

Software Design & Specification (ENGI-9874-001)

Professor : Raja Abbas

Library Management System (LMS)

Presented by

Priyanka Saha (ID: 202387432)

Sifat Sabrina Rahman (ID: 202286725)

Introduction

- Demonstrates software design principles using a Library Management System (LMS) as a case study.
- Focuses on applying UML modeling and SOLID principles for a modular, scalable, and maintainable system.
- Documents system requirements, object-oriented design, and role-based behavior.
- Aims to guide learners and developers in applying design theories to real-world systems.
- **Scope:** Automates library operations like book tracking and lending with role-based access.

Objectives

- Streamline borrowing/returning processes.
- Enable secure user and role management.
- Provide a scalable digital catalog with multimedia support

Methodology

- **Design Approach:**

Followed an object-oriented design (OOD) methodology using incremental modeling based on user stories.

- **Requirement Analysis:**

Functional requirements were captured through structured user stories for Admin and Member roles, ensuring traceability and testability.

- **UML Modeling:**

- **Class Diagrams:** Modeled core entities (User, Book, Loan) and their relationships.
- **Sequence Diagrams:** Visualized dynamic interactions like borrowing and returning books.
- **Tools Used:** PlantUML and PlantText.

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.

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.

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.

Key User Stories

- **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.

Key User Stories

- **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.

Key User Stories

- ****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**

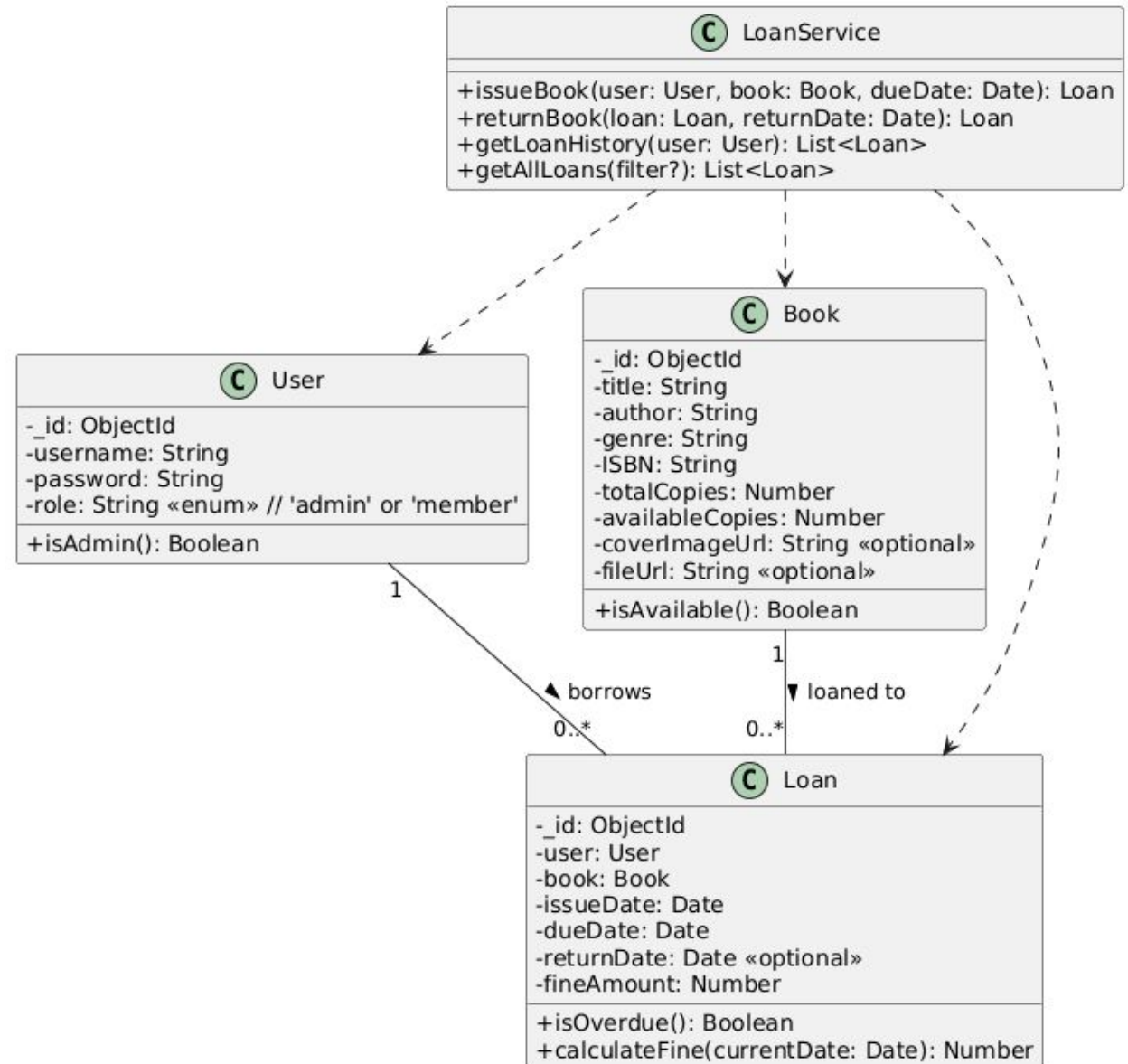
- Members can reserve unavailable books.
- The system notifies members when books are available.
- Reservation auto-expires after a pickup deadline.
- Admin can view and manage reservation queues.

Class Diagram

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.

