Library Management System (OOP – Java)

Table of Contents

2. Introduction

The rapid development of software systems has increased the need for well-structured and maintainable applications. One of the most effective approaches to building such systems is **Object-Oriented Programming (OOP)**, which helps in organizing code, improving reusability, and simplifying maintenance.

This project, **Library Management System**, is developed as part of the Object-Oriented Programming course. The main objective of the project is to design and implement a simple yet functional library system that demonstrates the practical use of OOP concepts using the Java programming language.

The system is implemented as a **console-based application**, focusing on clarity, correctness, and proper software design rather than graphical user interfaces. Through this project, fundamental software engineering practices such as modular design, testing, and version control are applied.

3. Project Overview

The Library Management System is designed to manage basic library operations in an organized and efficient manner. It allows the user to manage books, library members, and borrowing transactions while tracking book availability and borrowing history.

The system supports multiple core functionalities, including adding and removing books, searching for books by title or author, borrowing and returning books, displaying currently borrowed books, and calculating late fees based on the borrowing duration. Different types of members are supported to demonstrate inheritance and polymorphism within the system.

This project emphasizes clean code structure, proper use of object-oriented principles, and real-world problem modeling. It also incorporates unit testing using JUnit and project management using GitHub tools to simulate a real software development workflow.

4. System Requirements

This section describes the functional and non-functional requirements of the Library Management System. These requirements define what the system should do and the constraints under which it operates.

4.1 Functional Requirements

The Library Management System is designed to support the following functional requirements:

- The system shall allow the user to add new books to the library.
- The system shall allow the user to remove existing books from the library.
- The system shall allow searching for books by title.
- The system shall allow searching for books by author.
- The system shall allow library members to borrow available books.
- The system shall allow library members to return borrowed books.
- The system shall track the availability status of each book.
- The system shall display a list of currently borrowed books.
- The system shall calculate late fees based on the borrowing duration.
- The system shall support different types of members with different fee calculations.

4.2 Non-Functional Requirements

The non-functional requirements define the quality and constraints of the system:

- The system shall be implemented using the Java programming language.
- The system shall follow Object-Oriented Programming principles.
- The system shall be console-based for simplicity and clarity.
- The system shall be modular and easy to maintain.
- The system shall include unit testing using JUnit.
- The system shall use Git and GitHub for version control.
- The system shall include UML diagrams for system design documentation.

5. Object-Oriented Design

This project is designed following the fundamental principles of **Object-Oriented Programming (OOP)**. These principles help in building a modular, reusable, and maintainable software system. The Library Management System applies the core OOP concepts as described below.

5.1 Encapsulation

Encapsulation is the practice of hiding internal data and allowing access to it only through well-defined methods. In this project, encapsulation is applied by declaring class attributes as private and providing public getter and setter methods to access and modify them.

For example, the `Book` class encapsulates its data such as book ID, title, author, and availability status. Direct access to these attributes is restricted, ensuring better data protection and control.

5.2 Inheritance

Inheritance allows a class to acquire properties and behaviors from another class, promoting code reusability. In this project, inheritance is demonstrated through the `StudentMember` class, which extends the `Member` class.

The `StudentMember` class inherits common attributes and methods from the `Member` class while providing its own implementation for fee calculation. This reduces code duplication and allows easy extension of member types in the future.

5.3 Polymorphism

Polymorphism enables a single method to behave differently based on the object that invokes it. This concept is implemented in the project through method overriding.

The `calculateFee()` method is defined in the `Member` class and overridden in the `StudentMember` class. When the method is called using a `Member` reference, the actual method executed depends on the object type at runtime. This behavior allows flexible and dynamic fee calculation based on the member type.

5.4 Interfaces

Interfaces are used to define a contract that classes must follow. In this project, the `Searchable` interface defines methods for searching books by title and author.

The `LibraryManager` class implements the `Searchable` interface and provides concrete implementations for the search operations. This design improves flexibility and supports separation of concerns by decoupling search behavior from class implementation details.

6. Class Description

This section provides a detailed description of the main classes used in the Library Management System, including their responsibilities, key attributes, and important methods.

6.1 Book Class

The `Book` class represents a book in the library. It stores essential information about each book and tracks its availability status.

