

Table of Contents

Acknowledgement	3
Executive Summary	8
Introduction	9
Task 01: UML Design Solutions	10
1. UML Diagrams	10
1.1. Use case Diagram	11
1.2. Class Diagram.....	13
1.3. Sequence Diagram	17
1.3.1. Login Sequence	17
1.3.2. View Book details Sequence	18
1.3.3. Add new book Sequence	19
1.3.4. Search book by category Sequence	20
1.3.5. Creating cashier account Sequence.....	21
Task 02: System Development	22
2. Little Book Haven System Development	22
2.1. What is Object Oriented Programing	22
2.2. Key OOP Concepts applied in the System.....	23
2.2.1. Class	23
2.2.2. Object	24
2.2.3. Abstraction	25
2.2.4. Inheritance	27
2.2.5. Encapsulation.....	31
2.2.6. Polymorphism.....	33
2.3. Main Functionalities	34
2.3.1. User Login.....	34
2.3.2. Main Menu	36

2.3.3. View all Books	37
2.3.4. Search Book by Category.....	39
2.3.5. Add New Book	40
2.3.6. Create Cashier Accounts	42
Task 03: User Manual	44
3. Introduction.....	44
3.1. Loading Screen	44
3.2. Login Screen.....	45
3.3. Main Menu	46
3.3.1. Manager Dashboard	46
3.3.2. Cashier Dashboard	46
3.4. View Book List.....	47
3.5. Search Book by Category.....	48
3.6. Add New Book	49
3.6.1. Adding the book	50
3.7. Create Cashier Accounts	51
3.7.1. Creating new cashier account	52
Conclusion	53
References	54
Appendix	55

Table of Figures

Figure 1 Use-Case	11
Figure 2 Class Diagram.....	13
Figure 3 Login-Sequence	17
Figure 4 View all Books-Sequence.....	18
Figure 5 Add New Book-Sequence	19
Figure 6 Search-by-Category-Sequence.....	20
Figure 7 Create Cashier Accounts-Sequence	21
Figure 8 Book Class	23
Figure 9 AddNewBookFrame Class	24
Figure 10 Object Created from AddNewBookFrame	24
Figure 11 DBConnection Class	25
Figure 12 DBConnection Method called to establish connection	26
Figure 13 Objects created from Swing Classes	27
Figure 14 User Class	28
Figure 15 Cashier inherit from User	29
Figure 16 Manager inherit from Cashier.....	29
Figure 17 Inheritance applied in Progress bar	30
Figure 18 Encapsulation in User-class	31
Figure 19 Encapsulation in Book-class.....	32
Figure 20 Polymorphism applied in Progress bar.....	33
Figure 21 User-Login.....	35
Figure 22 Main-Menu	36
Figure 23 View-all-book-details	38
Figure 24 Search-book-by-category.....	39
Figure 25 Add-New-Book.....	41
Figure 26 Create-cashier-account	43
Figure 27 Loading Screen	44
Figure 28 Login Interface	45
Figure 29 Manager-Menu.....	46
Figure 30 Cashier-Menu.....	46
Figure 31 Books List.....	47
Figure 32 Filter Books.....	48
Figure 33 Add new book interface.....	49

Figure 34 Filling Book Form.....	50
Figure 35 Cashier-account-creation Interface	51
Figure 36 Filling New Cashier Details.....	52
Figure 37 Appendix A-Gantt Chart	55

Executive Summary

The “Little Book Haven” system represents a comprehensive automation solution for a newly established bookstore aiming to streamline its transaction processes and enhance customer service. This report outlines the design and development of an Object-Oriented Programming based application tailored for two user levels Cashier and Manager. Utilizing UML diagrams designed Use Case, Class, and Sequence. The system’s architecture was meticulously planned to ensure scalability and efficiency. The implementation leverages Java with Swing for the user interface developed in NetBeans IDE, and MySQL database integration using JDBC for data handling. The application incorporates core OOP principles like Object, Class, Abstraction, Inheritance, Encapsulation, and Polymorphism to deliver functionalities such as viewing book details, adding new books, searching by category, and creating cashier accounts. A detailed user manual supports the system’s deployment, ensuring accessibility for all users. This solution promises to improve operational efficiency and customer satisfaction at “The Little Book Haven”.

Introduction

“The Little Book Haven” a new bookstore located on a charming city side street, aims to meet growing customer demand by automating its transaction processes. This project presents a software system supporting two user roles Cashier and Manager each with distinct functionalities designed to enhance efficiency and service quality. The system design began with UML diagrams Use Case, Class, and Sequence to define requirements and interactions, serving as a blueprint for development. The system was then implemented in Java with graphical interfaces, and MySQL with JDBC for database management. By applying core OOP concepts, the solution ensures modularity, reusability, and maintainability. This report details the design process, system development, and the user manual to help users interact smoothly with the application.

Task 01: UML Design Solutions

1. UML Diagrams

Unified Modeling Language diagrams are standardized visual representations used to design, model and document software systems. They provide ways to visualize a system's architecture, components, relationships and behaviors in a clear and consistent manner. These diagrams help to understand, communicate and analyze structure and functionality of a system during its design and development stages (Fowler, 2004). In the context of "The Little Book Haven" bookstore management system, UML diagrams help in visualizing different user roles and their interactions with the systems, structure and organize classes and relationships using OOP concepts, understand the flow of operations, and to communicate design before coding to reduce future errors. For this project we'll focus on three UML diagrams.

1.1. Use case Diagram

A use case diagram is a visual representation that models the interactions between a system and its external entities called actors, to define the system's functional requirements. It shows what the system does from the perspective of its users, capturing specific behaviors and use cases that the system supports. The diagram uses simple notations, providing a high-level view of the system functionality without detailing its implementation (Booch, 2005).

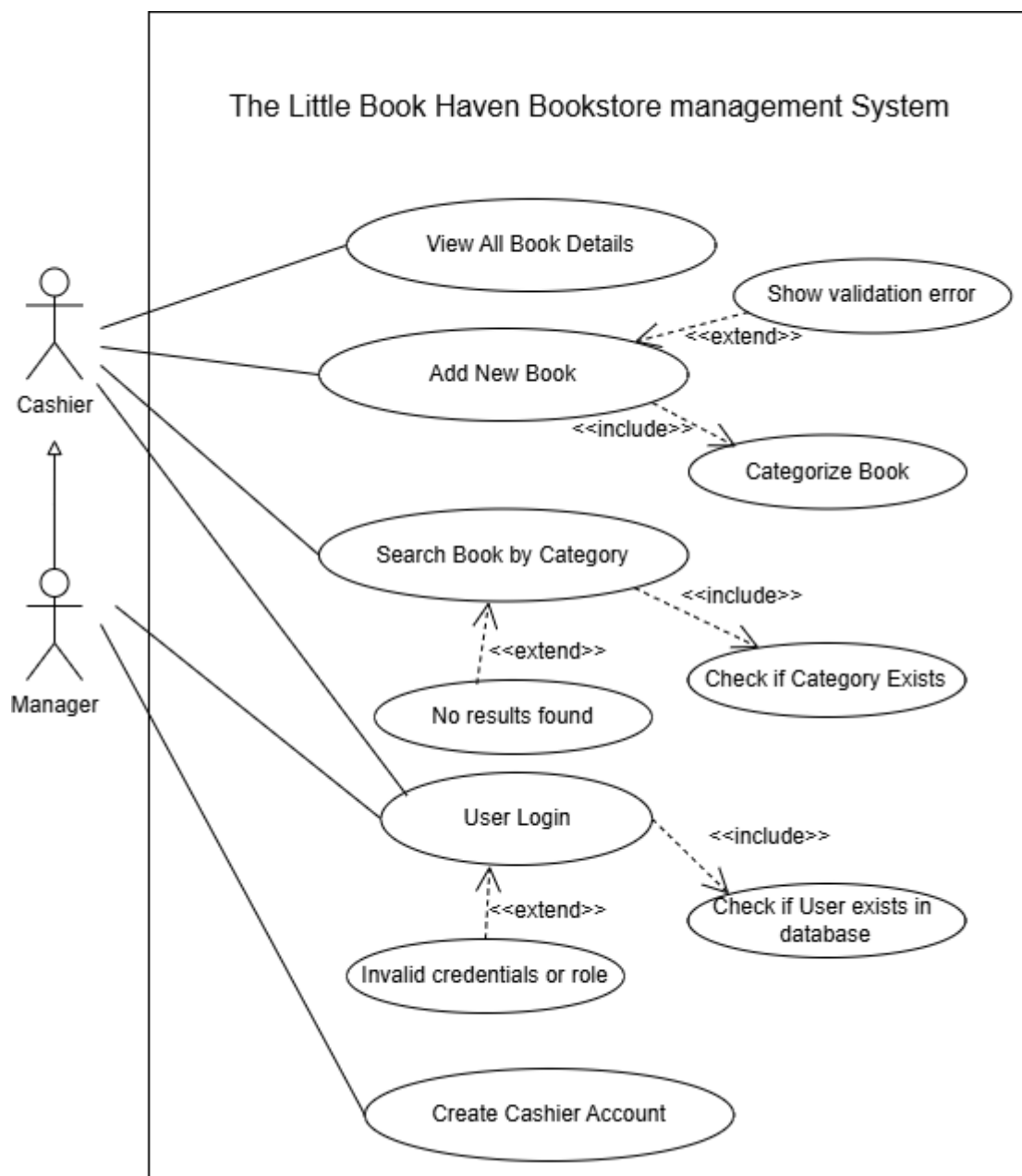


Figure 1 Use-Case

The use case diagram designed successfully models the “The Little Book Haven” bookstore system, aligning with its goal of automating transaction processes using two user levels Cashier and Manager. The following points justify the design:

- 1. Role-Based Functionality:** The inheritance relationship from Manager to Cashier is depicted with a generalization arrow showing that Managers possess all Cashier capabilities with the additional function of creating cashier accounts. This hierarchical structure reduces redundancy and adheres to UML practices for role extension.
- 2. Functionality Coverage and Stereo types:** “Categorize Book” is included in “Add New Book” as categorization is part of adding a book since books are searched by category. “Search Book by Category” includes “Check if Category Exists” ensuring that the category exists to fetch and display the search results. “User Login” includes “Check if User exists in database” to verify the user’s credentials and extends to “Invalid credentials or role” for failed attempts. Similarly, “Add New Book” and “Search Book by Category” extend to “Show validation error” and “No results found” respectively, representing optional flows triggered by specific conditions like invalid input or empty results.
- 3. Login Requirement:** The “User Login” use case serves as a central authentication point, with both Cashier and Manager connecting to it via associations, reflecting that every action requires initial authentication to the system.

1.2. Class Diagram

A class diagram is a fundamental UML diagram that shows the static structure of a system by illustrating its classes, attributes, operations, and relationships among classes. It helps as a blueprint for the system's design, showing how the system's classes are organized and how they interact structurally providing a clear overview of the system's design at a high level (Visual Paradigm, 2025).

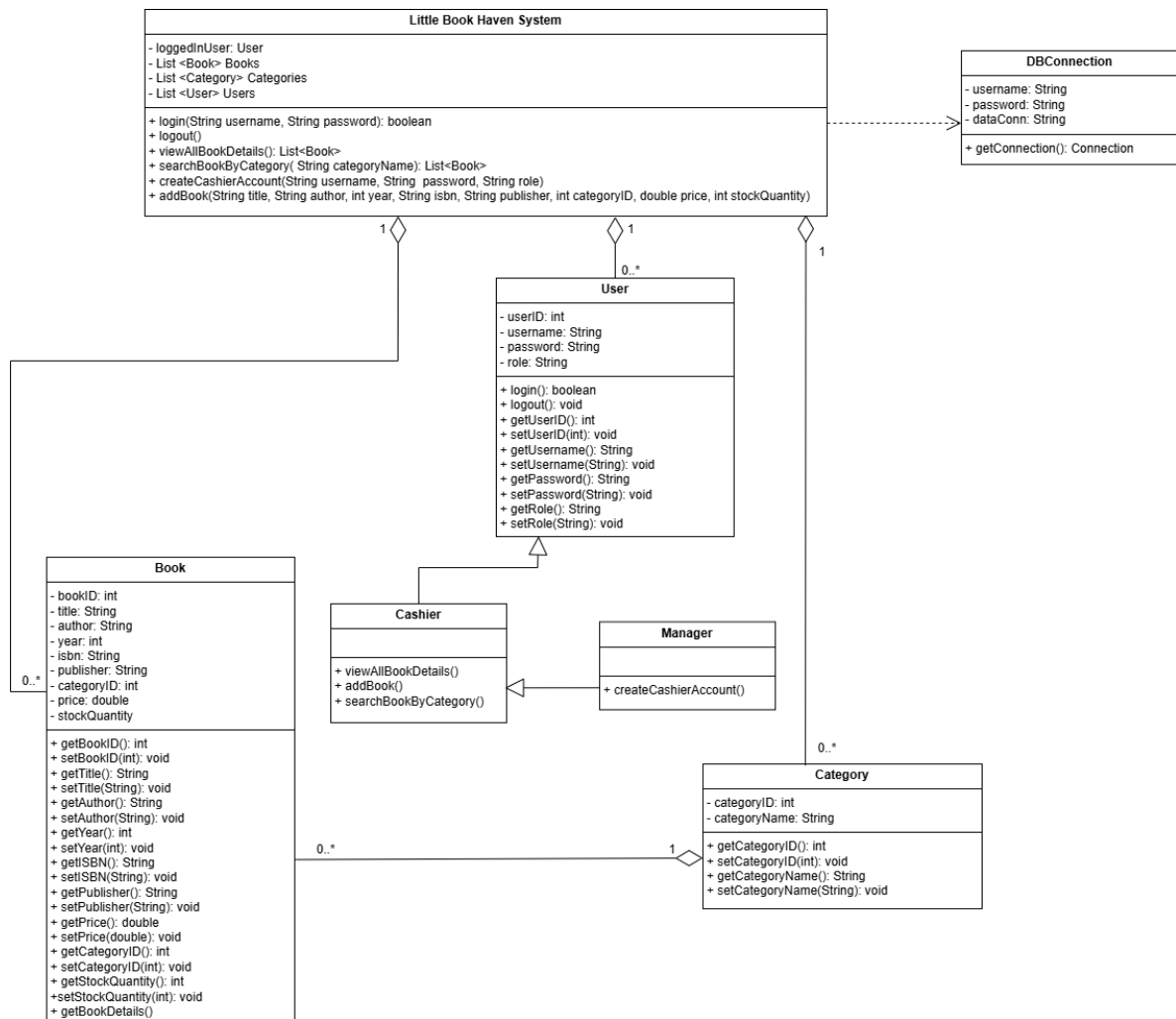


Figure 2 Class Diagram

The provided class diagram effectively models all required entities User, Cashier, Manager, Book, Category, DBConnection and the central system, with their attributes and relationships. The justification is as follows:

1. Entity Modeling

The system consists of several key entities that represent the core components of the bookstore's automated system.

