Project:
E-Book Library

NAME: SHAKEEL AHMAD
SAP ID: 48237
SEMESTER: 5TH
INSTRUCTOR: DR. MUHAMMAD YASEEN

Table of Contents

1. Modeling and Analysis 2
1.1 Overview 2
1.2 Abstract 3
1.3 Background of project 3

1.4 Project Aims and Objective 3
1.5 Functional and Non-Functional Requirements 3
2. UML Modeling 4
2.1 Use case diagram 4
2.2 Class diagram 6
2.3 Sequence diagram 7
2.4 Activity diagram 8
2.5 Data Flow Diagrams 9
2.6 Communication Diagram 10
2.7 Object Diagram 11
2.8 Component Diagram 12
2.9 Deployment DiagramError! Bookmark not defined. 13
3. Architecture Design 14
3.1.1 Horizontal Partitioning 14
3.1.2 Vertical Partitioning 15
3.2 Architecture Style 16
4. Code Design 18
4.1 Class diagram to Code Mapping 18
4.2 Design Principles 21
4.2.1 User 21
4.2.2 Guest-User Class 22
4.2.3 Premium-User 25
4.2.4 Books 27
4.2.5 Admin 29
4.2.6 Download 32
4.2.7 Payment Gateway 35
4.3 Design Patterns 38
4.3.1 Creational Design Patterns 38
4.3.2 Structural Design Patterns 38
4.3.3 Behavioral Design Patterns 38
5. Prototyping 40
5.1 Home 40
5.2 Catalog 41
5.3 Login 42
5.4 Sign up 42

1. Modeling and Analysis

1.1 Overview

The E-book Library project aims to streamline library operations by providing a digital platform for book cataloging, member management, and secure book issuing. It features advanced search capabilities, user-friendly interfaces, and secure authentication. Compatible with multiple devices and supporting local mobile payments, this project enhances the digital reading experience while reducing staff workload.

1.2 Abstract

E- book Library is a system which maintains the information about the books present in the library, their authors, the members of library to whom books are issued, library staff and all. This is very difficult to organize manually. Maintenance of all this information manually is a very complex task. Owing to the advancement of technology, organization of an Online Library becomes much simple. The Online Book Library has been designed to computerize and automate the operations performed over the information about the members, book issues and returns and all other operations. This computerization of library helps in many instances of its maintenances. It reduces the workload of management as most of the manual work done is reduced.

1.3 Background of project

The E-book Library project aims to provide a comprehensive online platform for users to access a vast collection of digital books in various genres and subjects. As the name suggests, the platform primarily deals with electronic books, allowing users to read them conveniently in PDF format. The project is designed to cater to the growing demand for digital reading materials, offering a convenient and accessible alternative to traditional printed books.

1.4 Project Aims and Objective

- Online book reading.
- A search column to search availability of books.
- Services is to offer free book reading facilities to all.
- An Admin login page where admin can add books, videos or page sources

1.5 Functional and Non-Functional Requirements

Functional Requirements:

- It'll be a library with online purchasing/shopping cart. **FR1**
- Local (Zimbabwe) mobile money payment is to be included. **FR2**
- Allow the admin to add, remove, and manage members. **FR3**
- User registration and login functionality. **FR4**
- Role-based access control (admin, librarian, member, guest). **FR5**
- Allow users to search for books by title, publication date, author, genre, etc. **FR6**
- Display book details, including description, author, publication date, and availability status. **FR7**
- Allow admins/librarians to upload and manage eBooks in various formats (PDF). **FR8**
- Categorize books into different categories. **FR9**
- The guest can able to view/read the online books. **FR10**

- Some books downloadable, some books read online. Premium members should be able to download books always. **FR11**
- There shall be an announcement panel for a member. **FR12**
- Allow users to leave feedback or report issues. **FR13**
- The system able to track visitor data. **FR14**
- Admin can set permission for users. **FR15**

Non-functional requirements:

- To have fast loading times and responsiveness design. **NF1**
- To have interested display and user friendly (easy to understand by user). **NF2**
- The system should support the HTML browser (Google, Opera, Firefox etc..). **NF3**
- The system should be compatibility with various devices (PCs, tablets, smartphones). **NF4**
- To have the secure payment processing. **NF5**
- To have secure login authentication system for different type of users (Admins, Librarians, Members, guests). **NF6**
- Clean and well-documented code. **NF7**

