**A Report**

**on**

**Library Management System**



Submitted in partial fulfillment of the requirements for

CS F213 - OBJECT ORIENTED PROGRAMMING

PREPARED FOR

Dr. Amit Dua

Department of Computer Science and Information Systems

PREPARED BY :

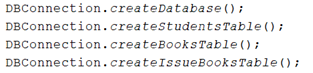
Kavish - 2021A7PS0559P

**ANTI- PLAGIARISM DECLARATION**

We, the members of group 14, hereby certify that our work is unique and free of plagiarism. Nothing in the code has been copied from the web or previously published work.The team members worked diligently to complete the project without using any plagiarized material. We all attest to the fact that each of us participated equally to the project.

**Analysis of Design Patterns**

**Singleton design pattern:** This is a creational design pattern meant to be used in scenarios where creation of multiple objects of a class is to be avoided, and only a single instance should be used throughout the application. In our project the DBConnection class that provides connection to the database and does other operations on the database seems ideal for this pattern since we only have one database and we’ll be storing and fetching all information from this common place. The usual way would be to make the constructor of the class private and have a method such as getInstance(). Every time this method is called, it would check if an object has already been created. If an object doesn’t exist, it would create one and return, otherwise it would return the already existing instance. However, our implementation of the singleton pattern is not through making the constructor private and having a method to get an instance. Instead, we made all the methods in the DBConnection class static. That way, we can call any method we want statically and we aren’t required to create an object of this class. This makes the class effectively work as a singleton design pattern. This works better than the usual way because the DBConnection class doesn’t really correspond to any real-world object (It’s merely a way to make the database accessible) and we don’t require an object for that.

// Static methods of DBConnection

**Factory design pattern**: In this design pattern we have a class acting as a factory that creates different kinds of subclass objects based on requirement using a superclass reference type. It allows the sub-classes to choose the type of objects to create thus promoting the loose-coupling by eliminating the need to bind application-specific classes into the code. In our project, we have Student and Admin classes extending User class. While logging in, for both Admin and Student, we are required to create a student or admin using only their username and password (which are provided in the User parent class). Therefore, in the AdminLogin and UserLogin, we have created Admin/Student using a User reference. However, we could have had a separate class such as UserFactory that creates Student/Admin object and returns using User reference to make the Factory design pattern more vivid.

// Creating Student in StudentLogin // Creating Admin in AdminLogin

**Analysis of code with respect to various OOP principles**

1. **Encapsulate what varies:**

Variable parts of code can be separated from the stable code to avoid any errors from the latter.

*Account Status Encapsulation not followed:*

Account status varies from account to account and depends on the admin’s discretion. Hence this part of the code could’ve been implemented using Enums to encapsulate the 3 different status of an account, ie. ‘Enabled’, ‘Disabled’, ‘Removed’.

This could’ve been implemented to cater better control for admin and avoid any variable errors while undergoing validation checks.

Currently only AddStudent and RemoveStudent methods have been implemented which only provides us with one third of the control the admin could’ve had.

**2.** **Favour Composition over Inheritance**

One principle of Object-Oriented Programming is to **prioritize composition over inheritance.** Polymorphic behaviour and code reuse should be achieved by classes through composition rather than through inheritance.

In our code, we've used **aggregation**, a weaker version of composition and **composition** both.

In fact, aggregation is preferable over composition if the two classes are not too tightly connected logically. At some places like we were attempting to get rid of composition because it encourages strong dependencies, comparable to inheritance. With the help of aggregation, both the classes can be used as standalone classes. Composition and aggregation make the code less rigid.

In the case of **RequestIssue, ReturnBook and ReissueBook,** all are dependent on **StudentHomepage** and can’t exist independently. Composition is more efficient as if we want to make some changes in the StudentHomepage, it’ll not affect the hierarchy and methods of the three classes aforementioned.

For classes  **StudentLogin and AdminLogin,** as these classes can very well exist independently, aggregation is used.

Similarly, for **AddStudent, RemoveStudent, RemoveBook, AddBook**, these classes will have little meaning without the class **AdminHomepage.** Here, composition was used over inheritance highlighting the principle. It might be a possibility that we want to add another class which doesn’t fulfill all the methods of the inherited class AdminHomepage, then we would have to modify our inheritance hierarchy leading to a lot of unnecessary hassle, therefore composition triumphs here.

