Library Management System Documentation

Names

Ahmed Abd El hakeem Abd El Rahman
2100379

Ahmed Hamada Ahmed
2100382

Ahmed Emad Hussein
2100373

Amr Mohamed Abdalmonam
2100493

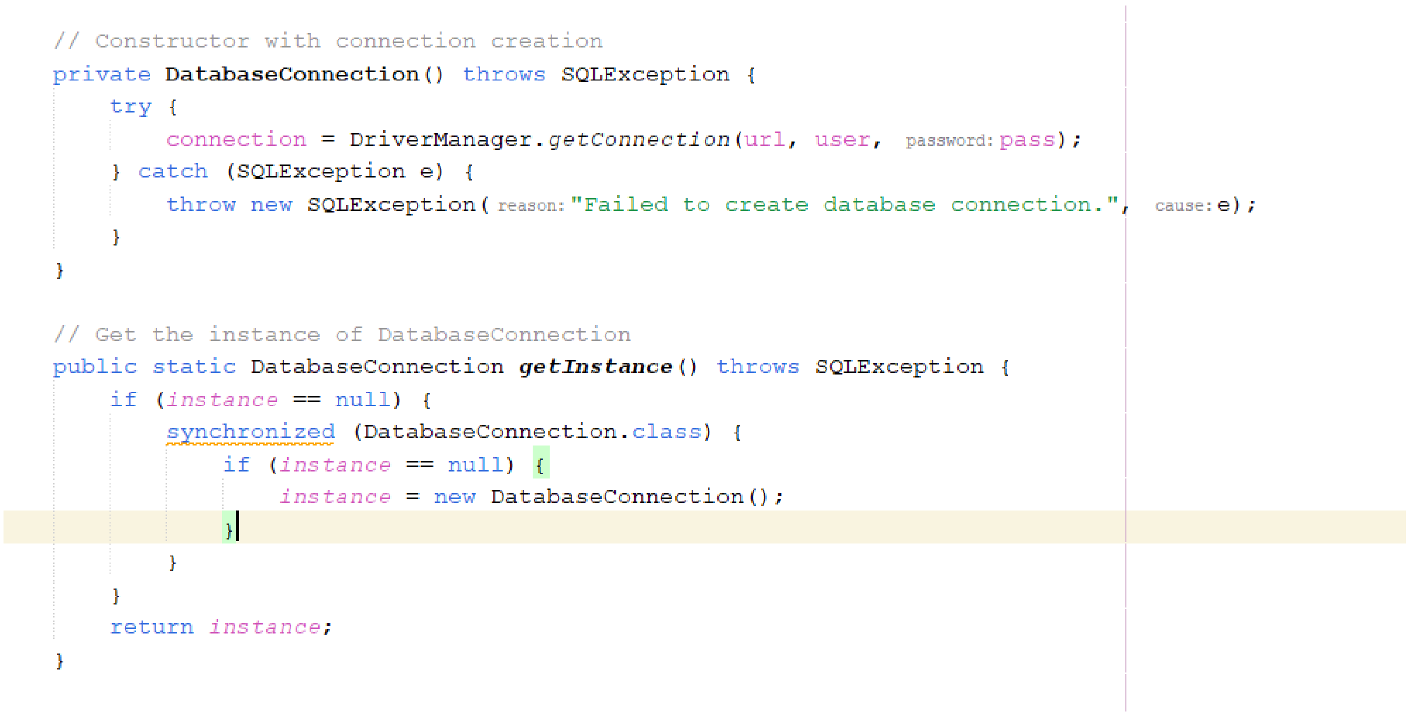
Abanob Eid Zaka
2100224

Abanoub Rady fakhry
2100227

This documentation describes the design patterns used in the Library Management System project. The project implements various design patterns such as Singleton, Strategy, Command, Factory, and State to handle different operations related to books and user management.
The aim is to manage books, users, and loans efficiently while keeping the system modular and flexible for future extensions.