2. UML Modeling

2.1 Use case diagram

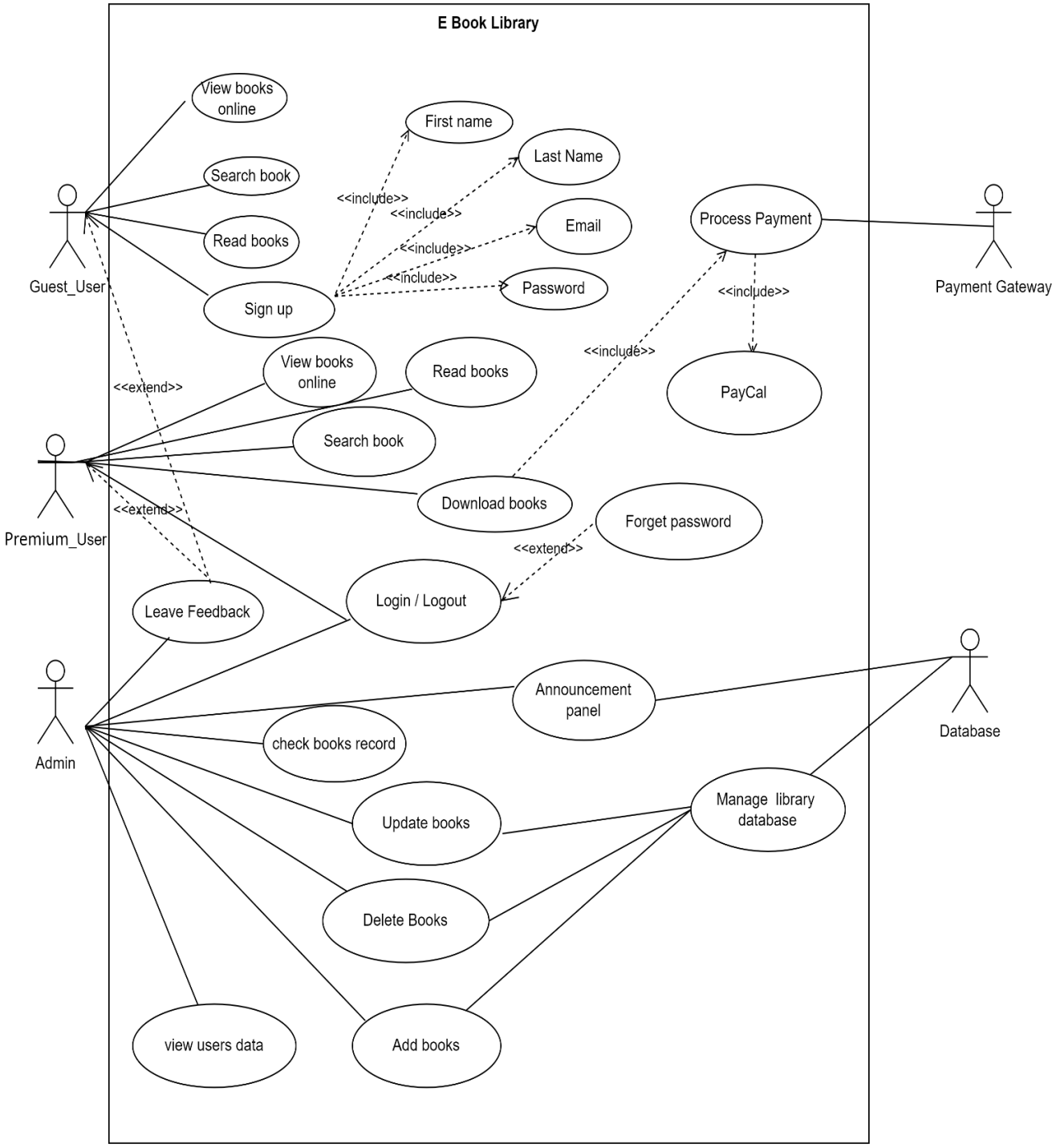


Figure #1: E-book Library Use-Case Diagram

2.2 Class diagram

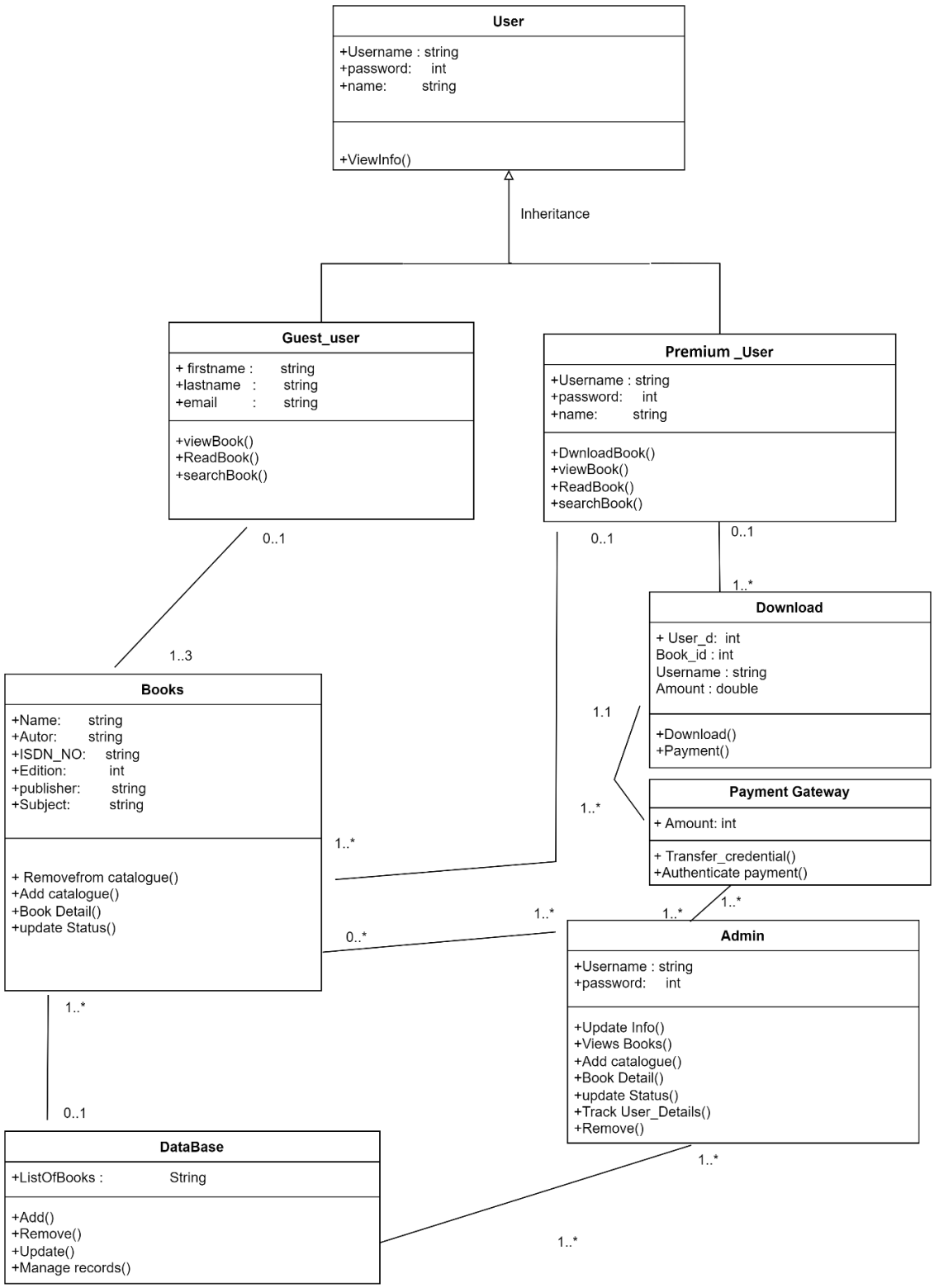


Figure #2: E-book Library Class Diagram

2.3 Sequence diagram

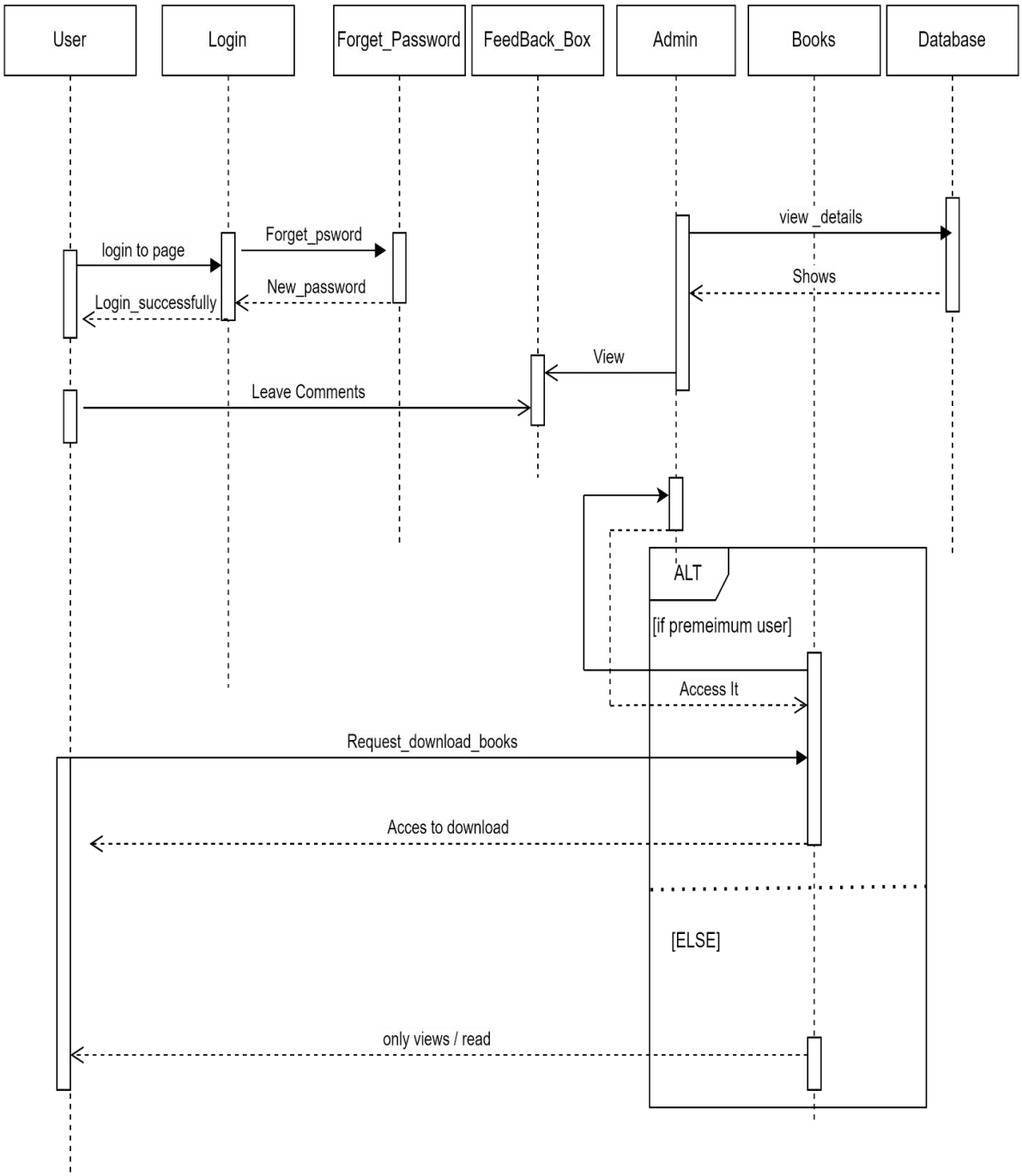


Figure #3: E-book Library Sequence Diagram

2.4 Activity diagram

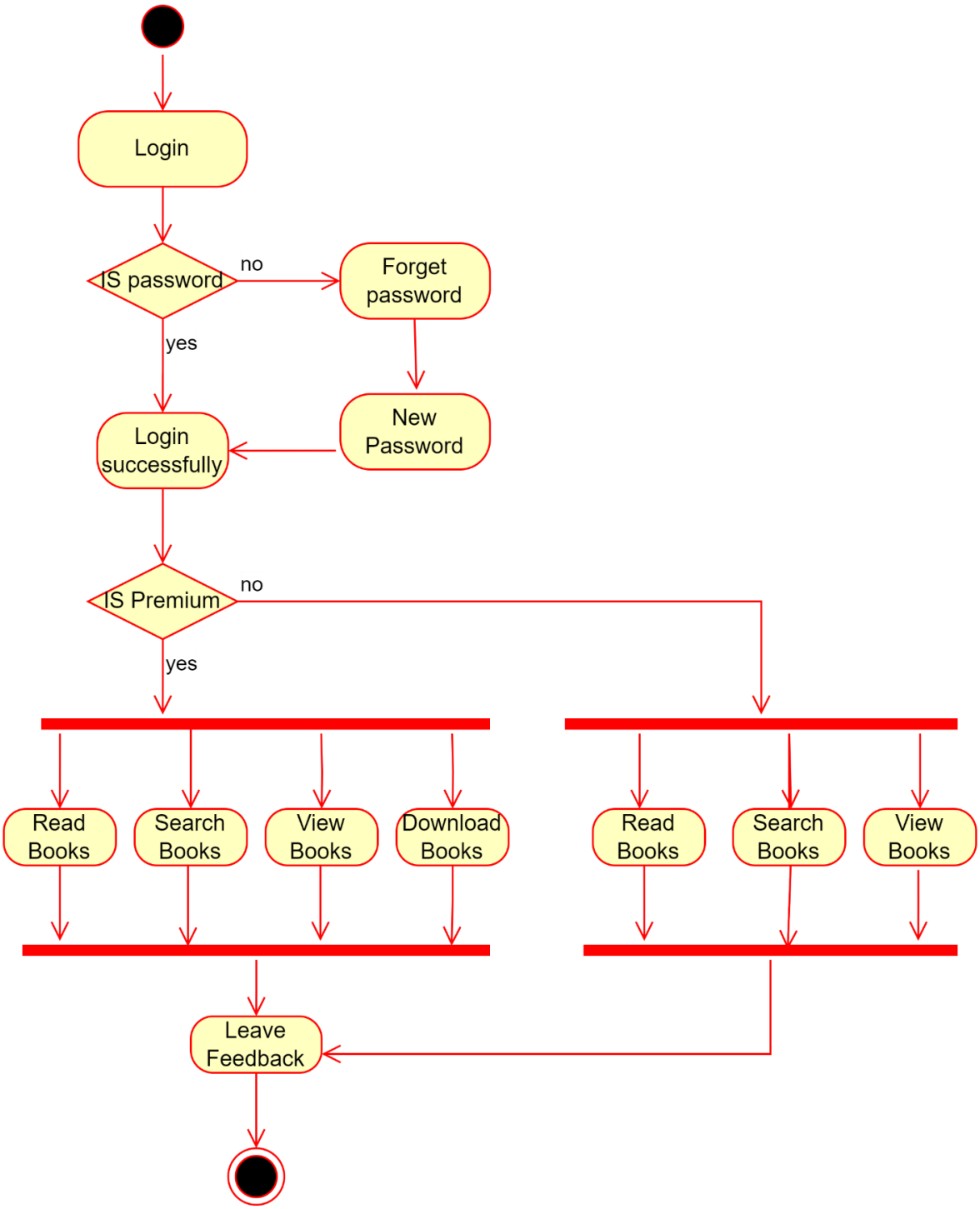


Figure #4: E-book Library Activity Diagram

2.5 Data Flow Diagrams