**3. Program to an interface not implementation**

**UserLogin** interface is implemented by both AdminLogin class and StudentLogin class and both have different method definitions which helps create polymorphism for methods. Instead of changing the complete concrete class, we can just provide different implementations for the same interface UserLogin in StudentLogin and AdminLogin.

Similarly , **StudentRegistration** interface is implemented by signup and addStudent. Both classes have provided a different implementation of the methods instead of changing complete classes for every class.

**4. Strive for loose coupling between objects that interact**

**The StudentRegistration** interface has been implemented by classes Signup and AddStudent. There is no dependency between the two classes and they have a breakable relationship with the parent interface. Hence they are loosely coupled.

Same case applies to the UserLogininterface as well, implemented by the AdminLogin and StudentLogin class. This is another example where loose coupling is achieved.

The class Student is a child class of Class User, and shares its methods and constructors. Inheritance usually leads to Close Coupling rather than Loose Coupling, and any changes in class User and its methods will directly affect the class Student, hence Loose Coupling is not achieved here, but since the objects of these classes will never interact hence it is not an issue.

**Analysis of code with respect to various SOLID principles**

**SOLID stands for**

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

**Open closed principle**

As the name suggests, this principle states that software entities should be open for extension, but closed for modification. As a result, when the business requirements change then the entity can be extended, but not modified

And we are using this principle at-

**· class AddStudent is implementing StudentRegistration class**

**· class Admin is extending User class**

**· class AdminLogin is implementing UserLogin class**

**· class ReturnBookThread is implementing Runnable class**

**· class SearchBookThread is implementing Runnable class**

**· class Student is extending User class**

**· class StudentLogin is implementing UserLogin class**

**· class StudentSignup is implementing StudentRegistration class**

**The Dependency Inversion Principle**

According to the Dependency Inversion concept, rather than actual classes and functions, our classes should depend on interfaces or abstract classes. The Open-Closed Principle is closely related to these two ideas, and we have used this pattern before.

We have rearranged our dependencies so that they now depend on interfaces rather than specific classes since we want our classes to be extensible.

In our code, we’ve used 2 interfaces, **StudentRegistration and UserLogin.** The **AddStudent** class and the **StudentSignup** class depend on **StudentRegistration** and, **AdminLogin and StudentLogin** depend on UserLogin. We tried to loosely couple the classes and this can be achieved through abstraction.

**The Single Responsibility Principle**

According to the single responsibility principle, a class should only have one cause to change and should only be responsible for one item. This implies that each class will perform

In our code, we’ve used different classes (**AddBook, RemoveBook**) for two functions, removing and adding books thereby ensuring that each of those classes have one singular responsibility.

Also, while making the classes **AddStudent and RemoveStudent,** we’ve employed the same principle of a single responsibility per class. This ensures a high level of cohesion and robustness in the code.

**Liskov Substitution Principle**

Liskov substitution principle is being followed in the code, which is basically that every subclass should be substitutable for their base class.

All methods declared by the interfaces are used in all classes that implement them, and