1.1. User (Abstract Class)

This defines a generic system user with attributes like userID, username, password, and role. As an abstract class, it provides a unified structure for subclasses like Cashier and Manager without allowing direct instantiation.

3.2. Cashier (Sub class of User)

This class inherits from User and represents staff handling book-related operations. It includes methods to view, add, and search books, reflecting the operational role of a cashier in the system.

3.3. Manager (Sub class of Cashier)

The Manager class extends Cashier, forming a multilevel inheritance hierarchy from User. This structure allows the Manager to inherit all user and cashier functionalities, while adding account creation as a managerial feature.

3.4. Book

This class represents individual books available in the bookstore's inventory, encapsulating key attributes. Allowing the system to effectively manage, search, categorize and view the books supporting various bookstore operations.

3.5. Category

This class models the classification of books into different genres. It contains attributes such as categoryID and categoryName, which enable the organization and filtering of books, thereby enhancing the user's ability to navigate and search the bookstore's collection efficiently.

3.6. DBConnection (Database connectivity Class)

This class represents the database connectivity utility of the system. It Encapsulates logic for establishing database connections via JDBC. Promoting centralized, reusable access to the database for actions like login, data retrieval, and insertion.

3.7. Little Book Haven System (Central Controller Class)

This class acts as the central system coordinator. Maintains collections of users, books, and categories and implements major operations such as login, adding books, searching, and account creation. By centralizing these functionalities, it acts as the bridge between users and system components, ensuring smooth and consistent operation.

2. Relationships:

2.1.Generalization (Inheritance).

The inheritance from User to Cashier and then to Manager models a clear user hierarchy. Cashier inherits common attributes, while Manager extends functionality by adding account creation, showcasing role-specific privileges and promoting code reuse through inheritance.

2.2. Aggregation

Aggregation is special type of weak relationship that depicts a “Has-a” where one object manages or contains several instances of another object, but those objects can exist independently of the whole part (Horstmann, 2005).

2.2.1. System aggregates User (1-to-many)

Indicates that the system manages collections of users without owning their lifecycle completely, enabling flexible user management.

2.2.2. System aggregates Book (1-to-many)

Indicates the system manages many books for inventory purposes but exist as independent entities, supporting scalable inventory handling.

2.2.3. System aggregates Category (1-to-many)

Indicates the system can consist multiple categories to classify books. This allows the system to organize and manage book genres while categories remain standalone entities.

2.2.4. Category aggregates Book (1-to-many)

This indicates how books are organized by genre within the system. One category can have one or more books, while each book linked to one category.

2.3. Dependency

Dependency relationship indicates that one class depends on another class (Horstmann, 2005).

2.3.1. System depends on DBConnection

This indicates that the system temporarily depends on the DBConnection class to establish database connectivity, but DBConnection exists independently and is not owned by the system.

Assumptions:

Database operations are managed via JDBC calls from the user interface classes, with the system class managing higher-level coordination.

The class design will serve as a blueprint for building the System by applying the core OOP concept promoting clean structure, logical separation of responsibilities, and code reuse.

1.3. Sequence Diagram

A sequence Diagram is an interaction diagram that models the dynamic behavior of a system by illustrating how objects interact through a sequence of messages over time to accomplish a specific task, typically tied to a use case or system operation. It emphasizes the chronological order of message exchanges, showing how objects collaborate to achieve a goal within a defined scenario (Booch, 2005).

1.3.1. Login Sequence

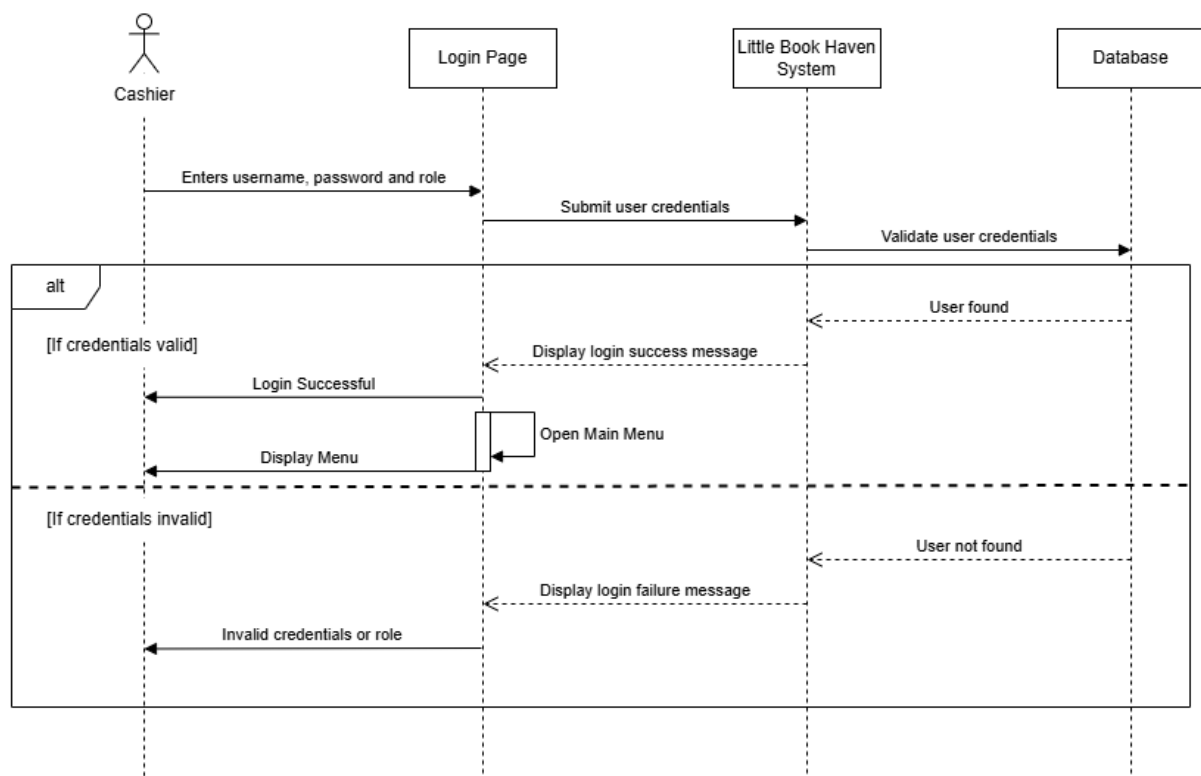


Figure 3 Login-Sequence

The login sequence diagram illustrates the user authentication process, showing how credentials are validated via the system and database. It handles both success and failure cases, ensuring secure access and role identification, which is essential for accessing the system's functionalities.

1.3.2. View Book details Sequence

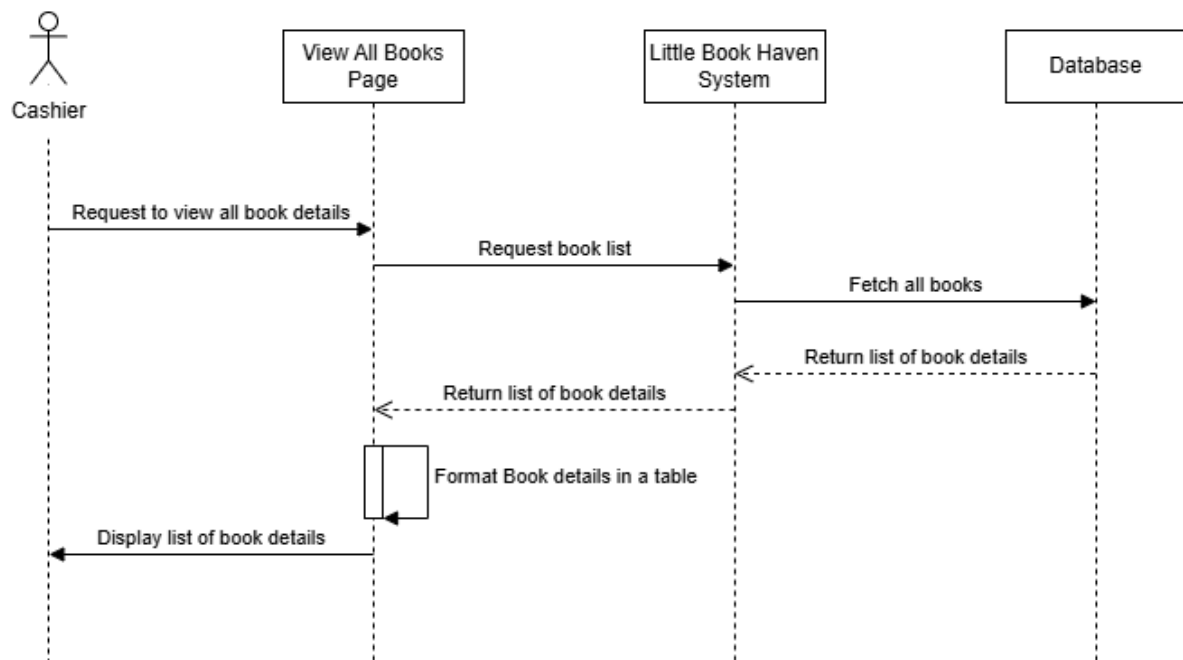


Figure 4 View all Books-Sequence

This diagram represents how the system retrieves and displays book details from the database upon user request. It highlights efficient data flow and formatting, supporting clear user interaction and decision-making.

1.3.3. Add new book Sequence

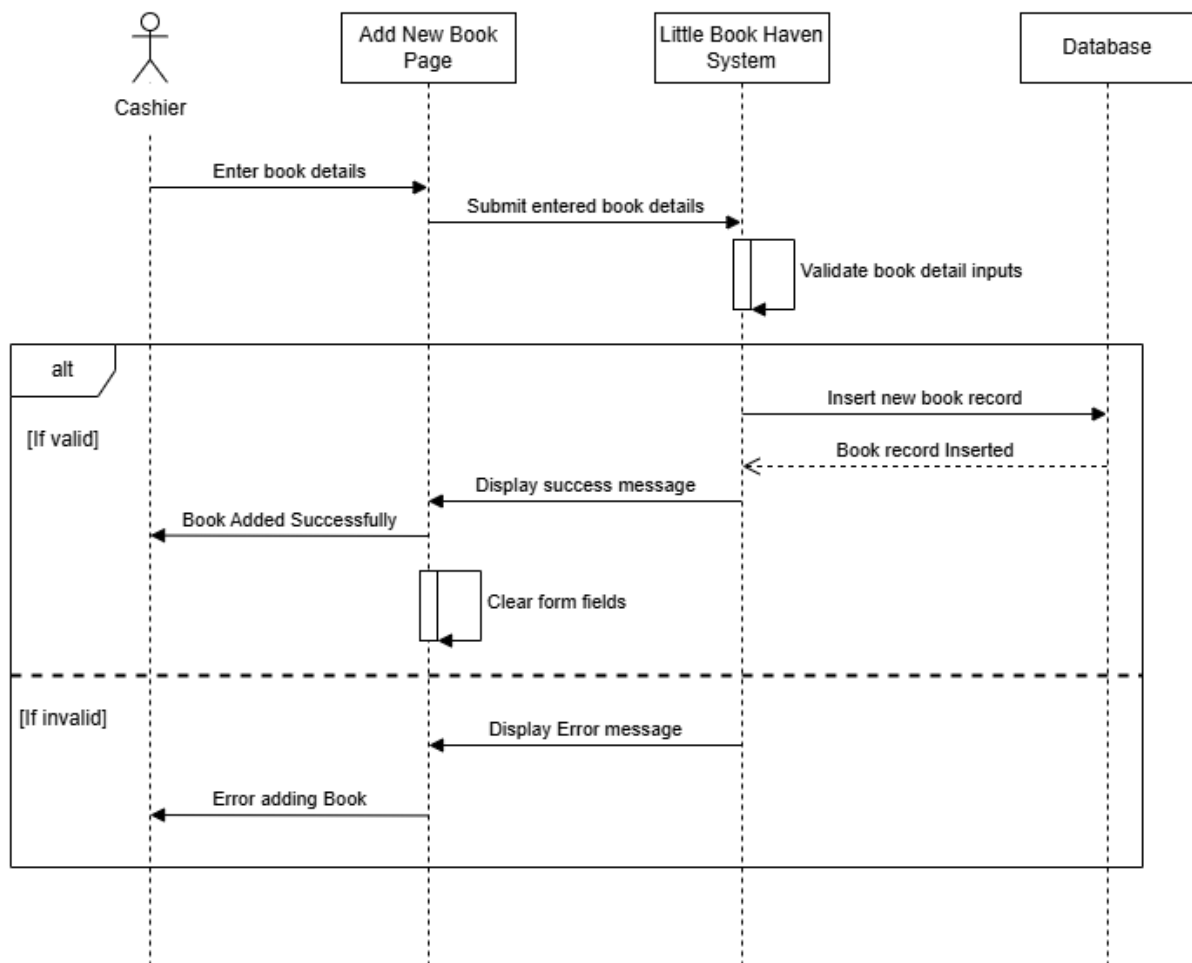


Figure 5 Add New Book-Sequence

This sequence illustrates how the system processes book addition by validating inputs, checking category existence, and inserting the book into the database. It also handles invalid input scenarios, ensuring data accuracy and consistency.

