# JavaFX Bookshop Application

## Table of Contents

1. [Introduction](#Xe3d0fc0bea9a42ce7605565d0964033d7f6ee47)
2. [Architecture](#X136a724f0ad6bedecc70d52a088da11b9355002)
3. [Design Patterns](#X9183dafefd8661a8c485a3331ef0a5ed8880e04)
4. [Features](#X55ace6873480125b3f0fc8b8d6c358074359e92)
5. [Database Design](#X6b97c217ebc75afcba20d43deee6c8307895657)
6. [Testing](#X892c32357cd0b9486fd4d79be9995276fe6bf29)
7. [Security](#Xda00d8566dd32e77b002d742b5b3e934dd60c1c)
8. [UML Diagrams](#Xb6c3eb7bfffdf64639c49c317306d287a241699)
9. [Conclusion](#Xf084a3aad2ba019113a3f84a3b857d0628c92ce)

## 1. Introduction

### Purpose and Overview

A comprehensive desktop application for an online bookshop enabling book browsing, purchase, and management.

### Target Users

* **Customers**: Browse books, manage shopping carts, place orders, write reviews
* **Administrators**: Manage inventory, view customer orders, moderate reviews

### Technologies Used

* **Java 17**: Core programming language
* **JavaFX and FXML**: UI construction and design
* **PostgreSQL**: Persistent data storage
* **BCrypt**: Secure password hashing
* **Maven**: Dependency management and build automation
* **JUnit 5 and Mockito**: Testing framework and mocking library

### Core Principles

#### Object-Oriented Programming (OOP)

The application is built on fundamental OOP principles: - **Encapsulation**: Data and methods are encapsulated within classes with appropriate access modifiers - **Inheritance**: Utilized for extending functionality in related classes - **Polymorphism**: Employed through interfaces and method overriding, particularly in the discount and payment system - **Abstraction**: Complex implementations are abstracted behind clean interfaces

#### Model-View-Controller (MVC) Architecture

The application separates concerns into three interconnected components: - **Model**: Represents data and business logic (Book, User, Order classes) - **View**: User interface that displays data (FXML files, JavaFX components) - **Controller**: Mediates between Model and View (Controller classes)

## 2. Architecture

### Project Structure

src/main/java/com/bookshop/  
├── models/ # Data models (Book, User, CartItem, etc.)  
├── controllers/ # JavaFX controllers for FXML views  
├── services/ # Business logic and operations  
├── repositories/ # Data access layer  
├── utils/ # Utility classes  
├── factory/ # Factory classes for object creation  
├── views/ # FXML view files  
└── Main.java # Application entry point

### Architectural Layers

1. **Model Layer**: Core business objects and logic
2. **Repository Layer**: Data persistence and retrieval
3. **Service Layer**: Business operations and rules
4. **Controller Layer**: User input handling and view updates
5. **View Layer**: User interface components

## 3. Design Patterns

### Pattern Selection Rationale

The application employs a combination of Gang of Four (GoF) and non-traditional patterns to address specific architectural and behavioral challenges. Each pattern was carefully selected based on the problem it solves and the benefits it provides to the application’s maintainability, scalability, and performance. The patterns were chosen to solve real-world problems in the bookshop context, such as managing complex object creation, handling real-time updates, and implementing flexible business rules.

### GoF Patterns

#### Factory Pattern

**Purpose**: The Factory Pattern was chosen to centralize and standardize object creation, particularly for complex objects like Books and Users. This pattern helps maintain consistency in object initialization, reduces code duplication, and makes it easier to modify object creation logic in one place. It’s especially useful when creating objects that require multiple steps or have complex initialization requirements. In our bookshop application, we use it extensively for creating Book objects from various sources (database, user input, DTOs) while ensuring consistent initialization and validation.

**Implementation**:

public class BookFactory {  
 // Creates a basic book with essential information  
 public static Book createBook(String title, String author, double price) {  
 Book book = new Book();  
 book.setTitle(title);  
 book.setAuthor(author);  
 book.setPrice(price);  
 book.setStockQuantity(0); // Default stock  
 book.setCreatedAt(LocalDateTime.now());  
 return book;  
 }  
   
 // Creates a book from a DTO with full details  
 public static Book createFromDTO(BookDTO dto) {  
 Book book = new Book();  
 book.setTitle(dto.getTitle());  
 book.setAuthor(dto.getAuthor());  
 book.setIsbn(dto.getIsbn());  
 book.setPublisher(dto.getPublisher());  
 book.setCategory(dto.getCategory());  
 book.setDescription(dto.getDescription());  
 book.setPrice(dto.getPrice());  
 book.setStockQuantity(dto.getStockQuantity());  
 book.setImageUrl(dto.getImageUrl());  
 book.setCreatedAt(LocalDateTime.now());  
 return book;  
 }  
   
 // Creates a book from database result set  
 public static Book createFromResultSet(ResultSet rs) throws SQLException {  
 Book book = new Book();  
 book.setId(rs.getInt("id"));  
 book.setTitle(rs.getString("title"));  
 book.setAuthor(rs.getString("author"));  
 // ... set other properties  
 return book;  
 }  
}

**Roles**: - BookFactory: Creator class that handles object creation with multiple creation methods - Book: Product class being created with complex initialization requirements - BookDTO: Data transfer object used for creation from external sources - ResultSet: Database result set used for creation from database records

**Usage Scenarios**: 1. Creating new books from admin input 2. Converting DTOs from API responses 3. Loading books from database results 4. Creating test books for unit testing

#### Singleton Pattern

**Purpose**: The Singleton Pattern was implemented for managing global resources like database connections and session state. This ensures that we have exactly one instance of these critical resources throughout the application’s lifecycle, preventing resource conflicts and maintaining consistent state. It’s particularly important for managing database connections to avoid connection pool exhaustion and for maintaining user session state across different parts of the application. The pattern helps us maintain a single source of truth for critical application-wide resources.

