Library Management System Report

Fantastic Two

July 20, 2025

Team Introduction

Team Members

Sifat Sabrina Rahman	Project Manager
Priyanka Saha	Senior Developer

Project Overview

This report documents the design and implementation of a Library Management System (LMS) developed as part of our Software Design and Specification course. The system was designed using UML modeling techniques and implements SOLID principles to create a modular, maintainable architecture.

Contents

Chapter 1

Introduction

1.1 Purpose

The purpose of this report is to explore and demonstrate the application of software design and specification principles through the development of a Library Management System (LMS). While the system itself provides practical functionality for managing users, books, and lending workflows, the focus of this report lies in how established design methodologies—such as UML modeling and SOLID principles—can be systematically applied to ensure a modular, scalable, and maintainable architecture.

This report serves as a case study to reinforce and extend the concepts covered in the Software Design and Specification course. It documents the system's requirements, object-oriented design, and role-based behavior using standard specification tools like use case diagrams, class diagrams, and sequence diagrams. Furthermore, the report highlights how design practices influence implementation choices, aiming to provide a structured reference for students, developers, and educators interested in the practical application of software design theories.

1.2 Scope

The Library Management System (LMS) is designed to automate traditional library operations, including book tracking, user management, and lending processes. This system addresses inefficiencies in manual library workflows by providing role-based access (Library Admin and Member) with features like digital book uploads, automated fine calculations, and real-time inventory updates.

Objectives:

- Streamline book borrowing/returning processes.
- Enable secure user/role management for admins.

- Provide a scalable digital catalog with multimedia support.

1.3 Definitions, Acronyms, and Abbreviations

LMS	Library Management System.
Admin	Library Administrator.
Member	Registered User of the Library.
ISBN	International Standard Book Number.

Chapter 2

Literature Review

Modern library systems have evolved significantly with the adoption of digital platforms. Traditional library systems often lacked real-time availability, user engagement, or robust access control mechanisms. Recent systems emphasize the use of design patterns, object-oriented principles, and clear architectural models to address these limitations [1].

UML is widely adopted for software specification due to its visual modeling capabilities [2]. Class diagrams, sequence diagrams, and use case diagrams are used to capture both static and dynamic aspects of systems. SOLID principles—introduced by Robert C. Martin—help ensure that software components are modular, maintainable, and extensible [3]. In web-based LMS solutions, RESTful APIs are now the standard for scalable client-server communication [4].

Chapter 3

Methodology

3.1 Design Approach

This project follows an object-oriented design (OOD) methodology supported by structured modeling and specification practices. The design process was incremental, starting with identifying key stakeholders and their interactions with the system. These were then captured in user stories, which helped guide the creation of design models, including class and sequence diagrams. The primary aim was to create a maintainable and extendable system by adhering to key software design principles.

3.2 Requirement Analysis and Modeling

User stories were developed to clearly capture the functional requirements of the system from the perspective of its two primary roles: Library Admin and Member. Each user story was structured with a narrative and acceptance criteria to ensure testable and traceable requirements.

These user stories formed the foundation for identifying system entities, relationships, and use cases, which were later formalized through UML diagrams. For example, the story "As a Member, I want to borrow and return books" led to the creation of the Loan class and borrowing sequence flow.

3.3 UML Modeling

UML (Unified Modeling Language) diagrams were used to visualize the system's design:

- **Class Diagram:** Modeled core entities (User, Book, Loan) and their relationships. Encapsulated responsibilities and fields (e.g., dueDate, totalCopies, role) based on user stories.

- **Sequence Diagram:** Illustrated dynamic interactions such as the borrowing process, showing how a member requests a loan and how the system updates availability and tracks loan metadata.

Modeling tools used: planttext and PlantUML.

3.4 Application of SOLID Principles

To ensure the system was modular and maintainable, the following SOLID principles were applied:

- **Single Responsibility Principle:** Each class or module (e.g., UserService, BookController) handles one well-defined task.
- **Open/Closed Principle:** The design supports future extensions like book reservations or admin reports without modifying core classes.
- **Liskov Substitution Principle:** The system differentiates user behavior by role while preserving polymorphism (e.g., Admin can borrow and manage books).
- **Interface Segregation Principle:** User functionalities are separated, so each role only sees the operations relevant to them (e.g., Members cannot see user management features).
- **Dependency Inversion Principle:** Although not heavily used in this small-scale system, abstractions are maintained where possible, e.g., in separating business logic from routing.

3.5 Technology Stack and Architecture

The system was designed as a 3-tier architecture consisting of:

- **Frontend:** React.js SPA that consumes backend APIs and supports role-based UI rendering.
- **Backend:** Node.js with Express.js to implement RESTful APIs handling authentication, CRUD operations, and transactions.
- **Database:** MongoDB to store collections for users, books, and loans, with support for flexible schema design.

Authentication is implemented using JWT, with middleware for role-based access control to protect sensitive endpoints.

3.6 From Requirements to Implementation

Each user story informed the modeling and design decisions:

- **User & Role Management:** Led to the creation of a User model with role distinction and access control logic.
- **Book Management:** Translated into CRUD APIs for Book objects, along with file/image handling features.
- **Lending Workflow:** Modeled using a Loan entity and sequence diagram. Fine calculation and return status were implemented in business logic based on dates.

This structured approach ensured that the transition from specification to implementation remained aligned with design principles and supported future extensibility.

Chapter 4

Design Artifacts

4.1 User Stories

The system was designed around two core user roles: Library Admin and Member. Each role has distinct responsibilities and access to specific system features. Functional requirements were captured through structured user stories with corresponding acceptance criteria. These guided the development of UML diagrams and code components.