Main Attributes:

- `id`: Unique identifier for the book
- `title`: Title of the book
- `author`: Author of the book
- `available`: Indicates whether the book is available for borrowing

Main Methods:

- Getter and setter methods for accessing and updating book data
- `isAvailable()`: Checks if the book is available
- `setAvailable()`: Updates the availability status

```
package library_system_;
public class Book {

    private int id;
    private String title;
    private String author;
    private boolean available;

    public Book(int id, String title, String author) {
```

```
        this.id = id;
        this.title = title;
        this.author = author;
        this.available = true; // book is available by default
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getAuthor() {
        return author;
    }

    public void setAuthor(String author) {
        this.author = author;
    }

    public boolean isAvailable() {
        return available;
    }

    public void setAvailable(boolean available) {
        this.available = available;
    }

    @Override
    public String toString() {
        return "Book ID: " + id +
                ", Title: " + title +
                ", Author: " + author +
                ", Available: " + (available ? "Yes" : "No");
    }
}
```

6.2 Member Class

The `Member` class represents a library member. It serves as a base class for different types of members and provides a general fee calculation method.

Main Attributes:

- `memberId`: Unique identifier for the member
- `name`: Name of the member

Main Methods:

- `calculateFee(int lateDays)`: Calculates late fees for borrowed books
- Getter and setter methods for member information

```
public class Member {

    protected int memberId;
    protected String name;
    public Member(int memberId, String name) {
        this.memberId = memberId;
        this.name = name;
    }
    public int getMemberId() {
        return memberId;
    }
    public void setMemberId(int memberId) {
        this.memberId = memberId;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public double calculateFee(int lateDays) {
        return lateDays * 2.0; // default fee per day
    }
    @Override
    public String toString() {
        return "Member ID: " + memberId +
                ", Name: " + name;
    }
```

6.3 StudentMember Class

The `StudentMember` class is a specialized type of library member that extends the `Member` class. It demonstrates inheritance and polymorphism.

Main Attributes:

- Inherits all attributes from `Member`

Main Methods:

- Overrides `calculateFee(int lateDays)` to apply a reduced fee for students

```
package library_system_;
public class StudentMember extends Member {

    public StudentMember(int memberId, String name) {
        super(memberId, name);
    }
    @Override
    public double calculateFee(int lateDays) {
        return lateDays * 1.0; // students pay less
    }

    @Override
    public String toString() {
        return "Student Member -> ID: " + memberId +
                ", Name: " + name;
    }
```

6.4 Loan Class

The `Loan` class represents the borrowing process of a book. It connects a book with a member and tracks borrowing and returning dates.

Main Attributes:

- `book`: The borrowed book
- `member`: The member who borrowed the book
- `borrowDate`: Date when the book was borrowed
- `returnDate`: Date when the book was returned

Main Methods:

- `returnBook()`: Marks the book as returned
- `isReturned()`: Checks whether the book has been returned
- `getBorrowedDays()`: Calculates the number of days the book has been borrowed

```
package library_system_;

import java.time.LocalDate;
import java.time.temporal.ChronoUnit;
public class Loan {

    private Book book;
    private Member member;
    private LocalDate borrowDate;
    private LocalDate returnDate;

    public Loan(Book book, Member member) {
        this.book = book;
        this.member = member;
        this.borrowDate = LocalDate.now();
        this.returnDate = null;
    }

    public Book getBook() {
        return book;
    }

    public Member getMember() {
        return member;
    }

    public LocalDate getBorrowDate() {
```

```
    return borrowDate;
}

public LocalDate getReturnDate() {
    return returnDate;
}


public void returnBook() {
    this.returnDate = LocalDate.now();
    book.setAvailable(true);
}

public boolean isReturned() {
    return returnDate != null;
}

public long getBorrowedDays() {
    return ChronoUnit.DAYS.between(borrowDate, LocalDate.now());
}

public void setBorrowDate(LocalDate borrowDate) {
    this.borrowDate = borrowDate;
}


@Override
public String toString() {
    return "Loan -> Book: " + book.getTitle() +
           ", Member: " + member.getName() +
           ", Borrowed on: " + borrowDate +
           ", Returned: " + (returnDate != null ? returnDate : "Not yet");
}
```

6.5 LibraryManager Class

The `LibraryManager` class acts as the core controller of the system. It manages books, loans, and library operations.