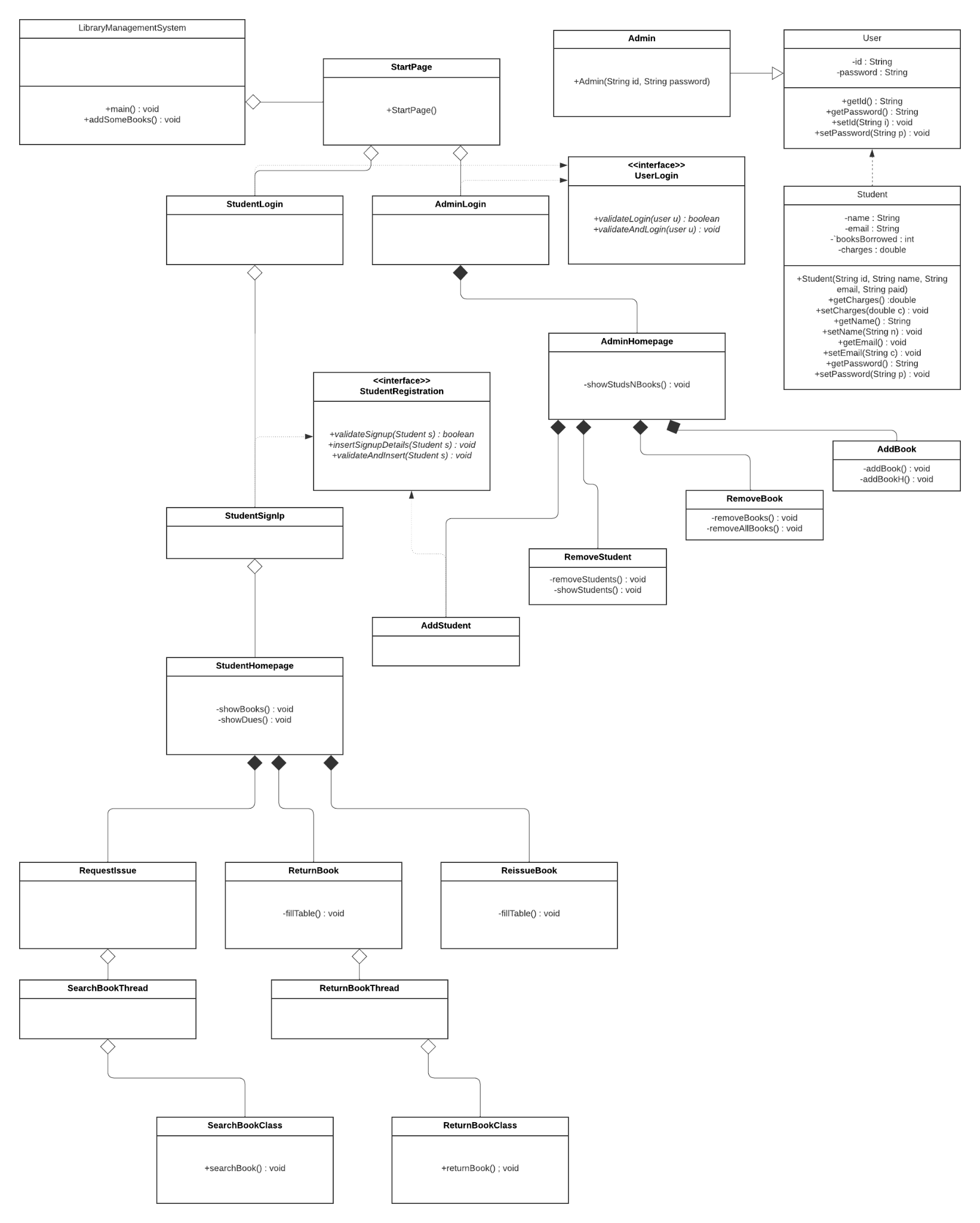
**StudentLogin,** we are substituting Student as a User, its parent class. This is one instance of the Liskov Substitution Principle.