Singleton Pattern

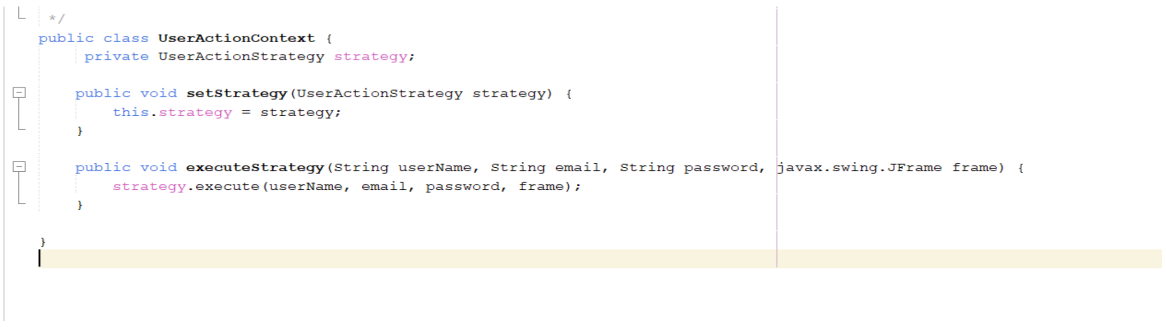
The Singleton pattern ensures that there is only one instance of the DatabaseConnection class throughout the application.
The instance is created lazily when needed and provides a global point of access to the connection. This pattern is used here to manage the connection to the database to ensure it is reused and not repeatedly established.



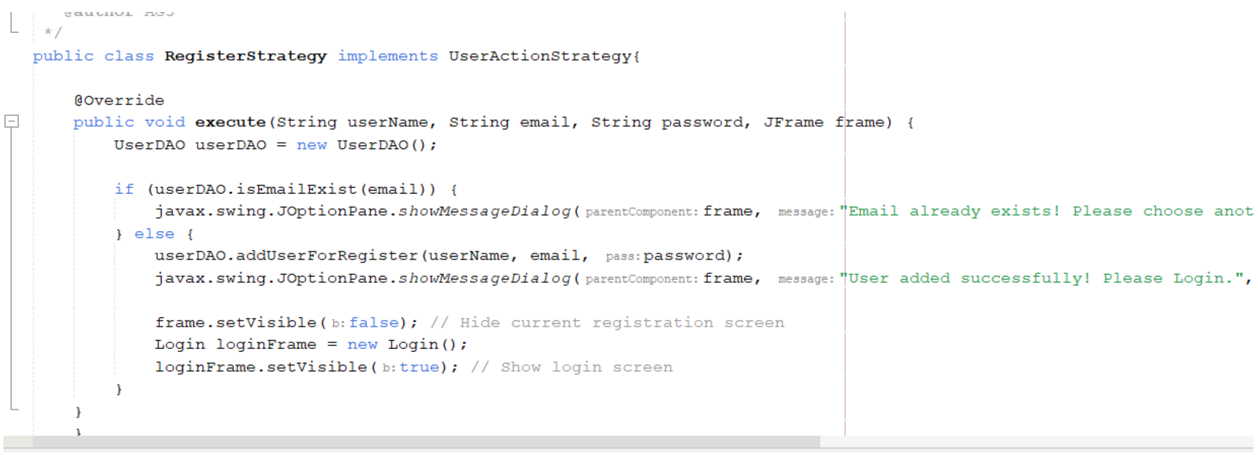
```
// Constructor with connection creation
private DatabaseConnection() throws SQLException {
    try {
        connection = DriverManager.getConnection(url, user,  password: pass);
    } catch (SQLException e) {
        throw new SQLException( reason: "Failed to create database connection.",  cause: e);
    }
}

// Get the instance of DatabaseConnection
public static DatabaseConnection getInstance() throws SQLException {
    if (instance == null) {
        synchronized (DatabaseConnection.class) {
            if (instance == null) {
                instance = new DatabaseConnection();
            }
        }
    }
    return instance;
}
```

