

Object Oriented Programming Project

LIBRARY MANAGEMENT SYSTEM

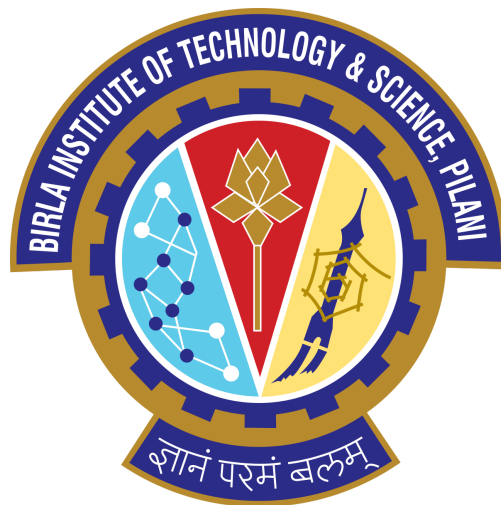
GROUP- 4

Stavya Puri (2020B5A70912P)

Radhika Gupta (2020B4A70600P)

Aditya Holikatti (2020A1PS1703P)

Agrawal Vansh Anil (2021A7PS2215P)



BITS PILANI

Anti-Plagiarism Statement:

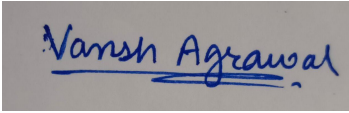
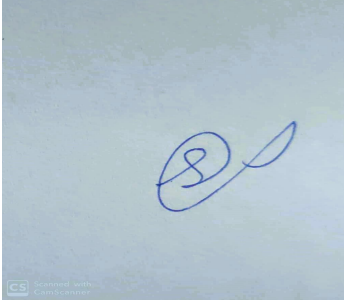

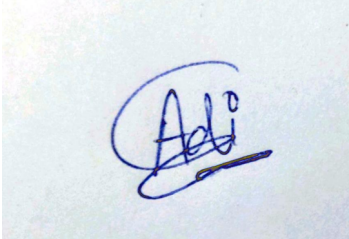
We declare that whatever has been written in this project and report has been written by all the group members itself. No part of this project has been copied from any source publications, articles, internet, or the projects of other students. Already provided resources materials are only used for reference purposes.

We all understand what plagiarism is and are aware of the University's policy in this regard. We declare that the work hereby submitted is our original work and no other people's work has been used.

We have not used any work previously produced by another student/group or any other person to hand in as our own.

We have not allowed, and will not allow anyone to copy our work with the intention of passing it off as his/her own work.

Agrawal Vansh Anil	2021A7PS2215P	
Stavya Puri	2020B5A70912P	
Radhika Gupta	2020B4A70600P	
Aditya Holikatti	2020A1PS1703P	

Name	Contribution	Signature
Agrawal Vansh Anil	<p>1) Register, Login, Main, Student , Book, Books, Catalog, studentData, bookData and Students class design.</p> <p>2) Use case diagram</p> <p>3) tried to implement file handling for data storage but was not able to implement it.</p> <p>4) Interface implementation</p> <p>5) Report Written</p> <p>6) Inheritance</p>	
Stavya Puri	<p>1) Use Case Diagram, UML Diagram, Sequence Diagram, SOLID Principles in Report.</p> <p>2) StudentMenu and AdminMenu Class design</p> <p>3) Tried to implement GUI but was unable to complete it in time so it was not included in the code.</p>	
Radhika Gupta	<p>1) Register, Login, Main, Student Class, Book, Books, studentData, bookData and Catalog Class design.</p> <p>2) Multi threading implementation.</p> <p>3) Generics implementation.</p> <p>4) Interface implementation.</p> <p>5) Inheritance</p>	
Aditya Holikatti	<p>1) Analyzed Design Pattern implementation for the code and in report, along with anti-plagiarism statement.</p> <p>2) Sequence, UML, use case diagram.</p> <p>3) StudentMenu and AdminMenu Classes.</p>	

Design Patterns:

Factory Design Pattern:

The Factory Design Pattern is to define an interface, it can be a java interface or an abstract class for creating an object and let the subclasses decide which class to instantiate. It talks about the creation of an object.

Implementing the Factory Design Pattern:

For each specified instance, a set and get function can be written in the class, and the main book class can be altered to become an abstract class. Then a function can be created in the specified book class above, to display the book item details, for the factory class to access them. The subclasses are inheriting the book class.

Then, a factory class is to be created and a method is to be created to access the book, with a specific parameter mentioned. The factory class must be called first whenever the student class needs to receive information from the book class.