1.3.4. Search book by category Sequence

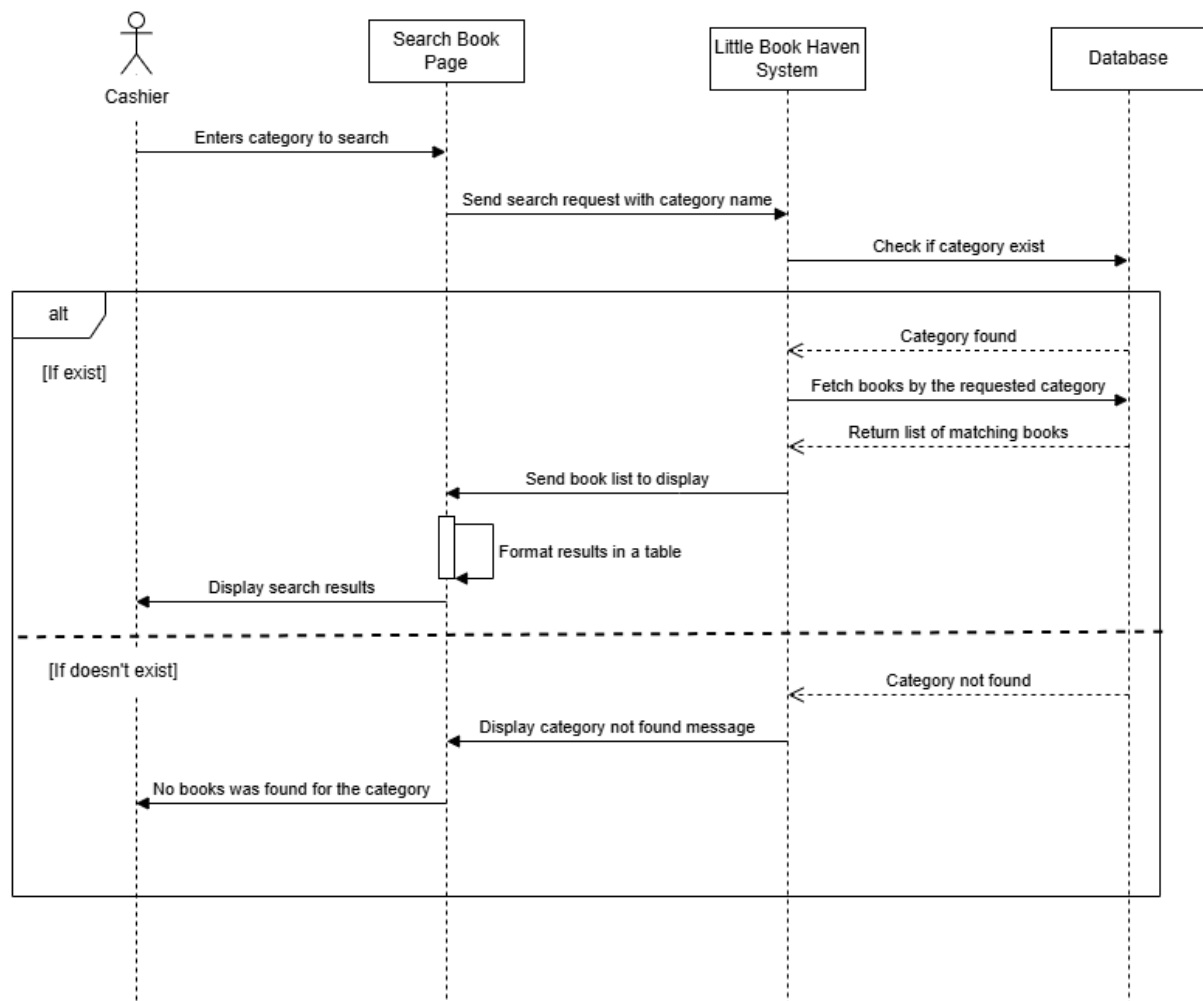


Figure 6 Search-by-Category-Sequence

This sequence shows how the system handles category-based book searches by validating input, retrieving matching records, and managing invalid categories through alternate flows, enhancing user experience and data accuracy.

1.3.5. Creating cashier account Sequence

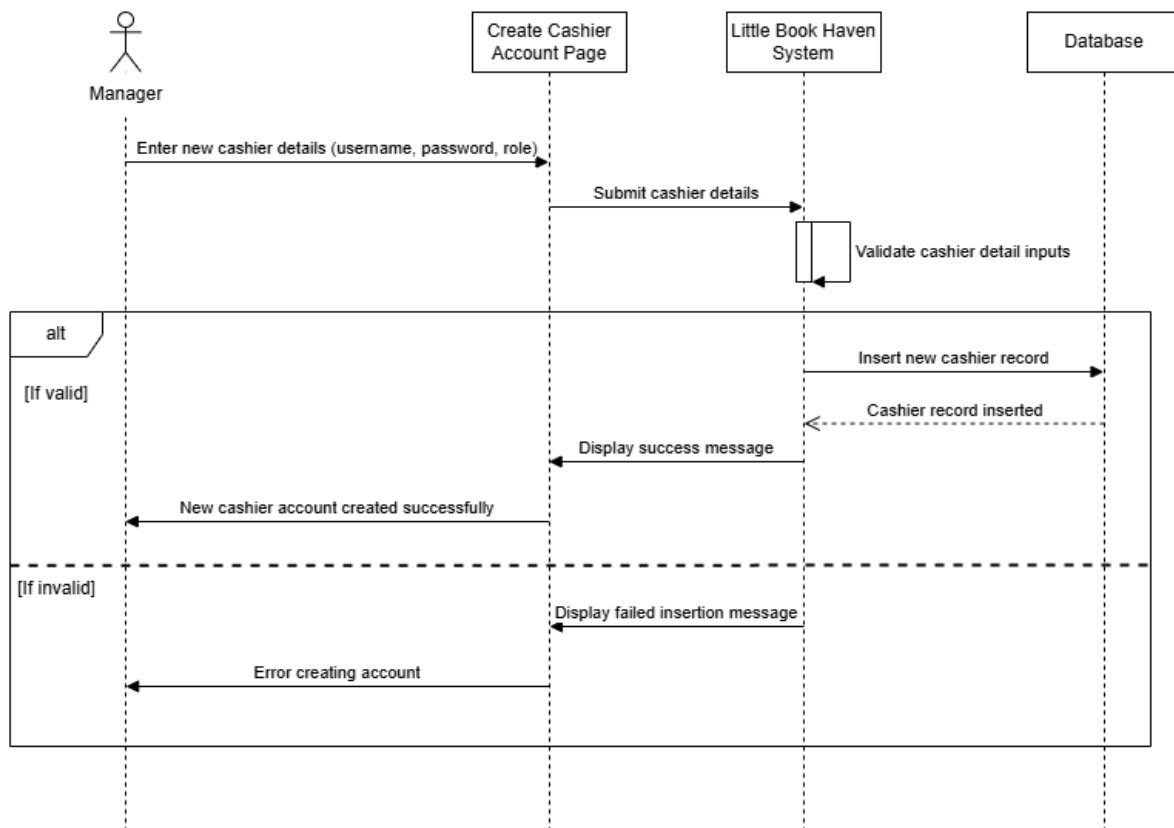


Figure 7 Create Cashier Accounts-Sequence

This diagram shows the process of a manager creating a new cashier account. It illustrates how the system validates input and handles the creation of a new cashier account, including alternate flows for success and failure, ensuring data integrity and clear feedback.

These sequence diagrams clearly represent key system interactions such as login, book management, search, and account creation. They show how users interact with the system and how it communicates with the database, using proper message types, alt blocks, and validation to ensure accurate and user-friendly flows.

The UML diagrams for the Little Book Haven system offer a clear and structured blueprint of both its design and behavior. The class diagram outlines core entities and role hierarchies, the use case diagram maps key user interactions, and the sequence diagrams detail essential process flows. Together, they support a well-organized, scalable system ready for implementation and future expansion.

Task 02: System Development

2. Little Book Haven System Development

The “Little Book Haven” bookstore management system was developed based on the UML designs from Task 01, which served as blueprints for automating operations across two user levels Cashier and Manager. Java was used for implementation, leveraging its object-oriented features and platform independence for both application logic and GUI. MySQL, connected via JDBC, handles secure storage and management of data such as user accounts, book details, and categories. Java Swing was used to create user-friendly interfaces with separate windows for each main operation.

2.1. What is Object Oriented Programming

Object-Oriented Programming (OOP) is a programming style that organizes software design around objects, which are instances of classes, rather than separating data and procedures as in procedural programming. Each object encapsulates data (attributes) and behavior (methods) to model real-world entities. The primary goal of OOP is to enhance flexibility, maintainability, and reusability in complex systems by promoting modularity and scalability (GeeksForGeeks, 2025). This is achieved through several core concepts which are explained below in the context of the “Little Book haven System” we developed.

2.2. Key OOP Concepts applied in the System

2.2.1. Class

A class is a fundamental building block of OOP, it is a user-defined data type that serves as a blueprint or a template for creating objects. It defines the properties (attributes) that an object could have and the behaviors (methods or functions) that an object can perform. While it helps to structure and organize the code, it also encapsulates the complex implementations details inside the class using private fields & public methods (Raut, 2020).

```
package littlebookhaven;

public class Book {

    private int bookID;
    private String title;
    private String author;
    private int year;
    private String isbn;
    private String publisher;
    private int categoryID;
    private double price;
    private int stockQuantity;

    public Book() {}

    public Book(int bookID, String title, String author, int year, String isbn, String publisher, int categoryID, double price, int stockQuantity) {
        this.bookID = bookID;
        this.title = title;
        this.author = author;
        this.isbn = isbn;
        this.publisher = publisher;
        this.categoryID = categoryID;
        this.price = price;
    }

    public Book(String title, String author, int year, String isbn, String publisher, int categoryID, double price, int stockQuantity) {
        this.title = title;
        this.author = author;
        this.isbn = isbn;
        this.publisher = publisher;
        this.categoryID = categoryID;
        this.price = price;
    }

    public int getBookID() { return bookID; }
    public void setBookID(int bookID) { this.bookID = bookID; }

    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 int getYear() { return year; }
    public void setYear(int year) { this.year = year; }

    public String getIsbn() { return isbn; }
    public void setIsbn(String isbn) { this.isbn = isbn; }

    public String getPublisher() { return publisher; }
    public void setPublisher(String publisher) { this.publisher = publisher; }

    public int getCategoryID() { return categoryID; }
    public void setCategoryID(int categoryID) { this.categoryID = categoryID; }

    public double getPrice() { return price; }
    public void setPrice(double price) { this.price = price; }

    public int getStockQuantity() { return stockQuantity; }
    public void setstockQuantity(int stockQuantity) { this.stockQuantity = stockQuantity; }
    @Override
    public String toString() {
        return "Book{" +
            "bookID=" + bookID +
            ", title=" + title + '\'' +
            ", author=" + author + '\'' +
            ", year=" + year +
            ", isbn=" + isbn +
            ", publisher=" + publisher + '\'' +
            ", categoryID=" + categoryID +
            ", price=" + price +
            ", stockQuantity=" + stockQuantity +
            '}';
    }
}
```

Figure 8 Book Class

The Book class in the system represents a real-world book and includes attributes like title, author, and price, along with methods for accessing and modifying these values. This demonstrates the concept of a class by acting as a blueprint for all book objects. It encapsulates data and related behavior, supporting code organization, reuse, and scalability.

2.2.2. Object

Objects are instances of a class with specifically defined data. These represent real-world objects or abstract entities, encapsulating attributes and methods that operate on that data. Each object has a unique identity and state, representing a specific instance of the class from which it was created, making them self-contained units (Weisfeld, 2008).

In the Little Book Haven system, objects are created from user-defined classes to perform actual operations. This includes user interfaces like Java Swings, which are themselves classes. When a button is clicked, a new object of the corresponding form class is created and made visible enabling user interaction and functionality.

```
package littlebookhaven;

import java.sql.*;
import java.util.HashMap;
import java.util.Map;
import javax.swing.JOptionPane;

public class AddNewBookFrame extends javax.swing.JFrame {

    private static final java.util.logging.Logger logger = java.util.logging.Logger.getLogger(AddNewBookFrame.class.getName());

    private User loggedInUser;

    private Map<String, Integer> categoryMap = new HashMap<>();

    public AddNewBookFrame(User user) {
        this.loggedInUser = user;
        initComponents();
        try {
            Connection conn = DBConnection.getConnection();
            String query = "SELECT categoryID, categoryName FROM category";
            PreparedStatement pst = conn.prepareStatement(query);
            ResultSet rs = pst.executeQuery();

            category.removeAllItems();

            while (rs.next()) {
                int id = rs.getInt("categoryID");
                String name = rs.getString("categoryName");
                category.addItem(name);
                categoryMap.put(name, id);
            }

            rs.close();
            pst.close();
            conn.close();
        } catch (Exception ex) {
```

Figure 9 AddNewBookFrame Class

```
private void addNewBookActionPerformed(java.awt.event.ActionEvent evt) {
    AddNewBookFrame ab = new AddNewBookFrame (user: loggedInUser);
    ab.setVisible(b: true);
    this.setVisible(b: false);
}
```

Figure 10 Object Created from AddNewBookFrame

Java Swing forms such as “AddNewBookFrame” are implemented as classes, and objects are created from these classes to carry out book-related operations. For example, when the user clicks the “Add New Book” button in the Menu, the system creates an object named “ab” from the “AddNewBookFrame” class. This object is then displayed to the user by calling “ab.setVisible(true)”. The object encapsulates all the components and functionality needed to add a new book, such as input fields, buttons, and database interaction. The parameter “loggedInUser” is a user object passed to the constructor, allowing the form to operate based on the current user’s role.

