DOKUZ EYLUL UNIVERSITY
ENGINEERING FACULTY
DEPARTMENT OF COMPUTER ENGINEERING

Phase-3 Library System

By
2022510164 Erol Çelik
2022510012 Rufat Tuhilar
2022510116 Kader Eraslan

IZMIR
17.06.2025

DEUCENG
Dokuz Eylül University
Dept of Computer Engineering

Library System

System Definition and Objectives

The Library Management System is a digital solution designed to streamline and automate the operations of a library. Its primary objective is to enhance the efficiency of library services by enabling users, librarians, and administrators to interact with a centralized system that facilitates book borrowing, returning, searching, and management activities.

This system addresses common challenges encountered in traditional library operations, such as manual tracking of borrowed materials, delayed returns, and inconsistent communication between staff and patrons. By automating core functions—including inventory updates, user management, penalty assessments, and notifications—the system minimizes human error, ensures data consistency, and improves the overall user experience. Furthermore, it supports scalability and accessibility, making it suitable for academic institutions, public libraries, and private organizations.

Functional Requirements

The Library Management System includes a comprehensive set of functionalities distributed among different user roles, ensuring secure and role-based access to system operations. The following are the core functional requirements categorized by user type:

1. User Functions (Members)

- Login/Authentication: Users must be able to securely log into the system.
- Book Search: Users can search for available books based on various filters.
- Reservation: Users can reserve books that are currently on loan.
- Penalty Payment: Enables users to pay late return penalties.
- Account Management: Users can manage their profile and change their password.
- Registration: New users can register by filling in the required member form.
- Borrowing History: Users can view their borrowing and penalty history.

2. Librarian Functions

- Approve Borrow / Return Requests: Librarians can validate and confirm book returns.
- Late Return Check: System checks for overdue items and flags them.
- Calculate Penalties: Automatic calculation of penalties for overdue items.
- Update Book Status: Librarians can change the availability or condition of books.
- Approve Reservation Requests: Librarians review and approve book reservations.
- Send Notifications: Librarians can send alerts or updates to users.

3. Administrator Functions

- User Management: Admins can add, delete, or suspend user accounts.
- Notify Users: Admins may issue system-level notifications or warnings.

- Book Management:

  - Add New Books: Insert new book entries into the catalog.
  - Remove Books: Delete outdated or damaged books from the system.
  - Update Book Details: Modify metadata such as title, author, or category.
  - Manage Categories: Organize books into appropriate classifications.

Non-Functional Requirements

In addition to its core functionalities, the Library Management System must meet several non-functional requirements that influence its overall quality, performance, and user satisfaction. These requirements ensure the system is reliable, secure, and adaptable to various contexts of use.

1. Performance

- Book search, login, and transaction processes (borrow/return) should be executed within a maximum of 2 seconds under normal load conditions.
- The system must be available.

2. Security

- All user data, including login credentials and personal information, must be securely stored using encryption protocols.
- Role-based access control must be enforced to restrict functionalities based on user roles (User, Librarian, Admin).
- The system must include session management and automatic logout after a period of inactivity to prevent unauthorized access.
- Audit logging must be enabled to track critical actions such as user deletions, book removals, and account modifications.
- Personal data such as member information and borrowing history must be kept confidential.
- Only authorized individuals (e.g. librarians and administrators) should have access to this data..

3. Scalability

- The architecture should support the addition of new modules (e.g., digital book lending, interlibrary loans) without requiring major reconfiguration.
- The database should efficiently handle large volumes of transactions and data entries with minimal impact on performance.

4. Usability

- The user interface must be intuitive and accessible, with consistent navigation and clear labeling of actions.
- Help documentation or tooltips must be available to guide users.

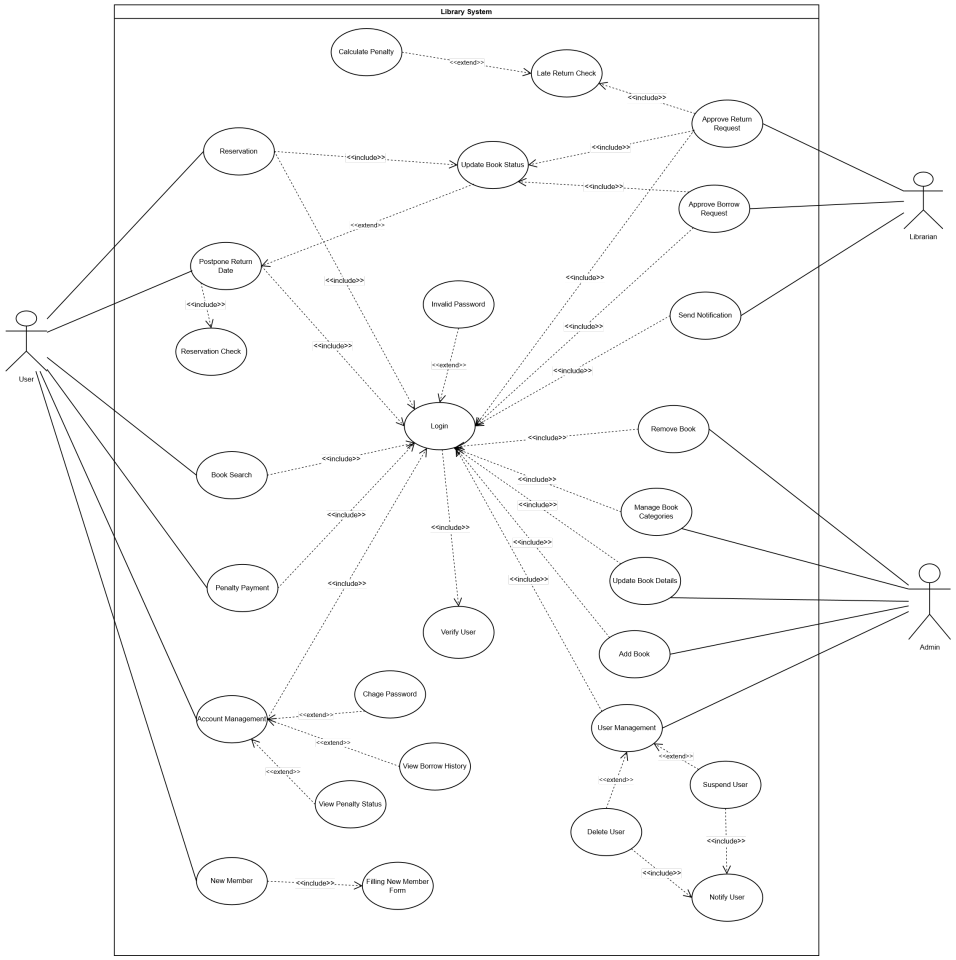
5. Maintainability and Extensibility

- The system's code should be clear and modular, so updates and bug fixes can be done easily.
- Adding new features or changing parts of the system should not break the main functions.

