OOPS Project Report

Team Members

Sr. no.	Name	ID No.
1.	Shubham Gupta	2021A7PS0468P
2.	Shardul Shingare	2021A7PS2539P
3.	Aryaman Chauhan	2020B5A7PS2006P

Video Recordings: -
https://drive.google.com/drive/folders/1rWchMAN4QFlvOxHEV4ZBbQaxdo-oibur

GitHub link for Code: -
https://github.com/Aryaman-Chauhan/OOP_Library_Management

Solid Principles

Design principles inspire us to build software that is more maintainable, intelligible, and versatile. As a result, as our apps increase in size, we may lower their complexity and save ourselves a lot of trouble later on.
The five solid principles are-

1. Single Responsibility
2. Open-Closed
3. Liskov Substitution
4. Interface Segregation
5. Dependency Inversion

I. Single Responsibility

A class should have only one cause to modify, according to the Single Responsibility Principle.
Technically, only one software modification (database logic, logging logic, etc.) should modify the class's specification.
If a class is a data container, like a Book or Student class, and it has fields about that object, it should only change when we modify the data model.
It's crucial to follow the SRP. First, because numerous teams might work on the same project and edit the same class, incompatible modules can result. Second, it simplifies versioning. Say a persistence class that handles database operations changes in GitHub contributions. Following the SRP tells us it's storage or database-related.
In our code, we have followed the SRP by ensuring that each class targets a single issue. For example, Books.java is used just to modify fields related to Books, and we have three separate interfaces for searching Books, one for searching with name, one for searching with ISBN and one for searching with author.

main OOP_Library_Management / Student.java / Jump to Go to file

Aryaman-Chauhan Corrected Database_DAO Latest commit 809b4a0 3 minutes ago History

4 contributors

74 lines (62 sloc) 2.64 KB Raw Blame

```java
import java.sql.SQLException;
import java.util.*;
import java.time.LocalDate;

public class Student extends User {
    private String name;
    private TreeMap<Book, LocalDate> currBooks;
//    private double dues;
    Student(String name, String id, String password) { //Constructor for Student which takes 3 arguments and initializes the parameters
        this.name = name;
        this.id = id;
        super.setPassword(password);
        currBooks = new TreeMap<>();
//        dues = 0;
    }

//    public double getDues() { //Gets the dues owed by the student
```

main OOP_Library_Management / Librarian.java / Jump to Go to file

Aryaman-Chauhan Corrected Database_DAO Latest commit 809b4a0 3 minutes ago History

4 contributors

54 lines (51 sloc) 2.25 KB Raw Blame

```java
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.HashSet;

public class Librarian extends User {

    Librarian() { //Constructor for Librarian initializing parameters if no argument is passed
        this.id = "FXXXXXXXXXXX";
        this.password = "admin";
    }
    public void setFine(double fine,String pass){ //Sets the fine value
        if(pass.equals(this.password))
            User.fine = fine;
    }
    public void addBook(Book b) throws SQLException, ClassNotFoundException { //Adds a book B to the SQL database
        Database_DAO dao = new Database_DAO();
        int a = dao.addBookToDB(b);
```

main OOP_Library_Management / Book.java / Jump to Go to file

shar-dul Add files via upload Latest commit 0b8f5d6 3 hours ago History

3 contributors

80 lines (64 sloc) 2.65 KB Raw Blame

```java
public class Book implements Comparable<Book>{
    private String title;
    private String author;
    private Boolean available;
    private String isbn;
    private String genre;
    private String publisher;

    Book(String name, String author, String isbn, String genre, String publisher) { //Constructor for Book which takes 5 arguments as parameters and initializes
        this.title = name;
        this.author = author;
        this.isbn = isbn;
        this.genre = genre;
        this.publisher = publisher;
        available = true;
    }
```