**Interface Segregation Principle**

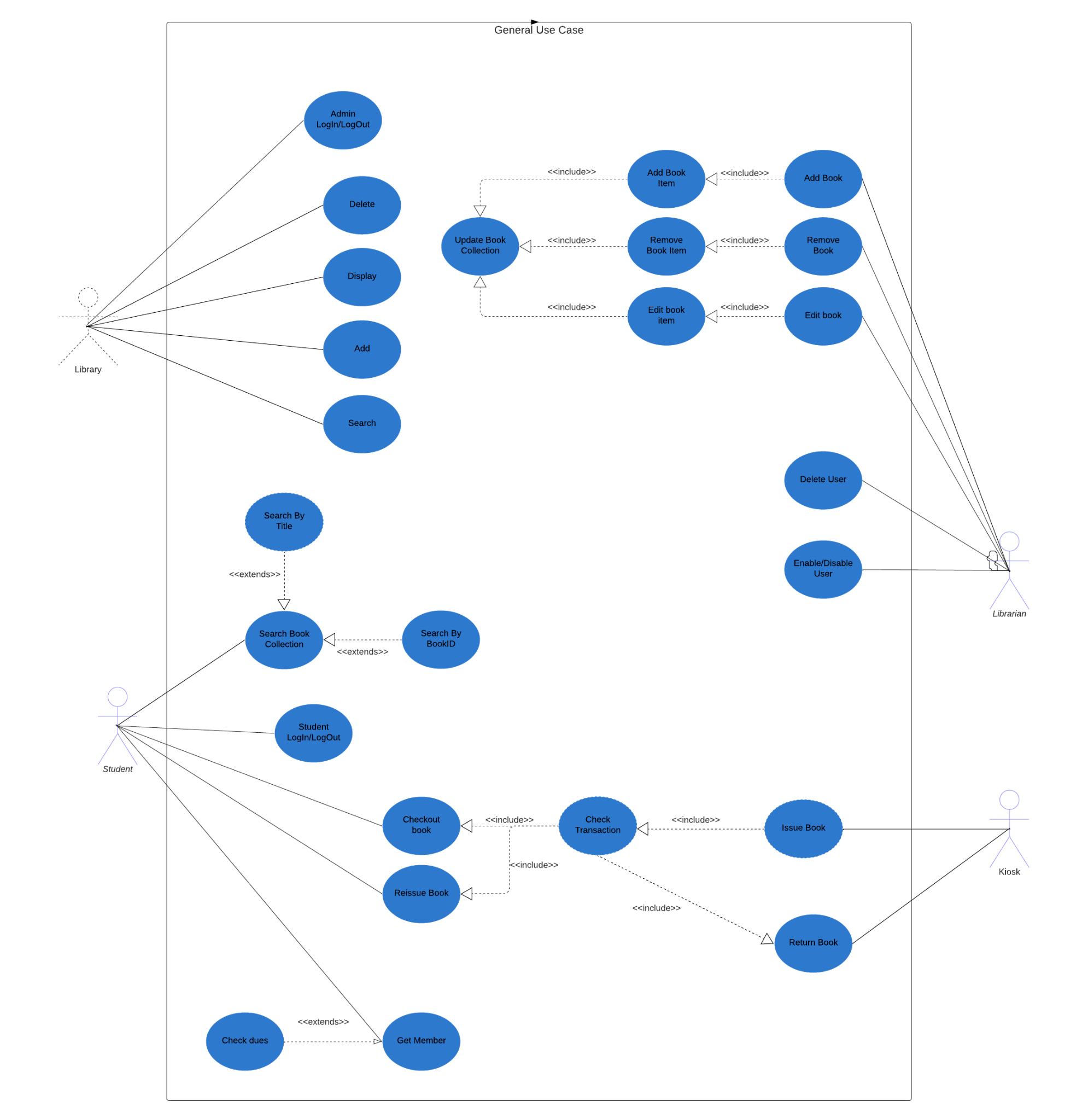
We have not used Interface Segregation principle in this code, as there was no need to with such few interfaces, it could be implemented however, if we made UserLogin interface into two interfaces StudentLoginInterface and AdminLoginInterface. But that is of no use as no class implementing any interface is implementing methods that are not of use to them.