**Implementation**:

public class DatabaseConnection {  
 private static volatile DatabaseConnection instance;  
 private Connection connection;  
 private static final String DB\_URL = "jdbc:postgresql://localhost:5432/bookshop";  
 private static final String DB\_USER = "bookshop\_user";  
 private static final String DB\_PASSWORD = "secure\_password";  
   
 private DatabaseConnection() {  
 try {  
 connection = DriverManager.getConnection(DB\_URL, DB\_USER, DB\_PASSWORD);  
 } catch (SQLException e) {  
 throw new RuntimeException("Failed to create database connection", e);  
 }  
 }  
   
 public static DatabaseConnection getInstance() {  
 if (instance == null) {  
 synchronized (DatabaseConnection.class) {  
 if (instance == null) {  
 instance = new DatabaseConnection();  
 }  
 }  
 }  
 return instance;  
 }  
   
 public Connection getConnection() {  
 return connection;  
 }  
   
 public void closeConnection() {  
 try {  
 if (connection != null && !connection.isClosed()) {  
 connection.close();  
 }  
 } catch (SQLException e) {  
 throw new RuntimeException("Failed to close database connection", e);  
 }  
 }  
}

**Roles**: - DatabaseConnection: Singleton class managing the single instance with thread-safe initialization - Connection: Resource being managed (database connection) - getInstance(): Static method providing global access with double-checked locking - closeConnection(): Method for proper resource cleanup

**Usage Scenarios**: 1. Database operations across the application 2. Connection pooling management 3. Transaction management 4. Resource cleanup on application shutdown

#### Observer Pattern

**Purpose**: The Observer Pattern was chosen to implement real-time updates in the shopping cart and order processing systems. This pattern allows objects to subscribe to changes in other objects without tight coupling, making it perfect for scenarios where multiple UI components need to update when the cart changes or when order status updates occur. It promotes loose coupling and makes the system more maintainable and extensible. In our bookshop, this pattern is crucial for keeping the UI in sync with the underlying data model and providing real-time feedback to users.

**Implementation**:

// Observer interface  
public interface CartObserver {  
 void onCartUpdated(CartEvent event);  
 void onItemAdded(CartItem item);  
 void onItemRemoved(CartItem item);  
 void onQuantityChanged(CartItem item, int newQuantity);  
}  
  
// Subject class  
public class CartManager {  
 private List<CartObserver> observers = new ArrayList<>();  
 private Map<Integer, CartItem> items = new HashMap<>();  
   
 public void addObserver(CartObserver observer) {  
 observers.add(observer);  
 }  
   
 public void removeObserver(CartObserver observer) {  
 observers.remove(observer);  
 }  
   
 public void addItem(Book book, int quantity) {  
 CartItem item = new CartItem(book, quantity);  
 items.put(book.getId(), item);  
 notifyItemAdded(item);  
 }  
   
 public void removeItem(int bookId) {  
 CartItem item = items.remove(bookId);  
 if (item != null) {  
 notifyItemRemoved(item);  
 }  
 }  
   
 public void updateQuantity(int bookId, int newQuantity) {  
 CartItem item = items.get(bookId);  
 if (item != null) {  
 item.setQuantity(newQuantity);  
 notifyQuantityChanged(item, newQuantity);  
 }  
 }  
   
 private void notifyItemAdded(CartItem item) {  
 observers.forEach(observer -> observer.onItemAdded(item));  
 }  
   
 private void notifyItemRemoved(CartItem item) {  
 observers.forEach(observer -> observer.onItemRemoved(item));  
 }  
   
 private void notifyQuantityChanged(CartItem item, int newQuantity) {  
 observers.forEach(observer -> observer.onQuantityChanged(item, newQuantity));  
 }  
}  
  
// Concrete Observer  
public class CartViewController implements CartObserver {  
 private CartManager cartManager;  
 private Label totalLabel;  
 private ListView<CartItem> cartListView;  
   
 public CartViewController(CartManager cartManager) {  
 this.cartManager = cartManager;  
 cartManager.addObserver(this);  
 }  
   
 @Override  
 public void onCartUpdated(CartEvent event) {  
 updateTotal();  
 refreshCartList();  
 }  
   
 @Override  
 public void onItemAdded(CartItem item) {  
 cartListView.getItems().add(item);  
 updateTotal();  
 }  
   
 @Override  
 public void onItemRemoved(CartItem item) {  
 cartListView.getItems().remove(item);  
 updateTotal();  
 }  
   
 @Override  
 public void onQuantityChanged(CartItem item, int newQuantity) {  
 updateTotal();  
 refreshCartList();  
 }  
   
 private void updateTotal() {  
 BigDecimal total = cartManager.calculateTotal();  
 totalLabel.setText(total.toString());  
 }  
   
 private void refreshCartList() {  
 cartListView.refresh();  
 }  
}

**Roles**: - CartObserver: Observer interface defining update contract with specific event types - CartManager: Subject class managing observers and cart state - CartViewController: Concrete observer implementing UI updates - CartItem: Data object representing cart items - CartEvent: Event object containing change information

**Usage Scenarios**: 1. Real-time cart total updates 2. Cart item list synchronization 3. Order status notifications 4. Inventory update notifications

#### Strategy Pattern

**Purpose**: The Strategy Pattern was implemented to handle the different discount calculation strategies for various user types. This pattern allows us to encapsulate each discount algorithm in its own class and switch between them at runtime, making it easy to add new discount types or modify existing ones without changing the core checkout logic. It’s particularly useful for the loyalty program where different user tiers get different discounts. The pattern helps us maintain clean separation between the discount calculation logic and the checkout process.