6. Reliability and Availability

- The system must be fault-tolerant, capable of recovering from errors without data loss.
- Backup and restore mechanisms must be implemented to ensure data integrity in case of system failure.

Use Case Diagram



Use Cases

1. User Manages Account

Who: User
Steps:

1. User logs into the system.
2. User navigates to "Account Management" section.
3. User views their loan history and penalty status.
4. User updates personal information (e.g., phone, email).
5. User changes password if needed.
6. System saves the updated information

2. Librarian Updates Book Status

Who: Librarian
Steps:

1. Librarian logs into the system.
2. Librarian selects a book from the inventory.
3. Librarian updates the status to “Available”, “Reserved”, “Borrowed”, “Returned”, or “Lost”.
4. System saves the updated book status for tracking and inventory management.

3. User Logs in and Verifies Identity

Who: Person
Steps:

1. User enters their login credentials.
2. System verifies username and password.
3. If credentials are valid, system grants access.
4. If invalid, system displays an error message.

4. Librarian Approves Borrow or Return Request

Who: Librarian
Steps:

1. Librarian logs in and opens the notification panel.
2. Librarian reviews pending borrow or return requests.
3. System shows relevant user and book details.
4. Librarian approves or rejects the request.
5. System updates book and user records accordingly.

5. Admin Manages Users

Who: Admin
Steps:

1. Admin logs into the system.
2. Admin selects "User Management" menu.
3. Admin performs one of the following:
    - Adds a new user by filling in user details.
    - Updates an existing user's information.
    - Suspends a user account.
    - Deletes a user account.
4. System applies the chosen action and updates records.

Library System Architecture

The system is designed as a centralized **management system** where all core operations—such as user account control, book inventory handling, reservation processing, and category-based organization are coordinated under a unified administrative structure. The Library class serves as the central controller, maintaining arrays of Book, Person, and Category objects. All user interactions such as searching for books, making reservations, borrowing items, and managing account details are routed through this central component.

The system follows an object-oriented structure based on the inheritance of a general Person class, from which the User, Librarian, and Admin classes are derived. Each role has a specific set of responsibilities and access permissions within the system. The User class is responsible for operations such as book reservation, borrowing, and managing personal account details. The Librarian processes user requests, including approving borrow or return actions and updating book statuses accordingly.

The Admin class has full privileges within the management system. It can manage the book catalog (add/remove/update), organize categories, and control user accounts by suspending or deleting users. All datasets are integrated within this management environment, and only the Admin has access to the entire dataset and the authority to perform changes.

Books and categories have a many-to-many relationship. Each book can belong to multiple categories, and each category can reference multiple books. This flexible structure enables users to list and filter books by category efficiently. The Category class stores its related books, supporting responsive and structured browsing capabilities.

System notifications are handled by the Notification class and are sent to users to inform them of system events such as approvals, account changes, or penalties. The overall architecture ensures a structured, role-driven, and centrally managed environment for effective library system operations.

Code Structure and Class Design

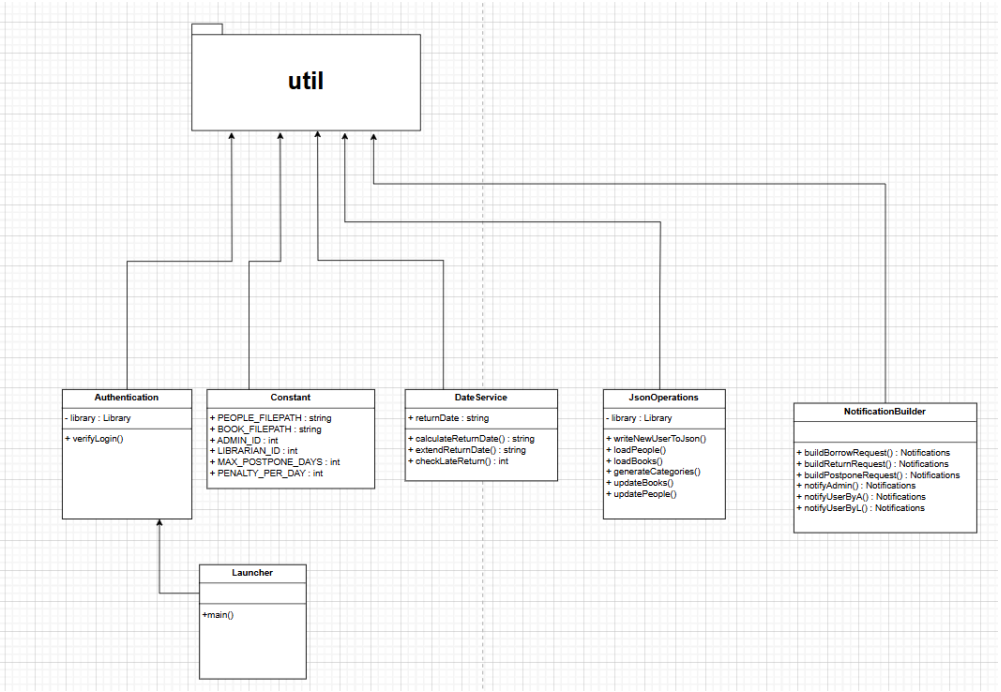
The system has been architected into three primary packages—**service, core, and util**—to ensure a clear separation of concerns, promote functional responsibility, and enhance logical modularity. This structured approach significantly improves code readability, maintainability, and extensibility.

1. The util Package: Utilities and General Functionalities

The util package forms a dedicated layer within the Library Management System, housing various utility classes and general-purpose functionalities. These classes are typically employed by other layers of the system to perform common tasks that are not directly tied to specific business logic but are essential for the overall operation of the application.

The Authentication class manages user authentication processes, ensuring secure access to the system. The Constant class defines application-wide fixed values, such as file paths, default IDs, and penalty rates, which significantly enhances code readability and maintainability. The DateService class abstracts complex date and time-related operations, including the calculation and extension of book return dates, and the checking of overdue returns.

The JsonOperations class handles data persistence by managing the reading from and writing to JSON files for system data, such as users and books. This class is crucial for ensuring data retention across application sessions and interacts directly with the Library class. Finally, the NotificationBuilder class applies a builder design pattern to consistently construct various types of notification objects (e.g., borrow requests, return requests), while the Launcher class encapsulates the application's main entry point (main method).