Main Attributes:

- `books`: A list of all books in the library
- `loans`: A list of all borrowing transactions

Main Methods:

- `addBook()`: Adds a new book to the library
- `removeBook()`: Removes a book from the library
- `borrowBook()`: Handles book borrowing operations
- `returnBook()`: Handles book return operations
- `displayBooks()`: Displays all books
- `displayBorrowedBooks()`: Displays currently borrowed books
- `calculateLateFee()`: Calculates late fees based on borrowing duration

```
package library_system_;

import java.util.ArrayList;
import java.util.List;
public class LibraryManager implements Searchable {

    private List<Book> books;
     List<Loan> loans;


    public LibraryManager() {
        books = new ArrayList<>();
        loans = new ArrayList<>();
    }

    // Add a new book
    public void addBook(Book book) {
        books.add(book);
    }

    public boolean removeBook(int bookId) {
        return books.removeIf(book -> book.getId() == bookId);
    }
```

```
    @Override
    public Book searchByTitle(String title) {
        for (Book book : books) {
            if (book.getTitle().equalsIgnoreCase(title)) {
                return book;
            }
        }
        return null;
    }

    // Search by author
    @Override
    public Book searchByAuthor(String author) {
        for (Book book : books) {
            if (book.getAuthor().equalsIgnoreCase(author)) {
                return book;
            }
        }
        return null;
    }

    // Borrow a book
    public boolean borrowBook(int bookId, Member member) {
        for (Book book : books) {
            if (book.getId() == bookId && book.isAvailable()) {
                book.setAvailable(false);
                loans.add(new Loan(book, member));
                return true;
            }
        }
        return false;
    }

    // Return a book
    public boolean returnBook(int bookId) {
        for (Loan loan : loans) {
            if (loan.getBook().getId() == bookId && !loan.isReturned()) {
                loan.returnBook();
                return true;
            }
        }
        return false;
    }

    // Display all books
    public void displayBooks() {
```

```
        for (Book book : books) {
            System.out.println(book);
        }
    }
    // Display all borrowed books

    public void displayBorrowedBooks() {
        boolean found = false;

        for (Loan loan : loans) {
            if (!loan.isReturned()) {
                System.out.println(
                    "Book: " + loan.getBook().getTitle() +
                    " | Borrowed by: " + loan.getMember().getName() +
                    " | Borrow date: " + loan.getBorrowDate()
                );
                found = true;
            }
        }

        if (!found) {
            System.out.println("No borrowed books at the moment.");
        }
    }

    public double calculateLateFee(int bookId) {
        for (Loan loan : loans) {
            if (loan.getBook().getId() == bookId && !loan.isReturned()) {
                long days = loan.getBorrowedDays();
                if (days > 7) { // سماح أسبوع
                    return loan.getMember().calculateFee((int)(days - 7));
                }
            }
        }
        return 0.0;
    }
```

6.6 Searchable Interface

The `Searchable` interface defines the search behavior for the system.

Declared Methods:

- `searchByTitle(String title)`
- `searchByAuthor(String author)`

This interface is implemented by the `LibraryManager` class to provide search functionality.

```
package library_system_;
public interface Searchable {
    Book searchByTitle(String title);
    Book searchByAuthor(String author);
}
```

6.7 Main Class

The `Main` class contains the entry point of the application. It provides a console-based user interface that allows users to interact with the system.

Responsibilities:

- Displaying menu options
- Handling user input
- Calling appropriate methods from `LibraryManager`