II. Open-Closed

The Available-Closed Principle says classes should be open for extension but closed for modification.
Modification means changing the code of an existing class, and extension means adding new functionality.
This principle states that new class functionality should be addable without changing existing code. We risk producing bugs whenever we tweak existing code. We should avoid touching (mainly) tested production code.
Interfaces and abstract classes help.
For example, in our database.java file, we added methods to calculate old dues and also to get new dews of the student without changing the existing codebase. The previously tested code was not changed so we did not get any bugs after adding the new code.
The green portion shows the new code that was added without changing the existing code.

```
190 +    private static void addNewDues(double odue, int idno) throws SQLException, ClassNotFoundException {
191 +        connect();
192 +        String query = "UPDATE student SET dues=? WHERE idno=?";
193 +        pst = con.prepareStatement(query);
194 +        pst.setDouble(1,odue);
195 +        pst.setInt(2,idno);
196 +        pst.executeUpdate();
197 +    }
198 +
199 +    private static double getOldDues(int idno) throws SQLException, ClassNotFoundException {
200 +        connect();
201 +        String query = "SELECT dues FROM student where idno=?";
202 +        pst = con.prepareStatement(query);
203 +        pst.setInt(1,idno);
204 +        ResultSet rs = pst.executeQuery();
205 +        rs.next();
206 +        return rs.getDouble(1);
207 +    }
```

III. Liskov Substitution

According to the Liskov Substitution Principle, subclasses should be interchangeable with their base classes.
This means that, because class B is a subclass of class A, we should be able to send an object of class B to any method that expects an object of type A, and the method should not produce any unexpected results.
This is expected behaviour because we assume that when we use inheritance, the child class inherits everything that the superclass contains. The child class extends but never narrows the behaviour.

As a result, when a class violates this principle, it results in some serious errors that are difficult to identify.
Liskov's principle is simple to grasp but difficult to detect in code.
In our file database_dao.java, we have created a method sign in which has return type of user but in the return statements we have returned Student and Librarian which are subclasses of User. Therefore, liskov substitution applies in our code.

```
static public User signIn(String id, String password) throws SQLException, ClassNotFoundException {
    connect();
    String query = "SELECT sname,sid,pwd FROM student WHERE sid=? and pwd=?";
    pst = con.prepareStatement(query);
    pst.setString(1, id.toUpperCase());
    pst.setString(2, password);
    ResultSet rs = pst.executeQuery();// This ResultSet can be used to extract data like idno, pwd, etc.
    // If null, then wrong info, null can be checked using boolean rs.next()
    if(!(id.equalsIgnoreCase("ADMIN"))) {
        if (rs.next()) return new Student(rs.getString(1), rs.getString(2), rs.getString(3));
        return null;
    }else if(rs.next()) return new Librarian();
    return null;

}
```

IV. Interface Segregation

The Interface Segregation Principle is about separating the interfaces, and segregation is keeping things separate.
According to the idea, numerous client-specific interfaces are preferable to one general-purpose interface. Clients should not be forced to use a function they do not require.
In the below snippets of the code, we can see that we have used different interfaces for different functionalities required and all the interfaces are properly segregated into different codes. We have not used one interface for different searching methods instead created 3 different interfaces for 3 different searching methods.

```java
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.HashSet;

public interface searchByAuthor { //Searches for a book by it's author
    static HashSet<Book> search(String authorname, Database_DAO dao) throws SQLException, ClassNotFoundException {
        return dao.bookDetailsByAuth(authorname);
    }
}
```

```java
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.HashSet;

public interface searchByISBN { //Searches for a book by it's isbn code
    static Book search(String isbn, Database_DAO dao) throws SQLException, ClassNotFoundException {
        Book retBook = null;
        ResultSet rs = dao.bookDetailsByISBN(isbn);
        if(rs.isFirst())
            retBook = new Book(rs.getString(1),rs.getString(2),rs.getString(3),rs.getString(4),rs.getString(5));

        return retBook;
    }
}
```

```java
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.HashSet;

public interface searchByName { //Searches for a book by it's name
    static HashSet<Book> search(String name, Database_DAO dao) throws SQLException, ClassNotFoundException {
        return dao.bookDetailsByName(name);
    }
}
```

V. Dependency Inversion

The Dependency Inversion principle states that our classes should depend upon interfaces or abstract classes instead of concrete classes and functions. We want our classes to be open to extension, so we have reorganized our dependencies to depend on interfaces instead of concrete classes.
For this principle, we have ensured that we use interfaces to implement the search of Books on the basis of Author, Name and ISBN, instead of implementing searching algorithm using full-fledged classes.

```java
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.HashSet;

public interface searchByAuthor { //Searches for a book by it's author
    static HashSet<Book> search(String authorname, Database_DAO dao) throws SQLException, ClassNotFoundException {
        return dao.bookDetailsByAuth(authorname);
    }
}
```

