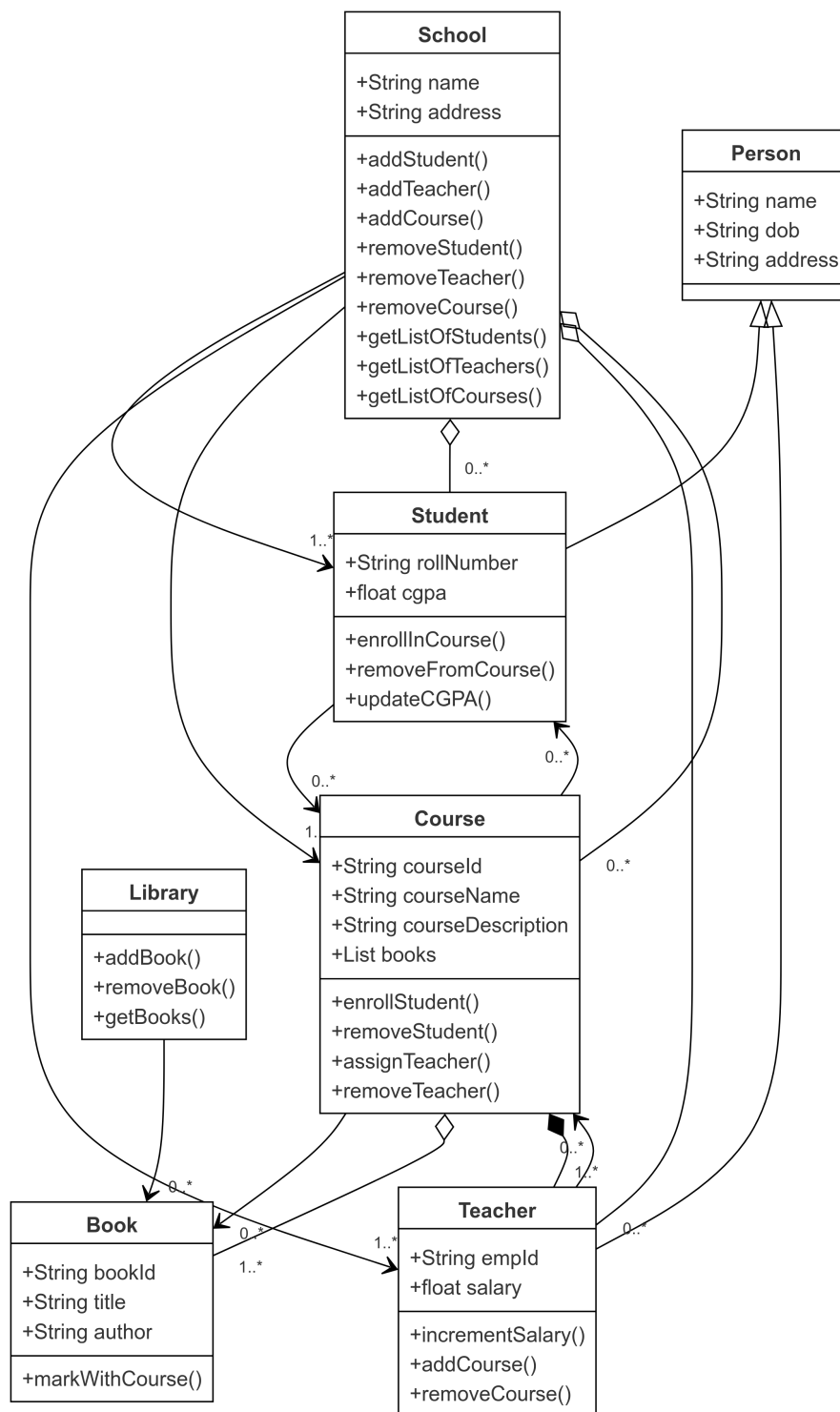


Figure 2.5: Class Diagram for JDBC School Management Tool



```
    schema.sql
src/
    com/schoolmanagement/
        dao/
        database/
        factory/
        main/
        models/
        services/
        tests/
```

## 3.3 Detailed Explanation of Each Component

### 3.3.1 Main Entry Point

- **Main.java:** This file serves as the primary entry point of the application. It interacts with the `SchoolService` class to perform various operations.
- **JdbcDemo.java:** Demonstrates example executions of database interactions, providing sample usage of the implemented functionalities.