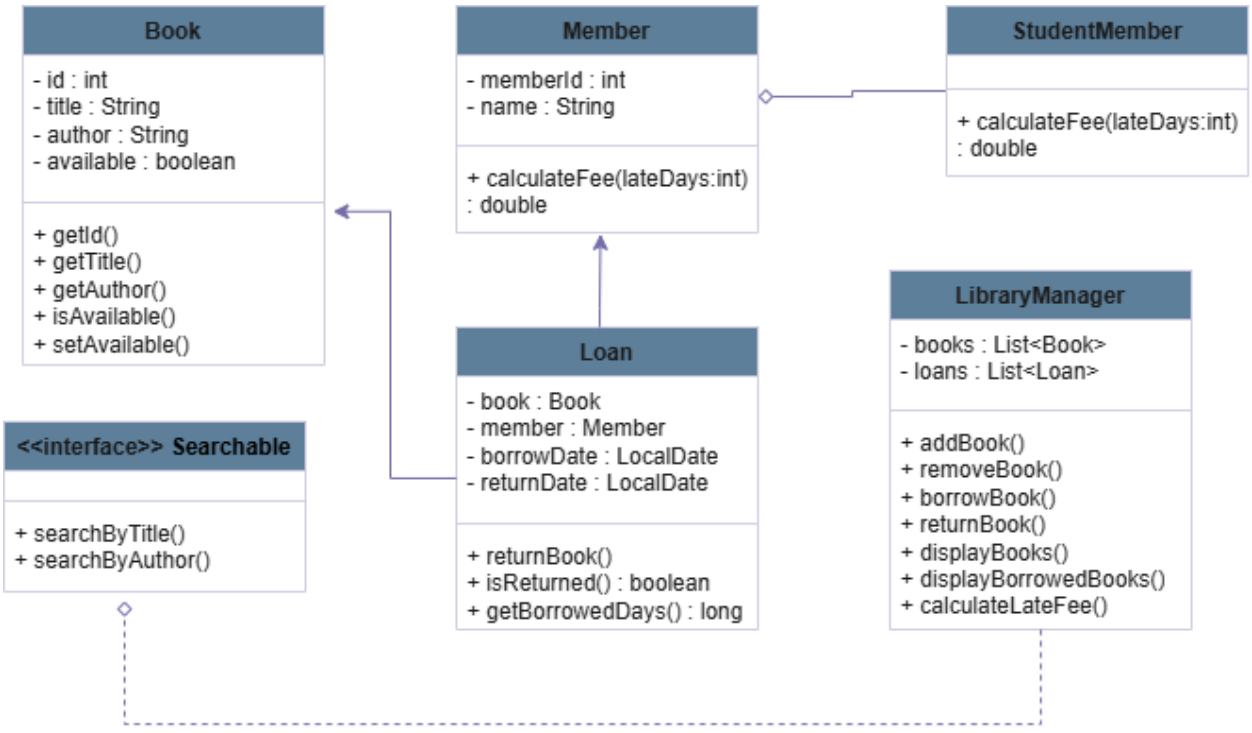
```
package library_system_
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        LibraryManager library = new LibraryManager();
        Scanner scanner = new Scanner(System.in);
        library.addBook(new Book(1, "Clean Code", "Robert Martin"));
        library.addBook(new Book(2, "Effective Java", "Joshua Bloch"));
        Member member = new Member(101, "Ahmed");
        StudentMember student = new StudentMember(102, "Sara");

        int choice;
```

```
        do {
            System.out.println("\n=== Library Management System ===");
            System.out.println("1. Display all books");
            System.out.println("2. Borrow book");
            System.out.println("3. Return book");
            System.out.println("4. Display borrowed books");
            System.out.println("0. Exit");
            System.out.print("Enter your choice: ");

            choice = scanner.nextInt();

            switch (choice) {
                case 1:
                    library.displayBooks();
                    break;

                case 2:
                    System.out.print("Enter book ID to borrow: ");
                    int borrowId = scanner.nextInt();
                    boolean borrowed = library.borrowBook(borrowId, student);
                    System.out.println(borrowed ? "Book borrowed successfully." : "Book not
available.");
                    break;

                case 3:
                    System.out.print("Enter book ID to return: ");
                    int returnId = scanner.nextInt();
                    boolean returned = library.returnBook(returnId);
                    System.out.println(returned ? "Book returned successfully." : "Return failed.");
                    break;
                case 4:
                    library.displayBorrowedBooks();
                    break;
                case 0:
                    System.out.println("Exiting system...");
                    break;
                default:
                    System.out.println("Invalid choice!");
            }
        } while (choice != 0);
        scanner.close();
    }
}
```

7. UML Diagrams

Unified Modeling Language (UML) diagrams are used in this project to visually represent the system structure and user interactions. UML helps in understanding the design of the system before and during implementation.