14 lines (12 sloc) | 522 Bytes

```java
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.HashSet;

public interface searchByISBN { //Searches for a book by it's isbn code
    static Book search(String isbn, Database_DAO dao) throws SQLException, ClassNotFoundException {
        Book retBook = null;
        ResultSet rs = dao.bookDetailsByISBN(isbn);
        if(rs.isFirst())
            retBook = new Book(rs.getString(1),rs.getString(2),rs.getString(3),rs.getString(4),rs.getString(5));

        return retBook;
    }
}
```

```java
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.HashSet;

public interface searchByName { //Searches for a book by it's name
    static HashSet<Book> search(String name, Database_DAO dao) throws SQLException, ClassNotFoundException {
        return dao.bookDetailsByName(name);
    }
}
```

Design Pattern

DAO stands for Data Access Object. DAO Design Pattern is used to separate the data persistence logic in a separate layer. This way, the service remains completely in dark about how the low-level operations to access the database is done. This is known as the principle of Separation of Logic.
With DAO design pattern, we have following components on which our design depends:

- The model which is transferred from one layer to the other.
- The interfaces which provides a flexible design.
- The interface implementation which is a concrete implementation of the persistence logic.

OOP Principles

Abstraction

We have used an abstract class named ‘User’ in user.java file. It has some defined variables and also has non-abstract methods defined in it.

Inheritance

The classes ‘Student’ and ‘Librarian’ inherits the parent abstract class ‘User’. The subclass uses the methods defined in parent class. These methods include getID etc.

Polymorphism - Overriding

The toString() method of ‘Student’ class overrides the toString() method.
The run() method of ‘BookIssuer’ class overrides the run() method of runnable interface.
The run() method of ‘ReissuerBook’ class overrides the run() method of runnable interface.

Encapsulation

We have used getter and setter in various classes. For example getName, setName, getAuthor, setAuthor, getISBN, setISBN, getPublisher, setPublisher are used in ‘Book’ class.

Project Description

Book.java
This class contains all information about a particular book like its name, author, genre, International Standard Book Number (ISBN), publisher as well as availability of the book. Next, we have the getters and setters for all the instance fields. Then we have a compareTo() method which compares two books on the basis of their names. We also have an equals() method which returns a Boolean value depending on if two books have the same name as well as author. The toString() method returns all instance fields of the class in a systematic format.

BookIssuer.java
This class has the instance fields bookname and idNo. This class creates and runs a thread.

BookNotAvailableException.java
This is an exception which is thrown if someone wants to issue a book which has already been issued.

BookNotFoundException.java
This is an exception that is thrown if someone wants to access a book which does not exist. Database_DAO*

Librarian.java
This class extends the User class and includes all the information about a Librarian, having instance fields ID (having initial value “ADMIN”) and password (having initial value “admin”). We have a method addBook() which enables the librarian to add a book to the book database. We also have a removeBook() method which helps the librarian to remove a book from the book database. The reviewUser() method displays the student database, while the reviewBook() method displays the book database.
Next, we have a method setFine() which sets the fine value. with functions to calculate the fine, add and remove books, and functions to review users and books.

MaxBookLimitException.java
This is an exception which is thrown if a student has already issued three books and tries to issue more.
ReissuerBook.java
This class has the instance field name. This class creates and runs a thread.

Student.java
This class extends the User class and has the instance field name. This method creates a TreeMap which stores the books issued by a student and the dates when they were issued. Then we have getters and setters for these. The toString() method returns the name and ID of a student in a systematic format. The borrowBook() method is used to issue a book to a student, with the condition to check if the student has already issued three books. The returnBook() method is used to return a book. The reissueBook() method enables a student to issue a book which he/she had already issued.
first of all has getters and setters for its fields, which are the user's Name, Dues and a map which maps Books to the date of withdrawal. It also has functions to return and borrow and return books.

User.java
The User class has instance fields id, password and fine, which is initially equal to one. There are getter methods for id and password and a setter method to set the password.

searchByAuthor.java
This interface is used to search for a book by its author

searchByISBN
This interface is used to search for a book on the basis of its ISBN

searchByName
This interface is used to search for a book by its name

Important Features of Project

The following functionalities have been implemented for Student:

- Registration/ Sign-In
- Book Search by:
    - Title
    - Author
    - ISBN
- Book Issue:
    - Can issue at maximum 3 books
    - Will have due date set to 15 days ahead of current date
- Book Re-Issue: Can't be re-issued again
- View total dues.
- Return book:
    - Will show dues incurred for the returned book
    - Will add this to the total dues

The following functionalities have been implemented for Librarian:

- Review all Books
- Review all students
- Add/Remove Books
- Set Fine(Default set to Rs 1.0/day delay)

This project uses MySQL database which is implemented in the Database_DAO.java, which requires a .jar file, which needs to be imported in order for MySQL Driver to work.