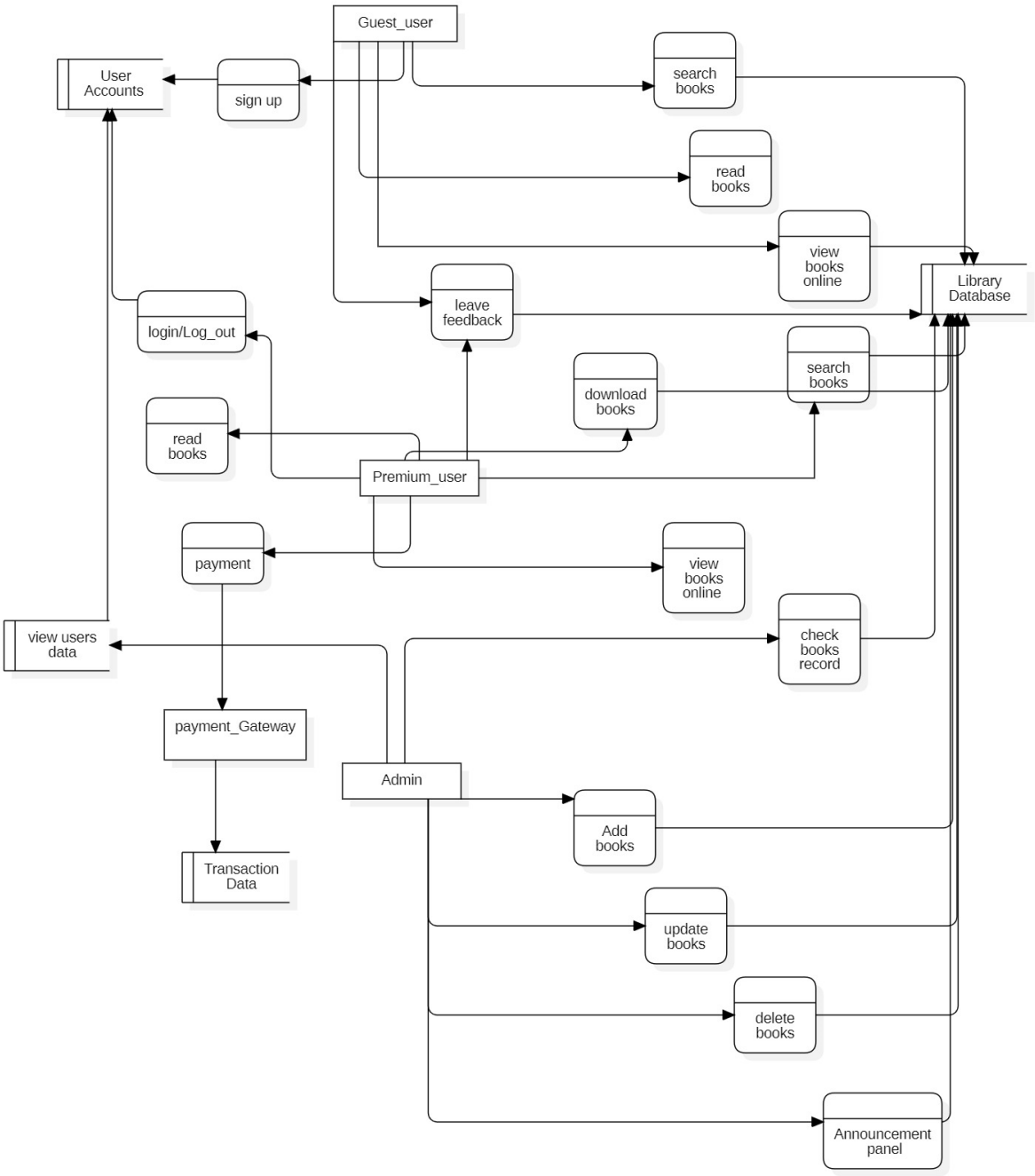


Figure #5: E-book Library Data-Flow Diagram

2.6 Communication Diagram

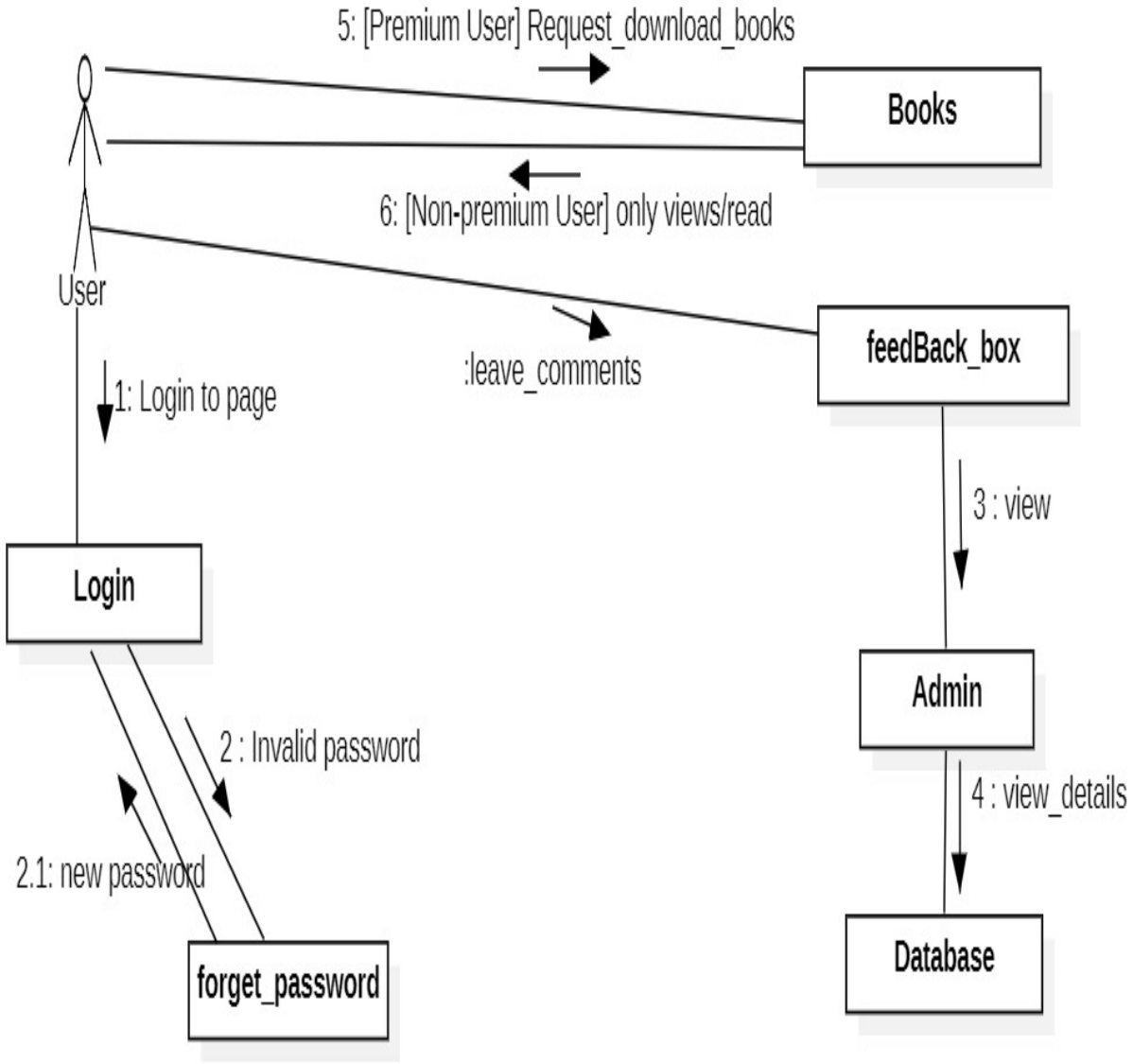


Figure #6: E-book Library Communication Diagram

2.7 Object Diagram

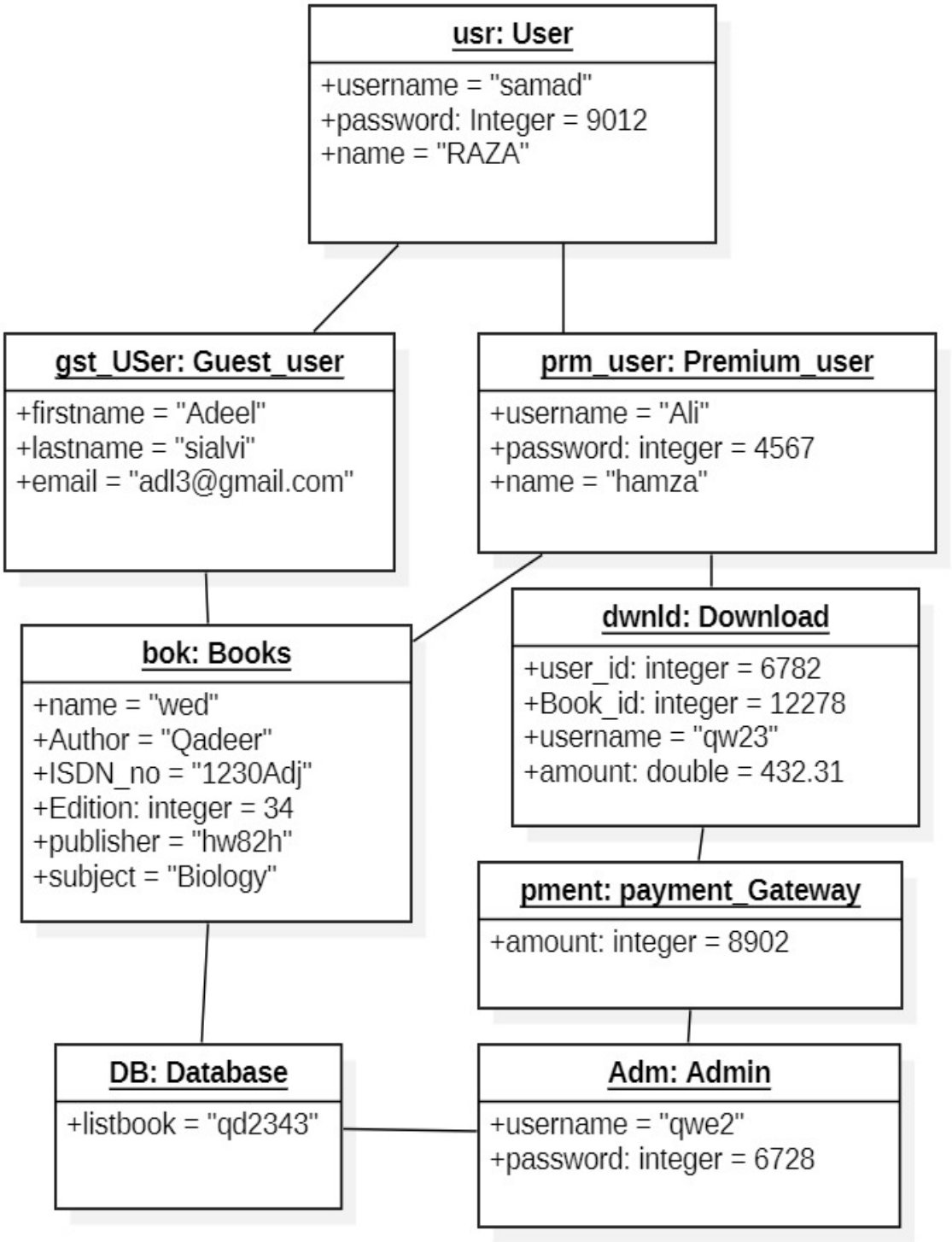


Figure #7: E-book Library Object Diagram

2.8 Component Diagram

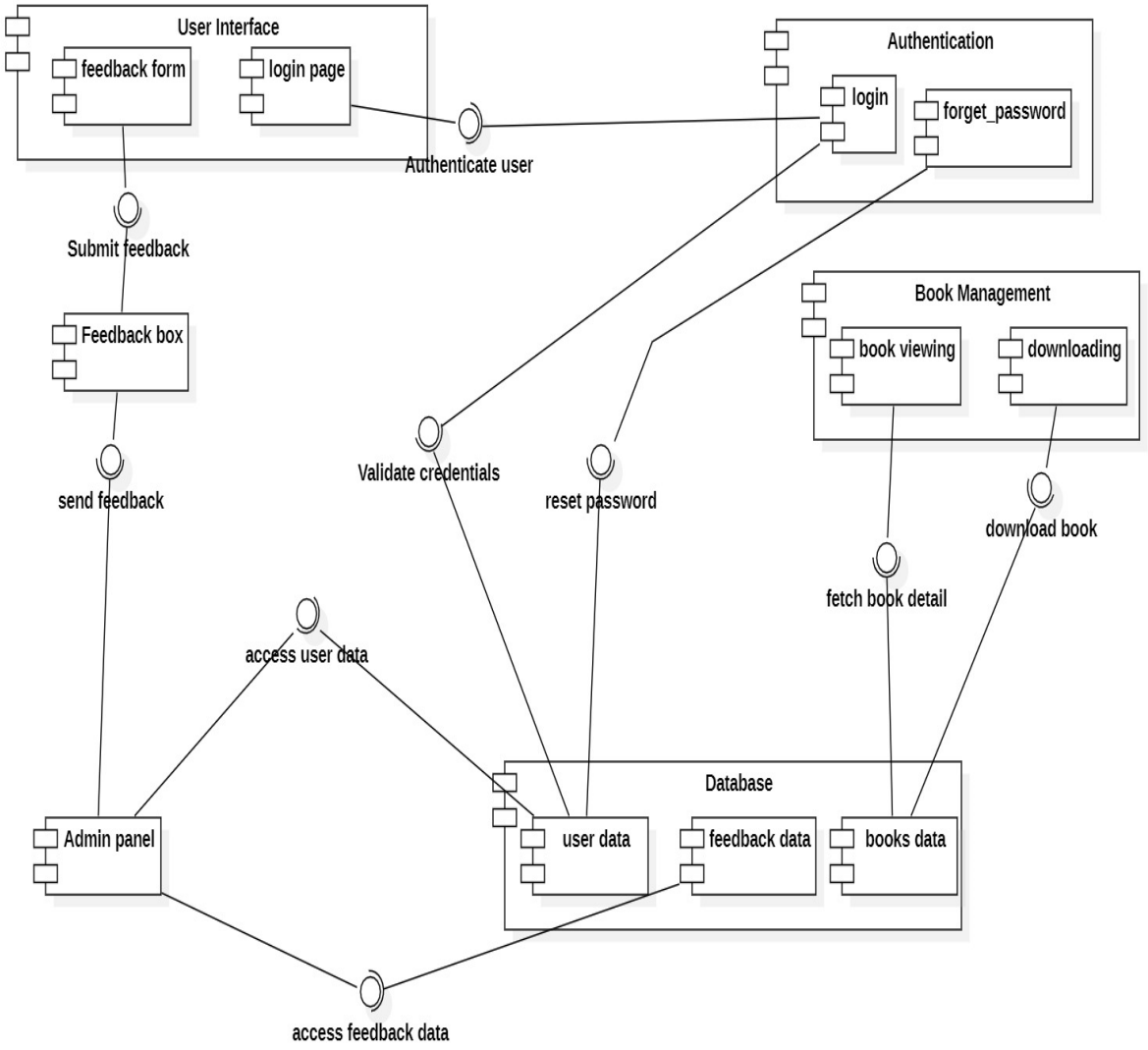


Figure #8: E-book Library Component Diagram

2.9 Deployment Diagram