**UML Diagrams for the current code**

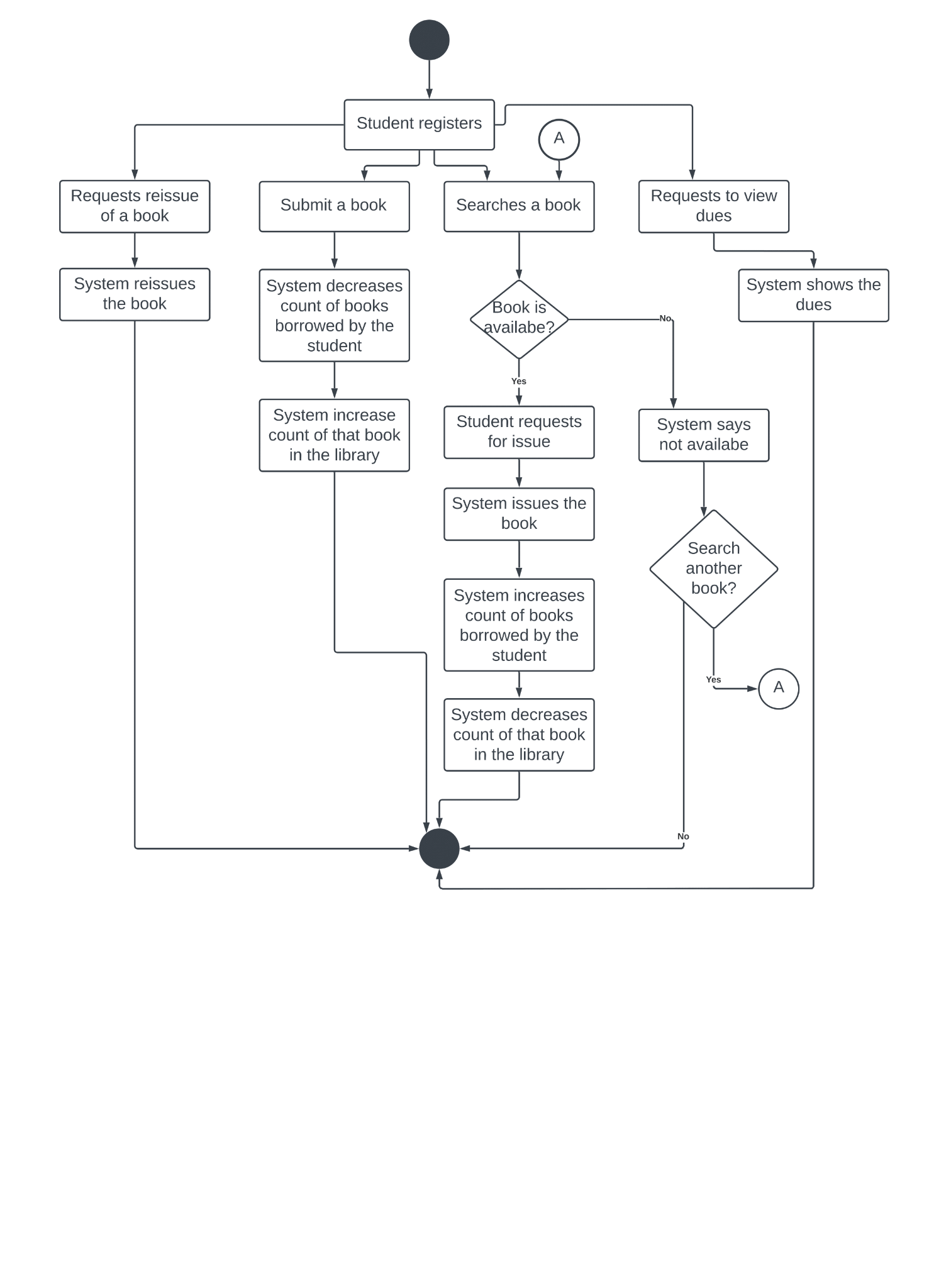
1) UML Class Diagram



2) Use Case Diagram



3) Sequence diagram

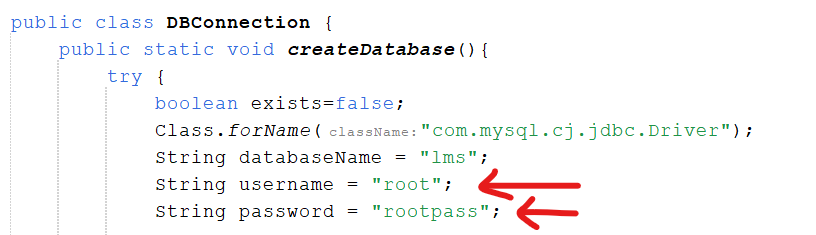


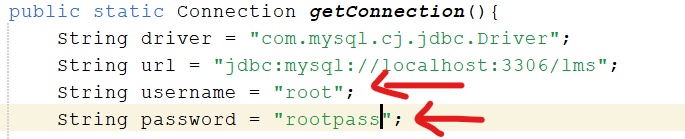
**Instructions to run the program**

* Have a local mysql server on your computer.
* No dump file is required as all operations on the database are done from the code itself, starting with creation of the database.
* All classes will be available in the path

\LibraryManagementSystem\src\main\java\com\mycompany\librarymanagementsystem\

* Before running, open the class DBConnection.java using any text editor.
* In the first method createDatabase() and the second method getConnection() you will see two lines assigning the strings “root” and “rootpass” to username and password as follows:





* Change these strings to match the username and password of your local mysql server.

In most systems by default the username is “root” and password is “”.

* Now open the LibraryManagementSystem.java class and run the program.
* For student operations, you can signup as a new student, then login (using the credentials you used for signup) and try out different features.
* For Admin, there will be no signup as there is only one admin and the login credentials are: Username: Admin1

Password: passforlms

You can then try out different actions the admin can do.