Multithreading

We have used multithreading to handle issue requests and reissue requests. There are two classes generated which implement the Runnable interface and override the run() method - BookIssuer and ReissuerBook.

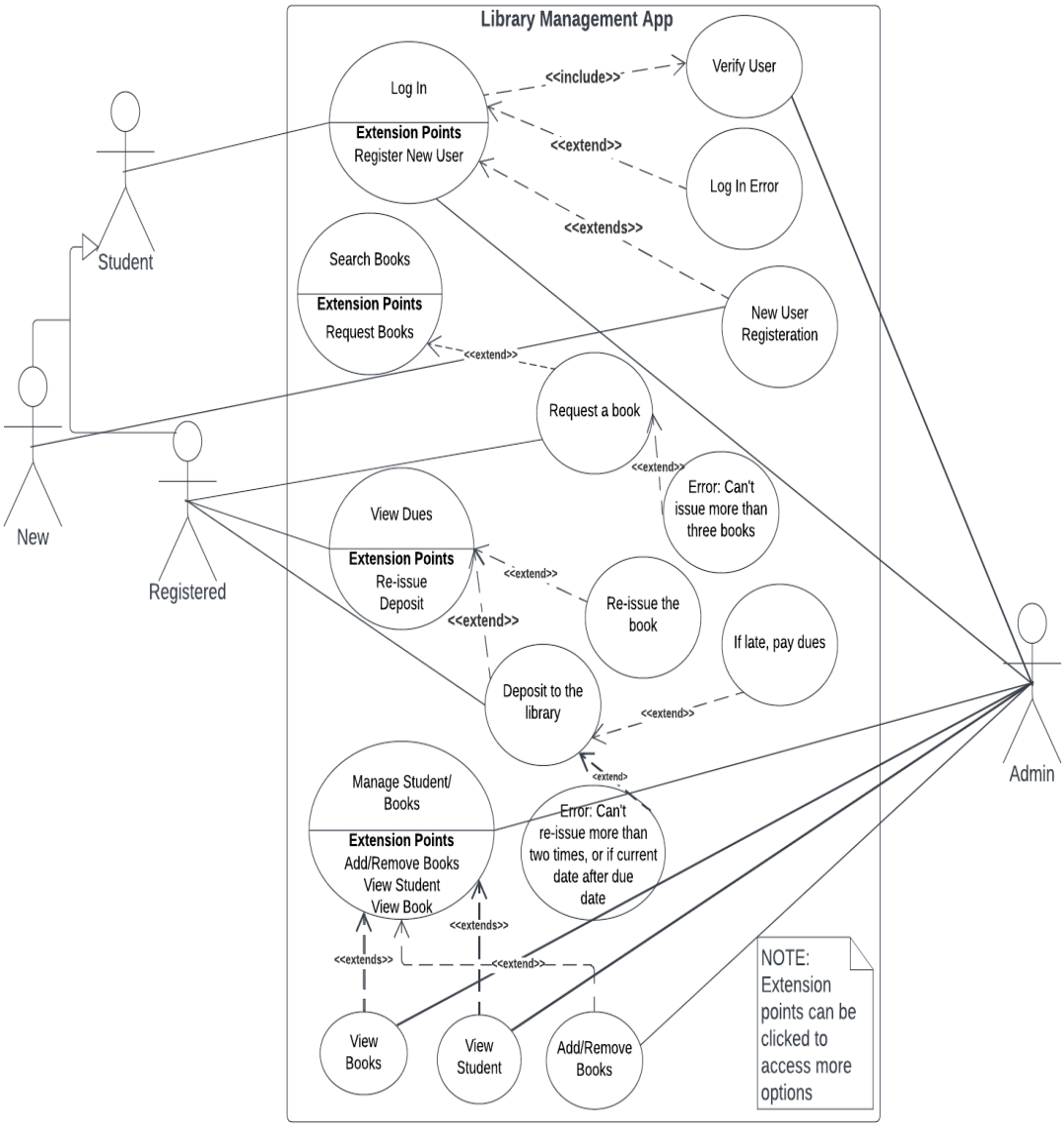
These 2 classes each have a synchronized function - issueBook() in BookIssuer and reissuerBook() in ReissuerBook().

The reason why they are synchronized is that if two students are trying to access the same book at the same time, then error would get generated. So we want to only one thread to access the function at one time.