Strategy Pattern

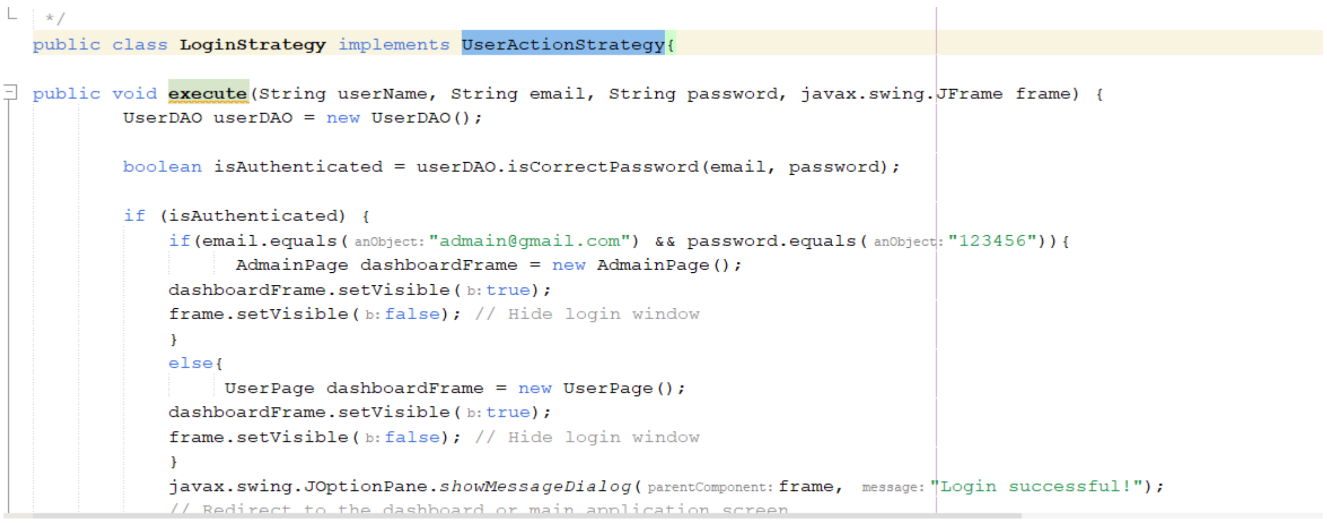
The Strategy pattern is used to encapsulate user actions such as login and registration in separate classes (LoginStrategy, RegisterStrategy).
The actions are interchangeable at runtime, and the UserActionContext class is responsible for selecting and executing the appropriate action.
This pattern provides flexibility and decouples the user actions from the rest of the system, making it easy to modify or extend them.



```java
 */
public class UserActionContext {
    private UserActionStrategy strategy;

    public void setStrategy(UserActionStrategy strategy) {
        this.strategy = strategy;
    }

    public void executeStrategy(String userName, String email, String password, javax.swing.JFrame frame) {
        strategy.execute(userName, email, password, frame);
    }

}
```



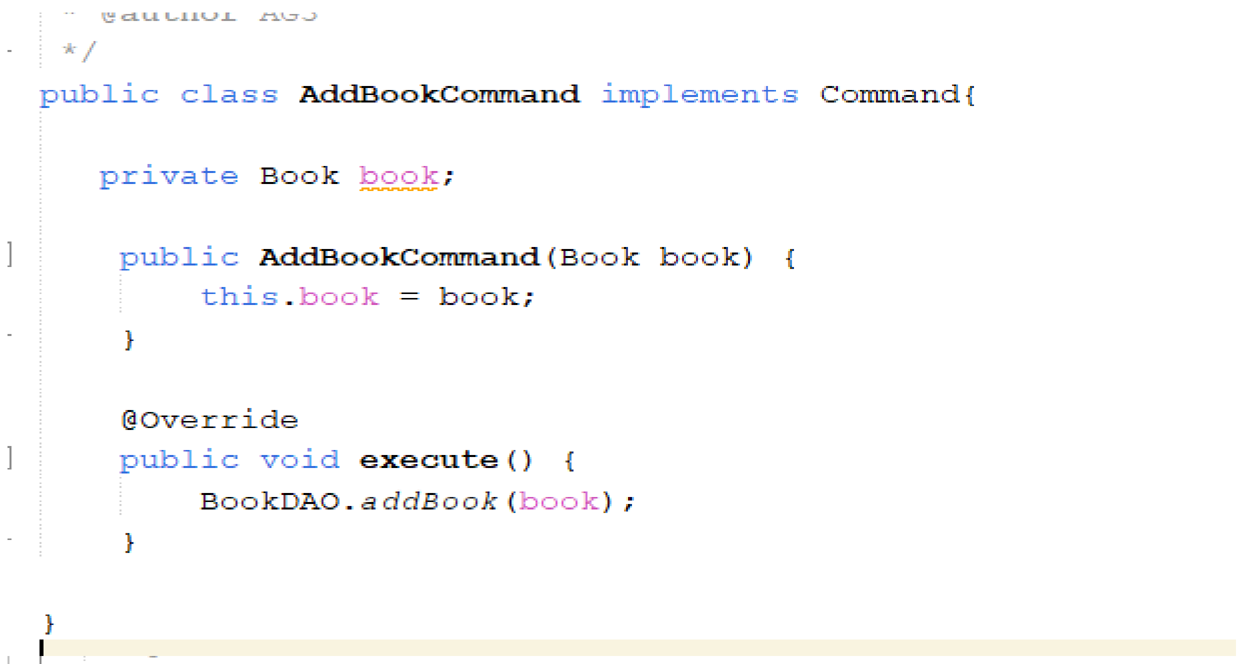
```java
 */
public class RegisterStrategy implements UserActionStrategy{

    @Override
    public void execute(String userName, String email, String password, JFrame frame) {
        UserDAO userDAO = new UserDAO();

        if (userDAO.isEmailExist(email)) {
            javax.swing.JOptionPane.showMessageDialog( parentComponent: frame,  message: "Email already exists! Please choose anot
        } else {
            userDAO.addUserForRegister(userName, email,  pass: password);
            javax.swing.JOptionPane.showMessageDialog( parentComponent: frame,  message: "User added successfully! Please Login.",

            frame.setVisible( b: false); // Hide current registration screen
            Login loginFrame = new Login();
            loginFrame.setVisible( b: true); // Show login screen
        }
    }
```