**Implementation**:

// Strategy interface  
public interface DiscountStrategy {  
 BigDecimal calculateDiscount(BigDecimal originalPrice);  
 String getDescription();  
}  
  
// Concrete strategies  
public class NoDiscount implements DiscountStrategy {  
 @Override  
 public BigDecimal calculateDiscount(BigDecimal originalPrice) {  
 return BigDecimal.ZERO;  
 }  
   
 @Override  
 public String getDescription() {  
 return "No discount applied";  
 }  
}  
  
public class RegularMemberDiscount implements DiscountStrategy {  
 private static final BigDecimal DISCOUNT\_RATE = new BigDecimal("0.10");  
   
 @Override  
 public BigDecimal calculateDiscount(BigDecimal originalPrice) {  
 return originalPrice.multiply(DISCOUNT\_RATE);  
 }  
   
 @Override  
 public String getDescription() {  
 return "Regular member discount (10%)";  
 }  
}  
  
public class PremiumMemberDiscount implements DiscountStrategy {  
 private static final BigDecimal DISCOUNT\_RATE = new BigDecimal("0.15");  
   
 @Override  
 public BigDecimal calculateDiscount(BigDecimal originalPrice) {  
 return originalPrice.multiply(DISCOUNT\_RATE);  
 }  
   
 @Override  
 public String getDescription() {  
 return "Premium member discount (15%)";  
 }  
}  
  
// Context class  
public class DiscountService {  
 public DiscountStrategy getDiscountStrategy(User user) {  
 if (user == null) {  
 return new NoDiscount();  
 }  
   
 if (user.getOrderCount() >= 10) {  
 return new PremiumMemberDiscount();  
 } else if (user.getOrderCount() >= 5) {  
 return new RegularMemberDiscount();  
 }  
   
 return new NoDiscount();  
 }  
   
 public BigDecimal calculateDiscountedPrice(BigDecimal originalPrice, User user) {  
 DiscountStrategy strategy = getDiscountStrategy(user);  
 BigDecimal discount = strategy.calculateDiscount(originalPrice);  
 return originalPrice.subtract(discount);  
 }  
}

**Roles**: - DiscountStrategy: Strategy interface defining discount calculation contract - NoDiscount, RegularMemberDiscount, PremiumMemberDiscount: Concrete strategy implementations - DiscountService: Context class selecting and applying strategies - User: Context object used for strategy selection

**Usage Scenarios**: 1. Checkout process discount calculation 2. Order preview with estimated discounts 3. Loyalty program status display 4. Discount strategy testing and validation

### Non-Traditional Patterns

#### Repository Pattern

**Purpose**: The Repository Pattern was chosen to abstract the data access layer and provide a clean interface for database operations. This pattern helps separate the business logic from data access concerns, making the code more maintainable and testable. It also provides a consistent way to access data across the application, regardless of the underlying storage mechanism. In our bookshop, this pattern is essential for managing complex database operations while keeping the business logic clean and focused.

**Implementation**:

// Repository interface  
public interface BookRepository {  
 List<Book> findAll();  
 Book findById(int id);  
 List<Book> findByAuthor(String author);  
 List<Book> findByCategory(String category);  
 List<Book> search(String query);  
 void save(Book book);  
 void update(Book book);  
 void delete(int id);  
 boolean existsById(int id);  
}  
  
// Concrete implementation  
public class BookRepositoryImpl implements BookRepository {  
 private final DatabaseConnection dbConnection;  
   
 public BookRepositoryImpl() {  
 this.dbConnection = DatabaseConnection.getInstance();  
 }  
   
 @Override  
 public List<Book> findAll() {  
 String sql = "SELECT \* FROM books ORDER BY title";  
 try (Connection conn = dbConnection.getConnection();  
 PreparedStatement stmt = conn.prepareStatement(sql);  
 ResultSet rs = stmt.executeQuery()) {  
   
 List<Book> books = new ArrayList<>();  
 while (rs.next()) {  
 books.add(BookFactory.createFromResultSet(rs));  
 }  
 return books;  
 } catch (SQLException e) {  
 throw new RuntimeException("Failed to fetch books", e);  
 }  
 }  
   
 @Override  
 public Book findById(int id) {  
 String sql = "SELECT \* FROM books WHERE id = ?";  
 try (Connection conn = dbConnection.getConnection();  
 PreparedStatement stmt = conn.prepareStatement(sql)) {  
   
 stmt.setInt(1, id);  
 try (ResultSet rs = stmt.executeQuery()) {  
 if (rs.next()) {  
 return BookFactory.createFromResultSet(rs);  
 }  
 return null;  
 }  
 } catch (SQLException e) {  
 throw new RuntimeException("Failed to fetch book", e);  
 }  
 }  
   
 // ... other method implementations  
}

**Roles**: - BookRepository: Repository interface defining data access methods - BookRepositoryImpl: Concrete implementation handling database operations - Book: Domain entity being managed - DatabaseConnection: Singleton providing database access

**Usage Scenarios**: 1. Book search and retrieval 2. Inventory management 3. Data persistence operations 4. Transaction management

#### Facade Pattern

**Purpose**: The Facade Pattern was implemented to simplify complex operations like the checkout process. This pattern provides a unified interface to a set of interfaces in a subsystem, making it easier to use. In our case, it hides the complexity of payment processing, order creation, and inventory management behind a simple checkout interface, making the code more maintainable and the system easier to use. The pattern helps us manage complex business processes while keeping the client code simple and focused.