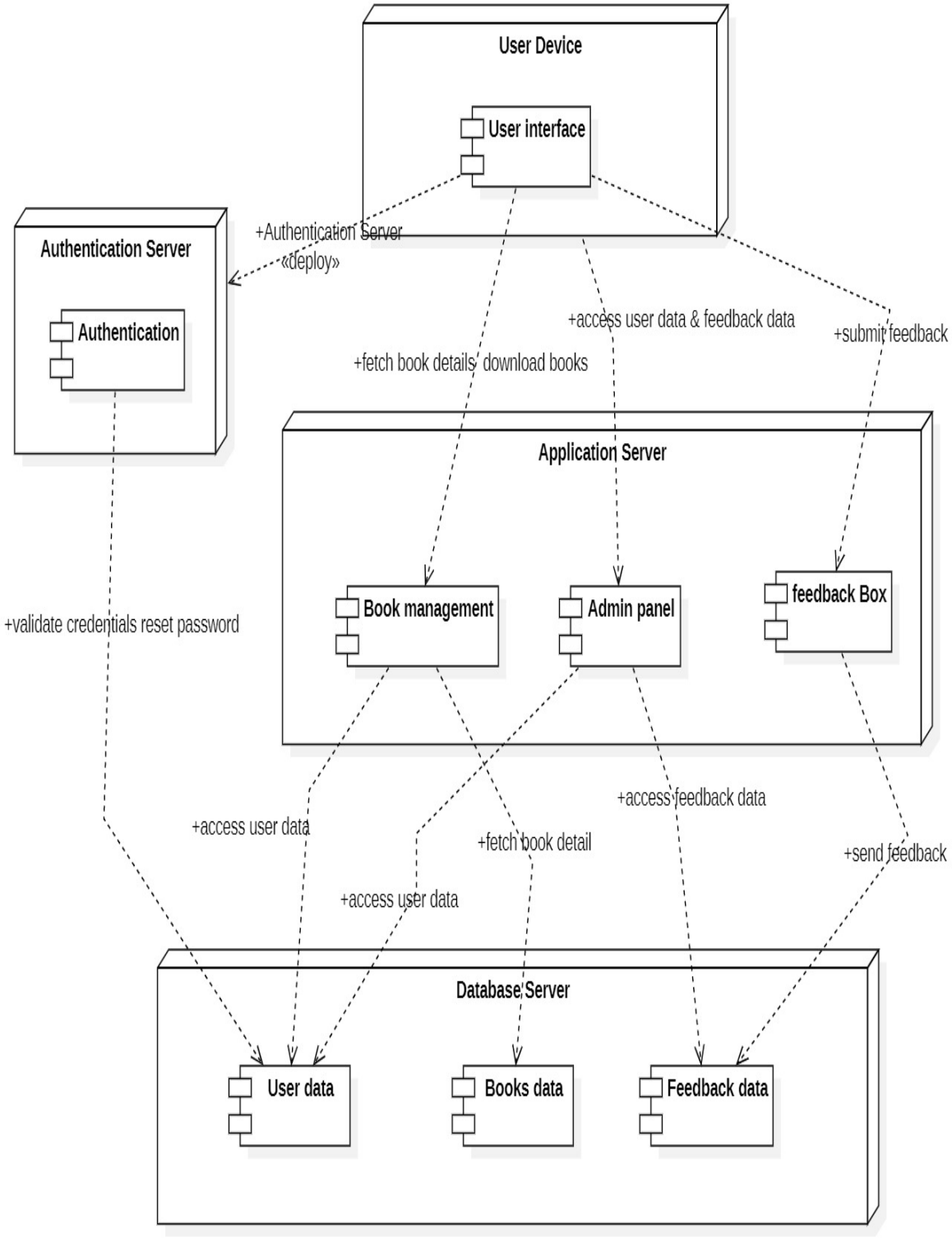


Figure #9: E-book Library Deployment Diagram

3. Architecture Design

3.1.1 Horizontal Partitioning

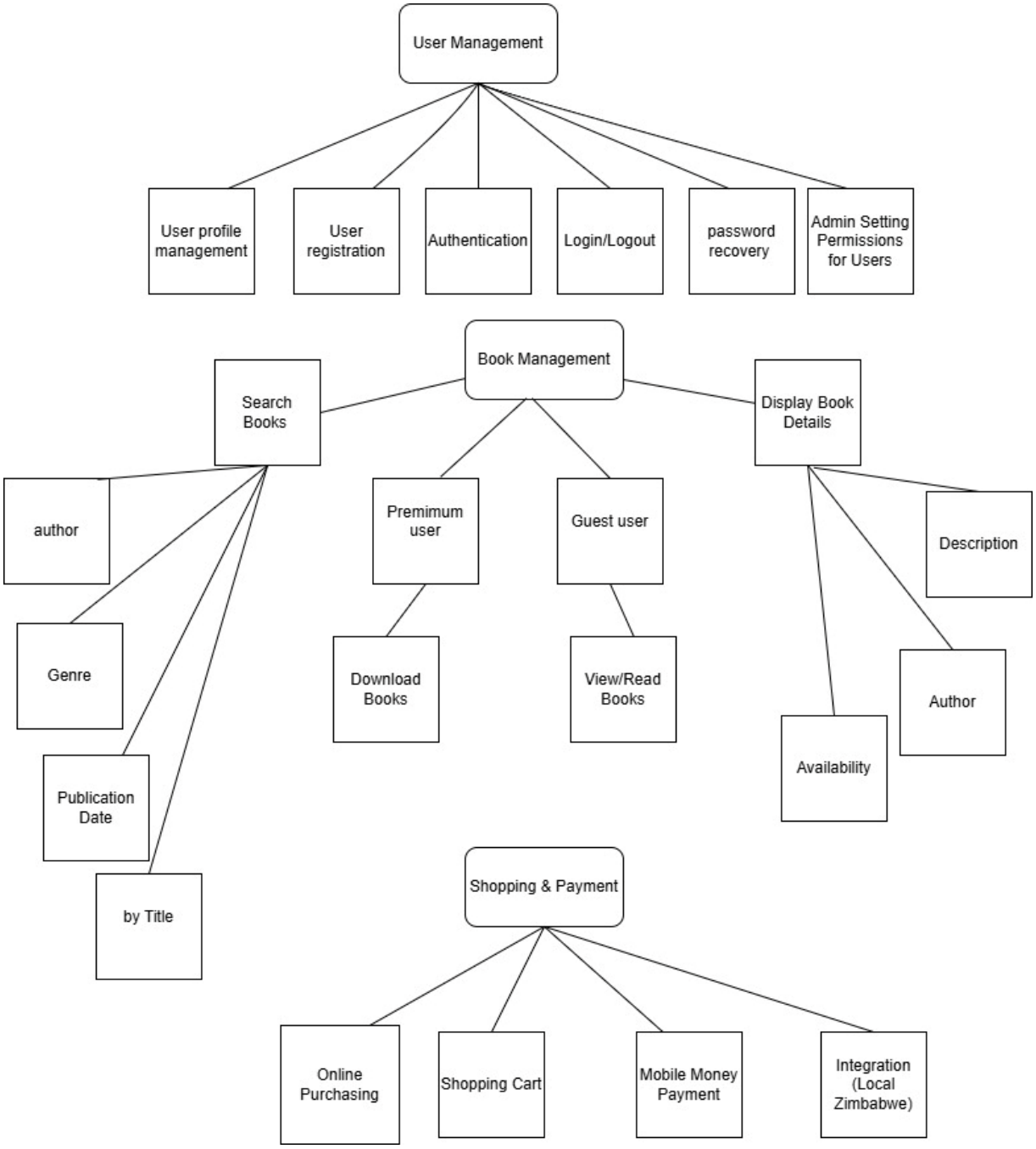


Figure #10: E-book Library Horizontal Partitioning

3.1.2 Vertical Partitioning

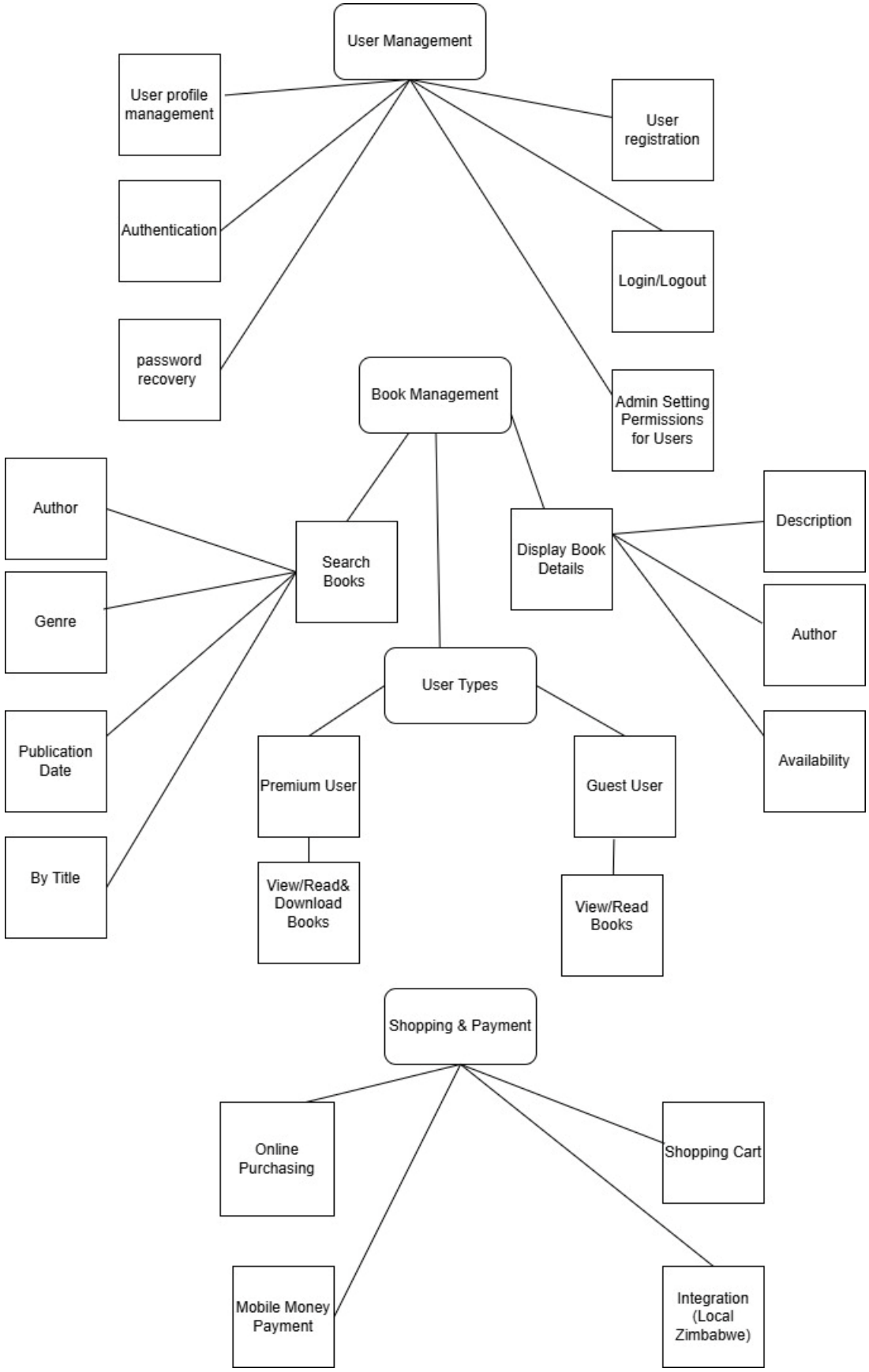


Figure #11: E-book Library Vertical Partitioning

3.2 Architecture Style

These are the following 3 architecture style that is suitable for my e book library project.