7.1 Class Diagram



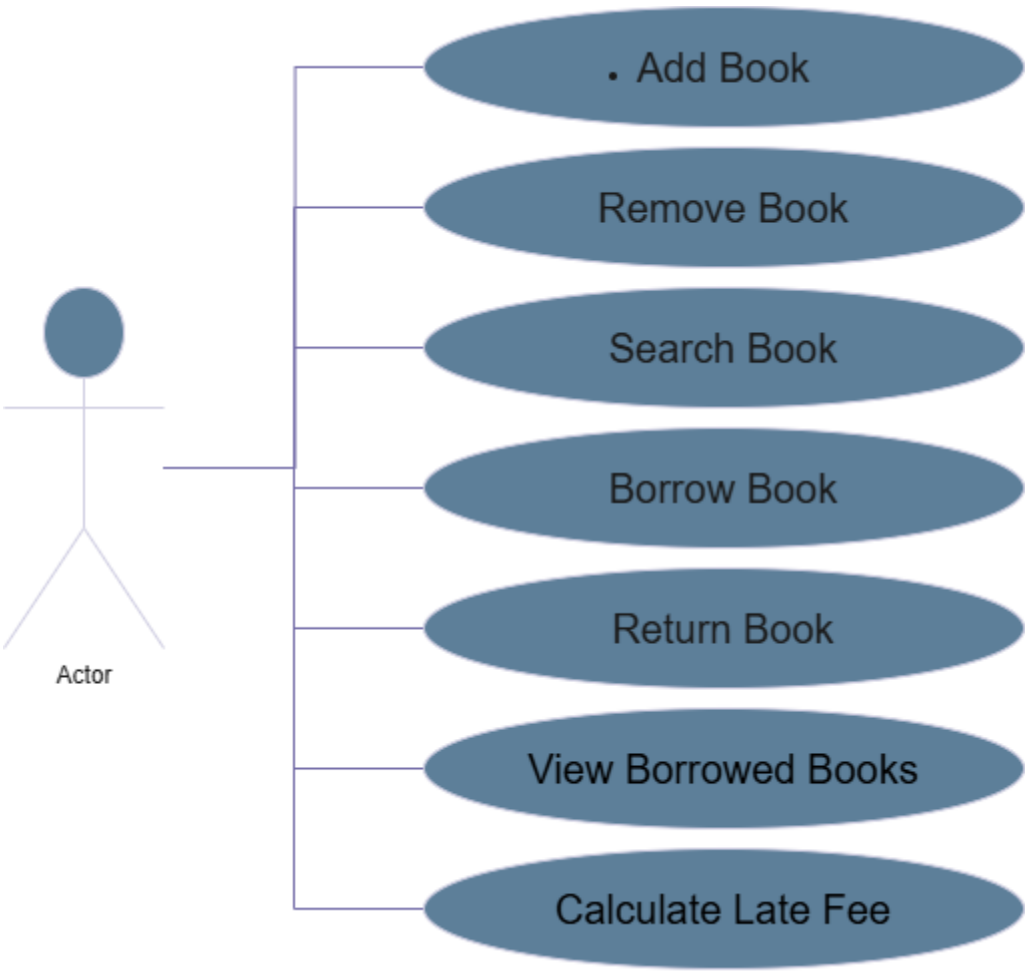
The Class Diagram illustrates the static structure of the Library Management System. It shows the main classes, their attributes, methods, and the relationships between them.

The diagram includes the following key relationships:

- **Inheritance** between the `Member` and `StudentMember` classes.
- **Interface implementation**, where the `LibraryManager` class implements the `Searchable` interface.
- **Association relationships** between the `Loan` class and both `Book` and `Member` classes.
- **Aggregation** within the `LibraryManager` class, which manages collections of `Book` and `Loan` objects.

The Class Diagram provides a clear overview of how the system components are structured and how they interact with each other at the class level.

7.2 Use Case Diagram



The Use Case Diagram represents the functional behavior of the system from the user's perspective. It illustrates the interactions between the system and the user.

In this diagram, the primary actor is the **Library User**, who can perform several actions such as:

- Adding and removing books
- Searching for books
- Borrowing and returning books
- Viewing borrowed books
- Calculating late fees

The Use Case Diagram helps in identifying the system requirements and ensures that all required functionalities are supported by the system.

8. Unit Testing (JUnit)

Unit testing is an essential part of software development as it ensures that individual components of the system work correctly and as expected. In this project, **JUnit 5** is used to test the core logic of the Library Management System.

The purpose of unit testing in this project is to validate class behavior, verify business logic, handle edge cases, and ensure system reliability. Each test focuses on a specific class or functionality to maintain test clarity and isolation.