Justification:

One of the patterns that is widely used in Java programming is the factory pattern. The factory design creates objects without revealing the logic behind their creation to the user, and it also utilizes interfaces, to refer to newly created objects. It is classified as a creational pattern because it involves making objects.

The book class uses the factory pattern in order to build many classes without having to repeatedly write the code to call methods in subclasses throughout the different classes.

Analysis of the code wrt S.O.L.I.D. Principles:-

Single Responsibility Principle:

The Single Responsibility Principle states that **a class should do one thing and therefore it should have only a single reason to change.**

In our code the '**Book**' class is one example implementing the above principle. It is being used to add book details to a given book object if created by the admin or the student and to access the details of the books due to the private fields of this class. Another example of implementation of this principle is the '**Login**' class, which provides only 'login' functionality to either admin or student based on the 'username' entered. Also '**StudentData**' and '**BookData**' class only contains the '**add**' method for data management.

Open-Closed Principle:

The Open-Closed Principle requires that **classes should be open for extension and closed to modification.**

In our code we have implemented this principle using the '**StudentAccount**' Interface. This interface has methods which are being implemented by the '**Student**' class. So, in the future if we wish to add some functionality to the methods and information students can access using their Student account details, we can add a method to this above mentioned interface and implement this in either the '**Student**' class itself or through some new class and access that through the '**StudentMenu**', which hence implements Open-Closed Principle. The '**Getter**' and '**Setter**' methods are also open for extension but closed for modification.

Liskov Substitution Principle:

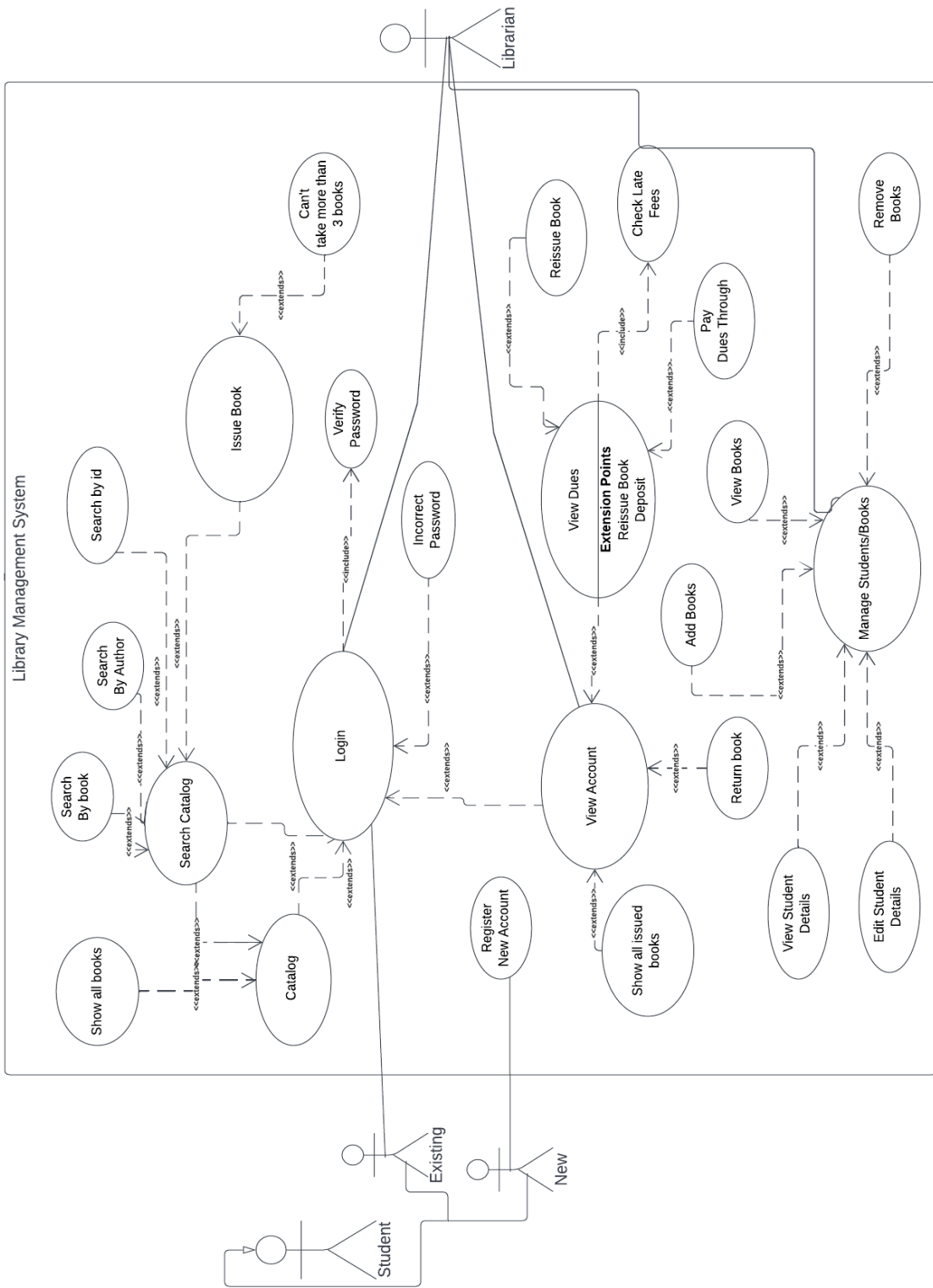
The Liskov Substitution Principle states that subclasses should be substitutable for their base classes. The class '**Students**' is being inherited by the class '**StudentData**'. Now if one changes the object occurrences of '**Students**' with object occurrences of '**StudentData**' there won't be a problem. Similar is the case with '**Catalog**' which is being inherited by '**BookData**'. But the problem can arise for '**Catalog**' class and '**StudentMenu**'.