**Implementation**:

public class CheckoutFacade {  
 private final PaymentService paymentService;  
 private final OrderService orderService;  
 private final InventoryService inventoryService;  
 private final NotificationService notificationService;  
   
 public CheckoutFacade() {  
 this.paymentService = new PaymentService();  
 this.orderService = new OrderService();  
 this.inventoryService = new InventoryService();  
 this.notificationService = new NotificationService();  
 }  
   
 public Order processCheckout(ShoppingCart cart, PaymentInfo paymentInfo, User user) {  
 try {  
 // Validate cart and payment  
 validateCart(cart);  
 validatePayment(paymentInfo);  
   
 // Process payment  
 PaymentResult paymentResult = paymentService.processPayment(paymentInfo, cart.getTotal());  
 if (!paymentResult.isSuccessful()) {  
 throw new PaymentException("Payment failed: " + paymentResult.getMessage());  
 }  
   
 // Create order  
 Order order = orderService.createOrder(cart, user, paymentInfo);  
   
 // Update inventory  
 inventoryService.updateStockForOrder(order);  
   
 // Clear cart  
 cart.clear();  
   
 // Send notifications  
 notificationService.sendOrderConfirmation(order, user);  
 notificationService.sendInventoryUpdate(order);  
   
 return order;  
 } catch (Exception e) {  
 // Handle and log the error  
 logError(e);  
 throw new CheckoutException("Checkout failed: " + e.getMessage());  
 }  
 }  
   
 private void validateCart(ShoppingCart cart) {  
 if (cart.isEmpty()) {  
 throw new ValidationException("Cart is empty");  
 }  
 // Additional validation logic  
 }  
   
 private void validatePayment(PaymentInfo paymentInfo) {  
 // Payment validation logic  
 }  
   
 private void logError(Exception e) {  
 // Error logging logic  
 }  
}

**Roles**: - CheckoutFacade: Facade class providing simplified interface - PaymentService, OrderService, InventoryService, NotificationService: Subsystem classes - ShoppingCart, PaymentInfo, User: Data objects used in the process - Order: Result object returned from the process

**Usage Scenarios**: 1. Complete checkout process 2. Order creation and processing 3. Inventory management 4. Payment processing 5. Notification handling

### Pattern Integration

The patterns work together to create a cohesive system: - Factory and Singleton patterns handle object creation and resource management - Observer pattern enables real-time updates and notifications - Strategy pattern provides flexible algorithm implementation - Repository pattern manages data access - Facade pattern simplifies complex operations

This combination of patterns results in a system that is: - Easy to maintain and extend - Flexible and adaptable to change - Efficient in resource management - Clear in its separation of concerns

The patterns complement each other: 1. Factory creates objects that Repository manages 2. Singleton provides resources that other patterns use 3. Observer notifies about changes that Strategy might affect 4. Facade coordinates the interaction of all patterns

## 4. Features

### User Authentication

* **Secure Login/Registration**
  + BCrypt password hashing
  + Session management
  + Password recovery system
* **Role-Based Access Control**
  + Admin privileges
  + Customer privileges
  + Guest access

### Book Management

* **Book Operations**
  + Comprehensive listing
  + Advanced search functionality
  + Detailed book information
* **Role-Specific Actions**
  + Customer features:
    - View book details
    - Purchase books
    - Write reviews
  + Admin features:
    - Add new books
    - Edit book details
    - Delete books
    - Manage inventory

### Shopping System

* **Cart Management**
  + Add/remove items
  + Update quantities
  + Save for later
* **Checkout Process**
  + Secure payment processing
  + Multiple payment options
  + Order confirmation
* **Order Tracking**
  + Order history
  + Status updates
  + Shipping information

### Review System

* **Rating Features**
  + Star ratings (1-5)
  + Written reviews
  + Review moderation
* **Admin Controls**
  + Review approval
  + Content moderation
  + User feedback management

### Loyalty Program

* **Discount Tiers**
  + Regular member (5+ orders): 10% discount
  + Premium member (10+ orders): 15% discount
* **Benefits**
  + Automatic discount application
  + Priority customer support
  + Exclusive offers

### Admin Dashboard

* **Inventory Management**
  + Stock tracking
  + Low stock alerts
  + Inventory reports
* **Order Processing**
  + Order status updates
  + Shipping management
  + Return processing
* **User Management**
  + User accounts
  + Role assignment
  + Account verification
* **Sales Reporting**
  + Revenue tracking
  + Sales analytics
  + Performance metrics

## 5. Database Design

### Theoretical Foundation

* **Entity-Relationship (ER) Model**
  + Visual representation of data relationships
  + Clear entity definitions
  + Relationship mapping
* **Normalization (3NF)**
  + Eliminates data redundancy
  + Reduces data anomalies
  + Ensures data integrity
* **Referential Integrity**
  + Foreign key constraints
  + Relationship validation
  + Data consistency
* **ACID Properties**
  + Atomic transactions
  + Consistent data
  + Isolated operations
  + Durable storage

### Tables Schema

#### users

| Column | Type | Description |
| --- | --- | --- |
| id | SERIAL | Primary key |
| username | VARCHAR(50) | Unique username |
| password\_hash | VARCHAR(255) | BCrypt hashed password |
| full\_name | VARCHAR(100) | User’s full name |
| email | VARCHAR(100) | User’s email address |
| address | TEXT | Shipping address |
| phone\_number | VARCHAR(20) | Contact phone number |
| role | VARCHAR(20) | User role (ADMIN or CUSTOMER) |
| order\_count | INTEGER | Number of orders placed (for loyalty) |