2.2.3. Abstraction

Abstraction is the approach of revealing only essential features of an object while hiding unnecessary or complex implementation details. It simplifies the complex reality by modeling classes based on essential properties and behaviors relevant to specific context, focusing on what an object does rather than how it does it (GeeksForGeeks, 2025; Raut, 2020).

In the Little Book Haven system, abstraction is demonstrated through the “DBConnection” class.

```
package littlebookhaven;

import java.sql.*;

public class DBConnection {

    private static final String username = "root";
    private static final String password = "";
    private static final String dataConn = "jdbc:mysql://localhost:3306/littlebookhaven";

    private DBConnection() {}

    public static Connection getConnection() {
        Connection conn = null;
        try {
            Class.forName(className: "com.mysql.jdbc.Driver");
            conn = DriverManager.getConnection(url: dataConn, user: username, password);
        } catch (Exception ex) {
            System.out.println("Database connection failed: " + ex.getMessage());
        }
        return conn;
    }
}
```

Figure 11 DBConnection Class

```

public AddNewBookFrame(User user) {
    this.loggedInUser = user;
    initComponents();
    try {
        Connection conn = DBConnection.getConnection();
        String query = "SELECT categoryID, categoryName FROM category";
        PreparedStatement pst = conn.prepareStatement(query);
        ResultSet rs = pst.executeQuery();

        category.removeAllItems();

        while (rs.next()) {
            int id = rs.getInt("categoryID");
            String name = rs.getString("categoryName");
            category.addItem(name);
            categoryMap.put(name, id);
        }

        rs.close();
        pst.close();
        conn.close();
    } catch (Exception ex) {
        JOptionPane.showMessageDialog(this, "Error loading categories: " + ex.getMessage());
    }
}

```

Figure 12 DBConnection Method called to establish connection

This class contains the method `getConnection()`, which handles the internal details of establishing a database connection including the JDBC driver setup, database URL, and login credentials. When this method is called in other parts of the system, such as in the “AddNewBookFrame” constructor (`Connection conn = DBConnection.getConnection();`). We don't handle the connection setup directly instead; we use the method to obtain a ready-to-use `Connection` object. This reduces redundancy, and centralizes the database logic.

In addition to the database connection, abstraction is also applied in GUI frame objects, where the internal UI design and event handling are hidden. The system interacts with these frames through simple methods like `setVisible(true)`, without needing to understand their internal structure or logic.

```

private void jButton2ActionPerformed(java.awt.event.ActionEvent evt) {
    loginFrame lf=new loginFrame();
    lf.setVisible(b: true);
    this.setVisible(b: false);
}

private void viewBookListActionPerformed(java.awt.event.ActionEvent evt) {
    BookDetailsFrame bd = new BookDetailsFrame (user: loggedInUser);
    bd.setVisible(b: true);
    this.setVisible(b: false);
}

private void searchBookBycateActionPerformed(java.awt.event.ActionEvent evt) {
    SearchBookFrame ab = new SearchBookFrame (user: loggedInUser);
    ab.setVisible(b: true);
    this.setVisible(b: false);
}

private void addNewBookActionPerformed(java.awt.event.ActionEvent evt) {
    AddNewBookFrame ab = new AddNewBookFrame (user: loggedInUser);
    ab.setVisible(b: true);
    this.setVisible(b: false);
}

private void createCashierAccActionPerformed(java.awt.event.ActionEvent evt) {
    CreateCashierAccountFrame cc = new CreateCashierAccountFrame (user: loggedInUser);
    cc.setVisible(b: true);
    this.setVisible(b: false);
}

```

Figure 13 Objects created from Swing Classes

2.2.4. Inheritance

Inheritance is a fundamental concept in OOP which allows one class known as the subclass or the child class to inherit all properties (attributes) and behaviors (methods) from another class identified as the superclass or parent class. These models a “Is-a type of” relationship meaning that the subclass is a type of the superclass. This concept reduce redundancy, improves code reusability, and establishes a logical hierarchy (Weisfeld, 2008).

Java supports three types of inheritance:

- Single Inheritance: where a class inherits from one parent class.
- Multilevel Inheritance: where a class inherits from a subclass, forming an inheritance chain.
- Hierarchical Inheritance: where multiple subclasses inherit from a single superclass.

However, Multiple Inheritance where a class inherits directly from more than one parent class is not supported in Java to avoid complexity and ambiguity issues.

In the Little Book Haven system, the User class acts as the superclass with common user properties like userID, username, password, and role. The subclasses Cashier and Manager inherit from User, gaining these common attributes and behaviors while adding their own specific methods.

This is an example of hierarchical inheritance because both Cashier and Manager extend the same superclass User but represent different types of users with specialized functionalities.

```
package littlebookhaven;

public class User {
    private int userID;
    private String username;
    private String password;
    private String role;

    public User() {}

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

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

    public int getUserID() { return userID; }
    public void setUserID(int userID) { this.userID = userID; }

    public String getUsername() { return username; }
    public void setUsername(String username) { this.username = username; }

    public String getPassword() { return password; }
    public void setPassword(String password) { this.password = password; }

    public String getRole() { return role; }
    public void setRole(String role) { this.role = role; }
}

class Cashier extends User {

    public Cashier(int userID, String username, String password) {
        super(userID, username, password, role: "Cashier");
    }

    public Cashier(String username, String password) {
        super(username, password, role: "Cashier");
    }

    public void viewBookDetails(User loggedInUser) {
        BookDetailsFrame bd = new BookDetailsFrame(user: loggedInUser);
        bd.setVisible(b: true);
    }

    public void addBook(User loggedInUser) {
        AddNewBookFrame ab = new AddNewBookFrame(user: loggedInUser);
        ab.setVisible(b: true);
    }

    public void searchByCategory(User loggedInUser) {
        SearchBookFrame ab = new SearchBookFrame(user: loggedInUser);
        ab.setVisible(b: true);
    }
}

class Manager extends Cashier {

    public Manager(int userID, String username, String password) {
        super(userID, username, password);
        setRole(role: "Manager");
    }

    public Manager(String username, String password) {
        super(username, password);
        setRole(role: "Manager");
    }

    public void createCashierAccount(User loggedInUser) {
        CreateCashierAccountFrame cc = new CreateCashierAccountFrame(user: loggedInUser);
        cc.setVisible(b: true);
    }
}
```

Figure 14 User Class

```

class Cashier extends User {

    public Cashier(int userID, String username, String password) {
        super(userID, username, password, role: "Cashier");
    }

    public Cashier(String username, String password) {
        super(username, password, role: "Cashier");
    }

    public void viewBookDetails(User loggedInUser) {
        BookDetailsFrame bd = new BookDetailsFrame(user: loggedInUser);
        bd.setVisible(b: true);
    }

    public void addBook(User loggedInUser) {
        AddNewBookFrame ab = new AddNewBookFrame(user: loggedInUser);
        ab.setVisible(b: true);
    }

    public void searchByCategory(User loggedInUser) {
        SearchBookFrame ab = new SearchBookFrame(user: loggedInUser);
        ab.setVisible(b: true);
    }
}

```

Figure 15 Cashier inherit from User

Cashier inherits common user properties from User and adds cashier-specific methods.

```

class Manager extends Cashier {

    public Manager(int userID, String username, String password) {
        super(userID, username, password);
        setRole(role: "Manager");
    }

    public Manager(String username, String password) {
        super(username, password);
        setRole(role: "Manager");
    }

    public void createCashierAccount(User loggedInUser) {
        CreateCashierAccountFrame cc = new CreateCashierAccountFrame(user: loggedInUser);
        cc.setVisible(b: true);
    }
}

```

Figure 16 Manager inherit from Cashier

Manager inherits from Cashier which itself inherits from User, allowing reuse of all user and cashier methods plus manager-specific methods such as creating cashier accounts.

Together, these relationships demonstrate hierarchical inheritance because multiple subclasses Cashier and Manager derive from a common base User, and also multilevel inheritance as Manager inherits from Cashier, which inherits from User.

```

static class Loader {
    public String getLoadingMessage() {
        return "Preparing to start...";
    }
}

static class LoadingStage1 extends Loader {
    public String getLoadingMessage() {
        return "Loading resources...";
    }
}

static class LoadingStage2 extends Loader {
    public String getLoadingMessage() {
        return "Initializing components...";
    }
}

static class LoadingStage3 extends Loader {
    public String getLoadingMessage() {
        return "Connecting database...";
    }
}

static class LoadingStage4 extends Loader {
    public String getLoadingMessage() {
        return "Generating User Interfaces...";
    }
}

static class LoadingStage5 extends Loader {
    public String getLoadingMessage() {
        return "Finalizing setup...";
    }
}

static class LoadingStage6 extends Loader {
    public String getLoadingMessage() {
        return "Launching application...";
    }
}

```

Figure 17 Inheritance applied in Progress bar

This is another example of inheritance used in the progress bar implementation, the class Loader is the superclass that defines a method getLoadingMessage(). The other classes LoadingStage1 to LoadingStage6 inherit from Loader and each overrides the getLoadingMessage() method to provide a specific loading message. This demonstrates inheritance because the child classes reuse the structure of the parent class but customize the behavior by overriding methods. It is a clear example of how subclasses extend a superclass to provide specialized functionality.

2.2.5. Encapsulation

Encapsulation also known as Information hiding is a process where it restricts direct access to some components of an object. It is the mechanism of bundling the data (attributes) and methods (functions) that operate on the data into a single unit class while hiding its internal implementation details. This is achieved by making attributes private or protected and providing public methods (getters & setters) to control access to them. This ensures both the data and functions are secure by preventing unauthorized access and direct modification from outside the object, showing only what is necessary (Weisfeld, 2008; Raut, 2020).

```
package littlebookhaven;

public class User {
    private int userID;
    private String username;
    private String password;
    private String role;

    public User() {}

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

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

    public int getUserID() { return userID; }
    public void setUserID(int userID) { this.userID = userID; }

    public String getUsername() { return username; }
    public void setUsername(String username) { this.username = username; }

    public String getPassword() { return password; }
    public void setPassword(String password) { this.password = password; }

    public String getRole() { return role; }
    public void setRole(String role) { this.role = role; }
}
```

Figure 18 Encapsulation in User-class

Encapsulation is demonstrated in the User class, where sensitive user data such as userID, username, password, and role are declared as private fields. These fields cannot be accessed directly from outside the class. Instead, they are accessed and modified through public getter and setter methods, which provide controlled interaction with the data. This hides the internal implementation and protects the integrity of the data, ensuring that any access to these fields is done in a secure and consistent manner.

```

package littlebookhaven;

public class Book {

    private int bookID;
    private String title;
    private String author;
    private int year;
    private String isbn;
    private String publisher;
    private int categoryID;
    private double price;
    private int stockQuantity;

    public int getBookID() { return bookID; }
    public void setBookID(int bookID) { this.bookID = bookID; }

    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 int getYear() { return year; }
    public void setYear(int year) { this.year = year; }

    public String getIsbn() { return isbn; }
    public void setIsbn(String isbn) { this.isbn = isbn; }

    public String getPublisher() { return publisher; }
    public void setPublisher(String publisher) { this.publisher = publisher; }

    public int getCategoryID() { return categoryID; }
    public void setCategoryID(int categoryID) { this.categoryID = categoryID; }

    public double getPrice() { return price; }
    public void setPrice(double price) { this.price = price; }

    public int getStockQuantity() { return stockQuantity; }
    public void setstockQuantity(int stockQuantity) { this.stockQuantity = stockQuantity; }
}

```

Figure 19 Encapsulation in Book-class

Similarly, encapsulation is also seen in the Book class where all its fields like title, author, price, etc. are declared as private, which hides them from direct access by external classes. Instead, public getter and setter methods like getTitle() are provided to safely access and modify the data. This ensures that the internal state of a book is controlled, protected, and can only be updated through defined interfaces.

2.2.6. Polymorphism

Polymorphism is a core OOP concept that allows different objects to be treated uniformly through a common superclass interface. This is achieved in two main methods. Method overriding, where subclasses provide their own unique implementation for a method inherited from a superclass, allowing dynamic behavior based on the object's actual type at runtime and method overloading, where a single class defines multiple methods with the same name but different parameters, providing ways to call related operations based on the inputs given at compile time (Weisfeld, 2008). This overall flexibility significantly enhances code extensibility, and reusability (GeeksForGeeks, 2025).