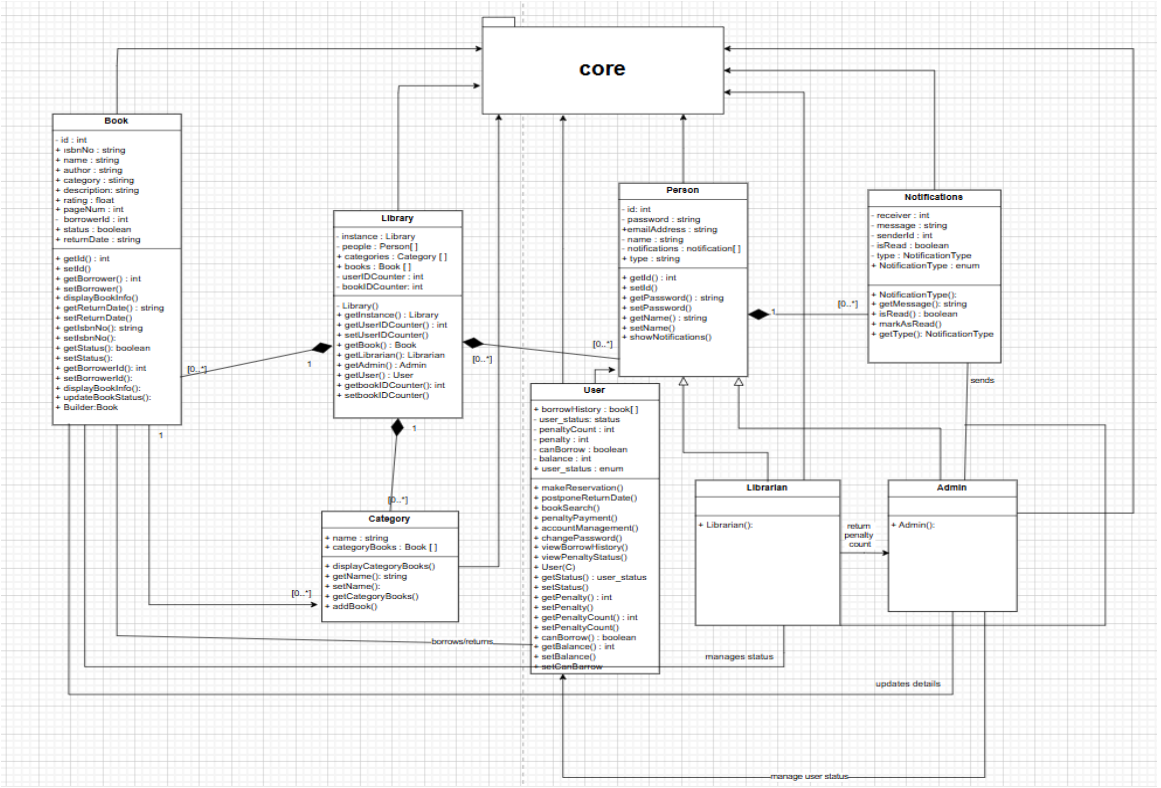
2. The core Package: Core Entities and Data Model

The core package constitutes the foundational layer of the Library Management System, defining its essential data structures and core entities. This layer encapsulates the system's fundamental business rules and its underlying data model, representing the most elemental components of the application.

At the heart of this package are classes such as Book, Category, and Person. The Book class meticulously defines all intrinsic attributes of a book, including id, name, author, and status, along with relevant methods like displayBookInfo(). The Category class facilitates the logical grouping of books, managing the collection of books contained within each specific category.

The Person class serves as an abstract base class, defining common attributes (ID, password, name, address) and behaviors for all user types within the system: User, Librarian, and Admin. This implementation leverages the principle of inheritance to minimize code duplication and establish a more consistent user modeling approach. The User class extends Person, incorporating user-specific details and methods such as borrowing history management, penalty status, and account operations. Similarly, Librarian and Admin classes inherit and extend functionalities pertinent to their respective roles.

The Library class functions as a centralized hub for the entire system, holding collections of core entities like Books, Categorys, and Person objects. Its design as a Singleton ensures that only a single instance of the Library object exists throughout the system's lifecycle, providing a global access point and thereby facilitating data consistency. The Notifications class defines the structure and status (read/unread) of messages dispatched to users, operating in close association with the Person class.