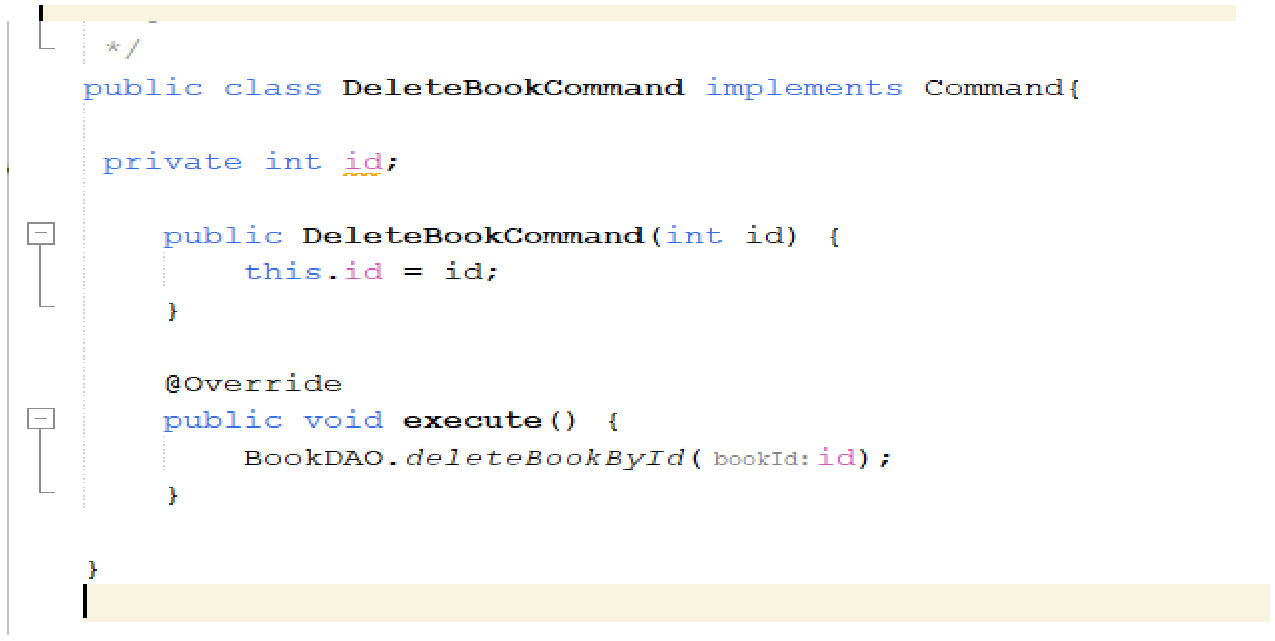
```java
 */
public class LoginStrategy implements UserActionStrategy{

public void execute(String userName, String email, String password, javax.swing.JFrame frame) {
        UserDAO userDAO = new UserDAO();

        boolean isAuthenticated = userDAO.isCorrectPassword(email, password);

        if (isAuthenticated) {
            if(email.equals( anObject: "admain@gmail.com") && password.equals( anObject: "123456")){
                AdmainPage dashboardFrame = new AdmainPage();
            dashboardFrame.setVisible( b: true);
            frame.setVisible( b: false); // Hide login window
            }
            else{
                UserPage dashboardFrame = new UserPage();
            dashboardFrame.setVisible( b: true);
            frame.setVisible( b: false); // Hide login window
            }
            javax.swing.JOptionPane.showMessageDialog( parentComponent: frame,  message: "Login successful!");
            // Redirect to the dashboard or main application screen
```

Command Pattern

The Command pattern encapsulates a request as an object, allowing parameterization of clients with queues, requests, and operations.
In this project, commands like AddBookCommand and DeleteBookCommand handle book-related actions, providing a clear separation between the request and its execution.
This pattern simplifies the execution of operations by abstracting the request into a command object that can be passed, executed, or undone if necessary.