#### books

| Column | Type | Description |
| --- | --- | --- |
| id | SERIAL | Primary key |
| title | VARCHAR(255) | Book title |
| author | VARCHAR(100) | Book author |
| isbn | VARCHAR(20) | ISBN number |
| publisher | VARCHAR(100) | Publisher name |
| category | VARCHAR(50) | Book category |
| description | TEXT | Book description |
| price | DECIMAL(10,2) | Book price |
| stock\_quantity | INTEGER | Available quantity |
| image\_url | TEXT | Book cover image URL |

#### reviews

| Column | Type | Description |
| --- | --- | --- |
| id | SERIAL | Primary key |
| book\_id | INTEGER | Foreign key to books |
| user\_id | INTEGER | Foreign key to users |
| rating | INTEGER | Rating (1-5) |
| comment | TEXT | Review text |
| review\_date | TIMESTAMP | Review submission date |

#### cart\_items

| Column | Type | Description |
| --- | --- | --- |
| id | SERIAL | Primary key |
| user\_id | INTEGER | Foreign key to users |
| book\_id | INTEGER | Foreign key to books |
| quantity | INTEGER | Quantity in cart |

#### orders

| Column | Type | Description |
| --- | --- | --- |
| id | SERIAL | Primary key |
| user\_id | INTEGER | Foreign key to users |
| order\_date | TIMESTAMP | Date of order |
| total\_amount | DECIMAL(10,2) | Total order amount |
| status | VARCHAR(20) | Order status |
| payment\_method | VARCHAR(50) | Payment method used |
| shipping\_address | TEXT | Delivery address |
| discount\_applied | DECIMAL(5,2) | Discount amount applied |

#### order\_items

| Column | Type | Description |
| --- | --- | --- |
| id | SERIAL | Primary key |
| order\_id | INTEGER | Foreign key to orders |
| book\_id | INTEGER | Foreign key to books |
| quantity | INTEGER | Quantity ordered |
| price | DECIMAL(10,2) | Price at time of order |

### Relationships

* **users** 1:N **reviews** (one user can write many reviews)
* **users** 1:N **orders** (one user can place many orders)
* **users** 1:N **cart\_items** (one user has one cart with many items)
* **books** 1:N **reviews** (one book can have many reviews)
* **books** 1:N **cart\_items** (one book can be in many carts)
* **books** 1:N **order\_items** (one book can be in many orders)
* **orders** 1:N **order\_items** (one order contains many items)

## 6. Testing

### Testing Strategy

* **Test-Driven Development (TDD)**
  + Write failing tests first
  + Implement minimum code to pass
  + Refactor while maintaining tests
* **Testing Pyramid**
  + Unit Tests: Class and method level
  + Integration Tests: Component interaction
  + System Tests: Full application testing
* **Mocking with Mockito**
  + Dependency simulation
  + Component isolation
  + Interaction verification

### Test Coverage

* **Model Classes**
  + Business logic validation
  + Data integrity checks
  + State management
* **Service Layer**
  + Business rules
  + Transaction handling
  + Error management
* **Data Access**
  + CRUD operations
  + Query optimization
  + Connection management
* **Security**
  + Authentication
  + Authorization
  + Data protection

### Sample Test Code

@Test  
public void testReduceStock() {  
 Book book = new Book();  
 book.setStockQuantity(10);  
   
 boolean result = book.reduceStock(3);  
 assertTrue(result);  
 assertEquals(7, book.getStockQuantity());  
   
 result = book.reduceStock(10);  
 assertFalse(result);  
 assertEquals(7, book.getStockQuantity());  
   
 result = book.reduceStock(-1);  
 assertFalse(result);  
 assertEquals(7, book.getStockQuantity());  
}  
  
@Test  
public void testAuthenticateUser() {  
 UserRepository mockRepo = mock(UserRepository.class);  
 User validUser = new User();  
 validUser.setId(1);  
 validUser.setUsername("testuser");  
 validUser.setPasswordHash(BCrypt.hashpw("password123", BCrypt.gensalt()));  
   
 when(mockRepo.findByUsername("testuser")).thenReturn(validUser);  
   
 UserService userService = new UserService(mockRepo);  
   
 User result = userService.authenticateUser("testuser", "password123");  
 assertNotNull(result);  
 assertEquals(1, result.getId());  
   
 result = userService.authenticateUser("testuser", "wrongpassword");  
 assertNull(result);  
   
 result = userService.authenticateUser("nonexistent", "password123");  
 assertNull(result);  
}  
  
@Test  
public void testCalculateDiscountedPrice() {  
 User regularUser = new User();  
 regularUser.setOrderCount(7);  
   
 User premiumUser = new User();  
 premiumUser.setOrderCount(12);   
   
 User standardUser = new User();  
 standardUser.setOrderCount(2);  
   
 DiscountService discountService = new DiscountService();  
 BigDecimal originalPrice = new BigDecimal("100.00");  
   
 BigDecimal regularDiscount = discountService.calculateDiscountedPrice(originalPrice, regularUser);  
 assertEquals(new BigDecimal("90.00"), regularDiscount);  
   
 BigDecimal premiumDiscount = discountService.calculateDiscountedPrice(originalPrice, premiumUser);  
 assertEquals(new BigDecimal("85.00"), premiumDiscount);  
   