Role	Description
Library Admin	Manages users, roles, books, and lending transactions.
Member	Registered library user who can browse, borrow, return, and reserve books.

Key User Stories

1. Library Admin – User & Role Management

Story
As a Library Admin, I want to register users and assign roles so that they can access appropriate features based on their responsibilities.

Acceptance Criteria

- Admin can create new users with username, password, and role.
- Admin can view all users.
- Admin can update roles (e.g., promote a Member to Admin).
- Admin can delete users.
- Access is strictly role-based.

2. Library Admin – Book Management

Story
As a Library Admin, I want to add, update, and delete books including images and files so that the catalog remains accurate and informative.

Acceptance Criteria

- Admin can add books with fields like title, ISBN, author, genre, totalCopies.
- Admin can upload cover images and digital content (e.g., PDFs).
- Admin can edit or delete book entries.
- Optional fields (e.g., image) are handled gracefully.
- Books are listed via API.

3. Member & Library Admin – Borrow & Return Books

Story
As a Member, I want to borrow and return books to enjoy library content and avoid overdue penalties.
As a Library Admin, I want to view all lending records and manage returns for users.

Acceptance Criteria

- Members can borrow available books.
- Members can return books.
- The system tracks issueDate, dueDate, and returnDate, and calculates fines.
- Members can view loan history.
- Admin can view all transactions and return books for users.

4. Member – Reservation System (Planned Feature)

Story
As a Member, I want to reserve checked-out books and get notified when they become available.

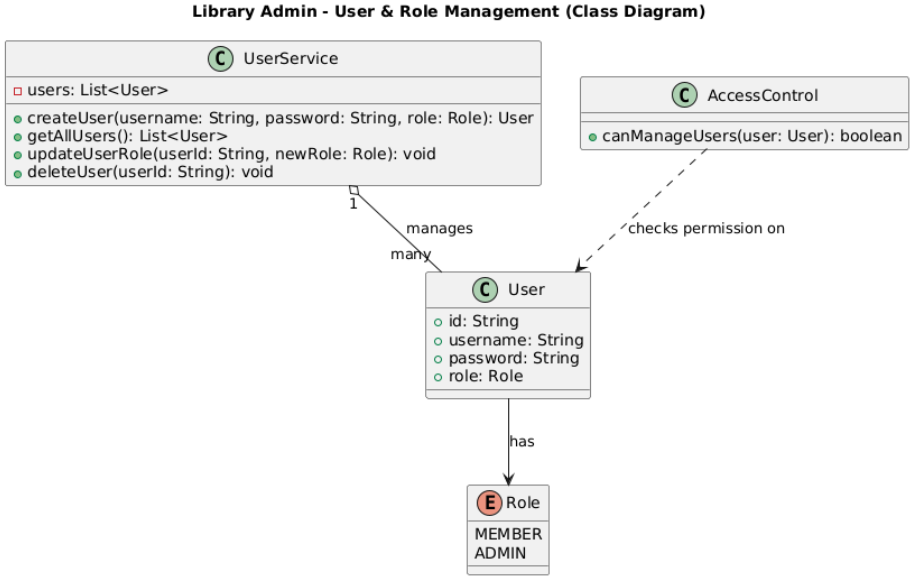
Acceptance Criteria

- Members can reserve unavailable books.
- The system notifies members when books are available.
- Reservation auto-expires after a pickup deadline.
- Admin can view and manage reservation queues.

4.2 Class Diagrams

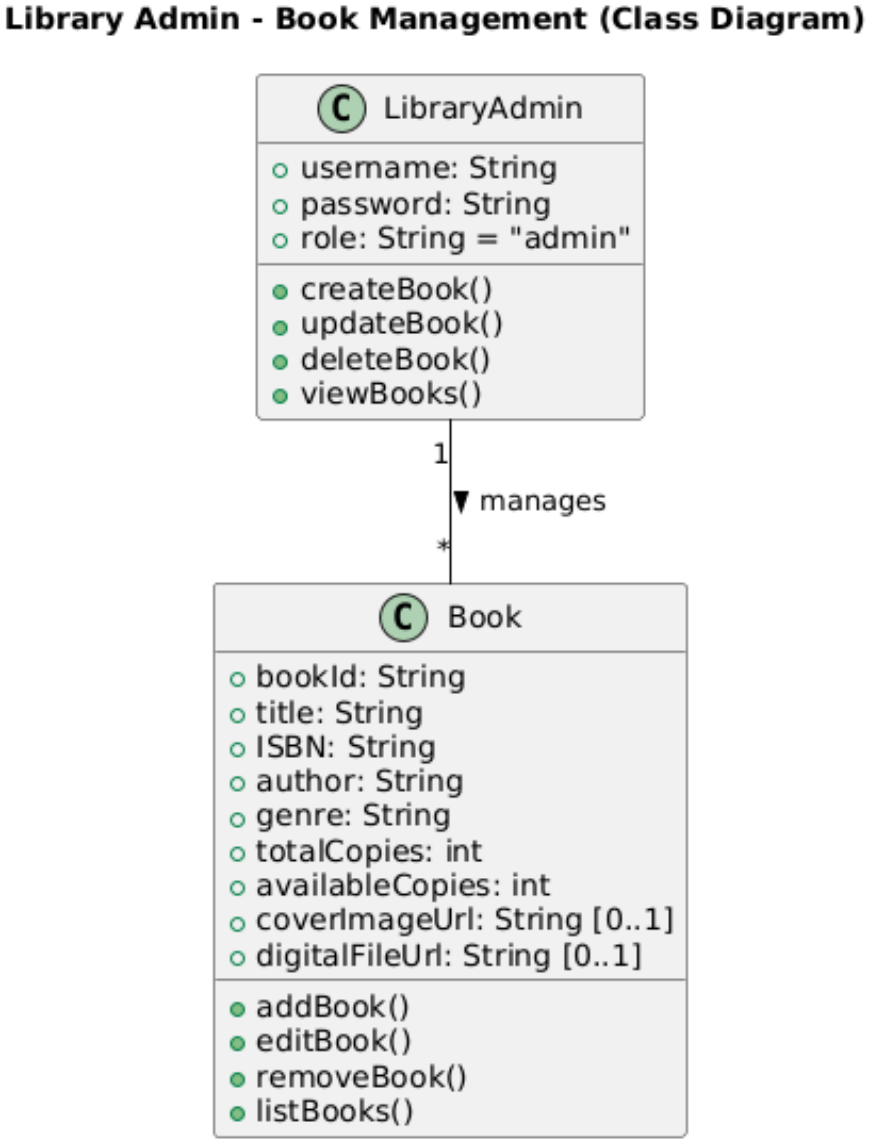
4.2.1 Library Admin – User & Role Management (Based on User story 1)