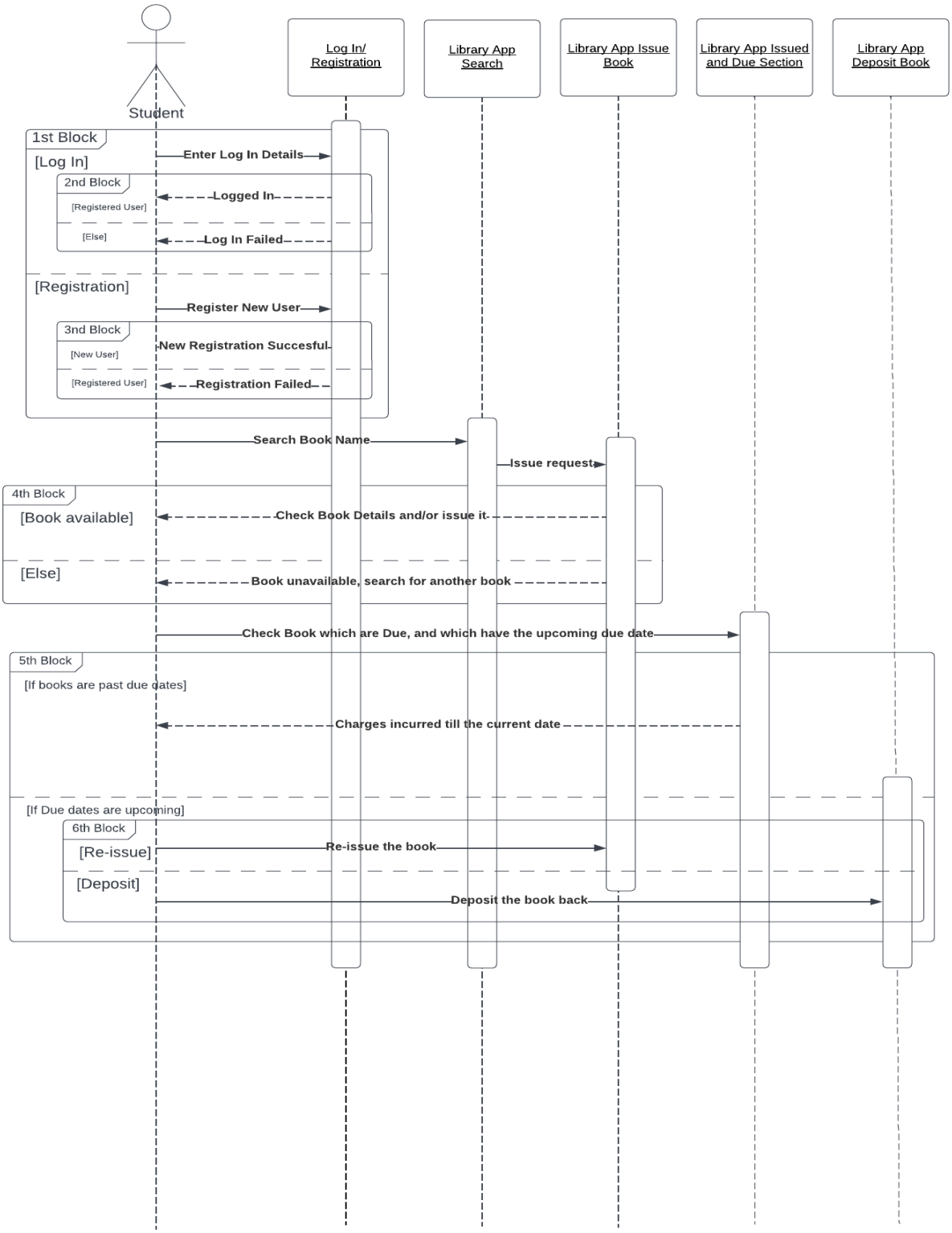
```
113     static public int issueBookDB(int idno, String bookname) throws InterruptedException {
114         BookIssuer r = new BookIssuer(idno,bookname);
115         Thread.sleep(100);
116         return r.ret;
117     }
118 
119     static public int reissueBookDB(String name) throws InterruptedException {
120         ReissuerBook r = new ReissuerBook(name);
121         Thread.sleep(100);
122         return r.ret;
123     }
```