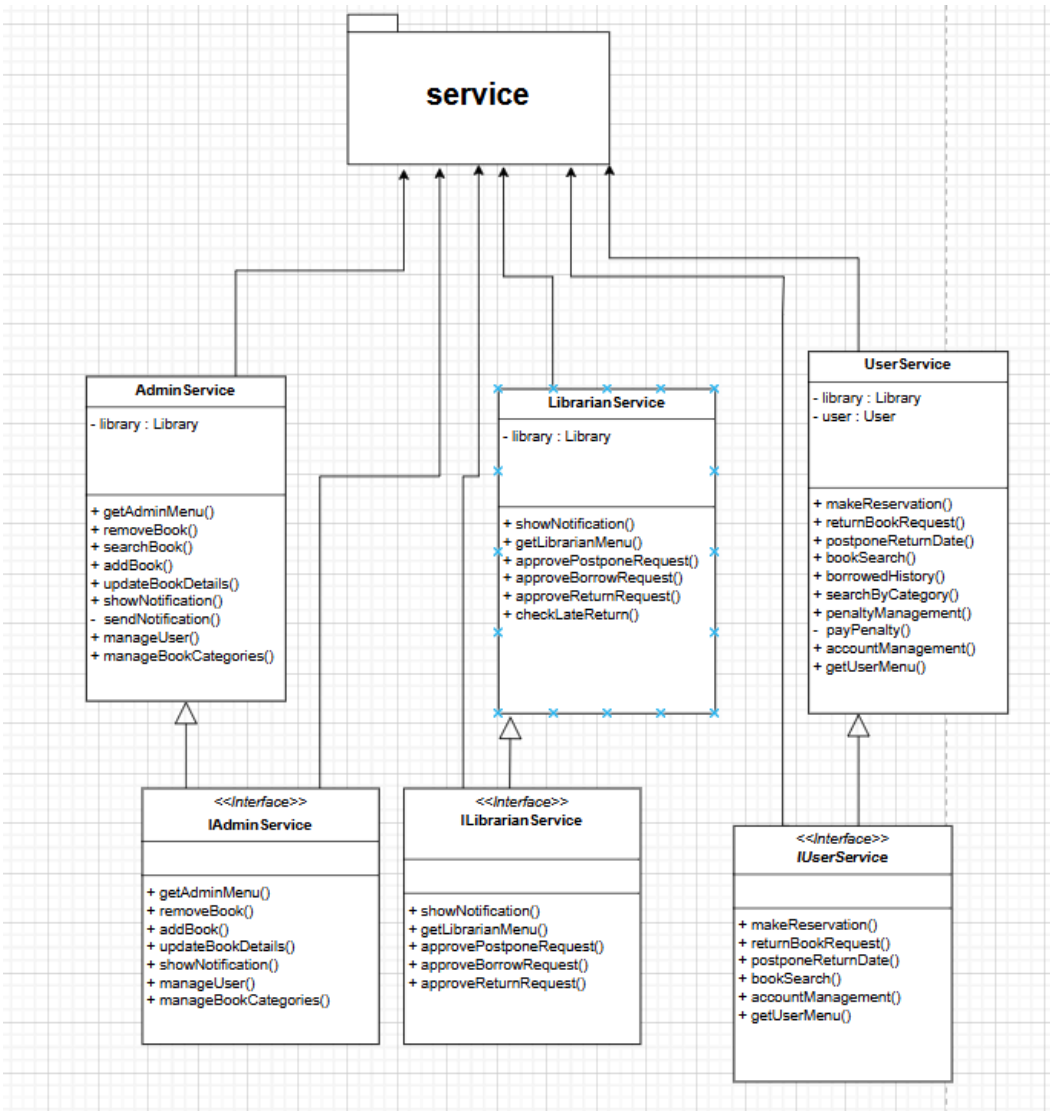


3. The service Package: Business Logic Layer

The service package embodies the business logic layer of the Library Management System. This layer is primarily responsible for processing requests originating from the user interface and orchestrating interactions with the core entities defined within the core package, thereby enforcing the system's business rules. This design paradigm effectively decouples the business functionality from the presentation layer, leading to a cleaner and more manageable codebase.

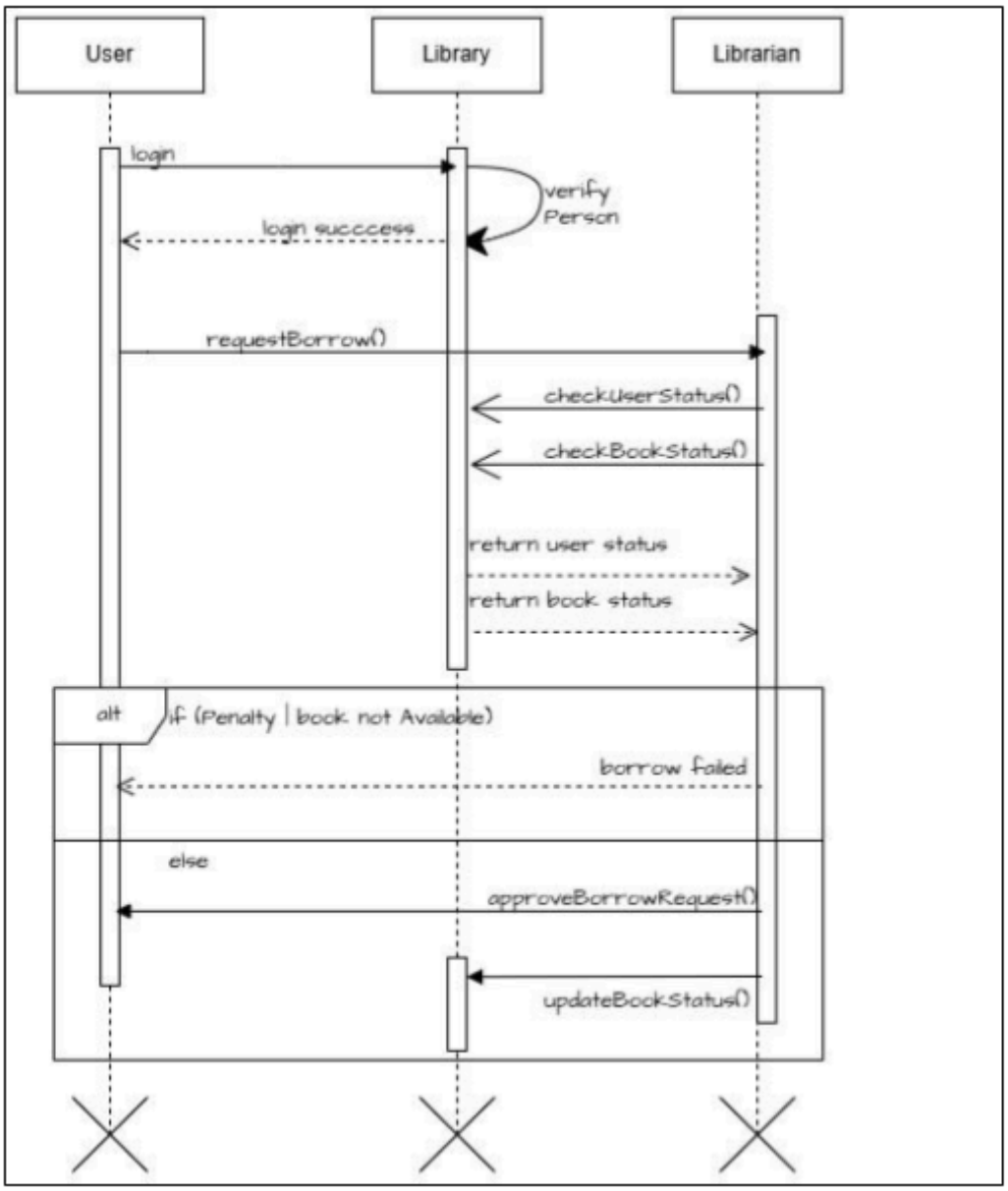
Within this package, distinct service classes are tailored for different user roles: AdminService, LibrarianService, and UserService. Each service class encapsulates the specific operations that a given user type can perform. For instance, AdminService manages administrative tasks such as adding, removing, and searching books, user management, and book category administration. Conversely, LibrarianService handles librarian-specific functionalities, including the approval of borrowing and return requests, and the monitoring of overdue returns. The UserService facilitates individual user operations, such as making book reservations, initiating return requests, viewing borrowing history, and settling penalties.

These service classes implement their respective interfaces—IAdminService, ILibrarianService, and IUserService. This adherence to interface-based programming enhances design flexibility and reduces inter-component dependencies. The use of interfaces aligns with the Interface Segregation Principle, defining more specific and focused contracts for each service, thereby exposing only the necessary methods. The services interact with core classes like Library and User to process and manipulate data.