8.1 Book Testing

Unit tests were written for the `Book` class to verify:

- Correct object creation
- Proper initialization of book attributes
- Default availability status
- Correct behavior of getter and setter methods

These tests ensure that book objects are created and managed correctly within the system.

```
package library_system_;

import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;
public class BookTest {
    @Test
    public void testBookCreation() {
        Book book = new Book(1, "Clean Code", "Robert Martin");

        assertEquals(1, book.getId());
        assertEquals("Clean Code", book.getTitle());
        assertEquals("Robert Martin", book.getAuthor());
        assertTrue(book.isAvailable());
    }
    @Test
    public void testBookAvailabilityChange() {
        Book book = new Book(2, "Effective Java", "Joshua Bloch");
        book.setAvailable(false);
        assertFalse(book.isAvailable());
        book.setAvailable(true);
        assertTrue(book.isAvailable());
```

8.2 Member and StudentMember Testing

The `Member` and `StudentMember` classes were tested to validate fee calculation logic. These tests demonstrate:

- Correct fee calculation for regular members
- Reduced fee calculation for student members
- Proper implementation of polymorphism through method overriding

The tests confirm that the system dynamically selects the correct fee calculation method based on the member type.

```
package library_system_;
public class StudentMember extends Member {
    public StudentMember(int memberId, String name) {
        super(memberId, name);
    }
    @Override
    public double calculateFee(int lateDays) {
        return lateDays * 1.0; // students pay less
    }
    @Override
    public String toString() {
        return "Student Member -> ID: " + memberId +
                ", Name: " + name;
```

```
Import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;
public class MemberTest {
    @Test
    public void testMemberFeeCalculation() {
        Member member = new Member(1, "Ahmed");
        double fee = member.calculateFee(3); // 3 late days
        assertEquals(6.0, fee);
    }
    @Test
    public void testStudentMemberFeeCalculation() {
        Member student = new StudentMember(2, "Sara");
        double fee = student.calculateFee(3); // polymorphism
        assertEquals(3.0, fee);
    }
    @Test
    public void testPolymorphismBehavior() {
        Member member = new StudentMember(3, "Omar");
        assertEquals(2.0, member.calculateFee(2))
```

8.3 Loan Testing

The `Loan` class was tested to verify the borrowing process and date handling logic. The unit tests check:

- Correct loan creation
- Proper initialization of the borrow date
- Correct return behavior
- Accurate tracking of returned and non-returned loans
- Calculation of borrowed days based on date differences

These tests ensure that loan transactions and time-based logic function correctly.

```
package library_system_;

import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

import java.time.LocalDate;

public class LoanTest {

    @Test
    public void testLoanCreation() {
        Book book = new Book(1, "Clean Code", "Robert Martin");
        Member member = new Member(1, "Ahmed");

        Loan loan = new Loan(book, member);

        assertEquals(book, loan.getBook());
        assertEquals(member, loan.getMember());
        assertNotNull(loan.getBorrowDate());
        assertNull(loan.getReturnDate());
        assertFalse(loan.isReturned());
    }

    @Test
    public void testBookAvailabilityAfterBorrow() {
        Book book = new Book(2, "Effective Java", "Joshua Bloch");
        Member member = new Member(2, "Sara");

        Loan loan = new Loan(book, member);

        assertNotNull(loan.getBorrowDate());
    }
```

```
    @Test
    public void testReturnBook() {
        Book book = new Book(3, "Design Patterns", "GoF");
        Member member = new Member(3, "Omar");

        Loan loan = new Loan(book, member);

        loan.returnBook();

        assertTrue(loan.isReturned());
        assertNotNull(loan.getReturnDate());
        assertTrue(book.isAvailable());
    }

    @Test
    public void testBorrowDateIsToday() {
        Book book = new Book(4, "Refactoring", "Martin Fowler");
        Member member = new Member(4, "Lina");

        Loan loan = new Loan(book, member);

        assertEquals(LocalDate.now(), loan.getBorrowDate());
    }

    @Test
    public void testBorrowedDaysCalculation() throws InterruptedException {
        Book book = new Book(5, "Algorithms", "CLRS");
        Member member = new Member(5, "Mona");

        Loan loan = new Loan(book, member);

        long borrowedDays = loan.getBorrowedDays();

        assertEquals(0, borrowedDays);
```

8.4 LibraryManager Testing

Unit tests for the `LibraryManager` class focus on validating the core system operations, including:

- Adding books to the system
- Borrowing available books
- Preventing borrowing of unavailable books
- Returning borrowed books
- Updating book availability status