In the “Little Book Haven” system, polymorphism is demonstrated in the progress bar loading process.

```
static class Loader {
    public String getLoadingMessage() {
        return "Preparing to start...";
    }
}

static class LoadingStage1 extends Loader {
    public String getLoadingMessage() {
        return "Loading resources...";
    }
}

static class LoadingStage2 extends Loader {
    public String getLoadingMessage() {
        return "Initializing components...";
    }
}

static class LoadingStage3 extends Loader {
    public String getLoadingMessage() {
        return "Connecting database...";
    }
}

static class LoadingStage4 extends Loader {
    public String getLoadingMessage() {
        return "Generating User Interfaces...";
    }
}

static class LoadingStage5 extends Loader {
    public String getLoadingMessage() {
        return "Finalizing setup...";
    }
}

static class LoadingStage6 extends Loader {
    public String getLoadingMessage() {
        return "Launching application...";
    }
}
```

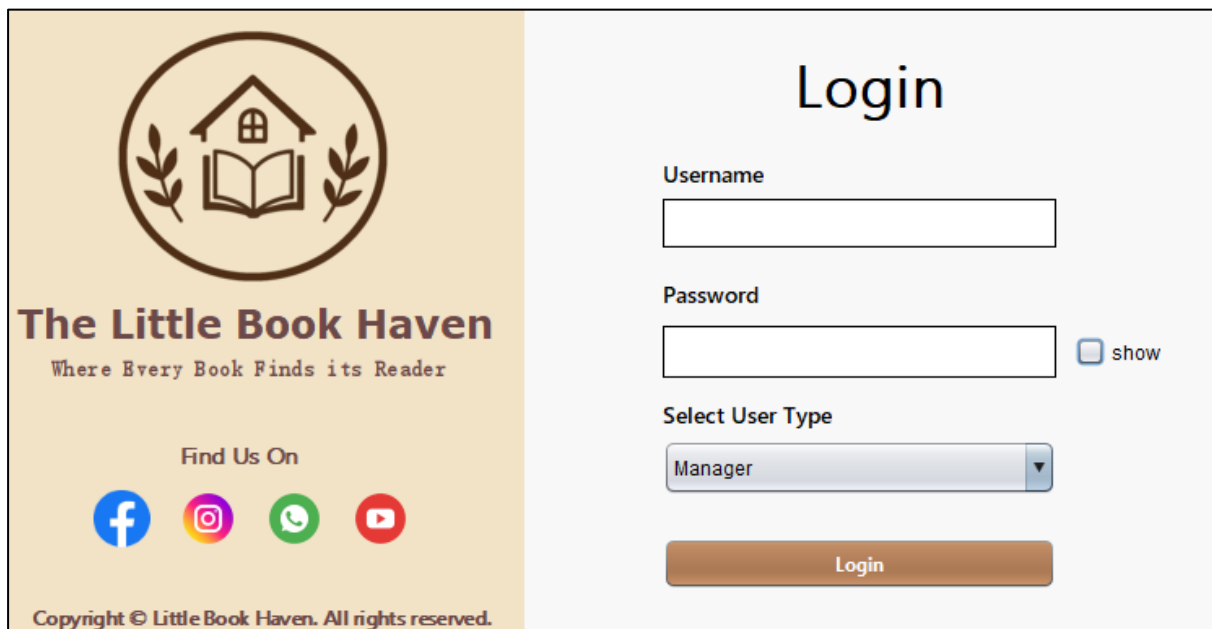
Figure 20 Polymorphism applied in Progress bar


The class Loader defines a method `getLoadingMessage()`, and its subclasses `LoadingStage1` to `LoadingStage6` each override this method to return different loading messages. For example, `LoadingStage1` returns “Loading resources...”, while `LoadingStage3` returns “Connecting database...”. Although all these classes are treated as Loader objects, the method behaves differently depending on which subclass object is used. This shows runtime polymorphism, where the same method call results in different outputs based on the actual object, making the system flexible and easier to extend.

By applying OOP principles, the system achieves modularity, scalability, and maintainability. Key functionalities like view books, adding books, searching, and user account creation are encapsulated in dedicated Java Swing classes and methods, ensuring robustness and easy extensibility. This approach aligns the application with modern development standards. The project’s planning and development timeline is detailed in the Appendix A-Gantt Chart.

2.3. Main Functionalities





2.3.1. User Login





The Little Book Haven
Where Every Book Finds its Reader

Find Us On



Copyright © Little Book Haven. All rights reserved.

Login

Username

Password
 ☐ show

Select User Type

Manager ▼

Login

```

private void loginBtnActionPerformed(java.awt.event.ActionEvent evt) {
    String username = txtUsername.getText();
    String password = new String(value: txtpassword.getPassword());
    String role = (String) userType.getSelectedItemAt();

    try {
        conn = DBConnection.getConnection();

        if (conn == null) {
            JOptionPane.showMessageDialog(parentComponent: this, message: "Database connection failed.");
            return;
        }

        String sql = "SELECT * FROM user WHERE BINARY username = ? AND BINARY password = ? AND BINARY role = ?";
        pst = conn.prepareStatement(string: sql);
        pst.setString(i: 1, string: username);
        pst.setString(i: 2, string: password);
        pst.setString(i: 3, string: role);

        rs = pst.executeQuery();

        if (rs.next()) {
            int userID = rs.getInt(string: "userID");
            User loggedInUser;

            if ("Manager".equalsIgnoreCase(anotherString: role)) {
                loggedInUser = new Manager(userID, username, password);
            } else {
                loggedInUser = new Cashier(userID, username, password);
            }

            JOptionPane.showMessageDialog(parentComponent: this, "Login Successful as " + role);

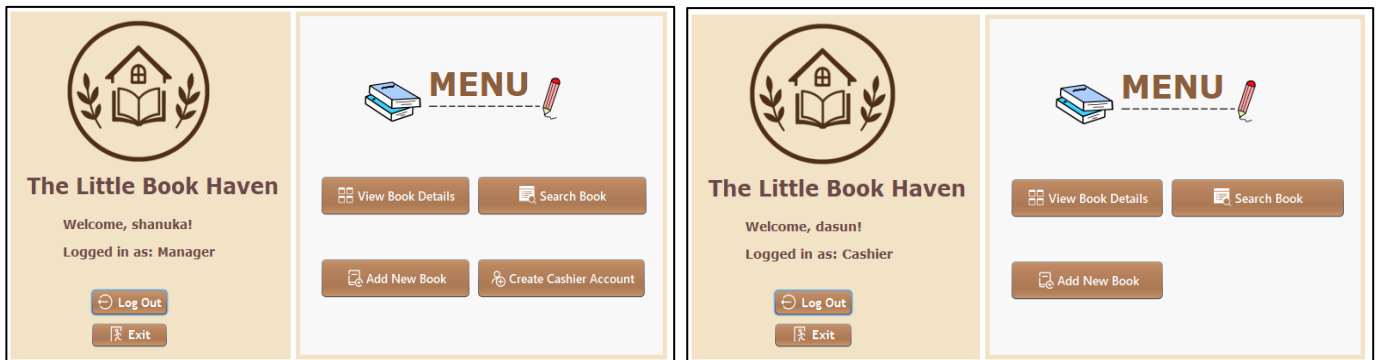
            this.dispose();
            new MainMenu(authenticatedUser: loggedInUser).setVisible(b: true);
        } else {
            JOptionPane.showMessageDialog(parentComponent: this, message: "Invalid credentials or role.", title: "Login Failed", messageType: JOptionPane.
        } catch (Exception ex) {
            JOptionPane.showMessageDialog(parentComponent: this, "Login error: " + ex.getMessage());
        } finally {
            try {
                if (rs != null) rs.close();
                if (pst != null) pst.close();
                if (conn != null) conn.close();
            } catch (SQLException ex) {
                System.out.println("Failed to close DB resources: " + ex.getMessage());
            }
        }
    }
}

```

Figure 21 User-Login

This functionality checks user credentials against the database, creates a Cashier or Manager object based on the role, displays a success message, and opens the main menu, ensuring secure access.

2.3.2. Main Menu



```
package littlebookhaven;

public class MainMenu extends javax.swing.JFrame {

    private static final java.util.logging.Logger logger = java.util.logging.Logger.getLogger(name: MainMenu.class.getName());

    private User loggedInUser;

    public MainMenu(User authenticatedUser) {
        this.loggedInUser = authenticatedUser;

        initComponents();

        welcomeLabel.setText("Welcome, " + loggedInUser.getUsername() + "!");
        loggedInUser.setText("Logged in as: " + loggedInUser.getRole());

        if (!(loggedInUser instanceof Manager)) {
            createCashierAcc.setVisible(aFlag: false);
        }
    }

    MainMenu() {
        throw new UnsupportedOperationException(message: "Not supported yet."); // Generated from nbfs://nbhost/SystemFileSystem
    }

    @SuppressWarnings("unchecked")
    Generated Code

    private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
        System.exit (status: 0);
    }

    private void jButton2ActionPerformed(java.awt.event.ActionEvent evt) {
        loginFrame lf=new loginFrame();
        lf.setVisible(b: true);
        this.setVisible(b: false);
    }

    private void viewBookListActionPerformed(java.awt.event.ActionEvent evt) {
        BookDetailsFrame bd = new BookDetailsFrame(user: loggedInUser);
        bd.setVisible(b: true);
        this.setVisible(b: false);
    }

    private void searchBookBycateActionPerformed(java.awt.event.ActionEvent evt) {
        SearchBookFrame ab = new SearchBookFrame (user: loggedInUser);
        ab.setVisible(b: true);
        this.setVisible(b: false);
    }

    private void addNewBookActionPerformed(java.awt.event.ActionEvent evt) {
        AddNewBookFrame ab = new AddNewBookFrame (user: loggedInUser);
        ab.setVisible(b: true);
        this.setVisible(b: false);
    }


    private void createCashierAccActionPerformed(java.awt.event.ActionEvent evt) {
        CreateCashierAccountFrame cc = new CreateCashierAccountFrame(user: loggedInUser);
        cc.setVisible(b: true);
        this.setVisible(b: false);
    }

    public static void main(String args[]) {
    }
}
```

Figure 22 Main-Menu

This serves as the central hub after login, displaying a welcome message with the user's username and role. This functionality displays a personalized interface based on the user's role hiding "Create Cashier Accounts" for Cashiers, enabling navigation to other features via buttons, and allows logout to return to the login screen or exit to close the system.

2.3.3. View all Books


The Little Book Haven
< Back to Menu
Exit

View All Book Details

Book ID	Title	Author	Published Ye...	ISBN	Publisher	Category	Price	Stock Quantity
1	1984	George Orwell	1949	9780451524...	Signet Classic	Fiction	500.00	20
2	Pride and Pr...	Jane Austen	1813	9780141439...	T. Egerton	Fiction	400.00	15
3	The Alchemist	Paulo Coelho	1988	9780061122...	HarperOne	Fiction	450.00	18
4	A Little Life	Hanya Yanag...	2015	9780804172...	Knopf Doubl...	Fiction	1200.00	10
5	Normal Peop...	Sally Rooney	2018	9781984822...	Crown Publ...	Fiction	700.00	12
6	Beloved	Toni Morrison	1987	9781400033...	Vintage	Fiction	700.00	14
7	All the Light ...	Anthony Doerr	2014	9781501173...	Scribner	Fiction	800.00	13
8	The Hobbit	J.R.R. Tolkien	1937	9780547928...	HarperCollins	Fiction	600.00	25
9	Harry Potter ...	J.K. Rowling	1997	9780747532...	Bloomsbury	Fiction	500.00	30
10	The Great G...	F. Scott Fitz...	1925	9780451524...	Signet Classic	Fiction	500.00	20

View All
Clear

```

package littlebookhaven;

import java.sql.*;
import javax.swing.JOptionPane;
import javax.swing.JFrame;
import javax.swing.table.DefaultTableModel;
import javax.swing.table.TableRowSorter;
import javax.swing.RowFilter;

public class BookDetailsFrame extends javax.swing.JFrame {

    private static final java.util.logging.Logger logger = java.util.logging.Logger.getLogger(name: BookDetailsFrame.class.getName());

    private User loggedInUser;

    public BookDetailsFrame(User user) {
        this.loggedInUser = user;
        initComponents();
    }

    private BookDetailsFrame() {
        throw new UnsupportedOperationException(message: "Not supported yet."); // Generated from nbfs://nbhost/SystemFileSystem/Template
    }

    @SuppressWarnings("unchecked")
    Generated Code

    private void jButton3ActionPerformed(java.awt.event.ActionEvent evt) {
        try {
            Connection conn = DBConnection.getConnection();

            if (conn == null) {
                JOptionPane.showMessageDialog(parentComponent: this, message: "Failed to connect to the database.");
                return;
            }

            String sql = "SELECT b.bookID, b.title, b.author, b.year, b.isbn, b.publisher, c.categoryName, b.price, b.stockQuantity FROM book b "
                + "JOIN category c ON b.categoryID = c.categoryID ORDER BY bookID ASC";
            PreparedStatement pst = conn.prepareStatement(sql);
            ResultSet rs = pst.executeQuery();

            DefaultTableModel tblModel = (DefaultTableModel) bookList.getModel();
            tblModel.setRowCount(rowCount: 0);

            while (rs.next()) {
                String bookID = rs.getString("bookID");
                String title = rs.getString("title");
                String author = rs.getString("author");
                String year = String.valueOf(rs.getInt("year"));
                String isbn = rs.getString("isbn");
                String publisher = rs.getString("publisher");
                String categoryName = rs.getString("categoryName");
                String price = String.format("%.2f", rs.getDouble("price"));
                String stockQuantity = String.valueOf(rs.getInt("stockQuantity"));

                String[] rowData = { bookID, title, author, year, isbn, publisher, categoryName, price, stockQuantity };
                tblModel.addRow(rowData);
            }

            rs.close();
            pst.close();
            conn.close();
        } catch (Exception ex) {
            JOptionPane.showMessageDialog(parentComponent: this, "Error loading books: " + ex.getMessage());
        }
    }

    private void jButton5ActionPerformed(java.awt.event.ActionEvent evt) {
        DefaultTableModel tblModel = (DefaultTableModel) bookList.getModel();
        tblModel.setRowCount(rowCount: 0);
    }

    private void jButton2ActionPerformed(java.awt.event.ActionEvent evt) {
        MainMenu mm = new MainMenu(authenticatedUser: loggedInUser);
        mm.setVisible(b: true);
        this.setVisible(b: false);
    }

    private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
        System.exit(status: 0);
    }

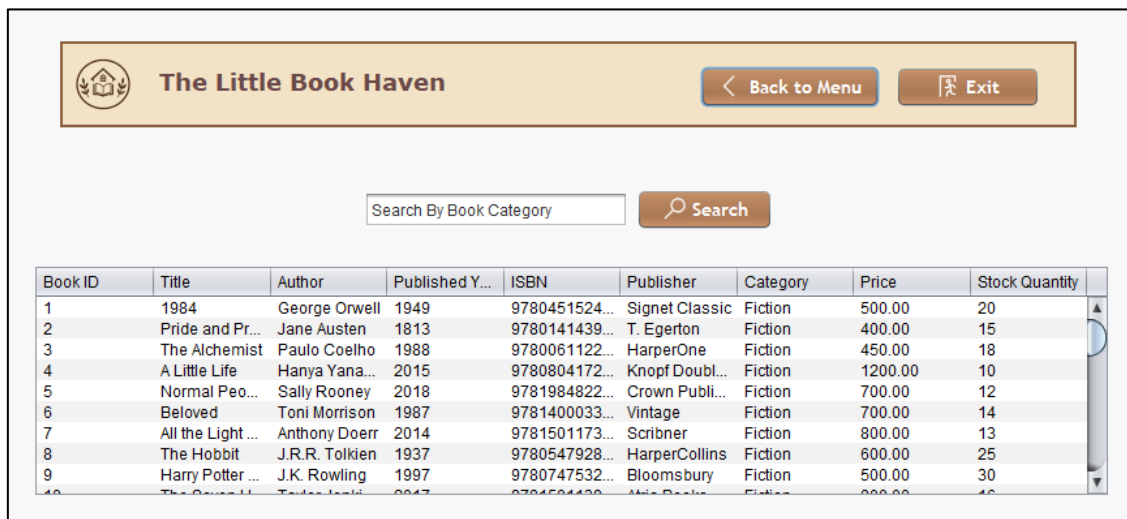
    public static void main(String args[]) {
    }
}

```

Figure 23 View-all-book-details

This functionality fetches all book details from the database and populates a table when “View All” is clicked, while “Clear” removes the data, and “Back to Menu” returns to the main menu.