Sequence Diagrams

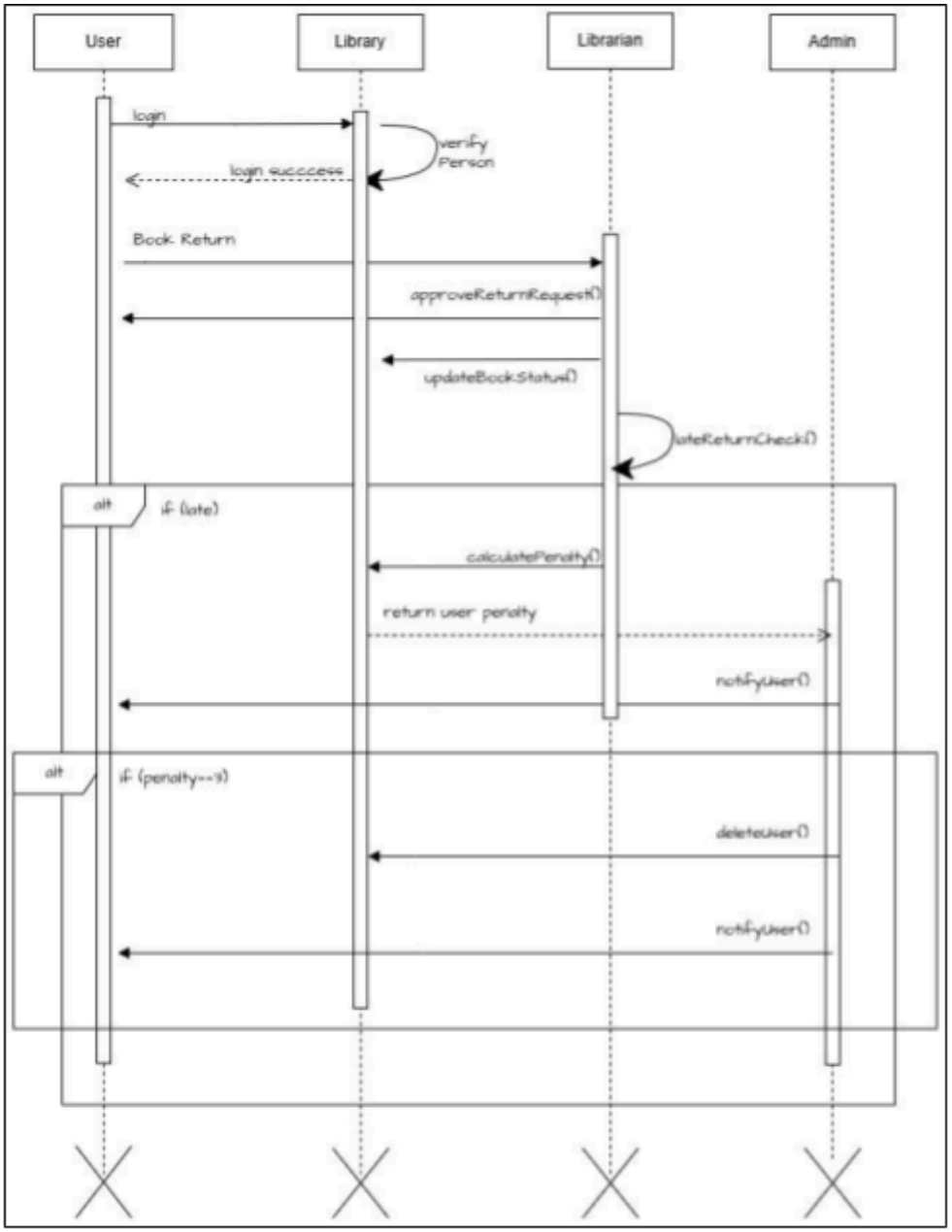
1.Borrowing a Book



This sequence shows the interaction between the User, Library, and Librarian classes to manage arule-based book borrowing process.

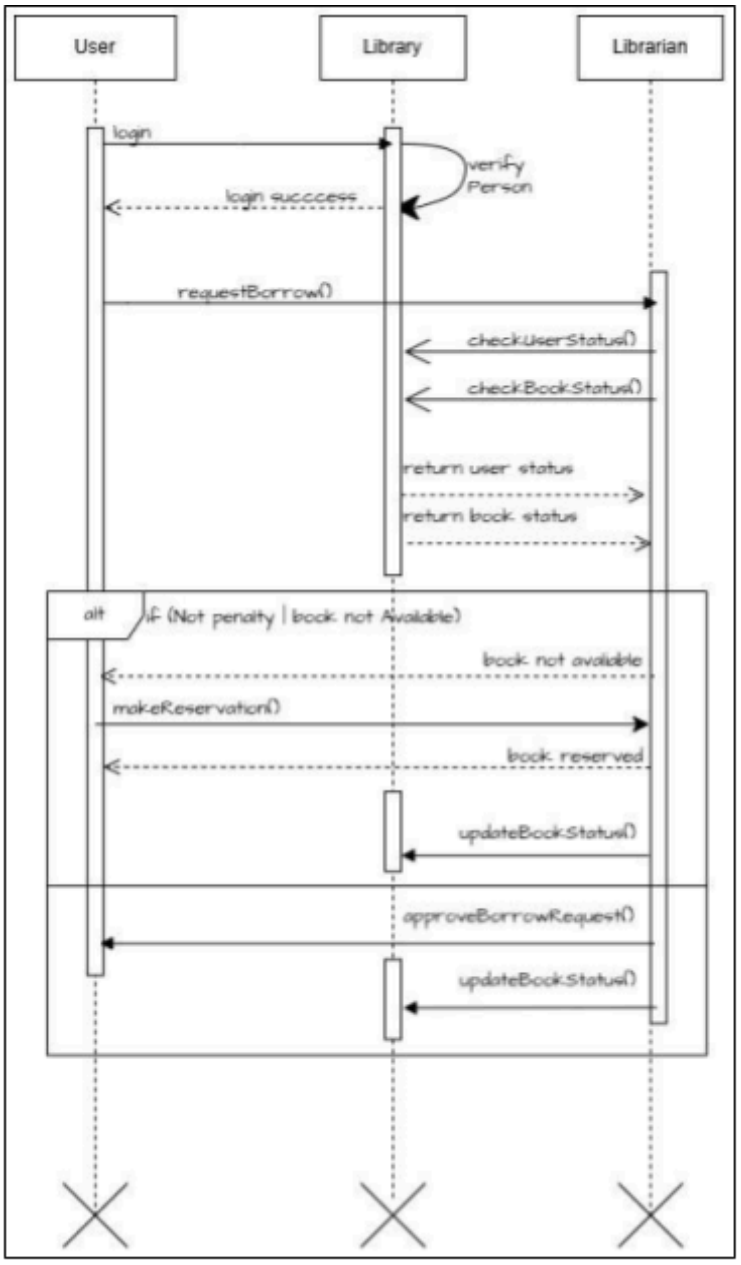
The diagram starts with the user logging in, after which the system verifies their identity. Once authenticated, the user sends a borrow request. The system checks both the user's status (e.g., existing penalties) and the availability of the book. If both conditions are satisfied, the librarian approves the request and updates the book's status. If not, the system informs the user that the borrow operation has failed.