### 3.3.2 Service Layer

- **SchoolService.java:** This class contains business logic and acts as an intermediary between the **DAO Layer** and the application logic.
  1. Handles calls from `Main.java`.
  2. Uses DAO classes to interact with the database.
  3. Processes business rules and integrates data from the `Models`.

### 3.3.3 DAO Layer

- **BaseDAO.java:** An abstract class providing basic database operation methods such as `create`, `read`, `update`, and `delete`.
- **BookDAO.java**, **CourseDAO.java**, **LibraryDAO.java**, **StudentDAO.java**, **TeacherDAO.java:** These classes extend `BaseDAO` and implement CRUD operations for specific entities such as books, courses, libraries, students, and teachers.



```

package com.schoolmanagement.services;

import com.schoolmanagement.dao.*;
import com.schoolmanagement.models.*;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.List;

public class SchoolService {

    private final StudentDAO studentDAO;
    private final TeacherDAO teacherDAO;
    private final CourseDAO courseDAO;
    private final BookDAO bookDAO;
    private final LibraryDAO libraryDAO;
    private ResultSet rs;

    public SchoolService(Connection connection) {
        this.studentDAO = new StudentDAO(connection);
        this.teacherDAO = new TeacherDAO(connection);
        this.courseDAO = new CourseDAO(connection);
        this.bookDAO = new BookDAO(connection);
        this.libraryDAO = new LibraryDAO(connection);
    }
}

```

Figure 3.2: Services Layer of the project and its file rendering due to its modularity

### 3.3.4 Models

- **Book.java, Course.java, Library.java, Person.java, Student.java, Teacher.java:**
  1. These classes represent database entities.
  2. **Person.java** acts as a parent class for **Student.java** and **Teacher.java**.
  3. Each class maps to a database table and contains fields that correspond to table columns.

### 3.3.5 Database Management

- **DatabaseManager.java:** Manages the database connection pool and provides a connection object to DAO classes. It acts as a singleton to ensure that a single instance handles all database connections.