The class diagram models the relationships between key entities involved in the management of users and their roles within the Library Management System. It captures how the Admin creates and manages users, assigns roles, and enforces access control. Each user has associated credentials and a role that governs their permissions. The diagram highlights the role-based design, ensuring that only users with administrative privileges can perform operations like updating roles or deleting accounts.



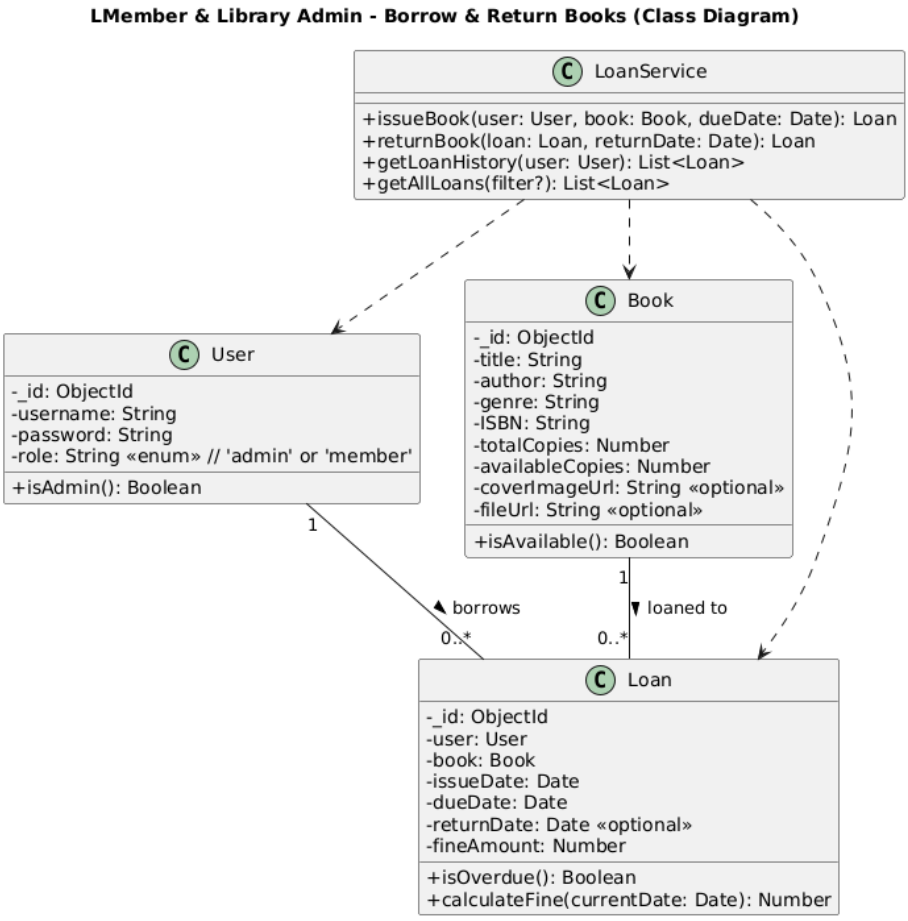
4.2.2 Library Admin – Book Management (Based on User story 2)

The class diagram illustrates the structure and relationships of entities involved in managing the book catalog within the Library Management System. It reflects how a LibraryAdmin interacts with Book entities to perform CRUD operations—such as adding, updating, or deleting book records.



4.2.3 Member & Library Admin – Borrow & Return Books (Based on User story 3)

The class diagram models the relationships between key entities involved in the borrowing and returning of books, including how loans are tracked, who borrows books, and how books are managed.