- Layered Architecture
- Client-Server Architecture
- Service-Oriented Architecture (SOA)

1. Layered Architecture

Layered architecture organizes the gadget into layers, wherein every layer has a specific function. Communication between layers is usually linear—a better layer requests offerings from a lower layer.

Role in the Project

- Presentation Layer

Interface for interactions like login, e-book seek, comments submission, and payment.

Technologies

HTML, CSS, JavaScript, or frameworks like React/Angular.

- Business Logic Layer

- Book categorization, importing, and downloading.
- Handling premium club and feedback processing.
- Ensures records is processed and choices are made in a based way.

Technologies

Backend languages like Java or Python and frameworks along with Spring Boot or Django.

- Data Access Layer

- Manages all interactions with the database.
- Stores user, e-book, and feedback records securely.
- Handles queries and updates successfully.

Benefits

- Separation of Concerns

Each layer handles specific functionality, making the gadget modular and less complicated to hold.

- Scalability

Individual layers can be up to date or replaced without affecting others.

- Security

Secure login and role-based totally get right of entry to manage can be implemented correctly.

2. Client-Server Architecture

In this structure, the device is split into two foremost components a patron that interacts with customers and a server that methods consumer requests and manages resources.

Role in the Project

- Client

Handles the front-quit interface for user interactions, together with Searching for books.

- Downloading or view eBooks.
- Providing comments or request premium books.
- Acts as a lightweight utility that sends requests to the server.

Technologies

A net browser-based totally interface or laptop/mobile app.

- Server

- Manages requests from the client, along with
- Processing user authentication and permissions.
- Handling database operations like fetching or storing e book and person information.
- Executing backend common sense and returning responses to the client.

Benefits

- Centralized Control

The server centrally manages resources, ensuring consistent statistics and secure operations.

- Flexibility

Clients can be constructed for specific gadgets (PCs, capsules, smartphones).

- Scalability

Multiple customers can have interaction with the server simultaneously.

4. Code Design

4.1 Class diagram to Code Mapping

```
// Base class
class User {
    protected String username;
    protected int password;
    protected String name;

    public User(String username, int password, String name) {
        this.username = username;
        this.password = password;
        this.name = name;
    }

    public void viewInfo() {
        System.out.println("User Info: " + name + ", Username: " +
username);
    }
}

// Guest User class
class GuestUser extends User {
    private String firstname;
    private String lastname;
    private String email;

    public GuestUser(String username, int password, String name, String
firstname, String lastname, String email) {
        super(username, password, name);
        this.firstname = firstname;
        this.lastname = lastname;
        this.email = email;
    }

    public void viewBook() {
        System.out.println("Viewing book as guest...");
    }

    public void readBook() {
        System.out.println("Reading book as guest...");
    }

    public void searchBook() {
        System.out.println("Searching books...");
    }
}

// Premium User class
class PremiumUser extends User {
    public PremiumUser(String username, int password, String name) {
        super(username, password, name);
    }

    public void downloadBook() {
        System.out.println("Downloading book...");
    }
```

```
    public void viewBook() {
        System.out.println("Viewing book as premium user...");
    }

    public void readBook() {
        System.out.println("Reading book as premium user...");
    }

    public void searchBook() {
        System.out.println("Searching books...");
    }
}

// Books class
class Books {
    private String name;
    private String author;
    private String isbnNo;
    private int edition;
    private String publisher;
    private String subject;

    public Books(String name, String author, String isbnNo, int edition,
String publisher, String subject) {
        this.name = name;
        this.author = author;
        this.isbnNo = isbnNo;
        this.edition = edition;
        this.publisher = publisher;
        this.subject = subject;
    }

    public void removeFromCatalogue() {
        System.out.println("Book removed from catalogue...");
    }

    public void addCatalogue() {
        System.out.println("Book added to catalogue...");
    }

    public void bookDetail() {
        System.out.println("Book details: " + name + ", Author: " +
author);
    }

    public void updateStatus() {
        System.out.println("Updating book status...");
    }
}

// Admin class
class Admin {
    private String username;
    private int password;

    public Admin(String username, int password) {
        this.username = username;
        this.password = password;
    }

    public void updateInfo() {
        System.out.println("Updating admin info...");
    }
```

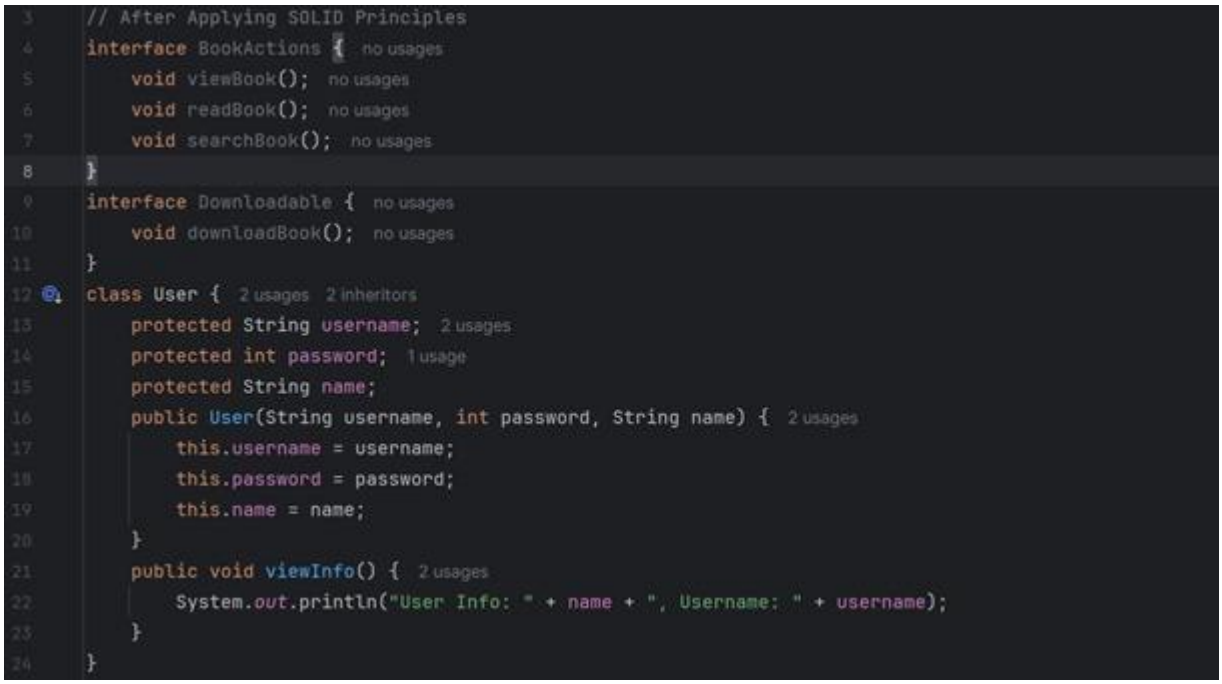
```
    public void viewBooks() {
        System.out.println("Viewing all books...");
    }

    public void addCatalogue() {
        System.out.println("Adding book to catalogue...");
    }

    public void bookDetail() {
        System.out.println("Viewing book details...");
    }

    public void updateStatus() {
        System.out.println("Updating book status...");
    }

    public void trackUserDetails() {
        System.out.println("Tracking user details...");
    }

    public void removeUser() {
        System.out.println("Removing user...");
    }
}

// Payment Gateway class
class PaymentGateway {
    private int amount;

    public void transferCredential() {
        System.out.println("Transferring credentials...");
    }

    public void authenticatePayment() {
        System.out.println("Authenticating payment...");
    }
}

// Download class
class Download {
    private int userId;
    private int bookId;
    private String username;
    private double amount;

    public void download() {
        System.out.println("Downloading book...");
    }

    public void payment() {
        System.out.println("Processing payment...");
    }
}
```

4.2 Design Principles

4.2.1 User

```
class User{ 2 usages 2 inheritors
    protected int password; 1 usage
    protected String name;
    public User(String username, int password, String name) { 2 usages
        this.username = username;
        this.password = password;
        this.name = name;
    }
    public void viewInfo() { 2 usages
        System.out.println("User Info: " + name + ", Username: " + username);
    }
    // Methods with behaviors differing for subclasses
    public void viewBook() { 2 usages 2 overrides
        System.out.println("Viewing book...");
    }
    public void readBook() { 2 usages 2 overrides
        System.out.println("Reading book...");
    }
    public void searchBook() { 2 usages
        System.out.println("Searching books...");
    }
    public void downloadBook() { 1 usage 1 override
        System.out.println("Downloading book is not available for regular users.");
    }
}
```

Figure #12: User Class Before



```
// After Applying SOLID Principles
interface BookActions { no usages
    void viewBook(); no usages
    void readBook(); no usages
    void searchBook(); no usages
}
interface Downloadable { no usages
    void downloadBook(); no usages
}
class User { 2 usages 2 inheritors
    protected String username; 2 usages
    protected int password; 1 usage
    protected String name;
    public User(String username, int password, String name) { 2 usages
        this.username = username;
        this.password = password;
        this.name = name;
    }
    public void viewInfo() { 2 usages
        System.out.println("User Info: " + name + ", Username: " + username);
    }
}
```

Figure #13: User Class After

Single Responsibility Principle (SRP)

- Before

The User class had methods such as viewBook, readBook, searchBook, downloadBook etc., although these methods were particular to sub classes. This was a violation of SRP since, the User class was assigned both user data and user actions which was not related to all users.

- After

The User class now is responsible only for keeping user data (username, password and name). All Book-specific actions are delegated to the Book Actions and Downloadable interfaces, which are implemented by the subclasses.

- Impact

The overall impact is that the User class is less dense and more targeted.

Liskov Substitution Principle (LSP)

- Before

Inconsistent behavior was quite common because the subclasses, particularly GuestUser and PremiumUser, inherited the whole lots of methods including ignore ones like downloadBook from the User class.

- After

User class is responsible only for shared attributes and methods. The subclasses use common interfaces (BookActions and/or Downloadable) to define their special behavior.

- Impact

Now subclasses can replace the User class and thus avoid the inconsistent behavior that was happened, so maintaining the consistency.

Open/Closed Principle (OCP)

- Before

Extending functionality for different user types required modifying the User class, risking unintended side effects.

- After

New user types can be created by implementing the appropriate interfaces (BookActions, Downloadable) without altering existing code.

- Impact

The system is now open to extension but closed to modification.