2.3.4. Search Book by Category



```
private void loadAllBooks() {
    try {
        Connection conn = DBConnection.getConnection();
        String sql = "SELECT b.bookID, b.title, b.author, b.year, b.isbn, b.publisher, c.categoryName, b.price, b.stockQuantity FROM book b "
            + "JOIN category c ON b.categoryID = c.categoryID ORDER BY bookID ASC";
        PreparedStatement pst = conn.prepareStatement(sql);
        ResultSet rs = pst.executeQuery();

        DefaultTableModel model = (DefaultTableModel) searchList.getModel();
        model.setRowCount(0);

        while (rs.next()) {
            String[] row = {
                rs.getString("bookID"),
                rs.getString("title"),
                rs.getString("author"),
                String.valueOf(rs.getInt("year")),
                rs.getString("isbn"),
                rs.getString("publisher"),
                rs.getString("categoryName"),
                String.format("%.2f", rs.getDouble("price")),
                String.valueOf(rs.getInt("stockQuantity"))
            };
            model.addRow(row);
        }

        rs.close();
        pst.close();
        conn.close();
    } catch (Exception e) {
        JOptionPane.showMessageDialog(parentComponent, "Error loading books: " + e.getMessage());
    }
}

private void searchBtnActionPerformed(java.awt.event.ActionEvent evt) {
    String searchTerm = searchBar.getText().trim();

    DefaultTableModel model = (DefaultTableModel) searchList.getModel();
    TableRowSorter<DefaultTableModel> sorter = new TableRowSorter<>(model);
    searchList.setRowSorter(sorter);


    if (searchTerm.isEmpty()) {
        sorter.setRowFilter(filter: null);
    } else {
        RowFilter<DefaultTableModel, Object> filter = RowFilter.regexFilter("(?i)" + Pattern.quote(searchTerm), indices:6);
        sorter.setRowFilter(filter);


        if (searchList.getRowCount() == 0) {
            JOptionPane.showMessageDialog(parentComponent, "No books found for the category: " + searchTerm);
        }
    }
}
}
```

Figure 24 Search-book-by-category

This functionality pre-loads all book data into a table and filters it based on the category entered, displaying matches or an error message if no books are found.

2.3.5. Add New Book

**The Little Book Haven**[< Back to Menu](#)[Exit](#)



Add New Book

Complete the form to add a new book.

Book Title	Publisher
<input type="text"/>	<input type="text"/>
Author of the Book	Category
<input type="text"/>	<input type="text" value="Fiction"/>
Published Year	Price
<input type="text"/>	<input type="text"/>
ISBN	Stock Quantity
<input type="text"/>	<input type="text"/>

[Add](#)[Clear](#)

```

package littlebookhaven;

import java.sql.*;
import java.util.HashMap;
import java.util.Map;
import javax.swing.JOptionPane;

public class AddNewBookFrame extends javax.swing.JFrame {

    private static final java.util.logging.Logger logger = java.util.logging.Logger.getLogger(name: AddNewBookFrame.class.getName());

    private User loggedInUser;

    private Map<String, Integer> categoryMap = new HashMap<>();

    public AddNewBookFrame(User user) {
        this.loggedInUser = user;
        initComponents();
        try {
            Connection conn = DBConnection.getConnection();
            String query = "SELECT categoryID, categoryName FROM category";
            PreparedStatement pst = conn.prepareStatement(query);
            ResultSet rs = pst.executeQuery();

            category.removeAllItems();

            while (rs.next()) {
                int id = rs.getInt("categoryID");
                String name = rs.getString("categoryName");
                category.addItem(name);
                categoryMap.put(name, id);
            }

            rs.close();
            pst.close();
            conn.close();
        } catch (Exception ex) {
            JOptionPane.showMessageDialog(parentComponent: this, "Error loading categories: " + ex.getMessage());
        }
    }

    @SuppressWarnings("unchecked")
    Generated Code

    private void jButton4ActionPerformed(java.awt.event.ActionEvent evt) {
        title.setText("");
        author.setText("");
        year.setText("");
        isbn.setText("");
        publisher.setText("");
        price.setText("");
        stockQuantity.setText("");
    }

    private void jButton5ActionPerformed(java.awt.event.ActionEvent evt) {
        String bookTitle = title.getText().trim();
        String bookAuthor = author.getText().trim();
        String bookYearStr = year.getText().trim();
        String bookISBN = isbn.getText().trim();
        String bookPublisher = publisher.getText().trim();
        String bookPriceStr = price.getText().trim();
        String bookStockQuantityStr = stockQuantity.getText().trim();
        String selectedCategory = (String) category.getSelectedItem();

        if (bookTitle.isEmpty() || bookAuthor.isEmpty() || bookYearStr.isEmpty() || bookISBN.isEmpty() || bookPublisher.isEmpty()
            || bookPriceStr.isEmpty() || bookStockQuantityStr.isEmpty() || selectedCategory == null) {
            JOptionPane.showMessageDialog(parentComponent: this, message: "Please fill all fields.");
            return;
        }

        try {
            int bookYear = Integer.parseInt(bookYearStr);
            double bookPrice = Double.parseDouble(bookPriceStr);
            int bookStockQuantity = Integer.parseInt(bookStockQuantityStr);
            int categoryID = categoryMap.get(selectedCategory);

            Connection conn = DBConnection.getConnection();
            String sql = "INSERT INTO book (title, author, year, isbn, publisher, categoryID, price, stockQuantity) VALUES (?, ?, ?, ?, ?, ?, ?, ?)";
            PreparedStatement pst = conn.prepareStatement(sql);
            pst.setString(1, bookTitle);
            pst.setString(2, bookAuthor);
            pst.setInt(3, bookYear);
            pst.setString(4, bookISBN);
            pst.setString(5, bookPublisher);
            pst.setInt(6, categoryID);
            pst.setDouble(7, bookPrice);
            pst.setInt(8, bookStockQuantity);

            int rowsInserted = pst.executeUpdate();
            if (rowsInserted > 0) {
                JOptionPane.showMessageDialog(parentComponent: this, message: "Book added successfully!");
                title.setText("");
                author.setText("");
                year.setText("");
                isbn.setText("");
                publisher.setText("");
                price.setText("");
                stockQuantity.setText("");
                category.setSelectedIndex(0);
            }


            pst.close();
            conn.close();
        } catch (NumberFormatException e) {
            JOptionPane.showMessageDialog(parentComponent: this, message: "Please enter valid numbers for year, price, ISBN and Stock Quantity.");
        } catch (Exception ex) {
            JOptionPane.showMessageDialog(parentComponent: this, "Error adding book: " + ex.getMessage());
        }
    }
}

```

Figure 25 Add-New-Book

This functionality loads categories into a dropdown, collects book details from the form, validates all fields and numeric inputs, adds the book to the database if valid, and clears the form upon success.

2.3.6. Create Cashier Accounts

**The Little Book Haven**

< Back to Menu

Exit



Create Cashier Accounts

Existing User Accounts

User ID	Username	Role
1	shanuka	Manager
2	kasun	Cashier
3	dasun	Cashier
4	pasan	Cashier
5	mathesha	Cashier

Refresh



Enter New Cashier Details

Username

Password

☐ show

Chosse User Role

Cashier

Create

42

```

package littlebookhaven;

import java.sql.*;
import java.util.regex.Pattern;
import javax.swing.JOptionPane;
import javax.swing.JFrame;
import javax.swing.table.DefaultTableModel;
import javax.swing.table.TableRowSorter;
import javax.swing.RowFilter;

public class CreateCashierAccountFrame extends javax.swing.JFrame {

    private static final java.util.logging.Logger logger = java.util.logging.Logger.getLogger(CreateCashierAccountFrame.class.getName());

    private User loggedInUser;

    public CreateCashierAccountFrame(User user) {
        this.loggedInUser = user;
        initComponents();
        loadUsers();
    }

    private void loadUsers() {
        try {
            Connection conn = DBConnection.getConnection();
            String sql = "SELECT userID, username, role FROM user";
            PreparedStatement pst = conn.prepareStatement(sql);
            ResultSet rs = pst.executeQuery();

            DefaultTableModel model = (DefaultTableModel) userList.getModel();
            model.setRowCount(0);

            while (rs.next()) {
                String[] row = {
                    rs.getString("userID"),
                    rs.getString("username"),
                    rs.getString("role")
                };
                model.addRow(row);
            }

            rs.close();
            pst.close();
            conn.close();
        } catch (Exception e) {
            JOptionPane.showMessageDialog(this, "Error loading users: " + e.getMessage());
        }
    }

    private void jCheckBox1ActionPerformed(java.awt.event.ActionEvent evt) {
        if (jCheckBox1.isSelected()) {
            newPassword.setEchoChar((char)0);
        } else {
            newPassword.setEchoChar('x');
        }
    }

    private void jButton5ActionPerformed(java.awt.event.ActionEvent evt) {
        String username = newUsername.getText().trim();
        String password = newPassword.getText().trim();
        String role = (String) userRole.getSelectedItem();

        if (username.isEmpty() || password.isEmpty() || role == null) {
            JOptionPane.showMessageDialog(this, "Please fill all fields.");
            return;
        }

        try {
            Connection conn = DBConnection.getConnection();
            String sql = "INSERT INTO user (username, password, role) VALUES (?, ?, ?)";
            PreparedStatement pst = conn.prepareStatement(sql);
            pst.setString(1, username);
            pst.setString(2, password);
            pst.setString(3, role);

            int rowsInserted = pst.executeUpdate();
            if (rowsInserted > 0) {
                JOptionPane.showMessageDialog(this, "New user created successfully!");
                newUsername.setText("");
                newPassword.setText("");
                userRole.setSelectedIndex(0);
            } else {
                JOptionPane.showMessageDialog(this, "Failed to create user.");
            }
        } catch (SQLException ex) {
            JOptionPane.showMessageDialog(this, "Error: " + ex.getMessage());
        }

        pst.close();
        conn.close();
    }

    private void jButton3ActionPerformed(java.awt.event.ActionEvent evt) {
        loadUsers();
    }

    private void jButton2ActionPerformed(java.awt.event.ActionEvent evt) {
        MainMenu mm = new MainMenu(authenticatedUser: loggedInUser);
        mm.setVisible(true);
        this.setVisible(false);
    }

    private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
        System.exit(0);
    }

    public static void main(String args[]) {
    }
}

```

Figure 26 Create-cashier-account

This functionality loads existing user accounts into a table, allows input of a new username and password with a fixed Cashier role, adds the account to the database upon creation, clears the form, and refreshes the table to reflect the update.

Task 03: User Manual

3. Introduction

Welcome to the “The Little Book Haven” system, a user-friendly bookstore management application designed to automate daily operations and enhance customer service. This system allows users to manage books, search inventory, and handle user accounts efficiently. This user manual will guide Cashiers and Managers step-by-step through the interfaces, from the loading screen to the final actions, with clear instructions and screenshots to help you navigate and use the system effectively.

3.1. Loading Screen

When the application launches, a loading screen with a progress bar and messages shows system initialization steps like loading resources and connecting to the database. After a few seconds, the window disappears, and the login screen appears automatically.

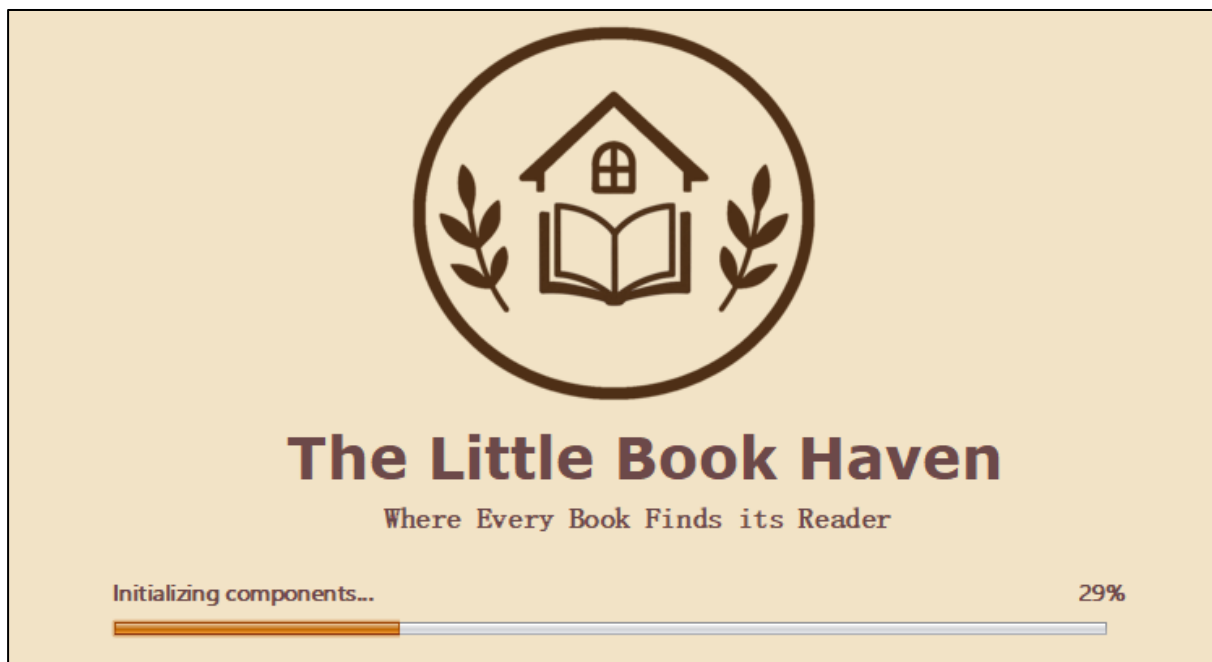


Figure 27 Loading Screen

3.2. Login Screen

When the login screen appears, a form with fields for username, password, and a dropdown to select the user type (Cashier or Manager) is displayed.