Explanation of Key Classes

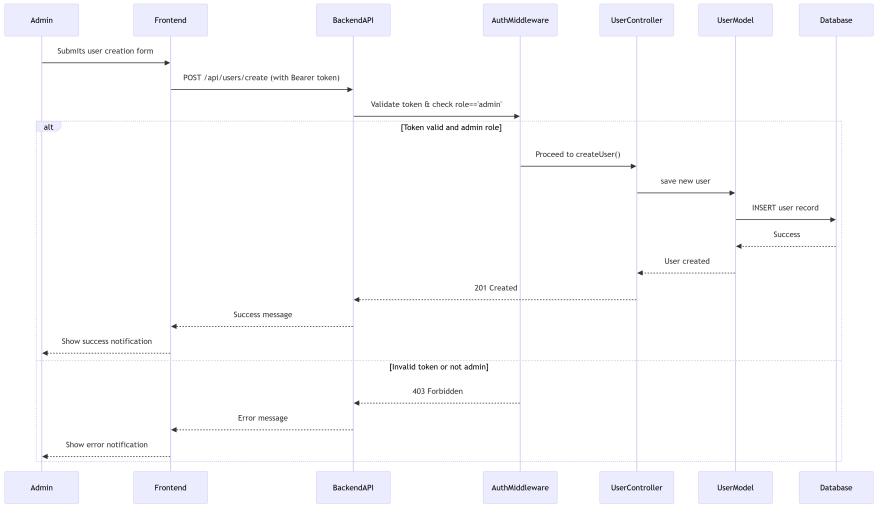
- **User:** Holds basic user info. Role-based access (admin/member) is derived from the role attribute.
- **Book:** Represents a catalog item. availableCopies is reduced/increased on borrow/return.
- **Loan:** Records the borrowing transaction. Includes timestamps and fine logic.
- **LoanService:** Business logic layer for issuing, returning, and fetching loan records.

Relationships in the Class Diagram

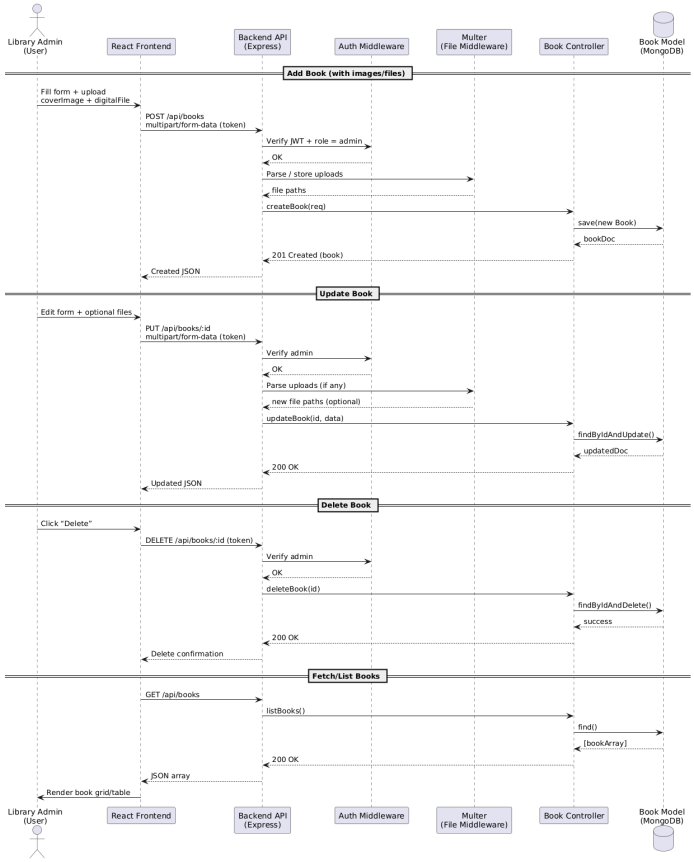
1. **User Loan**
    - Type: One-to-Many (1 — 0..*)
    - Meaning: A single user (usually a Member) can have multiple loans, but each loan is associated with exactly one user.
    - Direction: User → Loan: A user can borrow multiple books over time.
    - Example: User Alice borrows Book A, Book B → two Loan records belong to Alice.
2. **Book Loan**
    - Type: One-to-Many (1 — 0..*)
    - Meaning: A single book can be borrowed many times, so it can be associated with multiple loan records, but each loan corresponds to one book only.
    - Direction: Book → Loan: A book can have many lending records.
    - Example: Book 1984 is borrowed 10 times by different users → 10 Loan entries reference the same book.
3. **LoanService Loan, Book, User**
    - Type: Dependency (uses association)
    - Meaning: The LoanService class depends on the Loan, User, and Book classes to perform its operations (like issuing, returning, fetching loan records).
    - Direction: LoanService → Loan, User, Book
    - Explanation:
        - When issuing a book: LoanService checks the Book for availability, associates it with a User, creates a new Loan record.
        - When returning: LoanService finds the Loan, updates returnDate, and recalculates available copies of the Book.

4.3 Sequence Diagrams

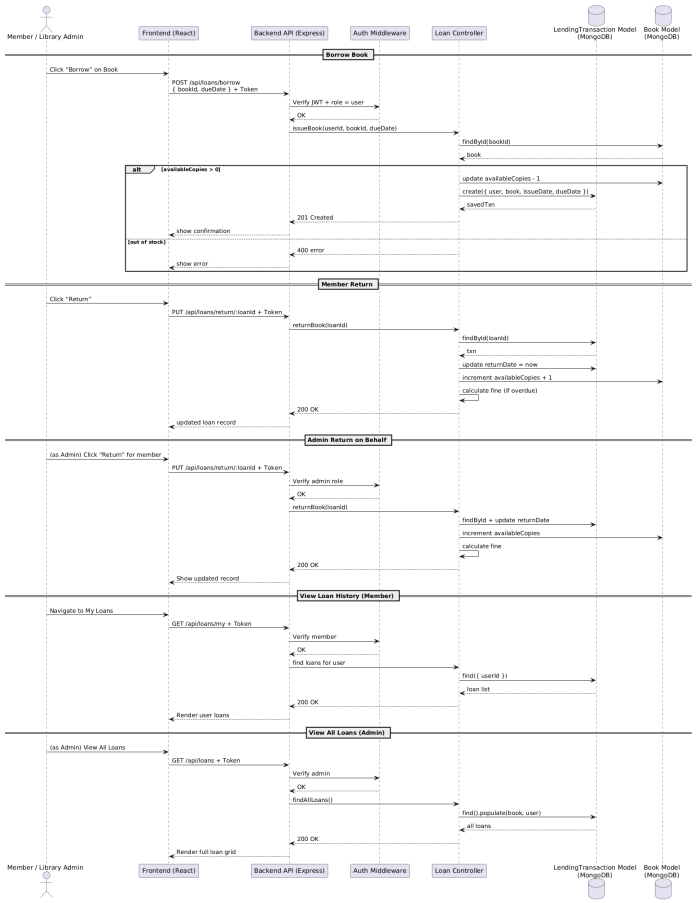
4.3.1 Library Admin – User & Role Management (Based on User story 1)