4.2.2 Guest-User Class

```java
class GuestUser extends User {  2 usages
    private String firstname;  1 usage
    private String lastname;  1 usage
    private String email;  1 usage
    public GuestUser(String username, int password, String name, String firstname, String lastname, String email) {  1 usage
        super(username, password, name);
        this.firstname = firstname;
        this.lastname = lastname;
        this.email = email;
    }
    @Override  2 usages
    public void viewBook() {
        System.out.println("Viewing book as guest...");
    }
    @Override  2 usages
    public void readBook() {
        System.out.println("Reading book as guest...");
    }
}
```

Figure #14: Guest-User Class Before

```java
class GuestUser extends User implements BookActions {  2 usages
    private String firstname;  1 usage
    private String lastname;  1 usage
    private String email;  1 usage
    public GuestUser(String username, int password, String name, String firstname, String lastname, String email) {  1 usage
        super(username, password, name);
        this.firstname = firstname;
        this.lastname = lastname;
        this.email = email;
    }
    @Override  2 usages
    public void viewBook() {
        System.out.println("Viewing book as guest...");
    }
    @Override  2 usages
    public void readBook() {
        System.out.println("Reading book as guest...");
    }
    @Override  2 usages
    public void searchBook() {
        System.out.println("Searching books...");
    }
}
```

Figure #15: Guest-User Class After

Single Responsibility Principle (SRP)

- Before

GuestUser had responsibilities for user-related information and specific behaviors like viewBook, readBook, and searchBook. These behaviors were inherited from the User class, leading to some methods being overridden while others were redundant.

- After

GuestUser now implements only the behaviors it needs by adopting the BookActions interface. The class focuses solely on defining the guest-specific attributes (firstname, lastname, and email) and implementing relevant methods for a guest user's behavior.

- Impact

Clear separation of concerns. GuestUser only deals with guest-specific logic, while

generic book-related actions are defined by the interface.

Interface Segregation Principle (ISP)

- Before

GuestUser inherited all methods from the User class, including irrelevant ones like downloadBook, which didn't apply to guests.

- After

GuestUser implements only the BookActions interface, which defines actions like viewBook, readBook, and searchBook. It does not inherit or depend on any unrelated methods.

- Impact

GuestUser depends only on the specific interfaces it needs, avoiding unnecessary or irrelevant functionality.

Liskov Substitution Principle (LSP)

- Before

GuestUser could replace User but had to override methods like downloadBook to prevent incorrect behavior.

- After

By implementing only relevant interfaces, GuestUser now cleanly replaces User in contexts where BookActions is expected without any overridden or inconsistent behavior.

- Impact

Reliable substitution of GuestUser in any system expecting a BookActions implementation.

Open/Closed Principle (OCP)

- Before

Extending or modifying GuestUser required changing the User class or overriding methods unnecessarily.

- After

GuestUser is extended by implementing interfaces, allowing new behaviors to be added (e.g., a new interface for guest-specific features) without altering the class itself.

- Impact

GuestUser is open for extension but closed for modification.

4.2.3 Premium-User

```
class PremiumUser extends User {  2 usages
    public PremiumUser(String username, int password, String name) {  1 usage
        super(username, password, name);
    }
    @Override  1 usage
    public void downloadBook() {
        System.out.println("Downloading book...");
    }
    @Override  2 usages
    public void viewBook() {
        System.out.println("Viewing book as premium user...");
    }
    @Override  2 usages
    public void readBook() {
        System.out.println("Reading book as premium user...");
    }
}
```

Figure #16: Premium-User Class Before

```
class PremiumUser extends User implements BookActions, Downloadable {  2 usages
    public PremiumUser(String username, int password, String name) {  1 usage
        super(username, password, name);
    }
    @Override  2 usages
    public void viewBook() {
        System.out.println("Viewing book as premium user...");
    }
    @Override  2 usages
    public void readBook() {
        System.out.println("Reading book as premium user...");
    }
    @Override  2 usages
    public void searchBook() {
        System.out.println("Searching books...");
    }
    @Override  1 usage
    public void downloadBook() {
        System.out.println("Downloading book...");
    }
}
```

Figure #17: Premium-User Class Before

Single Responsibility Principle (SRP)

- Before

The PremiumUser class inherited methods like searchBook and viewBook from User, even though some methods (like searchBook) were shared functionality that didn't require specific customization for premium users.
The downloadBook method was tightly coupled to the PremiumUser class, mixing shared

behaviors and unique premium user actions in the base class.

- After

The PremiumUser class now focuses solely on premium-specific behaviors by implementing the BookActions and Downloadable interfaces.
Shared behaviors (like searchBook) remain in the BookActions interface, ensuring clarity and separation of responsibilities.

- Impact

Each class or interface has a single, well-defined purpose. PremiumUser deals only with premium-specific functionality.

Interface Segregation Principle (ISP)

- Before

The PremiumUser class was forced to inherit unrelated or unnecessary methods like searchBook and viewBook directly from User.

- After

The PremiumUser class implements only the interfaces (BookActions and Downloadable) that define its required behaviors.
Methods unrelated to premium users are no longer inherited.

- Impact

The PremiumUser class depends only on the methods it needs, reducing code bloat and complexity.

Liskov Substitution Principle (LSP)

- Before

The PremiumUser class inherited and potentially overrode methods from User, which could lead to inconsistencies if User was used interchangeably with its subclasses.

- After

The PremiumUser class implements specific interfaces, ensuring it behaves consistently and predictably in any context requiring BookActions or Downloadable.

- Impact

The PremiumUser class can replace User or any interface implementation without

unexpected behavior.

Open/Closed Principle (OCP)

- Before

Extending the functionality of PremiumUser required modifying either the User class or the subclass itself.

- After

The PremiumUser class can be extended by implementing additional interfaces or adding new premium-specific features without altering the class or its parent.

- Impact

The PremiumUser class is open for extension but closed for modification, ensuring flexibility and stability.

4.2.4 Books

```
class Books {  no usages  2 inheritors
    private String name;  2 usages
    private String author;  2 usages
    private String isbnNo;  1 usage
    private int edition;  1 usage
    private String publisher;  1 usage
    private String subject;  1 usage
    public Books(String name, String author, String isbnNo, int edition, String publisher, String subject) {  no usages
        this.name = name;
        this.author = author;
        this.isbnNo = isbnNo;
        this.edition = edition;
        this.publisher = publisher;
        this.subject = subject;}
    public void removeFromCatalogue() {  no usages  2 overrides
        System.out.println("Book removed from catalogue...");
    }
    public void addCatalogue() {  no usages  2 overrides
        System.out.println("Book added to catalogue...");
    }
    public void bookDetail() {  no usages  1 override
        System.out.println("Book details: " + name + ", Author: " + author);
    }
    public void updateStatus() {  no usages  1 override
        System.out.println("Updating book status...");
```

Figure #18 Books Class Before

```java
interface BookOperations {  no usages
    void add();
    void remove();  no usages
    void update();}  no usages
class Books implements BookOperations {  no usages
    private String name;  2 usages
    private String author;  2 usages
    private String isbnNo;  1 usage
    private int edition;  1 usage
    private String publisher;  1 usage
    private String subject;  1 usage
    public Books(String name, String author, String isbnNo, int edition, String publisher, String subject) {  no usages
        this.name = name;
        this.author = author;
        this.isbnNo = isbnNo;
        this.edition = edition;
        this.publisher = publisher;
        this.subject = subject;}
    @Override
    public void add() {
        System.out.println("Book added to catalogue...");}
    @Override  no usages
        public void remove() {
            System.out.println("Book removed from catalogue...");}
        @Override  no usages
        public void update() {
            System.out.println("Updating book status...");}
        public void bookDetail() {  no usages
            System.out.println("Book details: " + name + ", Author: " + author);}}
```

Figure #19: Books Class After

Single Responsibility Principle (SRP)

- Before

The Books class now focuses only on managing the book details (bookDetail(), updateStatus()) and catalog-related operations (addCatalogue(), removeFromCatalogue()), which are book-specific actions.

- After

By splitting these responsibilities into separate interfaces, we ensure that the Books class has only one reason to change: when the book-related functionality changes.

Open/Closed Principle (OCP)

- Before

The Books class is now open for extension but closed for modification. For example, if

we need to create a special type of book, like an EBook or a PrintedBook, we can create a new class that extends Books or implements the BookDetails and Catalogue interfaces without changing the existing Books class.

- After

Additionally, we have made it easy to extend the catalog management and database operations by using interfaces. We can add new features or modify existing ones by simply adding new methods or implementing new classes.

Liskov Substitution Principle (LSP)

By using interfaces (BookDetails and Catalogue), any derived class that implements these interfaces can be substituted for Books. For example, we could create a SpecialEditionBook class that still behaves as a Book (through the interface), and it would not break the existing system.

Interface Segregation Principle (ISP)

- Before

I have created smaller, more specific interfaces (BookDetails, Catalogue) to avoid forcing the Books class to implement methods it doesn't need. This makes the class more focused on its specific responsibility and improves maintainability.

- After

The Books class implements both BookDetails (responsible for book-specific actions like displaying details or updating the book status) and Catalogue (responsible for adding/removing the book from the catalog), allowing more flexible use of the class.

4.2.5 Admin

```
// Admin class
class Admin {  no usages
    private String username;  1 usage
    private int password;  1 usage
    public Admin(String username, int password) {  no usages
        this.username = username;
        this.password = password;}
    public void updateInfo() {  no usages
        System.out.println("Updating admin info...");}
    public void viewBooks() {  no usages
        System.out.println("Viewing all books...");}
    public void addCatalogue() {  no usages
        System.out.println("Adding book to catalogue...");}
    public void bookDetail() {  no usages
        System.out.println("Viewing book details...");}
    public void updateStatus() {  no usages
        System.out.println("Updating book status...");}
    public void trackUserDetails() {  no usages
        System.out.println("Tracking user details...");
    }
    public void removeUser() {  no usages
        System.out.println("Removing user...");
    }
}
```

Figure #20: Admin Class Before

```java
// Interface for Book Management
interface BookManagement {  1 usage
    void viewBooks();  no usages
    void addCatalogue();  no usages
    void bookDetail();  no usages
    void updateStatus();}  no usages
// Interface for User Management
interface UserManagement {  1 usage
    void trackUserDetails();  no usages
    void removeUser();}  no usages
// Interface for Admin Info Management
interface AdminInfoManagement {  1 usage
    void updateInfo();}  no usages
// Admin class - Implements specific interfaces
class Admin implements AdminInfoManagement, BookManagement, UserManagement {  no usages
    private String username;  1 usage
    private int password;  1 usage
    public Admin(String username, int password) {  no usages
        this.username = username;
        this.password = password;}
    // Implementing AdminInfoManagement
    @Override  no usages
        System.out.println("Updating admin info...");}
    // Implementing BookManagement
    @Override  no usages
    public void viewBooks() {
        System.out.println("Viewing all books...");}
    @Override  no usages
    public void addCatalogue() {
        System.out.println("Adding book to catalogue...");}
    @Override  no usages
    public void bookDetail() {
        System.out.println("Viewing book details...");}
    @Override  no usages
    public void updateStatus() {
        System.out.println("Updating book status...");}
    // Implementing UserManagement
    @Override  no usages
    public void trackUserDetails() {
        System.out.println("Tracking user details...");}
    @Override  no usages
    public void removeUser() {
        System.out.println("Removing user...");}}
```

Figure #21: Admin Class After

Single Responsibility Principle (SRP)

- Before

The Admin class was handling user management, book management, and admin info updates, which made it violate SRP by having multiple responsibilities.

- After

The Admin class only implements the methods necessary for its role, and book and user management functionalities are handled by separate interfaces.

Open/Closed Principle (OCP)

- Before

  To add new features or roles, you'd need to modify the Admin class directly.

- After

  You can create new classes that implement new features or roles without changing the existing Admin class, ensuring that the system remains flexible and stable.

Liskov Substitution Principle (LSP)

- Before

  The Admin class had too many hardcoded responsibilities, making it difficult to swap out or extend with other roles.

- After

  The introduction of interfaces allows different classes to substitute for Admin while still adhering to the expected contract, promoting flexibility.

- Impact

  The Admin class is now more easily replaceable by any class that implements the AdminInfoManagement, BookManagement, or UserManagement interfaces. If you introduce a new role (like Manager or BookCoordinator), it can be substituted in place of Admin without breaking the system.