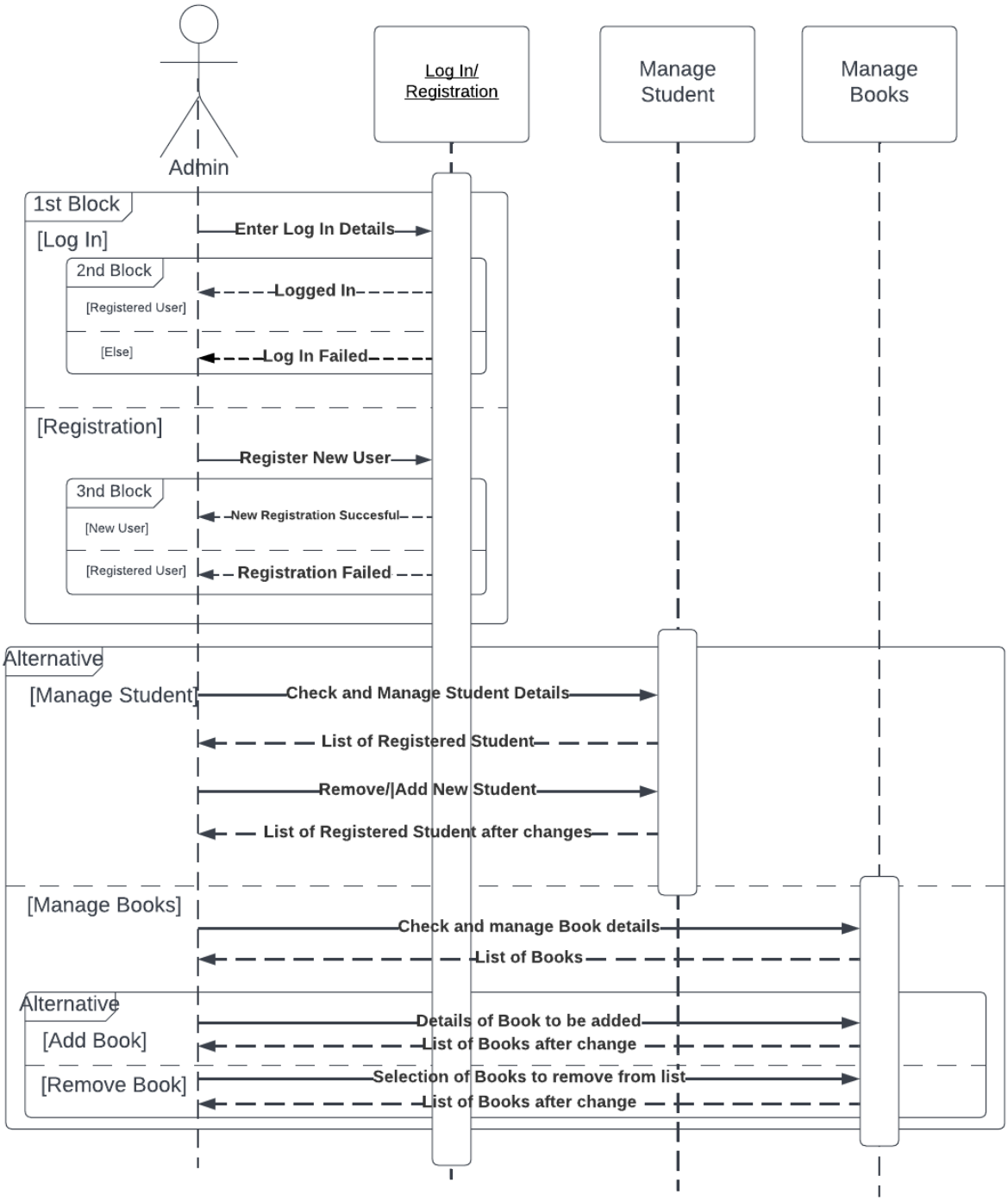
In Database_DAO class, we are creating separate threads for separate requests.