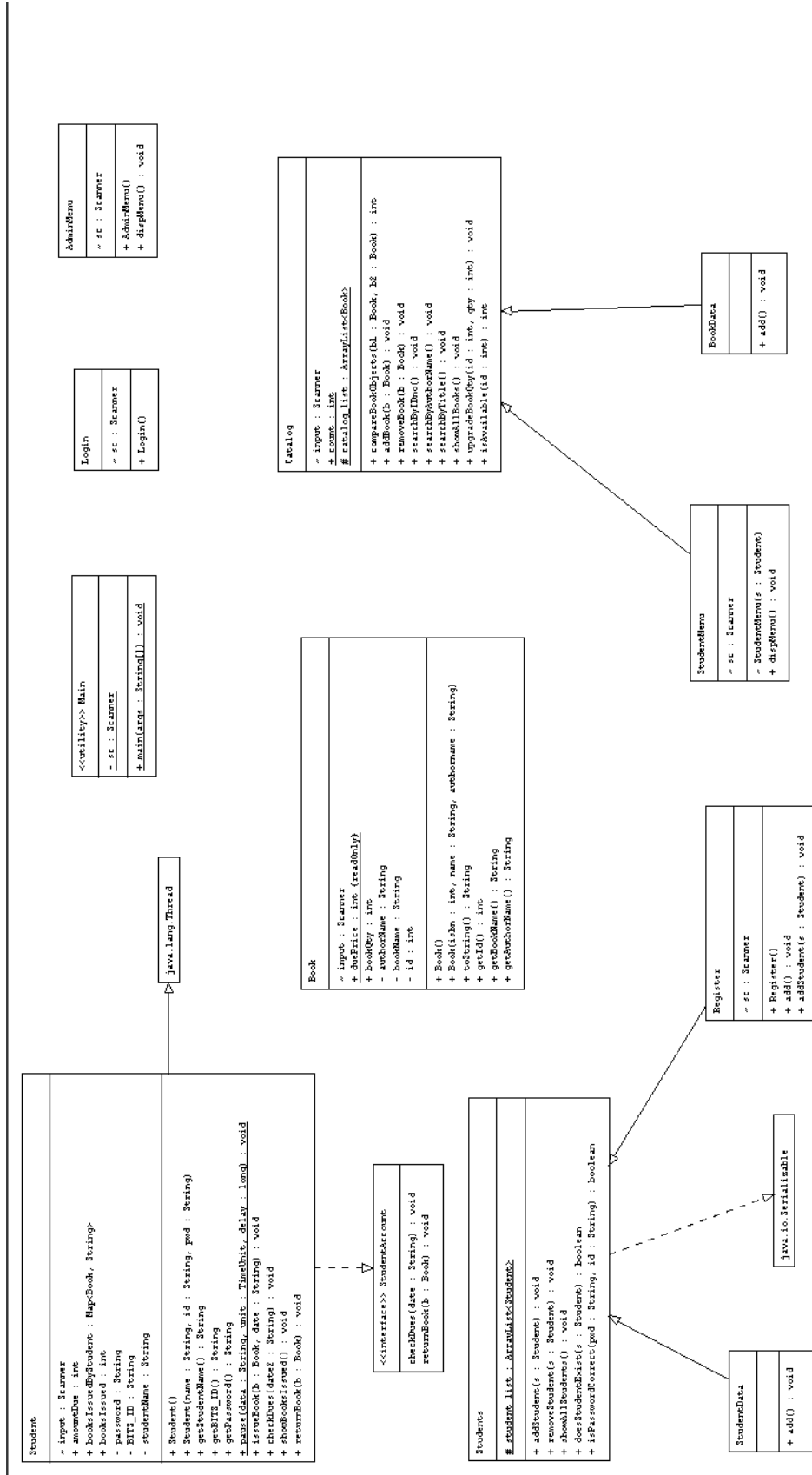
Interface Segregation Principle:

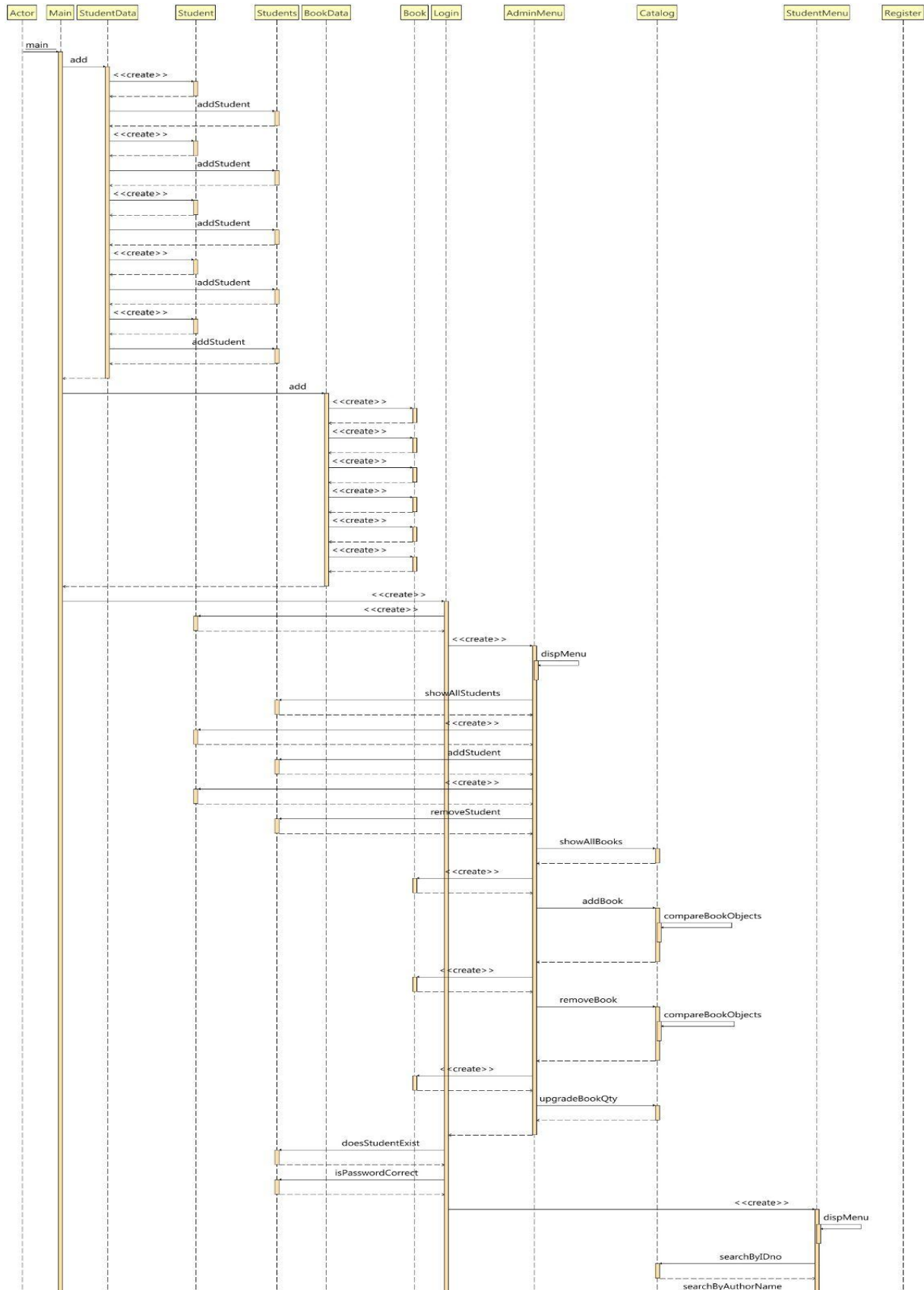
The principle states that many client-specific interfaces are better than one general-purpose interface. Clients should not be forced to implement a function they do not need. Here we have not used this Principle as such and have implemented the methods using inheritance i.e. by using classes and not using a lot of interfaces.