Interface Segregation Principle (ISP)

- Before

  The Admin class had to implement all methods regardless of its role's needs, violating ISP.

- After

  Each interface is more focused on specific functionality, so if a class only needs book management or user management, it doesn't have to implement all the methods that are irrelevant to its responsibility.

- Impact

  The functionalities are split into smaller, more specific interfaces: BookManagement, UserManagement, and AdminInfoManagement. Classes only implement the interfaces they actually need, avoiding unnecessary methods for roles that don't need them.

Dependency Inversion Principle (DIP)

- Before

The Admin class was tightly coupled to its own concrete methods and functionality, making it harder to change or extend.

- After

By depending on interfaces, the Admin class is now loosely coupled and more flexible, allowing the underlying functionality to be changed without affecting the higher-level code.

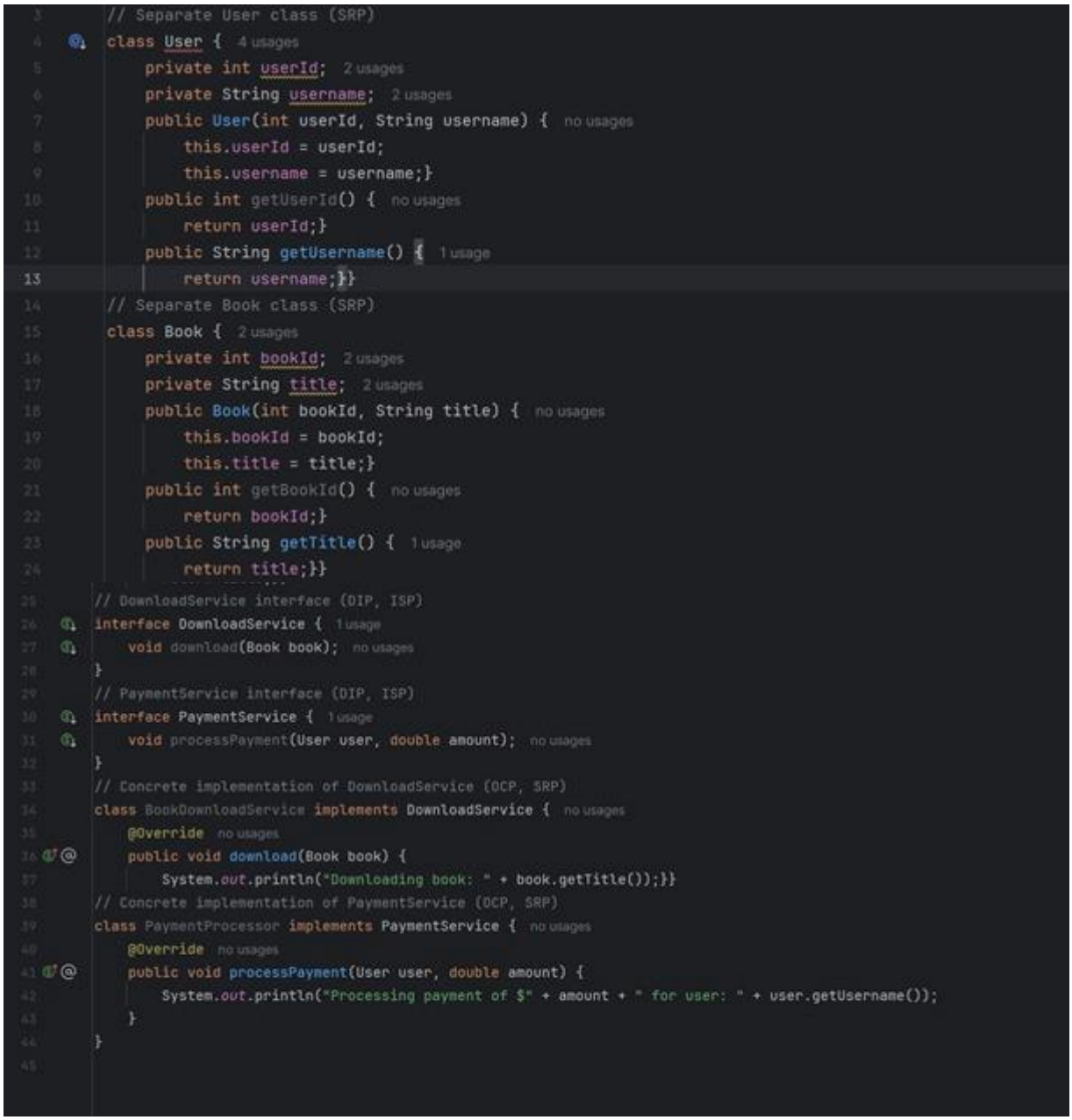
- Impact

The Admin class now depends on abstractions (interfaces) like BookManagement, UserManagement, and AdminInfoManagement, rather than concrete implementations. This allows the system to be more flexible, as higher-level components no longer need to know about specific implementations of user or book management.

4.2.6 Download

```
3   // Download class (without SOLID principles)
4   class Download {  no usages
5       private int userId;  no usages
6       private int bookId;  no usages
7       private String username;  no usages
8       private double amount;  no usages
9       public void download() {  no usages
10          System.out.println("Downloading book...");
11      }
12      public void payment() {  no usages
13          System.out.println("Processing payment...");
14      }
15  }
16
```

Figure #22: Download Class Before



```
// Separate User class (SRP)
class User {  4 usages
    private int userId;  2 usages
    private String username;  2 usages
    public User(int userId, String username) {  no usages
        this.userId = userId;
        this.username = username;}
    public int getUserId() {  no usages
        return userId;}
    public String getUsername() {  1 usage
        return username;}}
// Separate Book class (SRP)
class Book {  2 usages
    private int bookId;  2 usages
    private String title;  2 usages
    public Book(int bookId, String title) {  no usages
        this.bookId = bookId;
        this.title = title;}
    public int getBookId() {  no usages
        return bookId;}
    public String getTitle() {  1 usage
        return title;}}
// DownloadService interface (DIP, ISP)
interface DownloadService {  1 usage
    void download(Book book);  no usages
}
// PaymentService interface (DIP, ISP)
interface PaymentService {  1 usage
    void processPayment(User user, double amount);  no usages
}
// Concrete implementation of DownloadService (OCP, SRP)
class BookDownloadService implements DownloadService {  no usages
    @Override  no usages
    public void download(Book book) {
        System.out.println("Downloading book: " + book.getTitle());}}
// Concrete implementation of PaymentService (OCP, SRP)
class PaymentProcessor implements PaymentService {  no usages
    @Override  no usages
    public void processPayment(User user, double amount) {
        System.out.println("Processing payment of $" + amount + " for user: " + user.getUsername());
    }
}
```

Figure #23: Download Class After

Single Responsibility Principle (SRP)

- Before

The original Download class had multiple responsibilities, including managing user and book data, handling the downloading of books, and processing payments.

- After

The User class handles user-related data, the Book class manages book-related data, and the BookDownloadService class oversees the downloading process. This approach ensures that each class has a single responsibility, making the system modular and easier to maintain.

- Impact

This separation of concerns improves modularity, maintainability, and scalability of the system.

Open/Closed Principle (OCP)

- Before

The original Download class was not open for extension but open to modification. Any new type of download (e.g., audiobook download) would require modifying the Download class.

- After

Interface DownloadService, for download functionality and implemented a concrete class, BookDownloadService, to focus on downloading books, allowing for the addition of other download types, like AudioBookDownloadService, without modifying existing code.

Dependency Inversion Principle (DIP)

- Before

In the original Download class, the downloading logic was tied directly to the class. This created a dependency on concrete implementations.

- After

The download functionality was abstracted into the DownloadService interface, implemented by the BookDownloadService class, enabling high-level modules, such as controllers or main logic, to depend on the abstraction rather than the concrete implementation, thereby decoupling the high-level logic from specific download implementations.

Interface Segregation Principle (ISP)

- Before

Although the original Download class did not use interfaces, it bundled unrelated operations like downloading and payment into one class. This could have led to an oversized interface if expanded.

- After

Downloading is handled by DownloadService and payments by PaymentService, ensuring each service has a distinct purpose without unnecessary dependencies.

Liskov Substitution Principle (LSP)

- Before

The original Download class didn't use inheritance or interfaces, so it wasn't substitutable.

- After

DownloadService interface allows any subclass, such as BookDownloadService or AudioBookDownloadService, to be used interchangeably, ensuring implementations can replace the base interface without breaking the system.

4.2.7 Payment Gateway

```
3   // Payment Gateway class (without SOLID principles)
4   class PaymentGateway {  no usages
5       private int amount;  no usages
6
7       public void transferCredential() {  no usages
8           System.out.println("Transferring credentials...");
9       }
10
11      public void authenticatePayment() {  no usages
12          System.out.println("Authenticating payment...");
13      }
14  }
15
```

Figure #24: Payment Gateway Class Before

```java
// PaymentDetails class (SRP)
class PaymentDetails {  1 usage
    private int amount;  2 usages
    public PaymentDetails(int amount) {  no usages
        this.amount = amount;}
    public int getAmount() {return amount;}  1 usage
}
// PaymentProcessor interface (DIP, ISP)
interface PaymentProcessor {  4 usages
    void transferCredential();  1 usage
    void authenticatePayment();  1 usage
}
// CreditCardPayment class (SRP, OCP)
class CreditCardPayment implements PaymentProcessor {  no usages
    @Override  1 usage
    public void transferCredential() {
        System.out.println("Transferring credit card credentials...");
    }
    @Override  1 usage
    public void authenticatePayment() {System.out.println("Authenticating credit card payment...");}}
// PayPalPayment class (SRP, OCP)
```

Figure #25.1: Payment Gateway Class After

```java
class PayPalPayment implements PaymentProcessor {  no usages
    @Override  1 usage
    public void transferCredential() {
        System.out.println("Transferring PayPal credentials...");}
    @Override  1 usage
    public void authenticatePayment() {
        System.out.println("Authenticating PayPal payment...");}}
// PaymentService class (SRP, DIP)
class PaymentService {  no usages
    private PaymentProcessor paymentProcessor;  3 usages
    public PaymentService(PaymentProcessor paymentProcessor) {  no usages
        this.paymentProcessor = paymentProcessor;}
    public void processPayment(PaymentDetails details) {  no usages
        paymentProcessor.transferCredential();
        paymentProcessor.authenticatePayment();
        System.out.println("Processing payment of amount: $" + details.getAmount());}}
```

Figure #25.2: Payment Gateway Class After

Single Responsibility Principle (SRP)

- Before

The original PaymentGateway class violated the Single Responsibility Principle (SRP) by handling multiple tasks, including transferring credentials, authenticating payments, and directly managing the amount attribute.

- After

Separated the responsibilities into the PaymentDetails class for managing payment-related data (e.g., amount) and the PaymentProcessor interface with its implementations for handling payment-specific tasks (e.g., transferring credentials, authenticating payments). This ensures each class has a single responsibility, making the system easier to maintain and extend.

Open/Closed Principle (OCP)

- Before

The original PaymentGateway class was not extendable without modification. Adding new payment methods, like PayPal or Credit Card, would require modifying the PaymentGateway class itself.

- After

The PaymentProcessor interface was introduced with separate implementations for each payment type, enabling easy extension without modifying existing code, in line with the open/closed principle.

Dependency Inversion Principle (DIP)

- Before

In the original design, high-level modules (e.g., a PaymentService class or main application logic) would depend directly on the concrete PaymentGateway class. This tight coupling made the system rigid and less adaptable.

- After

The PaymentProcessor interface decouples the high-level PaymentService class from specific implementations, with classes like CreditCardPayment and PayPalPayment implementing it, ensuring flexibility and adaptability.