4.3.2 Library Admin – Book Management (Based on User story 2)



4.3.3 Member & Library Admin – Borrow & Return Books (Based on User story 3)



Explanation of the Borrow & Return Books Sequence Diagram

This sequence diagram illustrates the interactions between users (Member and Library Admin), the frontend, backend API, services (Loan and Book), and the database in the context of borrowing and returning books in the Library Management System.

Borrow Book Flow

1. **Initiation:** The Member initiates the borrow action by clicking a "Borrow" button in the frontend interface on an available book.

2. **Frontend Request:** The frontend sends a POST /loans API request to the backend, including the bookId and userId.

3. **Book Availability Check:** The API delegates to the BookService to check if the book is available. It queries the database for the book's total and available copies. If the book is available (i.e., availableCopies ¿ 0), the flow proceeds.

4. **Loan Creation:** The API calls LoanService to create a loan record with issueDate and dueDate. The LoanService inserts the loan into the database.

5. **Update Book Copies:** After creating the loan, the BookService decrements the book's availableCopies by 1 in the database.

6. **Response:** The API returns a success message and loan details to the frontend, which updates the UI.

Return Book Flow

1. **Initiation:** The Member clicks the "Return" button on a currently borrowed book.

2. **Frontend Request:** The frontend sends a PUT /loans/return/:loanId API request to the backend.

3. **Return Handling:** The LoanService updates the returnDate of the loan to today's date. It also calculates a fine if the return is overdue (returnDate ¿ dueDate).

4. **Restore Book Copies:** The BookService increments availableCopies by 1 for the returned book. The updated value is stored in the database.

5. **Response:** The API responds with confirmation and fine details (if applicable), and the frontend updates the UI.

Admin View Lending Records

1. **View Request:** The Library Admin opens the "Lending Records" section in the UI.

2. **Fetch Records:** The frontend calls GET /loans with optional filters (e.g., by user, status).

3. **Backend Aggregation:** The API performs a join query to fetch loans along with user and book data.

4. **Display:** The results are returned to the frontend and displayed in a table or dashboard for admin review.

Access Control & Roles

- Members can only borrow and return their own books.
- Library Admins can return books on behalf of users and access all loan data.

Chapter 5

Implementation Support for Design Artifacts

This section demonstrates how the implemented Library Management System aligns with the design artifacts presented earlier, including user stories, class diagrams, and sequence diagrams. It validates that the implementation supports the intended functionalities and design principles outlined in the methodology. The implementation uses the MERN stack (MongoDB, Express.js, React, Node.js) and follows a layered architectural pattern to separate concerns effectively.

5.1 Alignment with Design

The Library Management System was implemented to reflect the structure and behavior specified in the design phase. Core system features such as user registration, role-based access control, book management, and lending operations directly map to the components defined in the class and sequence diagrams.

- Each class in the UML diagram corresponds to a module in the codebase (e.g., User, Book, LendingTransaction).
- The sequence of interactions in the borrow/return workflows are implemented through coordinated API routes and service logic.
- Access control was implemented using JWT middleware, reflecting the responsibility separation defined in user stories and role modeling.

5.2 Technology Stack

The following technologies were used to implement the system:

- **Frontend:** React (with React Router and Axios)

- **Backend:** Express.js (REST API)
- **Database:** MongoDB
- **File Upload:** Multer
- **Authentication:** JWT-based token system
- **Tools:** VS Code, GitHub, Postman, PlantUML

This stack enables modular development, RESTful communication, and scalable data handling—supporting the separation of concerns emphasized in the system design.

5.3 Project Structure and Layered Architecture

The system follows a layered architecture which maps well to the separation of responsibilities modeled in the class diagrams:

- **Presentation Layer:** React components
- **Business Logic Layer:** Express controllers and services
- **Data Access Layer:** MongoDB models and Mongoose queries.

This structure is in alignment with the class diagram's modular representation of User, Book, LendingTransaction.

5.4 Support for SOLID Principles

The Library Management System (LMS) backend and frontend were developed with adherence to the SOLID design principles, which are essential for writing clean, maintainable, and scalable software.

5.4.1 Single Responsibility Principle (SRP)

Each module or class should have one, and only one, reason to change.

Implementation:

- The backend models such as Book.js, User.js, and LendingTransaction.js in the /models directory each handle schema definitions and data interactions for specific entities only.
- The authController.js manages only authentication logic, separated from loan or book handling.
- Middleware components like authMiddleware.js and upload.js are split by responsibility (authentication vs. file handling).

```
exports.login = async (req, res) => {
  const { username, password } = req.body;
  try {
    const user = await User.findOne({ username });
    if (!user) return res.status(400).json({ error: 'User not found' });

    const valid = await bcrypt.compare(password, user.password);
    if (!valid) return res.status(401).json({ error: 'Invalid credentials' });

    // □ Update lastLogin on successful login
    user.lastLogin = new Date();
    await user.save();

    const token = jwt.sign(
      { id: user._id, username: user.username, role: user.role }, // □ include role
      process.env.JWT_SECRET,
      { expiresIn: '1h' }
    );
    console.log('token:', token);

    res.json({ token });
  } catch (err) {
    res.status(500).json({ error: err.message });
  }
};
```

Figure 5.1: Evidence of SRP

```
// routes/bookRoutes.js
router.post('/add', addBook);
router.get('/', getAllBooks);
// New routes can be added without modifying existing ones
```

Figure 5.2: Evidence of OCP

- On the frontend, pages like Login.js, UserBooks.js, and AdminLoans.js are clearly divided by responsibility, making the interface easy to understand and maintain.