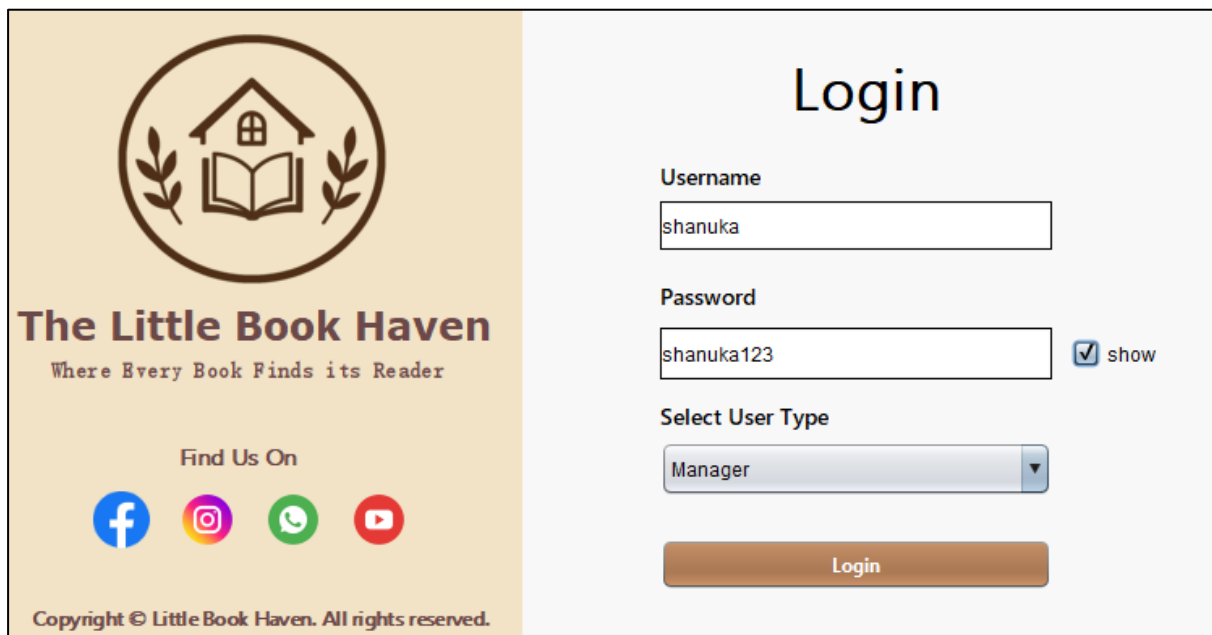
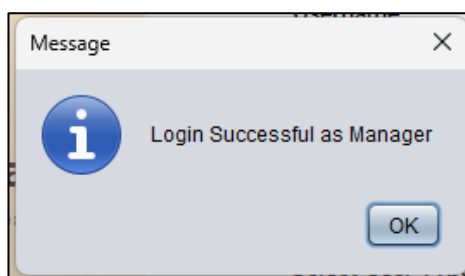
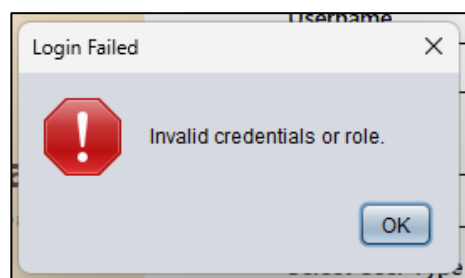
The login interface is divided into two main sections. The left section has a light orange background and features a circular logo with a house and an open book, flanked by laurel leaves. Below the logo is the text 'The Little Book Haven' and the tagline 'Where Every Book Finds its Reader'. Further down are social media icons for Facebook, Instagram, WhatsApp, and YouTube, with the text 'Find Us On' above them. At the bottom, it says 'Copyright © Little Book Haven. All rights reserved.' The right section has a light gray background and is titled 'Login'. It contains three input fields: 'Username' with the value 'shanuka', 'Password' with the value 'shanuka123', and a 'Select User Type' dropdown menu currently set to 'Manager'. To the right of the password field is a checkbox labeled 'show'. At the bottom of the right section is a large brown 'Login' button.

Figure 28 Login Interface

Users must enter their credentials, with an option to click “Show” next to the password field to reveal the hidden characters (e.g., from * to plain text), choose their role, and click the “Login” button to proceed.



- **Success Message:** When correct credentials are entered and Login is clicked, a message “Login Successful” with the correct user type (Cashier or Manager) is displayed.



- **Error Message:** If wrong credentials are entered, a message “Invalid credentials or role” is displayed and the user will need to re-enter the correct credentials to log in again.

3.3. Main Menu

When login is successful, the main menu opens with the bookstore logo and name “The Little Book Haven” on the left, displaying a welcome message with the logged-in username and their role. Below these, “Logout” returns the user back to the login page, and “Exit” safely closes the system. On the right, a menu with buttons for “View Book Details”, “Search Book”, “Add New Book”, and “Create Cashier Account” allows navigation to other interfaces and perform specific operations.

3.3.1. Manager Dashboard

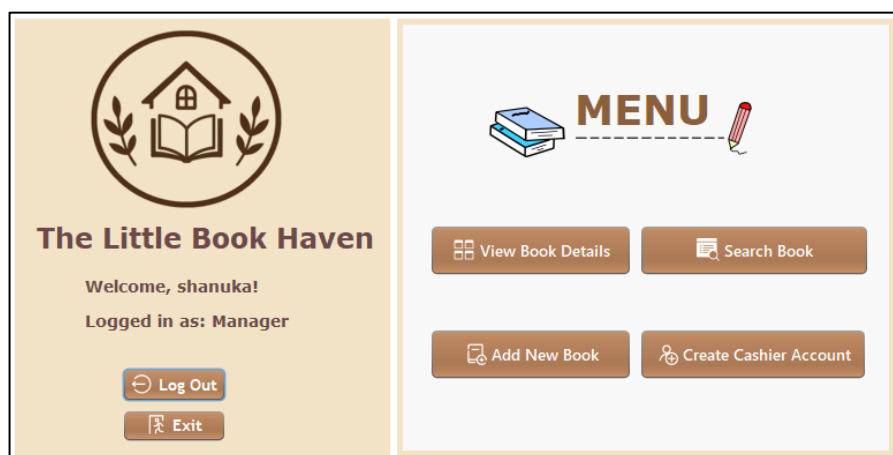


Figure 29 Manager-Menu

3.3.2. Cashier Dashboard


The Cashier menu is similar, showing a welcome message with the logged-in username and “Cashier” role. On the right, it includes all buttons for book operations but excludes the “Create Cashier Accounts” button.



Figure 30 Cashier-Menu

3.4. View Book List

When the “View Book Details” button is clicked from the main menu, this interface opens with a top section featuring “Back to Menu” and “Exit” buttons. “Back to Menu” returns the user to the main menu, while “Exit” safely closes the system. Below, a table displays all book details, including book ID, title, author, year, ISBN, publisher, category, price, and stock quantity. Clicking “View All” populates the table with book data, and the “Clear” button removes the details, allowing “View All” to reload the data when clicked again.

**The Little Book Haven**

[< Back to Menu](#)[Exit](#)

View All Book Details

Book ID	Title	Author	Published Ye...	ISBN	Publisher	Category	Price	Stock Quantity
1	1984	George Orwell	1949	9780451524...	Signet Classic	Fiction	500.00	20
2	Pride and Pr...	Jane Austen	1813	9780141439...	T. Egerton	Fiction	400.00	15
3	The Alchemist	Paulo Coelho	1988	9780061122...	HarperOne	Fiction	450.00	18
4	A Little Life	Hanya Yanag...	2015	9780804172...	Knopf Doubl...	Fiction	1200.00	10
5	Normal Peop...	Sally Rooney	2018	9781984822...	Crown Publi...	Fiction	700.00	12
6	Beloved	Toni Morrison	1987	9781400033...	Vintage	Fiction	700.00	14
7	All the Light ...	Anthony Doerr	2014	9781501173...	Scribner	Fiction	800.00	13
8	The Hobbit	J.R.R. Tolkien	1937	9780547928...	HarperCollins	Fiction	600.00	25
9	Harry Potter ...	J.K. Rowling	1997	9780747532...	Bloomsbury	Fiction	500.00	30
10	The Grapes of ...	John Steinbeck	1939	9781501173...	Scribner	Fiction	800.00	13

[View All](#)[Clear](#)

Figure 31 Books List

3.5. Search Book by Category

When the “Search Book” button is clicked from the main menu, this interface opens with a table already populated with book details.

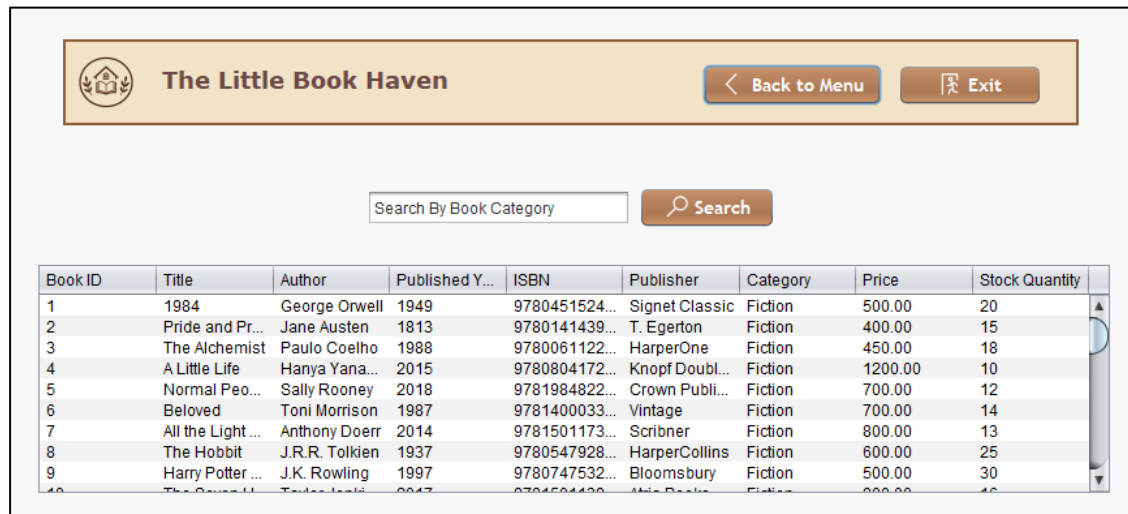
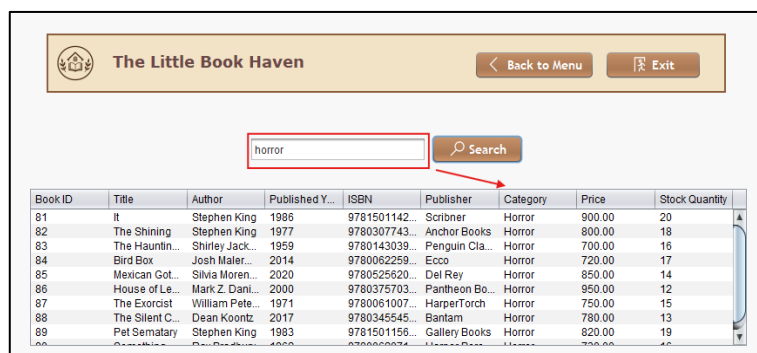
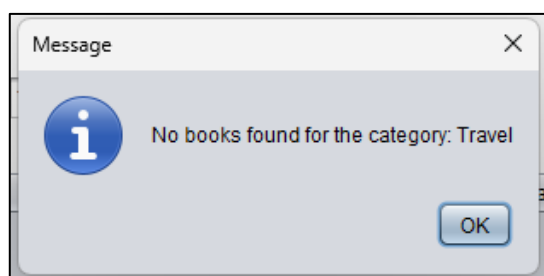


Figure 32 Filter Books



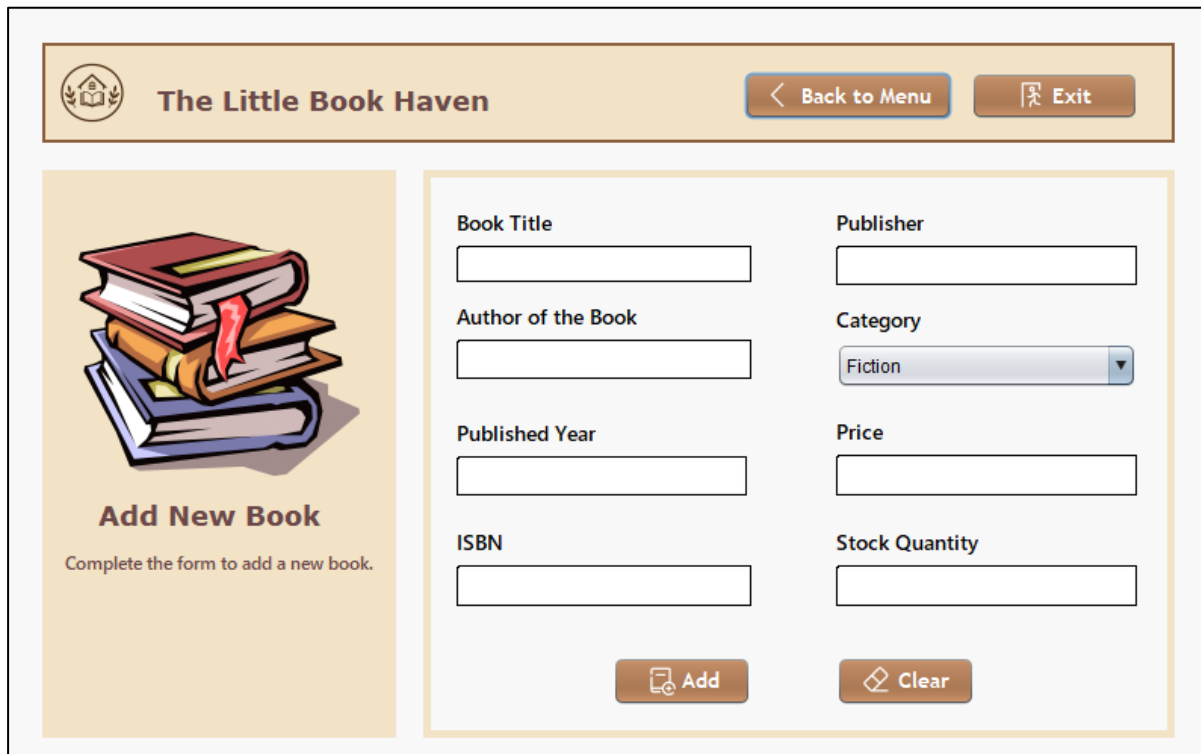
- **Results found:** Users type the category and click the “Search” button to filter the results.