Use Case Diagram

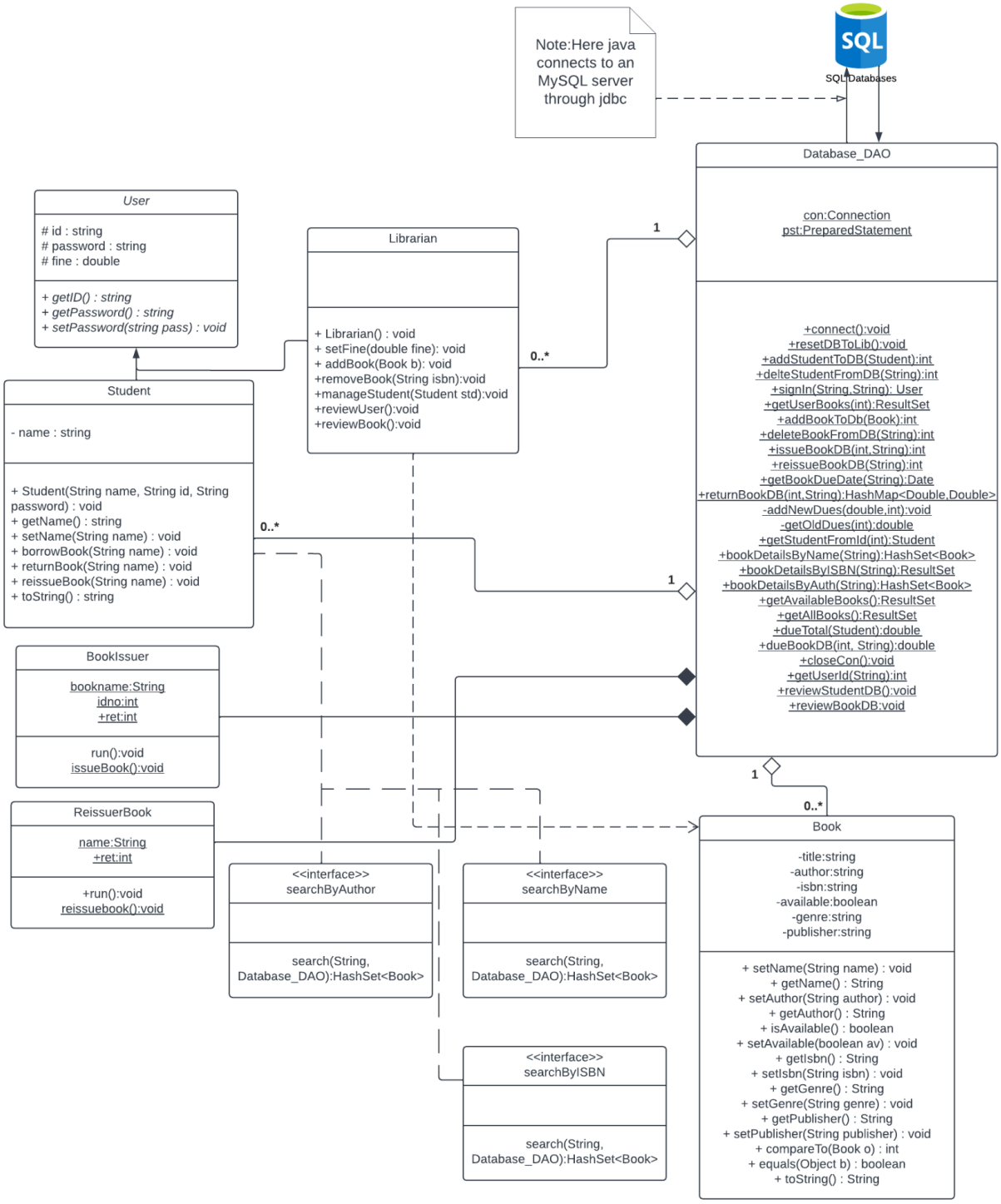


Sequence Diagrams

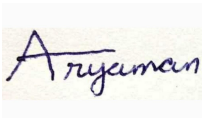
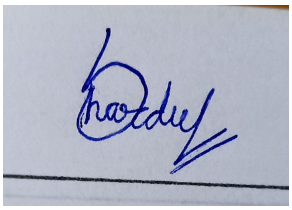
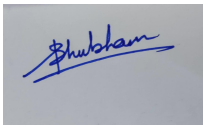



Admin
Log In/ Registration
Manage Student
Manage Books
1st Block
[Log In]
Enter Log In Details
2nd Block
[Registered User]
Logged In
[Else]
Log In Failed
[Registration]
Register New User
3nd Block
[New User]
New Registration Succesful
[Registered User]
Registration Failed
Alternative
[Manage Student]
Check and Manage Student Details
List of Registered Student
Remove/|Add New Student
List of Registered Student after changes
[Manage Books]
Check and manage Book details
List of Books
Alternative
[Add Book]
Details of Book to be added
List of Books after change
[Remove Book]
Selection of Books to remove from list
List of Books after change