5.4.2 Open/Closed Principle (OCP)

Software components should be open for extension but closed for modification.
Implementation:

- Route files like bookRoutes.js, auth.routes.js, and loanRoutes.js are organized in a way that allows adding new endpoints without altering the existing ones.
- Frontend layouts such as AdminLayout.js and UserLayout.js enable extending the UI with additional pages or roles without changing core layout logic.

5.4.3 Liskov Substitution Principle (LSP)

This usage supports the Liskov Substitution Principle (LSP) — which states that subclasses or derived types must be substitutable for their base types with-

out affecting the correctness of the program.

Implementation: In the React frontend, RequireUser expects a child component that behaves like a valid layout. Whether we pass in ¡UserLayout /¿, ¡AdminLayout /¿, or any other layout component, the component hierarchy remains stable and functional — proving that each layout can be substituted without altering the behavior of RequireUser.

```
<Route
  path="/dashboard"
  element={
    <RequireUser>
      <UserLayout />
    </RequireUser>
  }
/>
```

Evidence of LSP

5.4.4 Interface Segregation Principle (ISP)

Clients should not be forced to depend on interfaces they do not use.

Implementation:

- API routes are segregated (auth.routes.js, bookRoutes.js, loanRoutes.js) such that each route file exposes only the relevant endpoints.
- Frontend pages (e.g., Books.js, Users.js) only use the API methods they need.
- Reusable components avoid bundling unnecessary props or logic, keeping the interface clean.

5.4.5 Dependency Inversion Principle (DIP)

High-level modules should not depend on low-level modules. Both should depend on abstractions.

Implementation:

- Books.js is a high-level module (UI logic).
- bookroutes.js (Express route handlers) is a low-level module (data storage/API logic).
- API from ../api is abstraction layer (likely Axios or a wrapper), which:
  - Hides low-level fetch or axios logic.
  - Enables Books.js to just call API.get(”/books”), etc.

```
const res = await API.get('/books');
```

Evidence of DIP

Here, the frontend depends on an abstract interface (API) rather than directly accessing backend route internals (e.g., hardcoded fetch('/api/books')).

5.4.6 Solid Principles Summary

This project follows the SOLID design principles throughout both backend and frontend development to ensure modular, extensible, and maintainable code. Each principle is supported with detailed examples in the respective report sections, along with real code implementations.

To aid in verification and deeper understanding, all referenced code (e.g., Books.js, bookroutes.js, authController.js, App.js, etc.) is publicly available on GitHub: `https://github.com/priyankaaust/LMS`

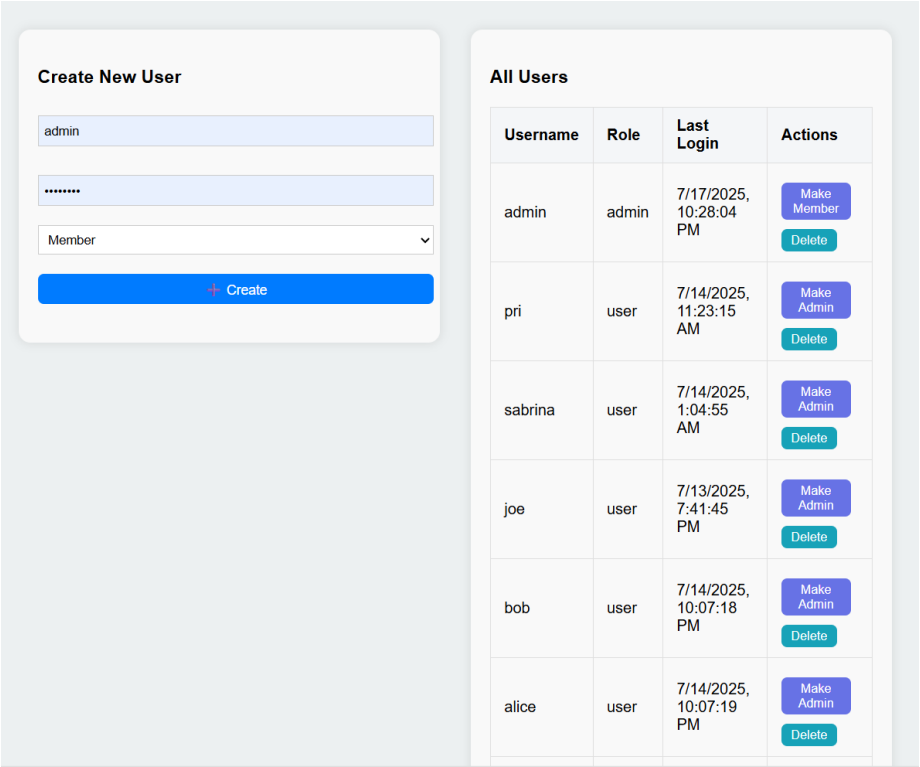
Chapter 6

Mapping Code to Design Artifacts

The following mappings show how key design models (user stories, class diagrams, sequence diagrams) are reflected in the system's functionality.