- **Results not found:** If no results are found for the searched category, an error message “No books found for category: [category user typed]” is displayed.

3.6. Add New Book

When the “Add New Book” button is clicked from the main menu, this interface opens with a form to fill in book details.



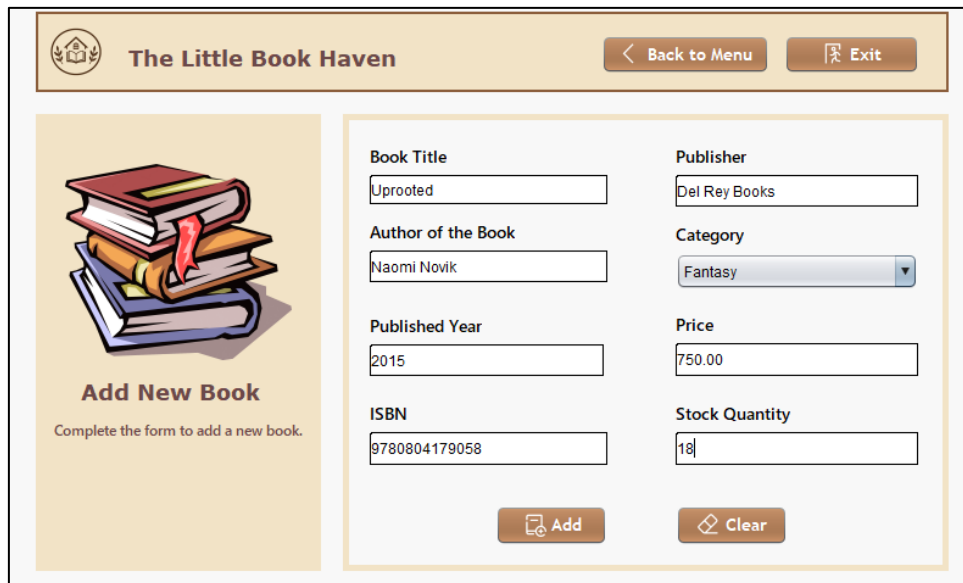
The screenshot displays the 'Add New Book' interface for 'The Little Book Haven'. The interface is divided into two main sections. On the left, there is a vertical panel with a light orange background. It features an illustration of a stack of books with a red bookmark, the title 'Add New Book' in bold, and the instruction 'Complete the form to add a new book.' in a smaller font. On the right, there is a form area with a light gray background. At the top of this area, there are two buttons: 'Back to Menu' with a left arrow icon and 'Exit' with a door icon. The form contains eight input fields arranged in two columns. The left column includes fields for 'Book Title', 'Author of the Book', 'Published Year', and 'ISBN'. The right column includes fields for 'Publisher', 'Category' (a dropdown menu currently showing 'Fiction'), 'Price', and 'Stock Quantity'. At the bottom of the form, there are two buttons: 'Add' with a plus icon and 'Clear' with an eraser icon.

Field	Type
Book Title	Text Input
Author of the Book	Text Input
Published Year	Text Input
ISBN	Text Input
Publisher	Text Input
Category	Dropdown Menu (Fiction)
Price	Text Input
Stock Quantity	Text Input

Figure 33 Add new book interface

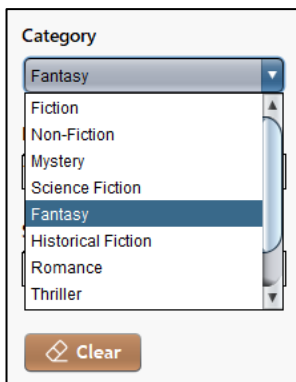
3.6.1. Adding the book

After filling in all details correctly and selecting a category from the dropdown to categorize the book, user should click the “Add” button.



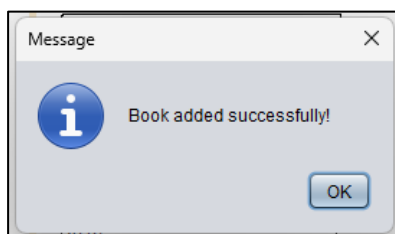
The screenshot shows a web application titled "The Little Book Haven". On the left, there is a graphic of a stack of books with the text "Add New Book" and "Complete the form to add a new book." On the right, there is a form with the following fields: "Book Title" (text input with "Uprooted"), "Publisher" (text input with "Del Rey Books"), "Author of the Book" (text input with "Naomi Novik"), "Category" (dropdown menu with "Fantasy" selected), "Published Year" (text input with "2015"), "Price" (text input with "750.00"), "ISBN" (text input with "9780804179058"), and "Stock Quantity" (text input with "18"). At the bottom of the form are two buttons: "Add" and "Clear". At the top right of the application are two buttons: "Back to Menu" and "Exit".

Figure 34 Filling Book Form



This screenshot shows a close-up of the "Category" dropdown menu. The menu is open, displaying a list of categories: "Fantasy", "Fiction", "Non-Fiction", "Mystery", "Science Fiction", "Fantasy", "Historical Fiction", "Romance", and "Thriller". The "Fantasy" option is currently selected and highlighted. Below the list is a "Clear" button.

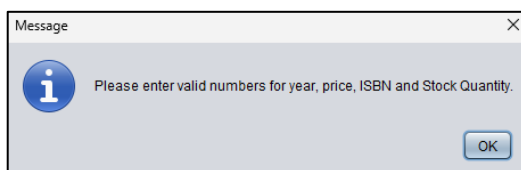
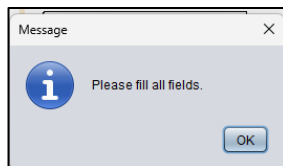
- Categorize book from the dropdown.



- If everything is valid, the book is added to the database successfully.

The Little Book Haven								
View All Book Details								
Book ID	Title	Author	Published Year	ISBN	Publisher	Category	Price	Stock Quantity
93	The Cat in th...	Dr. Seuss	1957	9780394800...	Random Ho...	Children's	450.00	22
94	Where the Wil...	Maurice Sen...	1963	9780064431...	HarperCollins	Children's	470.00	18
95	The Tale of P...	Beatrix Potter	1902	9780723247...	Warne	Children's	430.00	24
96	The Very Hun...	Eric Carle	1969	9780399225...	Philomel Bo...	Children's	400.00	28
97	Green Eggs ...	Dr. Seuss	1960	9780394800...	Random Ho...	Children's	450.00	26
98	Winnie-the-P...	A.A. Milne	1925	9780525417...	Dutton Juven...	Children's	520.00	19
99	Harry Potter ...	J.K. Rowling	1997	9780747532...	Bloomsbury	Children's	580.00	30
100	Diary of a W...	Jeff Kinney	2007	9780810993...	Amulet Books	Children's	480.00	23
101	Uprooted	Naomi Novik	2015	9780804179...	Del Rey Books	Fantasy	750.00	18

- Users can verify the addition of the book by navigating to the “View Book Details” interface to see the new book listed.



- If all fields are left empty, an error message “Please fill all fields” is displayed. If price, ISBN, year, or stock quantities are entered but not valid, an error message to enter valid numbers for these fields is shown.

3.7. Create Cashier Accounts

When the “Create Cashier Account” button is clicked from the main menu (available only for Managers), this interface opens with a form for the Manager to create a new cashier account.

The Little Book Haven
Back to Menu
Exit

Create Cashier Accounts

User ID	Username	Role
1	shanuka	Manager
2	kasun	Cashier
3	dasun	Cashier
4	pasan	Cashier
5	mathesha	Cashier

Refresh

Enter New Cashier Details

Username

Password
show

Chosse User Role

Cashier

Create

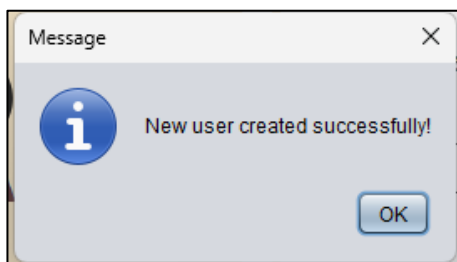
Figure 35 Cashier-account-creation Interface

3.7.1. Creating new cashier account

The Manager should enter a new username, password, and select the role “Cashier” and click the “Create” button to add the new Cashier to the database, creating their account.

User ID	Username	Role
1	shanuka	Manager
2	kasun	Cashier
3	dasun	Cashier
4	pasan	Cashier
5	mathesha	Cashier

Figure 36 Filling New Cashier Details



- If everything is valid, the new cashier is added to the database successfully.



- On the left side, a table displays existing user accounts. To verify the addition, the Manager can click the “Refresh” button to reload the table and see the new Cashier account.

This user manual provides a clear guide to navigating the “The Little Book Haven” system, from the loading screen to creating cashier accounts. By following the step-by-step instructions and referring to the screenshots, both Cashiers and Managers can efficiently manage book inventory and user accounts. For any issues, contact your system administrator. Enjoy using the system to streamline your bookstore operations.

Conclusion

The development of the “Little Book Haven” system successfully demonstrates how Object-Oriented Programming and systematic design using UML diagrams can result in an efficient, reliable, and user-friendly application. UML diagrams guided the structure and workflow, while Java and MySQL enabled practical implementation. Key OOP concepts such as abstraction, encapsulation, inheritance, and polymorphism were applied to create a maintainable and extensible system. The functionalities were built to meet the specific needs of both cashiers and managers, making book and user management streamlined and error-resistant. With the support of the user manual, this system is ready for use and scalable enhancements in the future.

References

Booch, G., 2005. *The Unified Modeling Language User Guide*. 2nd ed. s.l.:Pearson Education.

Fowler, M., 2004. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. 3rd ed. s.l.:Addison-Wesley Professional.

GeeksForGeeks, 2025. *Abstraction in Java*. [Online] Available at: <https://www.geeksforgeeks.org/java/abstraction-in-java-2/> [Accessed 27 June 2025].

GeeksForGeeks, 2025. *Java OOP(Object Oriented Programming) Concepts*. [Online] Available at: <https://www.geeksforgeeks.org/java/object-oriented-programming-oops-concept-in-java/> [Accessed 27 June 2025].

GeeksForGeeks, 2025. *Polymorphism in Java*. [Online] Available at: <https://www.geeksforgeeks.org/java/polymorphism-in-java/> [Accessed 27 June 2025].

Horstmann, C., 2005. *Object-Oriented Design & Patterns*. 2nd ed. s.l.:John Wiley & Sons Inc. [Online] Available at: https://amudhasjavatechnology.wordpress.com/wp-content/uploads/2015/10/objectorienteddesignnpatterns_cayhorstmann_2nd.pdf [Accessed 30 June 2025].

Raut, R. S., 2020. Research Paper on Object-Oriented Programming (OOP). *International Research Journal of Engineering and Technology (IRJET)*, 07(10), p. 1456. [Online] Available at: <https://www.irjet.net/archives/V7/i10/IRJET-V7I10247.pdf> [Accessed 30 June 2025].

Visual Paradigm, 2025. *What is Class Diagram?*. [Online] Available at: <https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-class-diagram/> [Accessed 29 June 2025].

Weisfeld, M., 2008. *The Object-Oriented Thought Process*. 3rd ed. s.l.:Addison-Wesley Professional.

Appendix

Gantt Chart

Task No.	Task Description	2025-06-21	2025-06-23	2025-06-23	2025-06-24	2025-06-25	2025-06-26	2025-06-27	2025-06-28	2025-06-29	2025-06-30	2025-07-01	2025-07-02	2025-07-03
	Planning & Research													
	Requirement Analysis													
1	System Design (Task 01)													
	Plan UML designs													
	Create Use case, Class & Sequence Diagrams													
	Documenting the designs and justifying													
2	System Development (Task 02)													
	Implementation setup													
	Coding core features & developing interfaces													
	Testing & Debugging													
	Documenting OOP concepts and the main system functionalities with screenshot evidences													
3	User manual (Task 03)													
	Documenting user interfaces, error messages with screenshots													
4	Documentation & Review													
	Report writing and formatting													
	Final Review & Submission													

Figure 37 Appendix A-Gantt Chart