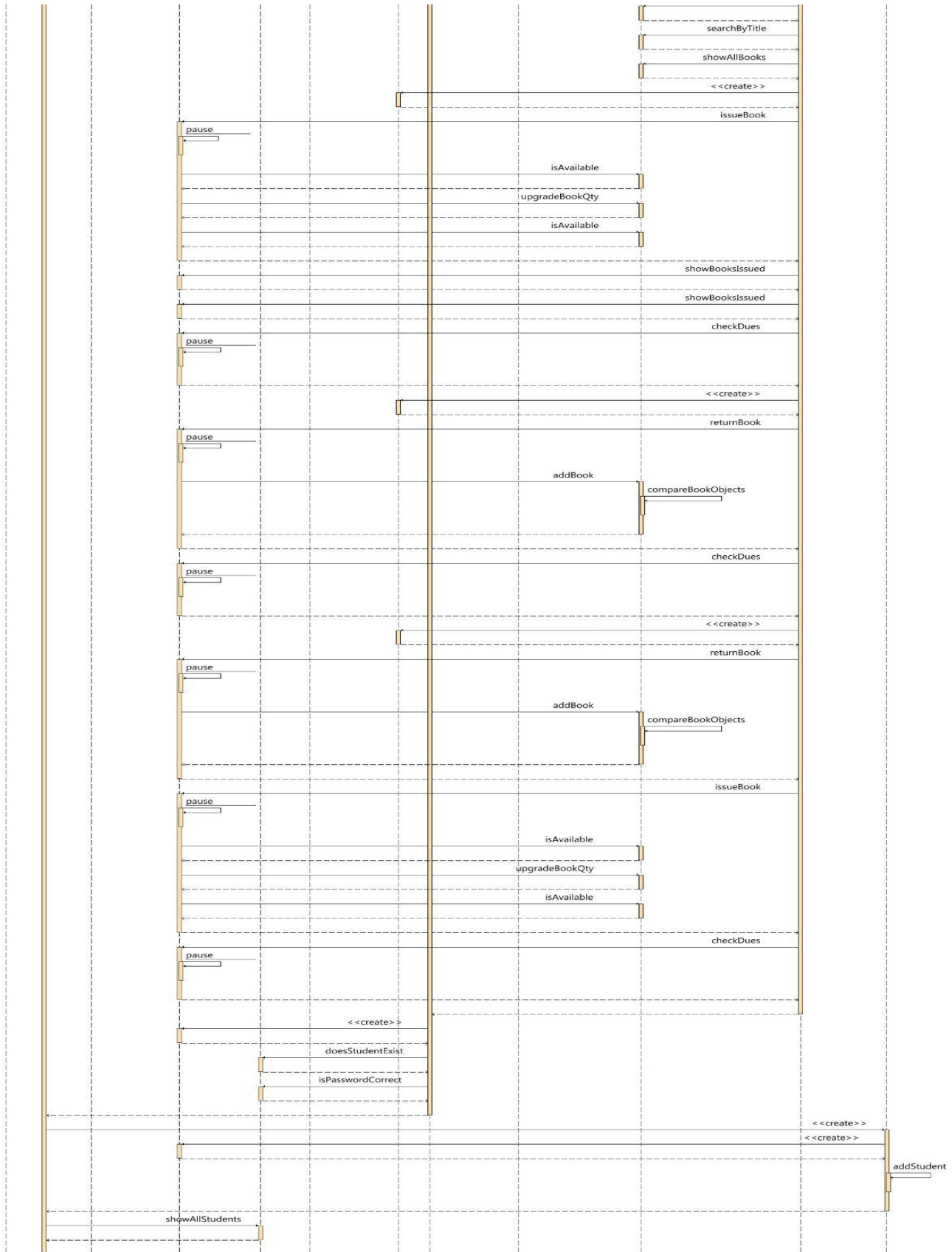
Dependency Inversion Principle:

The Dependency Inversion principle states that our classes should depend upon interfaces or abstract classes instead of concrete classes and functions. In our code we have used and implemented this principle by using '**StudentAccount**' Interface whose methods are implemented by the '**Student**' Class.









Link For Videos:

<https://drive.google.com/drive/folders/1Y1Bx4fGn73b22IDy2iCrDwJOsQJNieUN?usp=sharing>