6.1 User Management

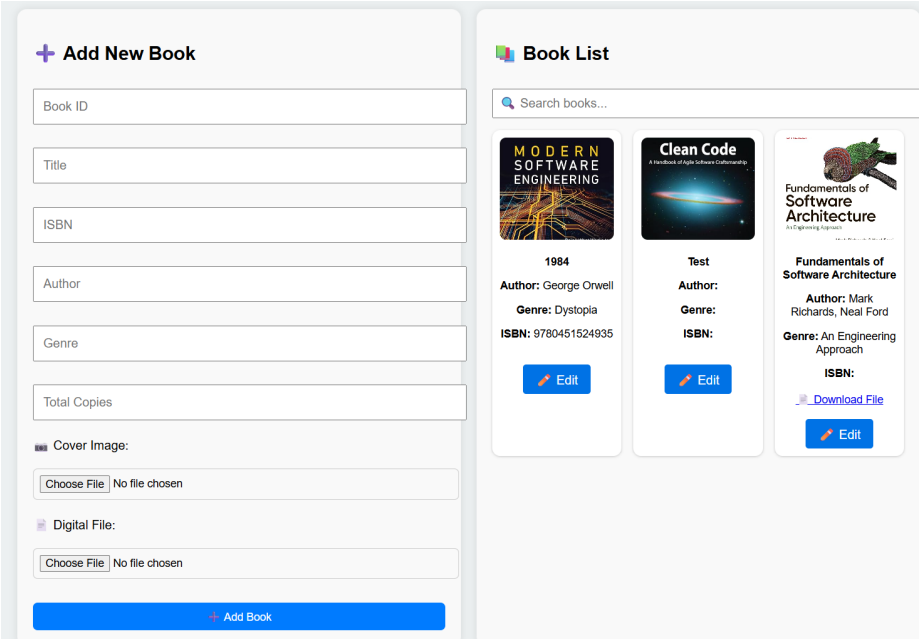
- **Related User Story:** Admin – User & Role Management
- **Class Diagram:** User, Role
- **Sequence Diagram:** Register User
- **Screenshot:** Admin UI for creating/editing users
- **Code:** userController.js, User.js (model)
- **Implements:** user creation, role assignment, view/delete/update



User Role Management

6.2 Book Management

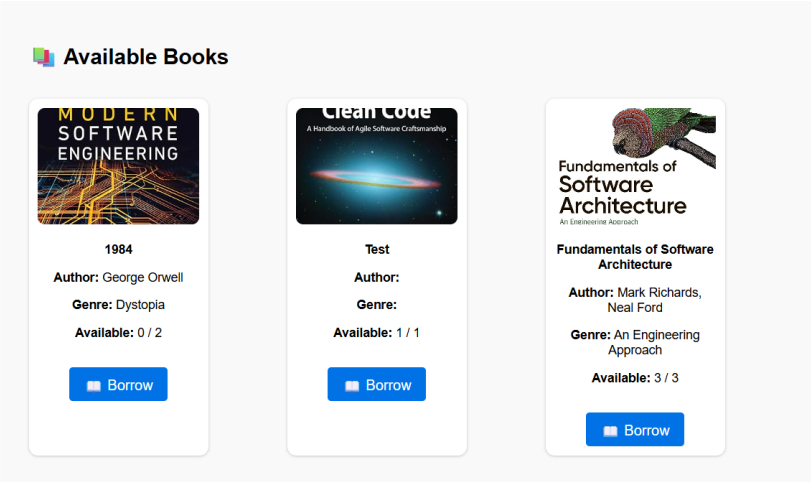
- **Related User Story:** Admin – Book Management
- **Class Diagram:** Book
- **Sequence Diagram:** Add Book
- **Screenshot:** Book form interface with image upload
- **Code:** bookController.js, Book.js, Multer middleware
- **Implements:** CRUD operations on books, file uploads, validations