These tests verify the correctness of the main business logic of the system.

```
package library_system_;

import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class LibraryManagerTest {

    @Test
    public void testBorrowBook() {
        LibraryManager library = new LibraryManager();
        Book book = new Book(1, "Clean Code", "Robert Martin");
        Member member = new Member(1, "Ahmed");

        library.addBook(book);

        boolean borrowed = library.borrowBook(1, member);

        assertTrue(borrowed);
        assertFalse(book.isAvailable());
    }

    @Test
    public void testReturnBook() {
        LibraryManager library = new LibraryManager();
        Book book = new Book(2, "Effective Java", "Joshua Bloch");
        Member member = new Member(2, "Sara");

        library.addBook(book);
        library.borrowBook(2, member);

        boolean returned = library.returnBook(2);
```

```
        assertTrue(returned);
        assertTrue(book.isAvailable());
    }

    @Test
    public void testBorrowUnavailableBook() {
        LibraryManager library = new LibraryManager();
        Book book = new Book(3, "Design Patterns", "GoF");
        Member member = new Member(3, "Omar");

        library.addBook(book);
        library.borrowBook(3, member);

        // Try to borrow again
        boolean borrowedAgain = library.borrowBook(3, member);

        assertFalse(borrowedAgain);
```

8.5 Late Fee Calculation Testing

Additional tests were implemented to validate late fee calculation based on borrowing duration. These tests cover:

- Grace period handling
- Late fee calculation for regular members
- Late fee calculation for student members
- Edge cases where no late fee is applied

This ensures that fee calculation logic is accurate and consistent with the system rules.

```
package library_system_;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;
import java.time.LocalDate;
public class LateFeeTest {

    @Test
    public void testLateFeeForRegularMember() {
        LibraryManager library = new LibraryManager();
        Book book = new Book(10, "Databases", "Elmasri");
        Member member = new Member(100, "Ahmed");

        library.addBook(book);
        library.borrowBook(10, member);
        Loan loan = library.searchByTitle("Databases") != null
```

```
            ? library.loans.get(0)
            : null;

    assertNotNull(loan);
    loan.setBorrowDate(LocalDate.now().minusDays(10));

    double fee = library.calculateLateFee(10);
    assertEquals(6.0, fee);
}

@Test
public void testLateFeeForStudentMember() {
    LibraryManager library = new LibraryManager();
    Book book = new Book(11, "Operating Systems", "Tanenbaum");
    Member student = new StudentMember(101, "Sara");

    library.addBook(book);
    library.borrowBook(11, student);

    Loan loan = library.loans.get(0);
    loan.setBorrowDate(LocalDate.now().minusDays(9));

    double fee = library.calculateLateFee(11);


    assertEquals(2.0, fee);
}

@Test
public void testNoLateFeeWithinGracePeriod() {
    LibraryManager library = new LibraryManager();
    Book book = new Book(12, "Networks", "Kurose");
    Member member = new Member(102, "Omar");

    library.addBook(book);
    library.borrowBook(12, member);

    Loan loan = library.loans.get(0);
    loan.setBorrowDate(LocalDate.now().minusDays(5));

    double fee = library.calculateLateFee(12);

    assertEquals(0.0, fee);6 Test Execution Results
```

All unit tests were successfully executed using JUnit 5, and the results indicate that the system components behave as expected. Screenshots of successful test execution are included in the report to demonstrate test coverage and correctness.