UML Diagram


Note:Here java connects to an MySQL server through jdbc
SQL
SQL Databases
Database_DAO
con:Connection
pst:PreparedStatement
+connect():void
+resetDBToLib():void
+addStudentToDB(Student):int
+delteStudentFromDB(String):int
+signIn(String,String): User
+getUserBooks(int):ResultSet
+addBookToDb(Book):int
+deleteBookFromDB(String):int
+issueBookDB(int,String):int
+reissueBookDB(String):int
+getBookDueDate(String):Date
+returnBookDB(int,String):HashMap<Double,Double>
-addNewDues(double,int):void
-getOldDues(int):double
+getStudentFromId(int):Student
+bookDetailsByName(String):HashSet<Book>
+bookDetailsByISBN(String):ResultSet
+bookDetailsByAuth(String):HashSet<Book>
+getAvailableBooks():ResultSet
+getAllBooks():ResultSet
+dueTotal(Student):double
+dueBookDB(int, String):double
+closeCon():void
+getUserId(String):int
+reviewStudentDB():void
+reviewBookDB:void
User
id : string
password : string
fine : double
+ getID() : string
+ getPassword() : string
+ setPassword(string pass) : void
Librarian
+ Librarian() : void
+ setFine(double fine): void
+ addBook(Book b): void
+removeBook(String isbn):void
+manageStudent(Student std):void
+reviewUser():void
+reviewBook():void
Student
- name : string
+ Student(String name, String id, String password) : void
+ getName() : string
+ setName(String name) : void
+ borrowBook(String name) : void
+ returnBook(String name) : void
+ reissueBook(String name) : void
+ toString() : string
BookIssuer
bookname:String
idno:int
+ret:int
run():void
issueBook():void
ReissuerBook
name:String
+ret:int
+run():void
reissuebook():void
<<interface>>
searchByAuthor
search(String, Database_DAO):HashSet<Book>
<<interface>>
searchByName
search(String, Database_DAO):HashSet<Book>
<<interface>>
searchByISBN
search(String, Database_DAO):HashSet<Book>
Book
-title:string
-author:string
-isbn:string
-available:boolean
-genre:string
-publisher:string
+ setName(String name) : void
+ getName() : String
+ setAuthor(String author) : void
+ getAuthor() : String
+ isAvailable() : boolean
+ setAvailable(boolean av) : void
+ getIsbn() : String
+ setIsbn(String isbn) : void
+ getGenre() : String
+ setGenre(String genre) : void
+ getPublisher() : String
+ setPublisher(String publisher) : void
+ compareTo(Book o) : int
+ equals(Object b) : boolean
+ toString() : String
1
0..*
1
0..*
1
0..*

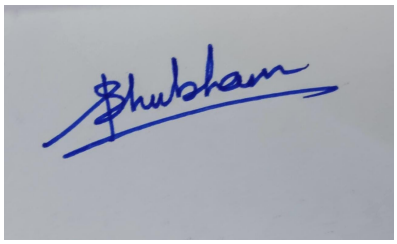
Contribution Table

Sr.No	Name	Contribution	Signature
1.	Aryaman Chauhan	SQL and Multithreading	
2.	Shardul Shingare	Classes and Methods, Project Report, UML	
3.	Shubham Gupta	Classes and Methods, Project Report	

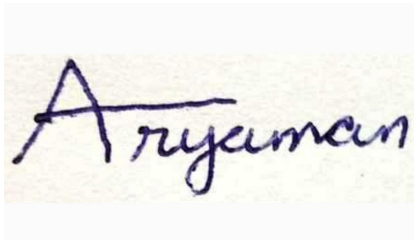
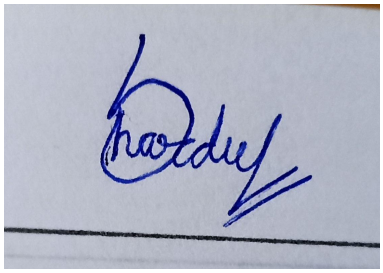
ANTI-PLAGIARISM STATEMENT

I/We certify that this assignment is my/our own work, based on my/our personal study and/or research and that I/We have acknowledged all material and sources used in its preparation, whether they be books, articles, reports, lecture notes, and any other kind of document, electronic or personal communication. I also certify that this assignment has not previously been submitted for assessment in any other unit, except where specific permission has been granted from all unit coordinators involved, or at any other time in this unit, and that I have not copied in part or whole or otherwise plagiarised the work of other students and/or persons.

Name: Shubham Gupta



Name: Aryaman Chauhan



Name: Shardul Shingare