2.Returning a Book with Penalty



This sequence shows how the system processes a late book return and enforces return policies through role-based interactions. After logging in and being verified by the system, the user returns a book. The librarian checks whether the return is late. If so, a penalty is calculated and returned. If the user's penalty count reaches a critical level (such as 3), the system automatically informs the admin. Then, the admin deletes the user from the system and sends a notification. This diagram reflects how the system responds to late actions and takes corrective measures.

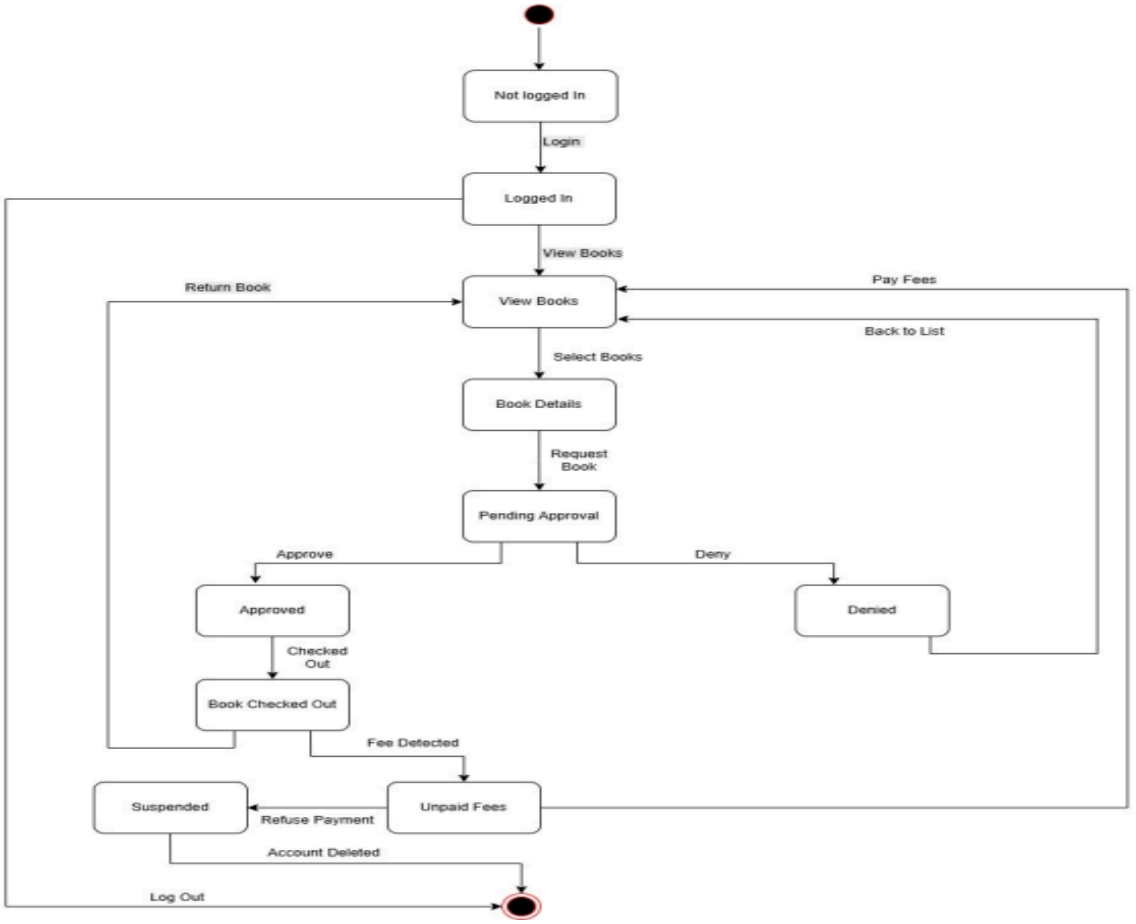
3.Book Reservation Process



This sequence shows how the system handles book reservations when a requested book is currently unavailable but the user is eligible. The user logs into the system, and their identity is verified by the Library class. After that, the user attempts to borrow a book. The system checks the user's status and the book's availability. If the user has no penalties but the book is not available, the system allows the user to place a reservation by calling the makeReservation() method.

Statechart Diagram

1.Book Request and Fee Handlin



This statechart diagram shows the behavior of a user in the library system, focusing on the book request process and penalty handling. The process begins with the user in the Not Logged In state. After a successful login, the user moves to the Logged In state and is allowed to browse and view books. Once a book is selected, the user enters the Book Details state and may send a request to borrow the book, leading to the Pending Approval state.

At this stage, the librarian may either approve or deny the request:

- If approved, the book is marked as Checked Out, and the user moves to the Book Checked Out state.

- If denied, the process moves to the Denied state and returns to the book list.

During the Book Checked Out state, if a late return or penalty condition is detected, the system transitions to the Unpaid Fees state. If the user fails to pay the fee, the account becomes Suspended. If the user refuses payment three times, the account is marked as Deleted. The user may return a book or pay the fee at any time, leading back to the main View Books state or resolving the suspension. This diagram demonstrates how the system dynamically transitions between states based on user actions, admin decisions, and policy rules.

Activity Diagram

1.Book Return and Penalty Process

This activity diagram models the workflow starting from a user's book return request and includes the penalty handling process in case of a late return. The process begins with the user sending a book return request. The librarian first checks whether the return is late.

- If it is not late, the return request is approved directly, and the book status is updated.