9. Git & GitHub Usage

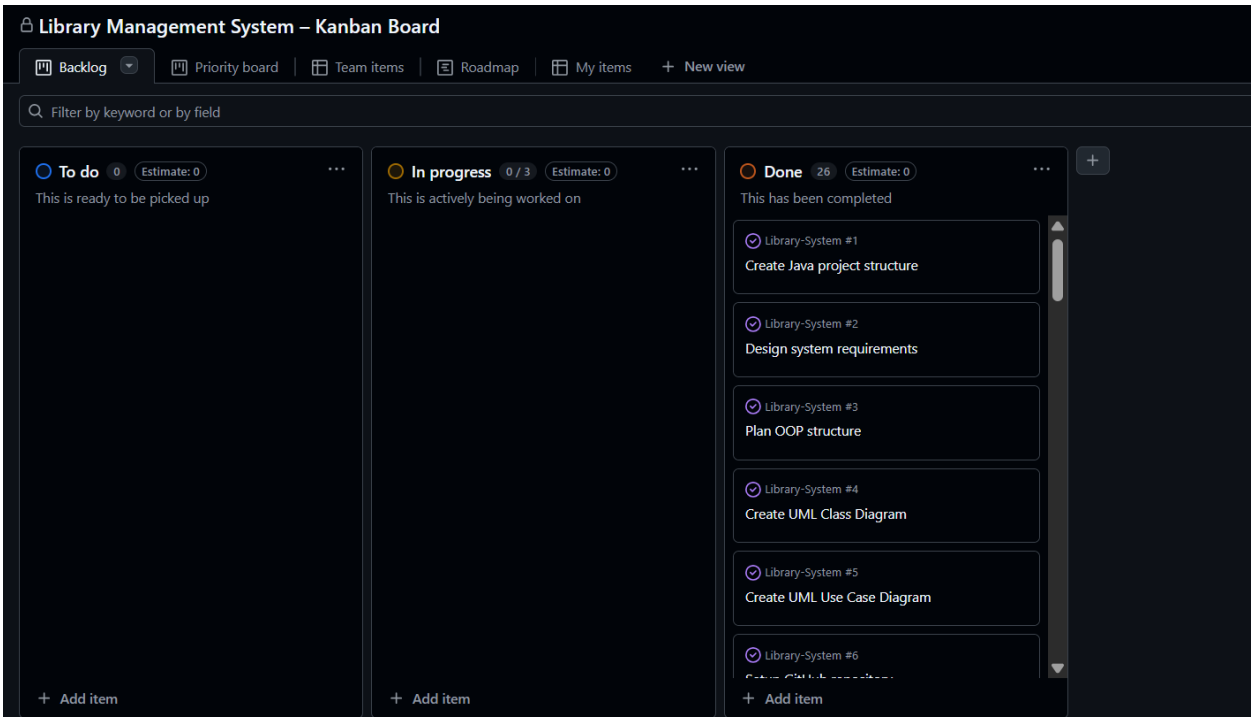
Version control is a critical aspect of modern software development. In this project, **Git** and **GitHub** were used to manage the source code, track changes, and maintain a clear development history.

The project repository was created on GitHub, and all development activities were committed incrementally throughout the project duration. This approach allowed better organization of work, easy tracking of changes, and safer experimentation during development.

Meaningful commit messages were used to clearly describe the purpose of each change. Commits were distributed across different stages of development, including project setup, class implementation, feature enhancements, testing, and documentation.

GitHub was also used as a collaboration and project management platform, providing transparency and traceability of the development process. Screenshots of the commit history are included in this report to demonstrate consistent and structured usage of version control.

10. Kanban Board (GitHub Projects)



To manage the development process of the Library Management System, a **Kanban Board** was created using **GitHub Projects**. The Kanban board was used to organize tasks, track progress, and visualize the different stages of the project.

The board was divided into three main columns:

- **To Do**: Contains tasks that were planned but not yet started.
- **In Progress**: Contains tasks that were actively being worked on.
- **Done**: Contains tasks that were completed successfully.

Each task in the project lifecycle, such as implementing classes, creating UML diagrams, writing unit tests, and preparing documentation, was represented as a card on the Kanban board. Tasks were moved across columns as work progressed, providing a clear overview of project status at any given time.

Using the Kanban board helped ensure structured task management, better time organization, and alignment with real-world software development practices. A screenshot of the Kanban board is included in this report to demonstrate task tracking and workflow management.

11. Demo Explanation

The project was demonstrated through a **console-based application** to showcase the core functionalities of the Library Management System. During the demo, the system was executed using the `Main` class, which provides a simple text-based menu for user interaction.

The demonstration included the following steps:

- Displaying the list of available books.
- Borrowing a book by providing its unique identifier.
- Updating and tracking book availability after borrowing.
- Displaying currently borrowed books along with their borrow dates.
- Returning a borrowed book and updating its availability status.
- Calculating late fees based on the borrowing duration and member type.

The demo successfully demonstrated how different system components interact with each other and how object-oriented principles are applied in a real execution scenario. The console-based interface ensured simplicity and clarity while effectively presenting the system behavior.

12. Conclusion

In this project, a **Library Management System** was designed and implemented using Java, with a strong focus on Object-Oriented Programming principles. The project provided practical experience in designing a modular system that models real-world scenarios such as book borrowing, member management, and fee calculation.

Through this project, key OOP concepts including encapsulation, inheritance, polymorphism, and interfaces were applied effectively. Additionally, modern software development practices such as unit testing with JUnit, version control using Git and GitHub, UML-based system design, and task management using a Kanban board were incorporated.

Overall, this project enhanced understanding of object-oriented design, testing methodologies, and structured software development workflows, making it a valuable learning experience.