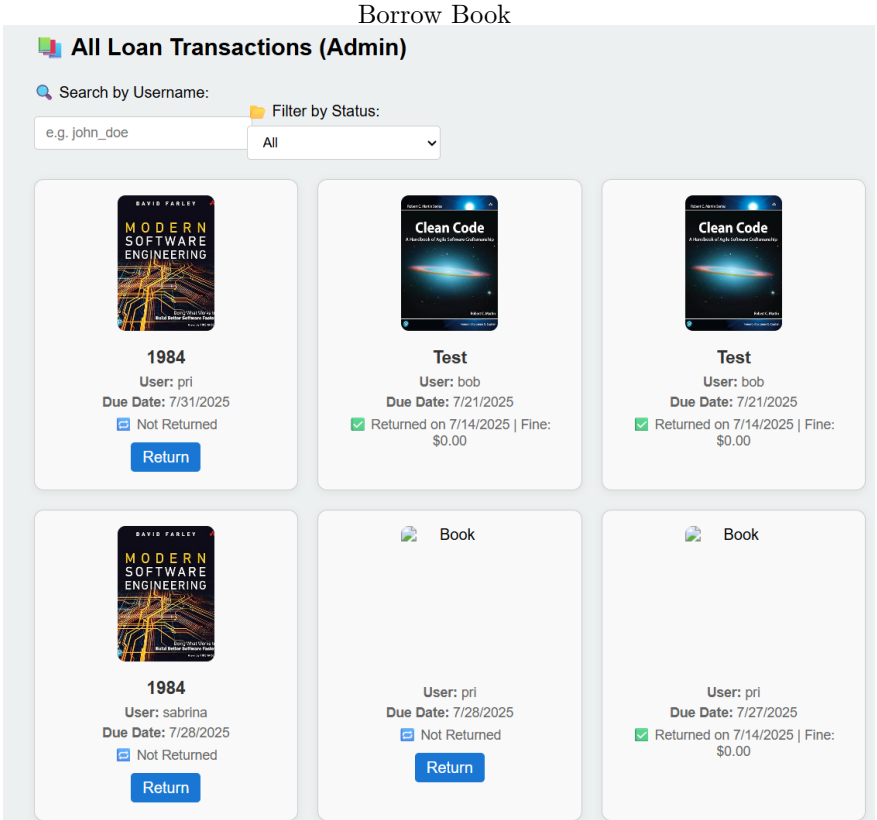
Enter Caption

6.3 Borrow & Return

- **Related User Story:** Member/Admin – Borrow & Return
- **Class Diagram:** Loan, Book, User
- **Sequence Diagram:** Borrow Book, Return Book
- **Screenshot:** Borrow button and current loan view for members
- **Code:** loanController.js, Loan.js (model), calculateFine.js utility
- **Implements:** track issueDate, dueDate, returnDate, calculate fine



Borrow Book



Return Book

Chapter 7

Limitations

While the Library Management System (LMS) implementation supports the designed architecture and fulfills key user stories, several limitations were encountered during development and research:

7.1 System Limitations

- **Simplified Authentication:** The system currently uses JWT-based authentication without multi-factor support. In a real-world deployment, stronger security measures such as OAuth2 or MFA should be implemented.
- **Basic UI/UX Design:** The frontend interface was kept minimal to focus on functionality. It lacks accessibility features and responsive design optimizations for various devices.
- **No Real-time Notifications:** Features such as reservation alerts or due-date reminders are not implemented with real-time notifications (e.g., via WebSocket or email). This limits the user experience for time-sensitive operations.
- **Limited Admin Control:** Admin functionality is limited to user and book management. More advanced controls such as analytics dashboards, batch uploads, or report generation are not included.
- **No Fine Payment Integration:** While the system calculates overdue fines, there is no integration with payment gateways or tracking for fine payment.
- **Single Language Support:** The system interface is in English only. Multilingual support would improve usability for diverse users.

7.2 References Supporting Research and Documentation Limitations

7.2.1 Limited Access to Paid Journals

Suggested mention in text: Some detailed works, including those discussing formal verification with Alloy and industrial adoption of the Z notation, were published in paid-access venues like ACM Digital Library or IEEE Xplore.

Suggested Reference: [5] [6] [7]

7.2.2 Incomplete Feature Implementation

The project timeline prioritized the implementation of core functionalities—such as user authentication, book management, and lending workflows—to ensure a stable and functional Minimum Viable Product (MVP). However, due to scope and time constraints, several advanced features were intentionally deferred (User story 4). These include:

- Real-time notifications (e.g., due date alerts, return reminders)
- Fine payment and penalty management
- Multilingual interface support

Implementing these features would have required integrating third-party services (e.g., notification APIs, payment gateways) or additional architectural layers, which were outside the scope of the current design and resource plan. Their exclusion was a deliberate decision to maintain system quality, reduce complexity, and ensure delivery of a robust MVP.

7.2.3 Tool Constraints

Suggested mention in text: Tools like Enterprise Architect and Papyrus support automatic code generation and deep traceability but were not used due to their commercial nature and learning curve.

Suggested Reference: [8] [9] [10]

Chapter 8

Conclusion

This report presented the design and specification of a Library Management System (LMS) as a case study to explore and apply foundational principles in software engineering. Through the use of user stories, UML diagrams (class and sequence), and structured methodology grounded in object-oriented practices and SOLID principles, the project demonstrated how theoretical concepts translate into maintainable, modular, and extensible software systems.

The implementation closely followed the design artifacts, with evidence from code snippets and user interface screenshots that confirm the traceability of system behavior from design to execution. While the system operates successfully on a small scale, limitations in scalability testing, tool usage, and access to advanced academic resources were noted.

Overall, the project reinforces the practical relevance of software design and specification techniques in real-world system development. It also highlights the importance of adhering to design patterns and modular architecture to enhance future maintainability and evolution of the system. Future work may include scaling the system, integrating advanced reservation features, and experimenting with formal specification methods to further validate and extend the system's design integrity.

Bibliography

[1] A. N. Chavan and M. P. Khedkar, "Web based library management system," *International Journal of Advanced Research in Computer Science and Software Engineering*, vol. 3, no. 9, 2013.

[2] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*, 2nd ed. Addison-Wesley, 2005.

[3] R. C. Martin, *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, 2002.

[4] L. Richardson and S. Ruby, *RESTful Web Services*. O'Reilly Media, 2007.

[5] J. M. Spivey, *The Z Notation: A Reference Manual*, 2nd ed. Prentice Hall, 1992, Commonly used reference for Z notation; full access often behind institutional paywalls.

[6] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*, Revised. MIT Press, 2012, Core reference for Alloy, often only partially accessible online.

[7] M. Broy and K. Stølen, *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Springer, 2001, Used in advanced design verification studies; many chapters behind SpringerLink paywall.

[8] J. Arlow and I. Neustadt, *UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design*, 2nd ed. Addison-Wesley, 2005.

[9] Object Management Group, *Omg unified modeling language (uml), version 2.5.1*, 2017. [Online]. Available: `https://www.omg.org/spec/UML/2.5.1/` (visited on 11/20/2023).

[10] PlantUML, *Plantuml: Open-source tool for uml modeling*, n.d. [Online]. Available: `https://plantuml.com` (visited on 11/20/2023).