 BigDecimal standardDiscount = discountService.calculateDiscountedPrice(originalPrice, standardUser);  
 assertEquals(originalPrice, standardDiscount);  
}

## 7. Security

### Security Principles

* **CIA Triad**
  + Confidentiality: Authorized access only
  + Integrity: Data accuracy and reliability
  + Availability: Access when needed
* **Defense in Depth**
  + Database layer security
  + Service layer protection
  + Presentation layer safeguards
* **Least Privilege**
  + Admin: Full system access
  + Customer: Personal data access
  + Guest: Read-only access

### Authentication & Authorization

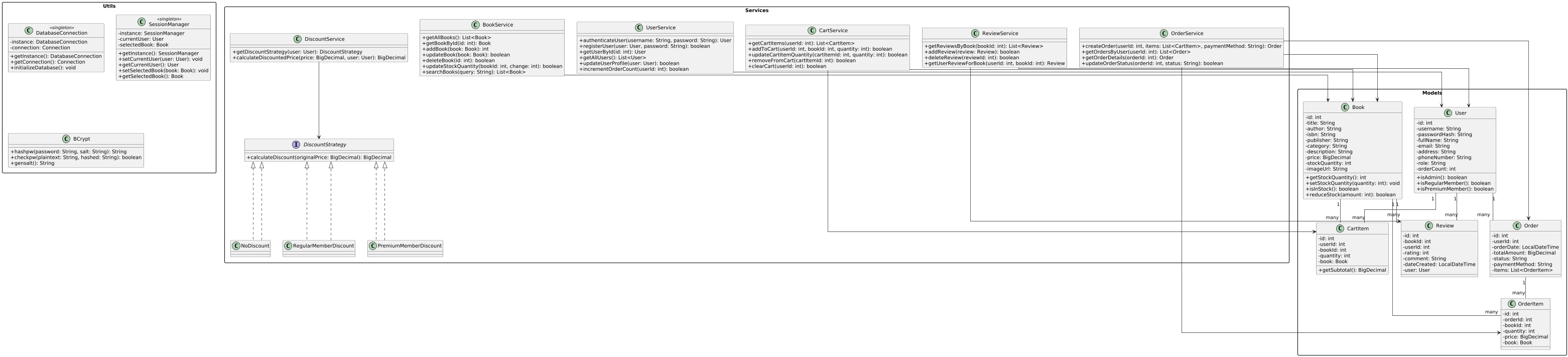
* **Password Security**
  + BCrypt hashing
  + Salt generation
  + Key stretching
* **Session Management**
  + Secure session creation
  + Timeout handling
  + Session invalidation
* **Access Control**
  + Role-based permissions
  + UI component visibility
  + Server-side validation

### Code Implementation

@FXML  
private void initialize() {  
 User currentUser = SessionManager.getInstance().getCurrentUser();  
   
 if (currentUser != null && currentUser.isAdmin()) {  
 adminControls.setVisible(true);  
 customerControls.setVisible(false);  
 } else {  
 adminControls.setVisible(false);  
 customerControls.setVisible(true);  
 }  
}  
  
public boolean deleteBook(int bookId) throws SQLException {  
 User currentUser = SessionManager.getInstance().getCurrentUser();  
   