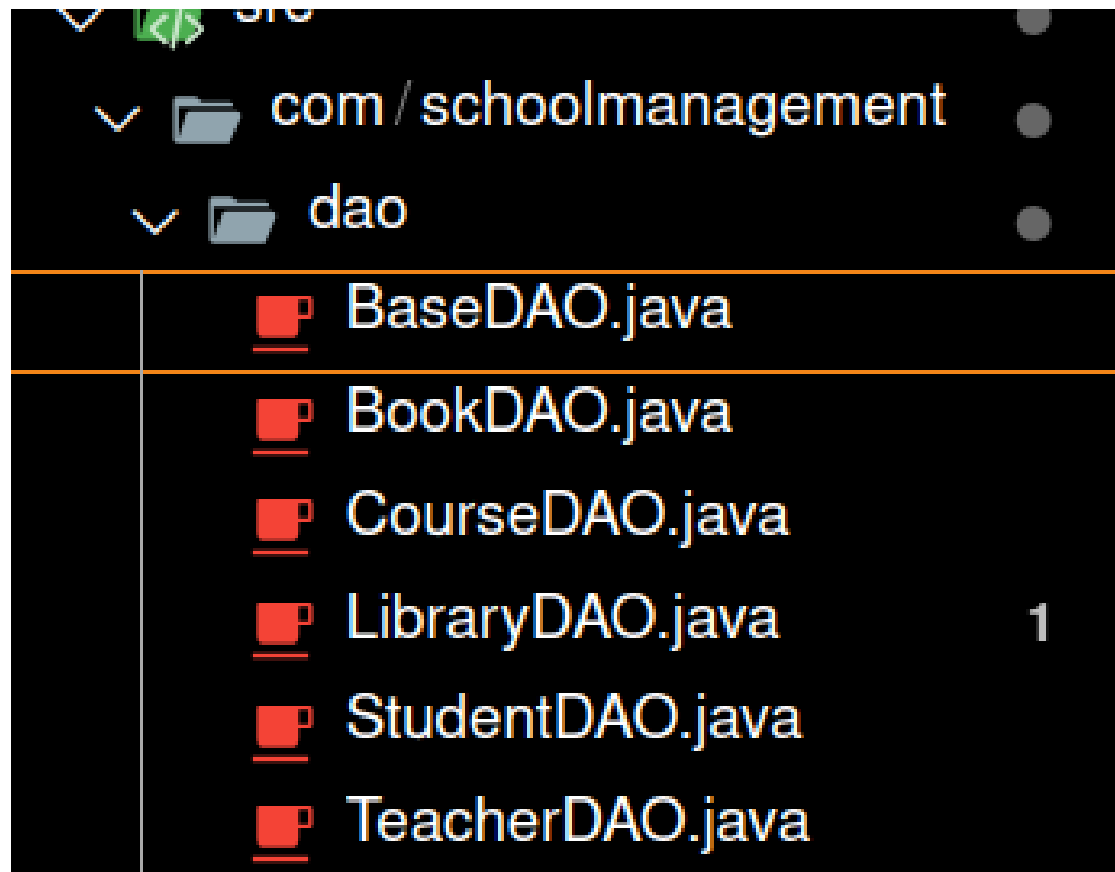


Figure 3.3: The use of DAO files and their folder structuring

### 3.3.6 SQL Scripts

- **schema.sql:** Contains SQL commands to set up the initial database schema, defining tables, primary keys, and relationships.
- **alter.sql:** Defines schema alterations, such as adding or modifying columns and constraints.
- **insert.sql:** Populates the database with initial data for testing purposes.

### 3.3.7 Testing Layer

- **BookDAOTest.java, CourseDAOTest.java, StudentDAOTest.java, TeacherDAOTest.java:**
  1. These files contain unit tests for the DAO classes to verify that CRUD operations are implemented correctly.
  2. They use mock data and assertions to validate the behavior of DAO methods.

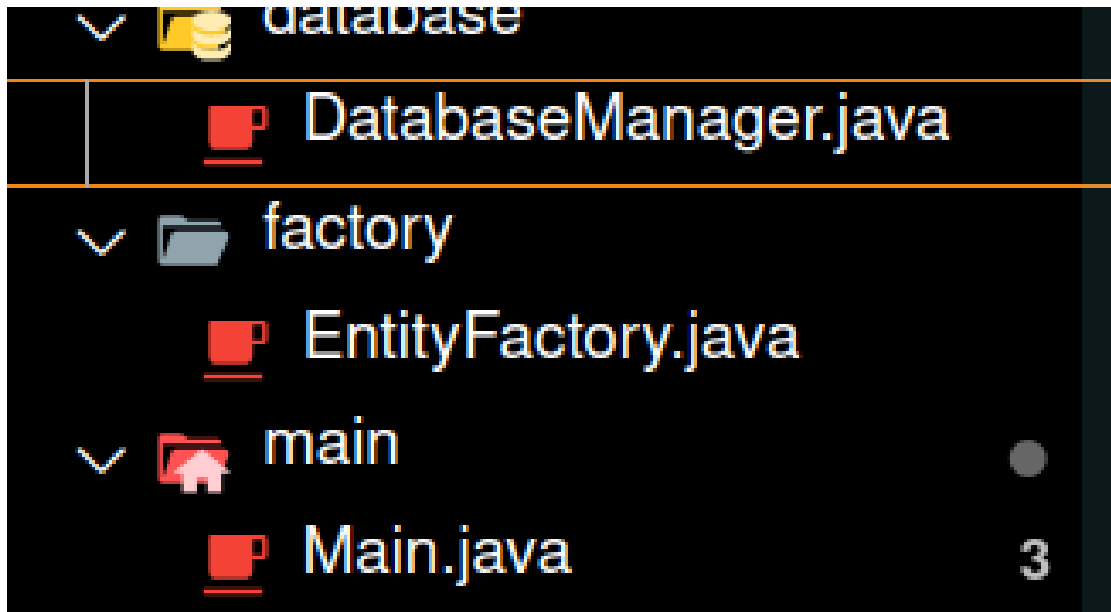


Figure 3.4: Folder structure for the entire project

- **imt2022025\_school.java:** Conducts integration tests to ensure that the service layer functions correctly with the DAO layer and database.

### 3.4 Folder Relationships and Dependencies

The folder structure shows clear separation of concerns, ensuring modularity and maintainability:

- The **Main.java** file communicates with the **SchoolService** class to handle user inputs and execute application logic.
- The **SchoolService** class interacts with the DAO layer to perform database operations.
- The DAO layer uses **DatabaseManager** to connect to the database and processes CRUD operations on entity classes.
- The models represent the structure of the database and are used throughout the service and DAO layers.
- SQL scripts ensure easy setup, alteration, and population of the database.
- Test files validate the functionality of individual components and their integration.