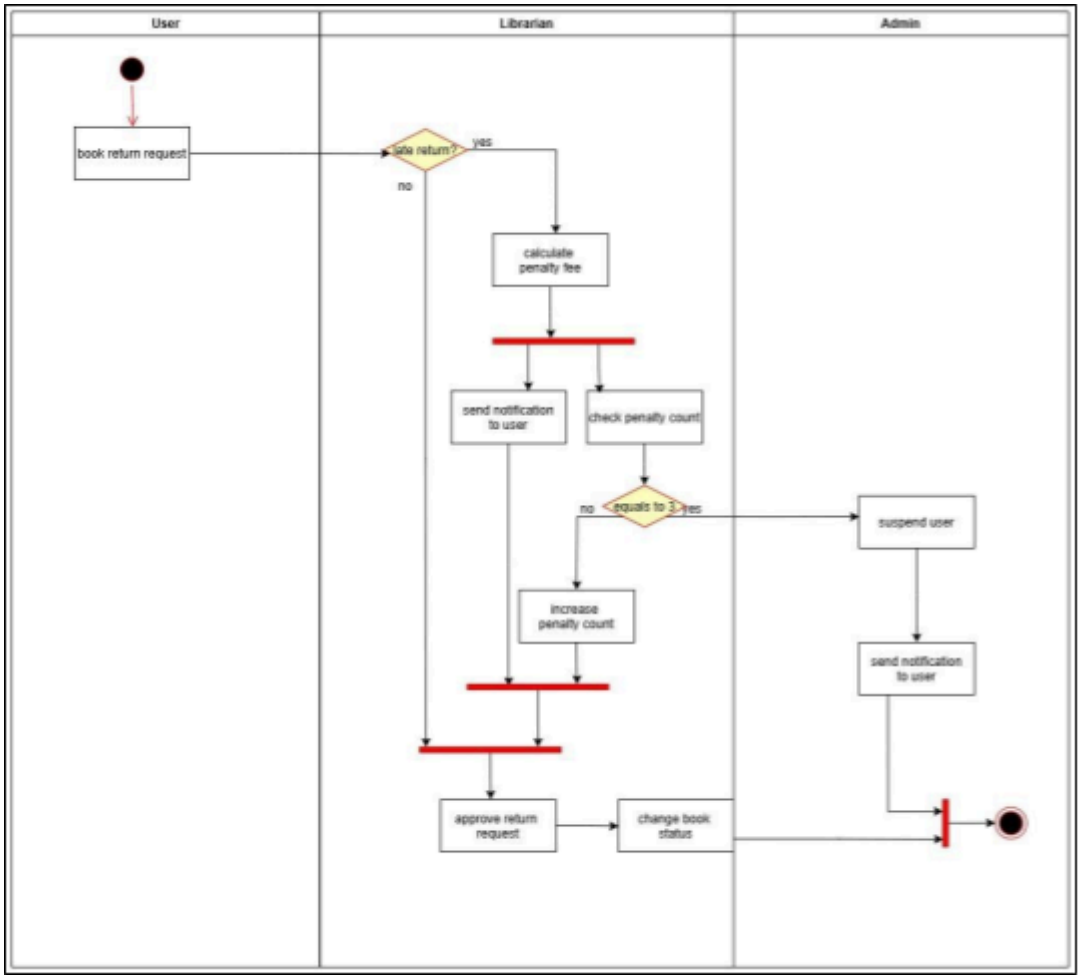
- If the return is late, the librarian calculates the penalty fee, sends a notification to the user, and checks the current penalty count.

Depending on the user's penalty history:

- If the count has reached 3, the process moves to the Admin, who suspends the user and sends a final notification.

- If the count is less than 3, the penalty count is increased, and the process continues.

Finally, the librarian approves the return request and updates the status of the book. This diagram shows how the system manages late returns and escalates the issue to the admin when policy limits are exceeded.



2. Book Borrowing Process

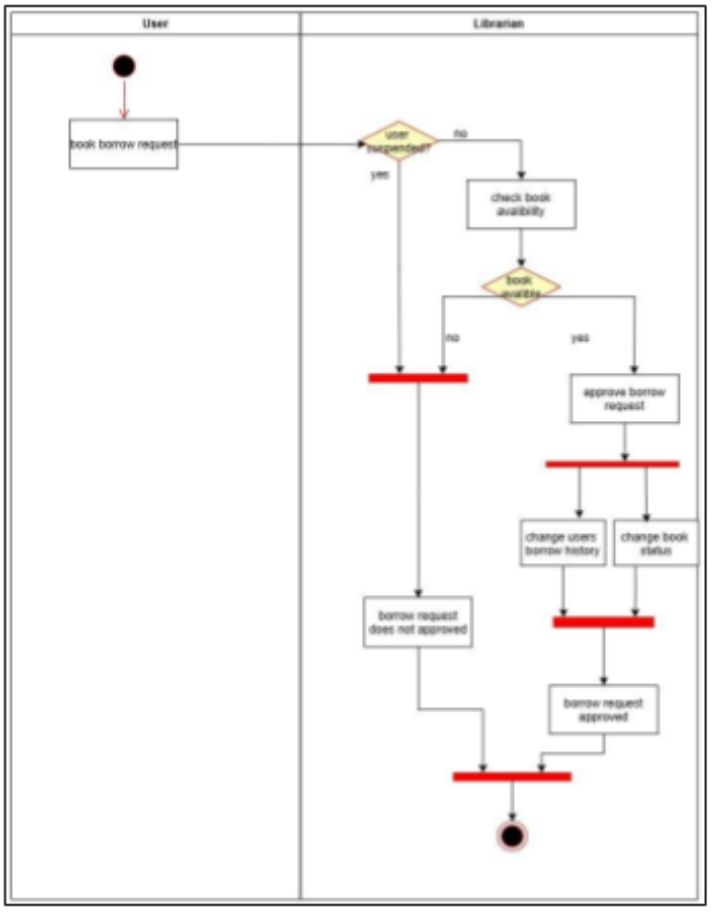
This diagram shows the process that takes place when a user requests to borrow a book and how the librarian handles this request. The process begins when the user sends a borrow request. Then, the librarian checks two conditions:
Is the user suspended the request is not approved, and the process ends.

If the user not suspended the librarian proceeds to the next step.

Is the book available?

- If not: The request is denied.
- If the book is available:

The request is approved, the user's borrow history is updated, and the book status is changed, completing the process successfully.This diagram illustrates how the system responds based on the user's status and the book's availability, and what updates are made to the system if the request is approved.



Overall System Design Description

This library management system is designed with a clear structure that separates responsibilities between different roles and keeps operations organized. The main parts of the system are based on object-oriented design, with classes like User, Admin, and Librarian all connected to a central Person and Library structure.

Each user role has access to specific functions: users can search and reserve books, librarians handle approvals and monitor returns, while admins have full access to update book, user, and category data. Admins are also responsible for taking action when rules are violated, such as suspending users.

To make the system easier to understand and manage, several types of diagrams are used. Class diagrams show how the parts are connected. Sequence diagrams explain how users interact with the system step by step. Statechart diagrams describe how things change based on actions, and activity diagrams help visualize workflows in decision-based processes.

Almost every important scenario in the system is supported by a related diagram, which helps make the overall design easier to follow and develop in the future.

Test Approach and Results

Our testing strategy encompassed both unit and functional testing to ensure the robustness, correctness, and adherence to requirements of the developed system. The insights gained from these testing phases were instrumental in identifying and rectifying critical issues, thereby enhancing the overall system quality.