 if (currentUser == null || !currentUser.isAdmin()) {  
 return false;  
 }  
   
 return bookRepository.delete(bookId);  
}

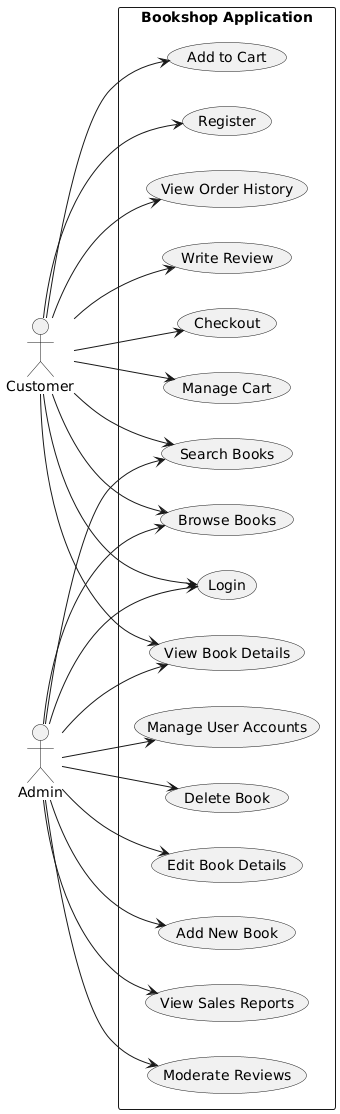
## 8. UML Diagrams

### Class Diagram



Extended Class Diagram

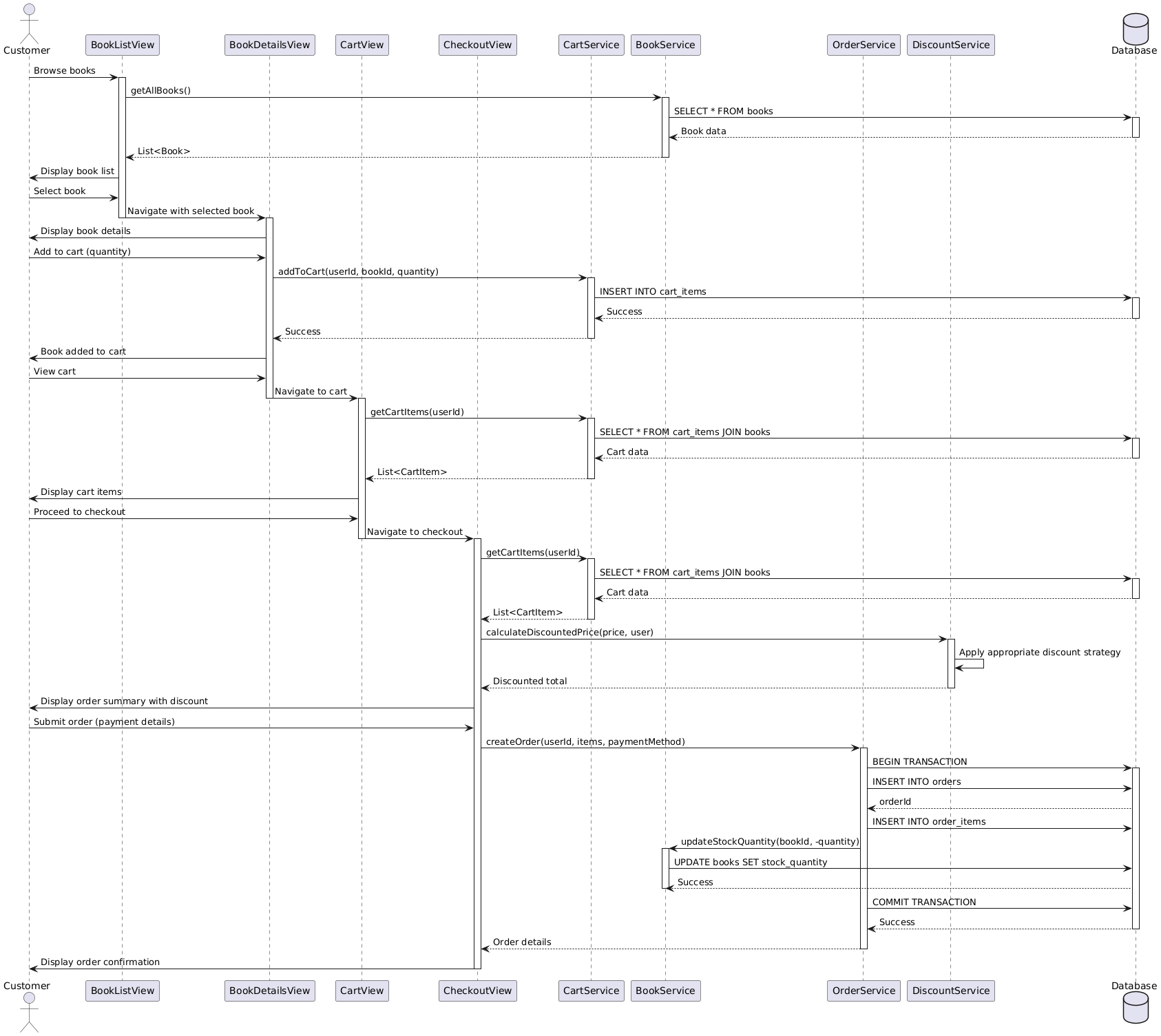
### Use Case Diagram



Use Case Diagram

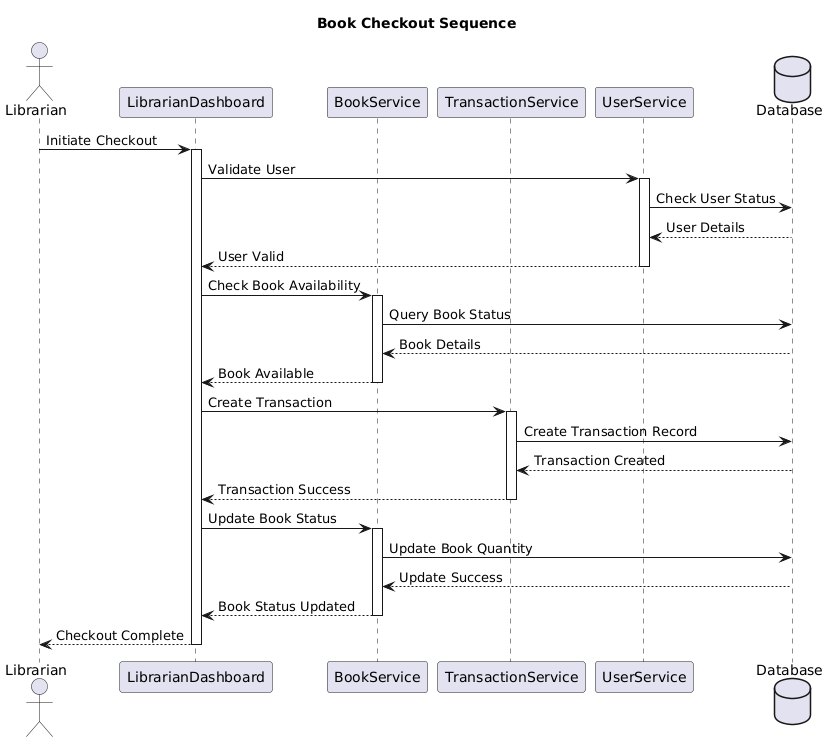
### Sequence Diagrams

#### Main Flow



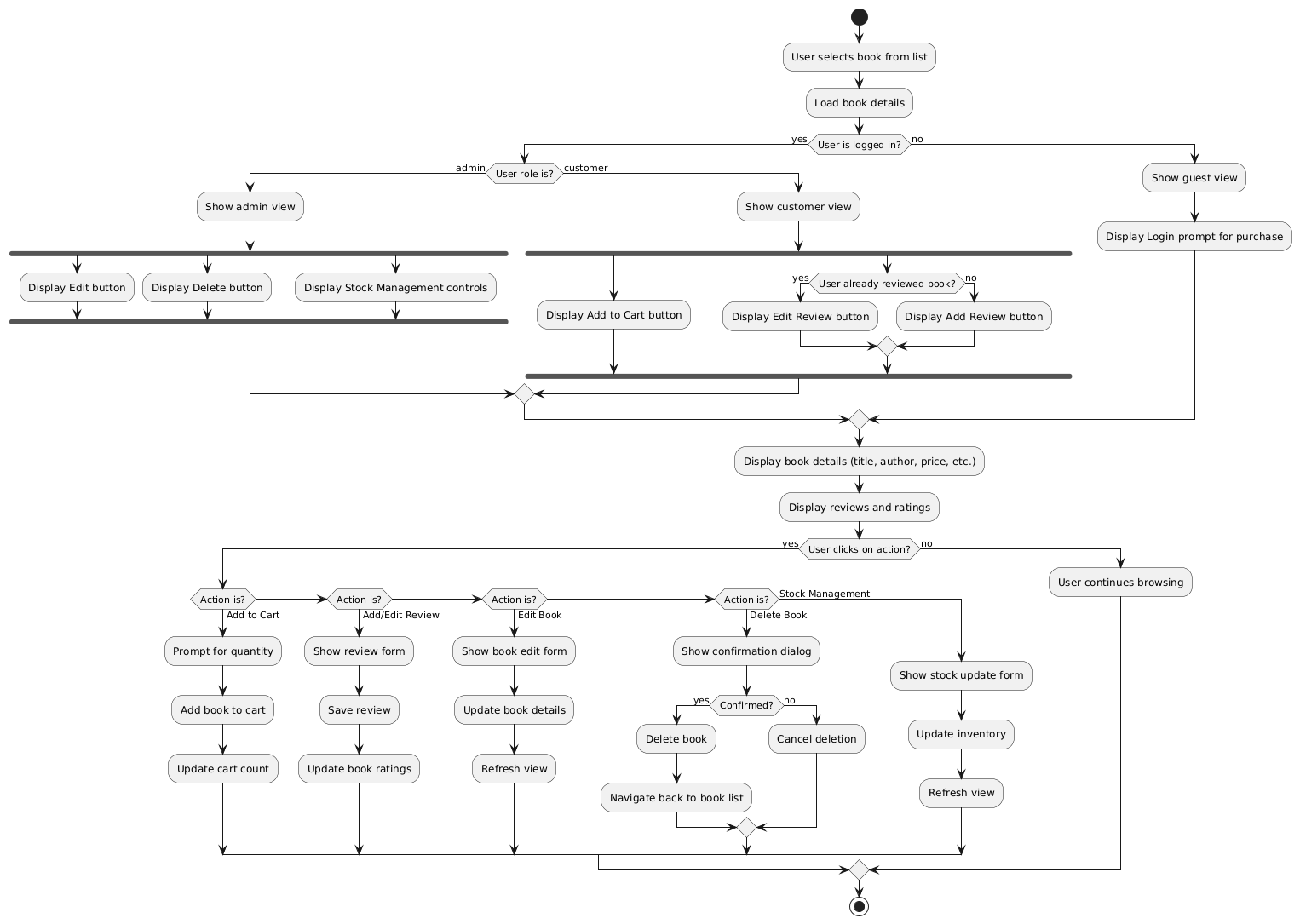
Sequence Diagram

#### Alternative Flow



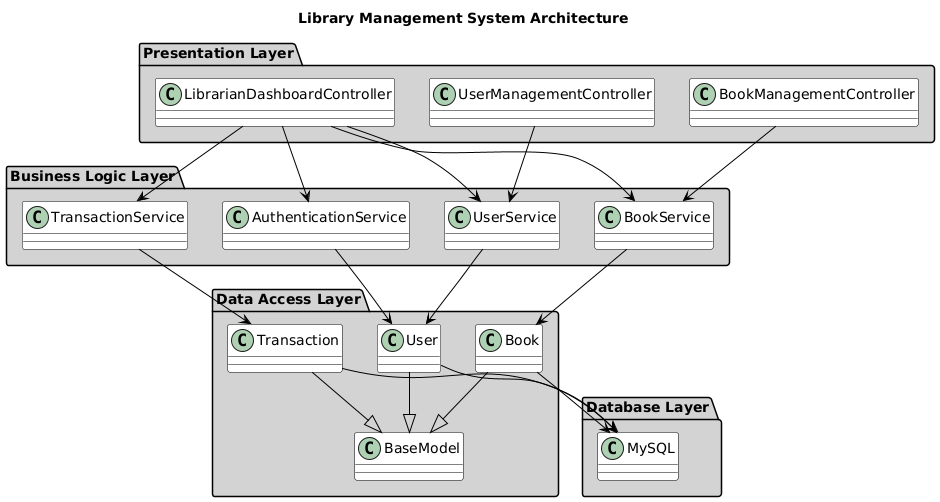
Sequence Diagram 2

### Activity Diagram



Activity Diagram

### System Architecture



System Architecture

## 9. Conclusion

### Key Achievements

* **Code Organization**
  + Clean architecture
  + Design pattern implementation
  + Modular structure
* **Security Implementation**
  + Secure authentication
  + Role-based access
  + Data protection
* **Feature Set**
  + Comprehensive functionality
  + User-friendly interface
  + Efficient performance
* **Data Management**
  + Optimized database design
  + Efficient queries
  + Data integrity
* **Design Patterns**
  + Appropriate pattern selection
  + Clean implementation
  + Maintainable code