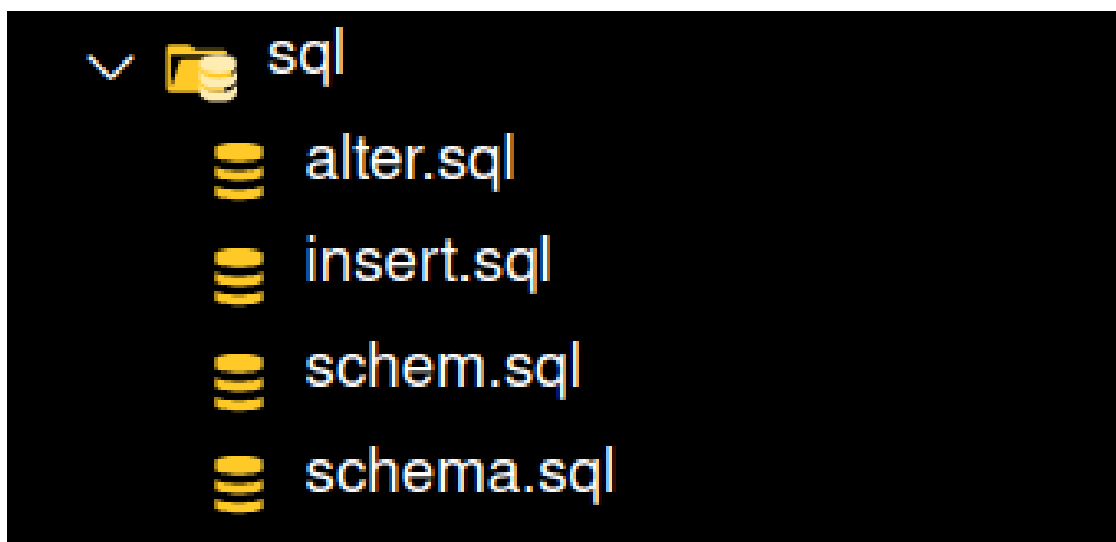


Figure 3.5: Folder structure for the entire project

## 4 Code Overview

### 4.1 Code Architectural Description

We have incorporated a **Layered Architecture** taught in class that makes the code structured into various modules that can be called for better

#### 4.1.1 SQL Scripts

The `sql/` folder contains scripts for database setup and data manipulation:

- **alter.sql**: Contains *SQL commands* to **modify the database schema**, such as altering tables or adding new columns.
- **insert.sql**: Provides initial *data insertion scripts* for testing and populating tables.
- **schem.sql** and **schema.sql**: Defines the **database schema**, including table structures and relationships.

#### 4.1.2 Main Application Files

Located in the `main/` folder, these files serve as the entry point for the application:

- **Main.java**: The *primary entry point*, orchestrating user interactions and triggering **service methods**.
- **JdbcDemo.java**: Demonstrates database operations by showcasing example queries and interactions.

#### 4.1.3 Service Layer

The `services/` folder contains the business logic:

- **SchoolService.java**: Serves as the *business logic layer*, connecting the **DAO layer** with the application and implementing core functionalities.

```
-- Create the school database
CREATE DATABASE IF NOT EXISTS school_db;
USE school_db;

-- Table for Students
CREATE TABLE IF NOT EXISTS students (
    id INT AUTO_INCREMENT PRIMARY KEY,
    roll_number VARCHAR(20) NOT NULL UNIQUE,
    name VARCHAR(100) NOT NULL,
    dob DATE NOT NULL,
    address VARCHAR(255),
    cgpa FLOAT NOT NULL
);

-- Table for Teachers
CREATE TABLE IF NOT EXISTS teachers (
    id INT AUTO_INCREMENT PRIMARY KEY,
    emp_id VARCHAR(20) NOT NULL UNIQUE,
    name VARCHAR(100) NOT NULL,
    dob DATE NOT NULL,
    address VARCHAR(255),
    salary FLOAT NOT NULL
);
```

Figure 4.1: Glimpses from SQL code

#### 4.1.4 DAO Layer

The `dao/` folder provides the **Data Access Object (DAO)** pattern for handling database interactions:

1. **BaseDAO.java**: An abstract base class providing generic *CRUD operations*.
2. **BookDAO.java**: Implements methods to **manage books** in the database.
3. **CourseDAO.java**: Handles *course-related data*, such as course information and registrations.
4. **LibraryDAO.java**: Provides functions to manage the **library table**, connecting books with courses.
5. **StudentDAO.java**: Manages database operations for *student records*.
6. **TeacherDAO.java**: Implements *CRUD operations* for teacher-related data.

```

110 import com.schoolmanagement.database.DatabaseManager;
111 import java.util.InputMismatchException;
112 import java.util.List;
113 import java.util.Scanner;
114
115 public class Main {
116
117     private static SchoolService schoolService;
118
119     Run | Debug
120     public static void main(String[] args) {
121         String jdbcURL = "jdbc:mysql://localhost:3306/school_db"; // Replace
122         String username = "root"; // Replace with your DB username
123         String password = "admin"; // Replace with your DB password
124         try {
125             // Get the database connection
126             // Connection connection = DatabaseManager.getInstance().getConne
127             Connection connection = DriverManager.getConnection(jdbcURL, user
128
129             // Initialize the service with the connection
130             schoolService = new SchoolService(connection);

```

Figure 4.2: Here we can see the main function being called and all the necessary libraries as the final landing page

### 4.1.5 Database Management

The `database/` folder contains utility classes:

- **DatabaseManager.java**: Implements a **singleton pattern** to provide a shared database connection pool.

### 4.1.6 Entity Models

The `models/` folder contains classes representing database entities:

1. **Book.java**: Represents the *Book* entity with fields like **title**, **author**, and **ISBN**.
2. **Course.java**: Maps to the **Course** table, including attributes like **name** and **code**.
3. **Library.java**: Represents a library with relationships to books and courses.
4. **Person.java**: A base class for **Student** and **Teacher**, defining common attributes like **name** and **ID**.
5. **Student.java**: Extends **Person**, adding fields like *roll number* and **CGPA**.
6. **Teacher.java**: Extends **Person**, adding attributes like *expertise* and **salary**.

```

public static Object createEntity(String entityType) {
    switch (entityType) {
        case "Student":
            return new Student(id:0, rollNumber:"", name:"", dob:"", addr
        case "Teacher":
            return new Teacher(id:0, empId:"", name:"", dob:"", address:"
        case "Course":
            return new Course(courseId:0, courseCode:"", courseName:"", c
        case "Book":
            return new Book(id:0, bookId:"", title:"", author:"", library
        case "Library":
            return new Library(id:0, name:""); // Creating a default libr
        default:
            throw new IllegalArgumentException("Unknown entity type: " +
    }
}

```

Figure 4.3: Here we can see the factory.java code is given where we can see the description

#### 4.1.7 Factory Pattern Implementation

The `factory/` folder provides:

- **EntityFactory.java**: Implements the **factory design pattern** to create entity objects dynamically.

#### 4.1.8 Testing Layer

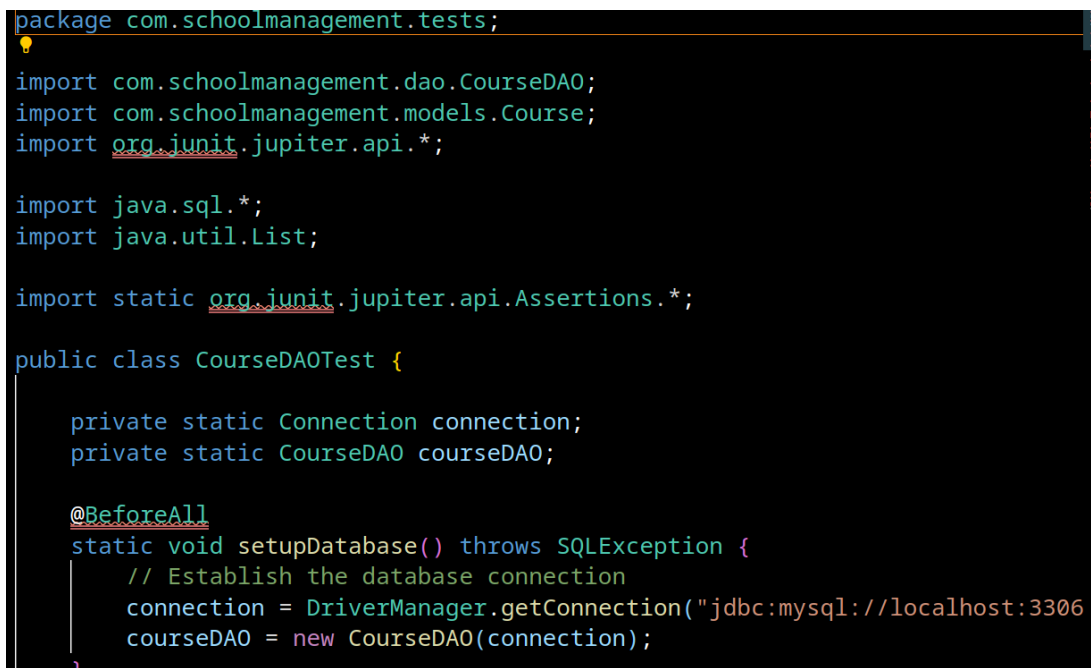
The `tests/` folder contains unit and integration tests:

1. **BookDAOTest.java**: Tests *CRUD operations* for books.
2. **CourseDAOTest.java**: Validates database operations for courses.
3. **StudentDAOTest.java**: Ensures the correctness of *student-related CRUD operations*.
4. **TeacherDAOTest.java**: Tests **teacher-related database interactions**.
5. **imt2022025\_school.java**: Validates the integration of **service and DAO layers**.

## 4.2 Conclusion

This project is organized into a **layered architecture** that separates concerns between the *DAO*, *service*, and *presentation* layers. The codebase follows principles of modularity, scalability, and maintainability, ensuring future extensibility.



A screenshot of a code editor with a dark background. The code is written in Java and includes package declarations, imports for CourseDAO, Course, JUnit, and standard Java classes like Connection and DriverManager. It defines a public class CourseDAOTest with static fields for a database connection and a CourseDAO object, and a @BeforeAll annotated static method setupDatabase that establishes the database connection.

```
package com.schoolmanagement.tests;

import com.schoolmanagement.dao.CourseDAO;
import com.schoolmanagement.models.Course;
import org.junit.jupiter.api.*;

import java.sql.*;
import java.util.List;

import static org.junit.jupiter.api.Assertions.*;

public class CourseDAOTest {

    private static Connection connection;
    private static CourseDAO courseDAO;

    @BeforeAll
    static void setupDatabase() throws SQLException {
        // Establish the database connection
        connection = DriverManager.getConnection("jdbc:mysql://localhost:3306
        courseDAO = new CourseDAO(connection);
    }
}
```

Figure 4.4: Calling the libraries and the code for testing is written

## 5 Conclusion

### 5.1 System Maintenance

The JDBC-based School Management System will require periodic maintenance to ensure its ongoing functionality. Updates and bug fixes will be necessary to adapt to evolving user needs and technological changes.

### 5.2 Performance and Scalability

The system should be designed to handle growing data and user traffic. Performance optimization and scalability are essential to maintain efficient operation, even with increased usage.

### 5.3 Flexibility and Adaptability

As requirements change, the system must be flexible enough to accommodate future features and modifications. It should allow for easy updates and integration with new modules as needed.

### 5.4 Code Conclusion and Folder Structuring

The **JDBC-based School Management System** adheres to a well-defined and modular *folder structure*, ensuring clear separation of concerns and maintainability. Each layer of the application—from **DAO**, **service**, **models**, to **main**—is structured to promote scalability and reusability.

#### 5.4.1 Layered Architecture

The project employs a **layered architecture** that clearly delineates responsibilities:

- The **DAO layer** handles all interactions with the database using `BaseDAO.java` and specialized classes like `BookDAO.java` and `StudentDAO.java`.
- The **Service layer**, represented by `SchoolService.java`, encapsulates the business logic, acting as a bridge between the DAO layer and the application logic.
- The **Model layer**, with entity classes such as `Book.java`, `Student.java`, and `Teacher.java`, maps the database schema to Java objects, enabling object-relational mapping.

- The **Main application files** (`Main.java` and `JdbcDemo.java`) serve as the entry point, managing user interaction and system execution.

### 5.4.2 File Organization

The folder structure is crafted to enhance modularity:

1. **SQL Folder:** Contains database setup scripts (`alter.sql`, `insert.sql`, `schema.sql`) to create, update, and populate the database.
2. **Database Folder:** Includes `DatabaseManager.java`, a central utility for managing database connections.
3. **Factory Pattern Implementation:** The `EntityFactory.java` file dynamically creates object instances, adhering to the *factory design pattern*.
4. **Testing Folder:** Contains unit test files such as `BookDAOTest.java` and `TeacherDAOTest.java` to ensure functionality at every layer.

### 5.4.3 Conclusion on Code

The entire system is built with maintainability and extensibility in mind. By following a structured approach, the codebase remains modular, making it easier to debug, test, and extend. Each folder serves a distinct purpose, reducing interdependency and ensuring that future developers can easily navigate and understand the system.