Interface Segregation Principle (ISP)

- Before

The original PaymentGateway class had no interfaces. While the class was small, expanding it to handle multiple payment methods (e.g., PayPal, Credit Card, Bank Transfer) could result in a bloated class with unused methods for some payment types.

- After

The PaymentProcessor interface ensures each payment method implementation defines and uses only relevant methods, avoiding the need for any class to implement unnecessary methods.

Liskov Substitution Principle (LSP)

- Before

The original PaymentGateway class didn’t implement any abstractions, so it couldn’t

be substituted. It was directly tied to the application logic.

- After

Introducing the PaymentProcessor interface allows any implementation, such as CreditCardPayment or PayPalPayment, to replace the interface in the application logic, ensuring new classes behave correctly without breaking the system.

4.3 Design Patterns

4.3.1 Creational Design Patterns

- Singleton Pattern (for Admin or Payment Gateway)

For managing the Admin or Payment Gateway, you want to ensure there is only one instance of these classes running at any given time to maintain consistent global state and prevent redundant operations.

- Factory Pattern (for User Creation)

To handle the creation of different types of users (GuestUser, PremiumUser, Admin) based on user roles dynamically. This avoids hardcoding user object creation and simplifies code extensibility.

4.3.2 Structural Design Patterns

- Decorator Pattern (for Book Features)

To handle different behaviors for books like view, read, or download without modifying the Books class. Extends functionality in a flexible way to support premium or guest-specific behaviors.

- Adapter Pattern (for Payment Gateway Integration)

To integrate local mobile money payment (specific to Zimbabwe) with the system's existing payment gateway structure.

4.3.3 Behavioral Design Patterns

- Observer Pattern (for Announcement Panel)

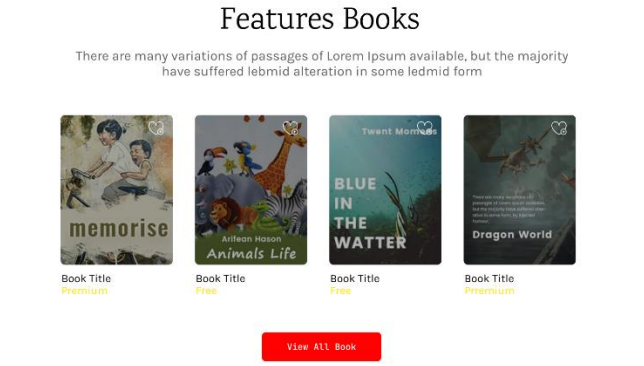
To notify all members when the admin posts a new announcement. Ensures a real-time update mechanism for connected users.

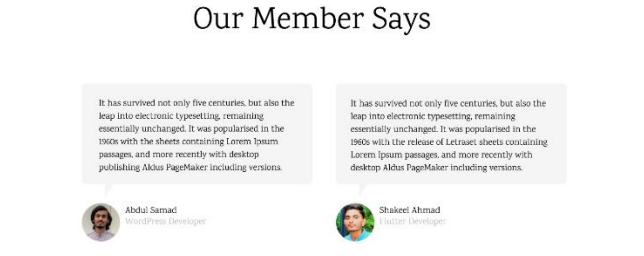
- Strategy Pattern (for Search Functionality)

To support multiple search criteria (title, author, genre) without hardcoding logic into the Books class. Simplifies the addition of new search criteria.

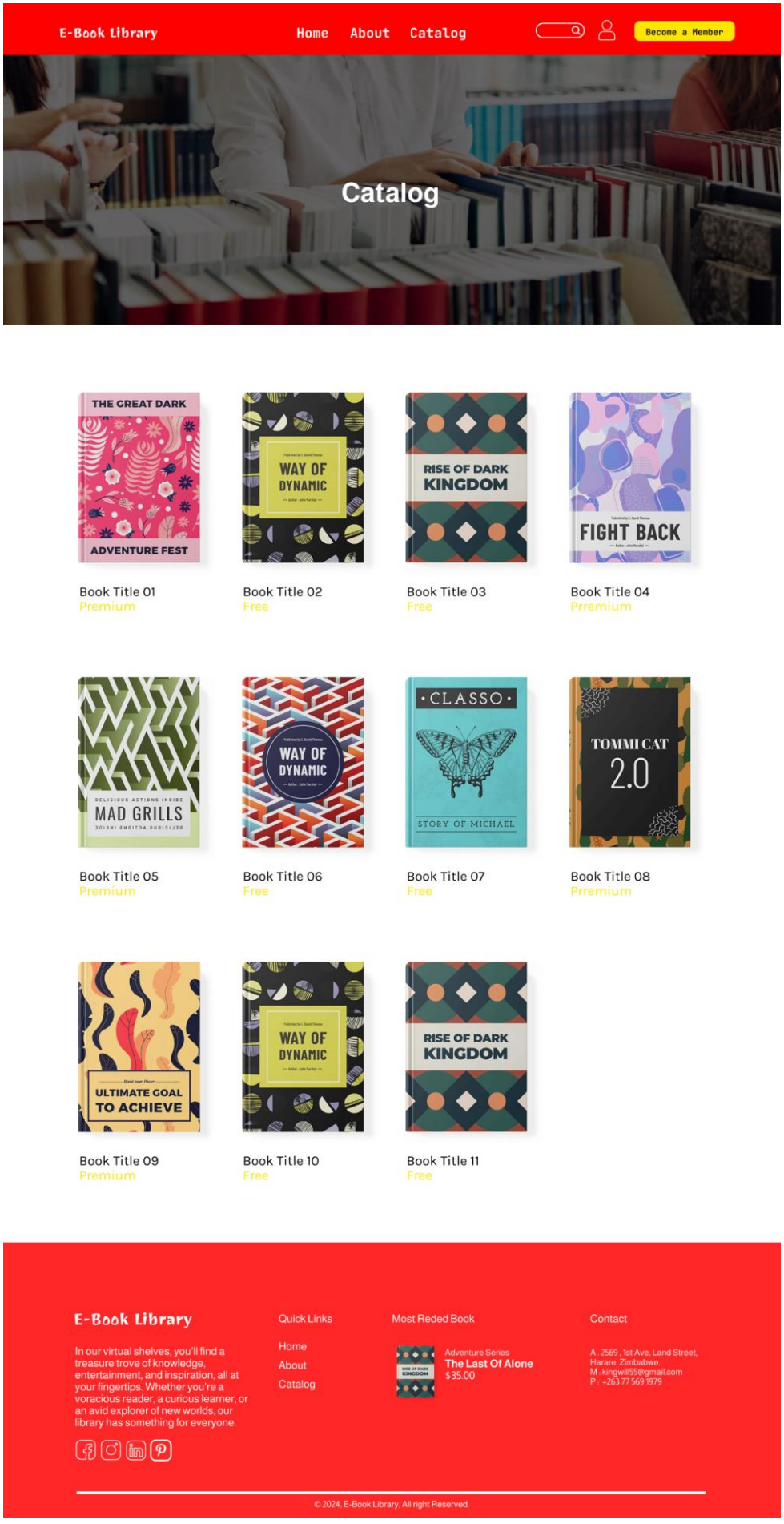
5. Prototyping

5.1 Home

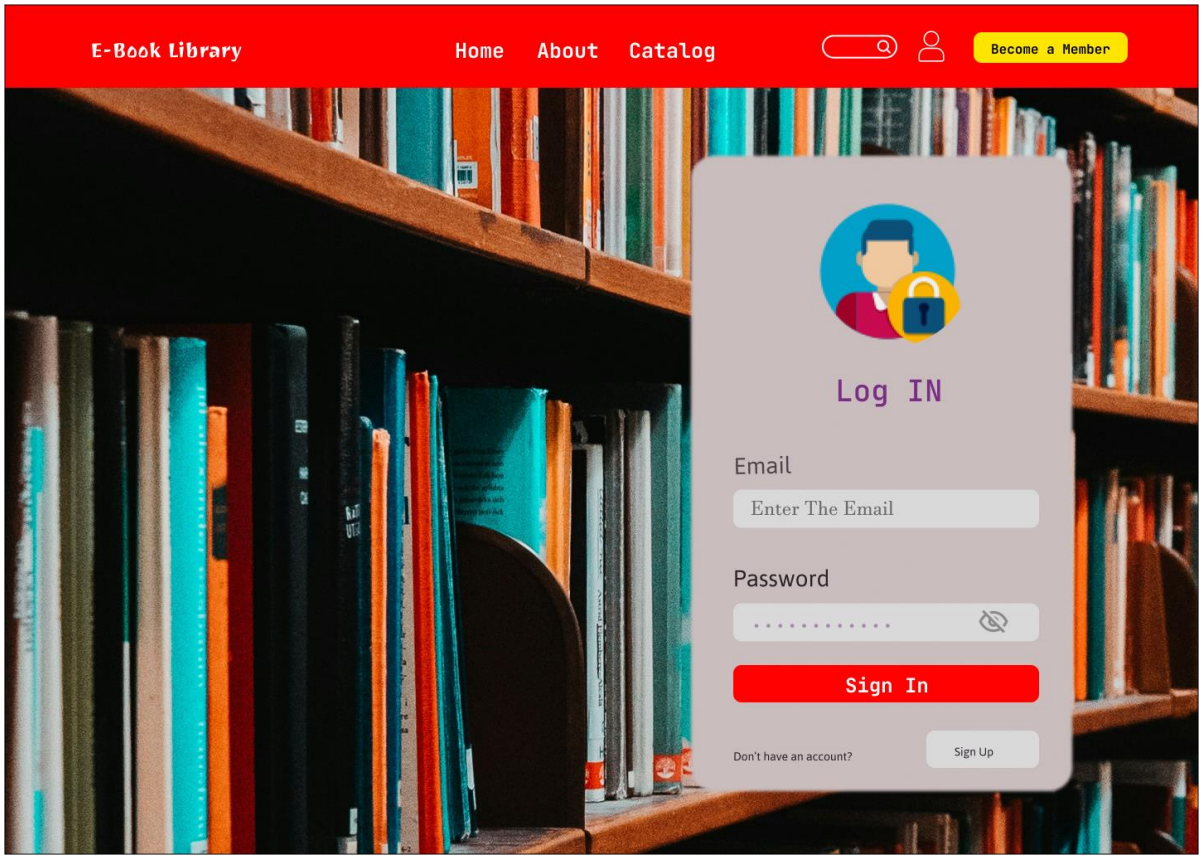




5.2 Catalog


E-Book Library
Home
About
Catalog
Become a Member
Catalog
THE GREAT DARK
ADVENTURE FEST
Book Title 01
Premium
WAY OF DYNAMIC
Book Title 02
Free
RISE OF DARK KINGDOM
Book Title 03
Free
FIGHT BACK
Book Title 04
Prremium
MAD GRILLS
Book Title 05
Premium
WAY OF DYNAMIC
Book Title 06
Free
CLASSO
STORY OF MICHAEL
Book Title 07
Free
TOMMI CAT 2.0
Book Title 08
Prremium
ULTIMATE GOAL TO ACHIEVE
Book Title 09
Premium
WAY OF DYNAMIC
Book Title 10
Free
RISE OF DARK KINGDOM
Book Title 11
Free
E-Book Library
In our virtual shelves, you'll find a treasure trove of knowledge, entertainment, and inspiration, all at your fingertips. Whether you're a voracious reader, a curious learner, or an avid explorer of new worlds, our library has something for everyone.
Quick Links
Home
About
Catalog
Most Reded Book
Adventure Series
The Last Of Alone
$35.00
Contact
A : 2569 , 1st Ave, Land Street, Harare, Zimbabwe.
M : kingwill55@gmail.com
P : +263 77 569 1979
© 2024, E-Book Library, All right Reserved.

5.3 Login



5.4 Sign up