LMember & Library Admin - Borrow & Return Books (Class Diagram)



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. available Copies is reduced/increased on borrow/return.
- **Loan:** Records the borrowing transaction. Includes timestamps and fine logic.
- **Loan Service:** Business logic layer for issuing, returning, and fetching loan records.

Sequence Diagram



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.

System Implementation Overview

- Built with MERN Stack (MongoDB, Express.js, React, Node.js)
- Follows layered architecture for separation of concerns
- Aligns with design artifacts:
 - User Stories
 - Class Diagrams
 - Sequence Diagrams

Alignment with Design

- 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.

Technology Stack

- Frontend: React (Router, Axios)
- Backend: Express.js (REST APIs)
- Database: MongoDB (Mongoose)
- Authentication: JWT Token System
- File Upload: Multer
- Tools: Postman, VS Code, GitHub, PlantUML

Layered Architecture

- Presentation Layer: React components
- Business Logic Layer: Express controllers & services
- Data Layer: Mongoose models
- Structure reflects modular class design (User, Book, LendingTransaction)

SOLID Principles in Practice

- SRP: Separate concerns by module (auth, loan, upload, etc.)
- OCP: Easily extend layout/routes without breaking logic
- LSP: Layouts interchangeable in RequireUser wrapper
- ISP: Clean interfaces, only required API methods used
- DIP: Frontend depends on abstract API interface, not raw logic

Single Responsibility Principle

```
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 });
  }
};
```

Open/Closed Principle (OCP)

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.

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

Liskov Substitution Principle (LSP)

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>
  }
/>
```

Interface Segregation Principle (ISP)

- 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.

Dependency Inversion Principle (DIP)

- `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.

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

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

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>

Mapping Code to Design Artifacts

User Management

- **Related User Story:** Admin – User & Role Management

Create New User

admin

.....

Member

▼

+ Create

All Users

Username	Role	Last Login	Actions
admin	admin	7/17/2025, 10:28:04 PM	<div>Make Member</div> <div>Delete</div>
pri	user	7/14/2025, 11:23:15 AM	<div>Make Admin</div> <div>Delete</div>
sabrina	user	7/14/2025, 1:04:55 AM	<div>Make Admin</div> <div>Delete</div>
joe	user	7/13/2025, 7:41:45 PM	<div>Make Admin</div> <div>Delete</div>
bob	user	7/14/2025, 10:07:18 PM	<div>Make Admin</div> <div>Delete</div>
alice	user	7/14/2025, 10:07:19 PM	<div>Make Admin</div> <div>Delete</div>

Book Management

Related User Story: Admin – Book Management

+ Add New Book

Book ID


Title

ISBN


Author

Genre

Total Copies

 Cover Image:

Choose File No file chosen


 Digital File:

Choose File No file chosen

+ Add Book


Book List

Search books...




1984
Author: George Orwell
Genre: Dystopia
ISBN: 9780451524935

Edit



Clean Code
A Handbook of Agile Software Craftsmanship
Test
Author:
Genre:
ISBN:

Edit



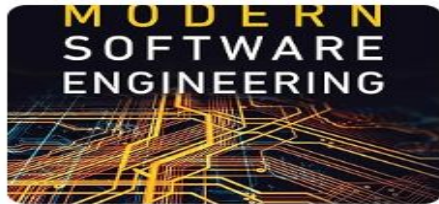
Fundamentals of Software Architecture
An Engineering Approach
Author: Mark Richards, Neal Ford
Genre: An Engineering Approach
ISBN:
[Download File](#)

Edit

Borrow & Return

Related User Story: Member/Admin – Borrow & Return

Available Books



1984

Author: George Orwell

Genre: Dystopia

Available: 0 / 2

 Borrow



Clean Code

Author:

Genre:

Available: 1 / 1

 Borrow



**Fundamentals of
Software
Architecture**

An Engineering Approach

**Fundamentals of Software
Architecture**


Author: Mark Richards,
Neal Ford


Genre: An Engineering
Approach


Available: 3 / 3

 Borrow

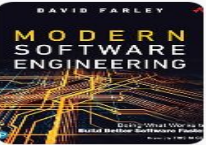





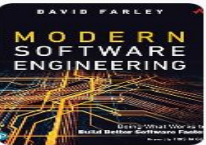





Return From Admin

 **All Loan Transactions (Admin)**

 Search by Username:

 Filter by Status:

All

<div></div> <div>1984 User: pri Due Date: 7/31/2025  Not Returned <div>Return</div></div>	<div></div> <div>Test User: bob Due Date: 7/21/2025  Returned on 7/14/2025 Fine: \$0.00</div>	<div></div> <div>Test User: bob Due Date: 7/21/2025  Returned on 7/14/2025 Fine: \$0.00</div>
<div></div> <div>1984 User: sabrina Due Date: 7/28/2025  Not Returned <div>Return</div></div>	<div> Book</div> <div>User: pri Due Date: 7/28/2025  Not Returned <div>Return</div></div>	<div> Book</div> <div>User: pri Due Date: 7/27/2025  Returned on 7/14/2025 Fine: \$0.00</div>

Limitation

Limited Access to Paid Journals

- 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.

Tool Constraints

- 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.

Incomplete Feature Implementation

- The project timeline prioritized core functionalities like user management, book handling, and lending workflows, leaving limited time for advanced features. To maintain quality and stability, the focus remained on delivering a functional MVP (Minimum Viable Product) rather than overextending the feature set. Certain features (e.g., real-time notifications, fine payment integration, multilingual support) required additional tools, APIs, or infrastructure that were beyond the current project setup.

Thank you !