Unit Testing: During the unit testing phase, individual components and methods were tested in isolation to verify their correct functionality. This included thorough checks of data manipulation, logical operations, and edge case handling for each class and method, ensuring that each part of the system performed as expected independently.

Functional Testing and Identified Issues: Functional testing was conducted to validate the system's compliance with specified requirements from an end-user perspective. This comprehensive testing revealed several critical issues that required immediate attention:

1. **Multiple Book Reservations by Users:** Initially, a significant functional flaw was identified where a User could make multiple book reservations, even if they already had books checked out. This unintended behavior was promptly addressed and corrected, ensuring that the system now properly restricts users from making excessive reservations, thereby promoting fair access to library resources.
2. **Unauthorized Book Deletion by Admin:** Another critical issue discovered was that an Admin user possessed the ability to delete a book from the system even if it was currently checked out by a User. This constituted a potential data integrity risk. We implemented a protective measure to prevent this action, ensuring that books actively in circulation cannot be deleted, thus safeguarding loan records and preventing data inconsistencies.
3. **Librarian Approval/Denial Order Inflexibility:** In the librarian module, it was observed that librarians lacked the flexibility to approve or deny book requests (e.g., borrowing or return requests) in a custom or preferred order. The system previously enforced a rigid processing sequence. To enhance operational efficiency and user experience for librarians, we introduced improvements that now allow them to manage and process these requests in a more convenient and prioritized manner.

These functional test findings were crucial in refining the system, ensuring that it operates according to the intended design and user expectations, while also fortifying its data integrity and usability. The successful resolution of these issues underscores the effectiveness of our iterative testing and development approach.

Reflection and Future Improvements

During the development of the library management system, several technical and design-related challenges were encountered. One of the primary difficulties involved establishing a clear separation of concerns between service, utility, and data processing layers. As observed in the system architecture, distinct responsibilities were assigned to components such as JsonOperations, DateService, and service classes (AdminService, LibrarianService, UserService), each aligned with their respective interface definitions. This modular approach contributed to system maintainability but required careful orchestration to ensure proper dependency management, especially in cross-component interactions.

Another significant challenge was implementing state transitions effectively, as outlined in the statechart diagram. Capturing the complete user flow — from login to book reservation and return, including scenarios like overdue penalties and account suspension — necessitated integrating business logic with user interface control structures. Ensuring that every possible transition (e.g., from "Pending Approval" to either "Approved" or "Denied") triggered the correct notifications and updates required meticulous validation and testing.

Throughout the process, we gained a deeper understanding of interface-driven design, particularly the benefits of polymorphism in service classes. Interfaces like IAdminService, ILibrarianService, and IUserService allowed us to decouple implementation logic and standardize behavior across user roles. Similarly, encapsulating static values in a dedicated Constant class simplified configuration changes and avoided hard-coded values.

For future enhancements, several improvements are proposed:

- **Enhanced Error Handling:** While the current system handles expected control flow well, adding comprehensive exception handling (especially in JsonOperations and file I/O processes) would increase robustness.
- **Unit Testing & Test Coverage:** Integrating a formal testing framework would allow us to systematically validate core modules (e.g., verifyLogin, updateBooks, buildBorrowRequest) and detect regressions early.
- **Extending Role Functionality:** Current role-based services could be extended with role-specific dashboards or analytics, especially for admins to track user activity or for librarians to monitor return rates.
- **UI/UX Improvements:** While the statechart defines user flow logically, actual user interface development could be enhanced with a GUI framework or web-based interaction for better accessibility.
- **Persistent Database Integration:** Transitioning from JSON-based file storage to a relational or NoSQL database would improve data integrity, scalability, and concurrent access performance.

Implementation Details

The system is implemented using Java with a modular and object-oriented approach, ensuring separation of concerns and extensibility across user roles: Admin, Librarian, and User. The architecture is composed of service classes, utility modules, domain logic, and interface-driven contracts to guarantee scalability and maintainability.

Key Classes and Responsibilities

1. **Launcher**

   Acts as the system entry point, responsible for initiating the login process and directing users to the appropriate service class (AdminService, LibrarianService, or UserService) based on their role.
2. **Authentication**

   Validates login credentials and identifies the user role. It collaborates with the Library object to verify user data loaded from JSON files and redirects users to the correct service layer.

3. **AdminService** / **LibrarianService** / **UserService**
   Each of these classes implements its respective interface (IAdminService, ILibrarianService, IUserService) to encapsulate role-specific behaviors:
   - AdminService provides functionality to add, remove, and update books, manage users, and assign categories.
   - LibrarianService handles notifications and approves borrow/return/postpone requests.
   - UserService allows users to reserve books, return or postpone return dates, pay penalties, and view history.

4. **JsonOperations**
   Centralized data-handling class responsible for reading from and writing to JSON files. It provides methods such as loadPeople(), loadBooks(), updateBooks(), and updatePeople() to persist user and book data.

5. **NotificationBuilder**
   This class builds structured notification messages (e.g., borrow requests, return confirmations) to be sent to users or admins depending on system actions.

6. **DateService**
   Manages date calculations for due dates, postponement, and lateness penalties. It abstracts time-related logic, ensuring consistent computation across the system.

7. **Constant**
   Stores system configuration such as file paths, penalty constants, and system-wide admin/librarian IDs.

8. **Use Case Mapping to Implementation**
    - Use cases like **"Login"**, **"Verify User"**, and **"Invalid Password"** are implemented via the Authentication class.
    - **"Make Reservation"**, **"Return Book"**, **"Postpone Return Date"**, and **"Penalty Payment"** are handled within UserService using methods like makeReservation(), returnBookRequest(), and payPenalty().
    - Administrative actions such as **"Add Book"**, **"Remove Book"**, **"Manage Book Categories"**, and **"User Management"** are implemented in AdminService.
    - Use cases related to approval workflows like **"Approve Return Request"** and **"Approve Borrow Request"** are executed within LibrarianService, which invokes notification and update mechanisms.
    - Cross-cutting concerns such as **"Update Book Status"** and **"Send Notification"** rely on helper classes like JsonOperations and NotificationBuilder.

Design Highlights

- **Interface Segregation**: Each user role is decoupled via dedicated service interfaces, promoting flexibility and easier testing.
- **Data Persistence**: JSON-based storage is used for simplicity and portability. Future extensions may consider database integration.
- **Single Responsibility Principle**: Classes like DateService and JsonOperations focus on one core concern, making the codebase easier to maintain and debug.
- **State-Driven Behavior**: System behavior changes based on user state and transitions (e.g., from reservation to checked out), as modeled in the statechart and use case diagrams.