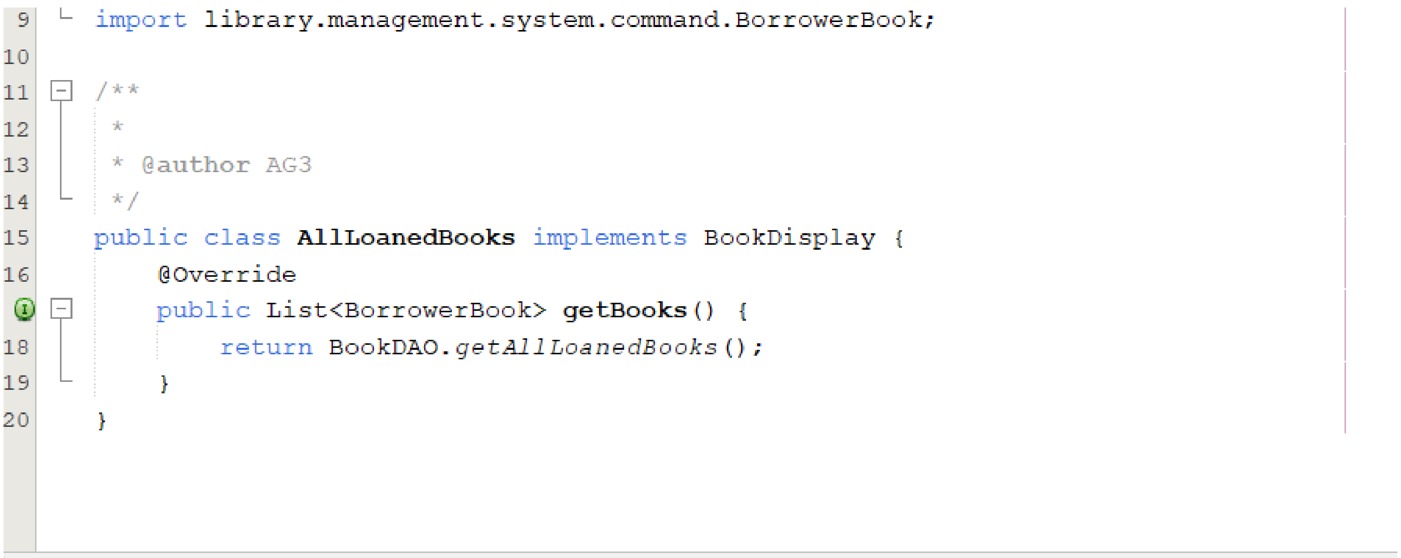
```java
 * @author AGS
 */
public class AddBookCommand implements Command{

   private Book book;

    public AddBookCommand(Book book) {
        this.book = book;
    }

    @Override
    public void execute() {
        BookDAO.addBook(book);
    }

}
```



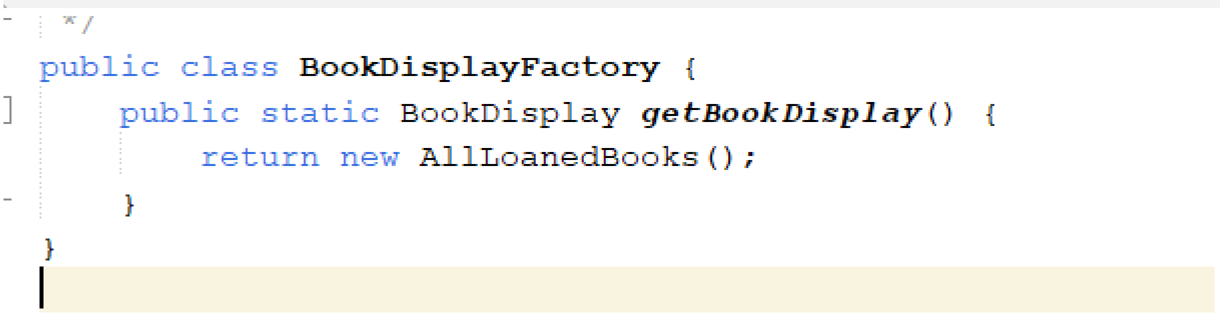
```java
 */
public class DeleteBookCommand implements Command{

 private int id;

    public DeleteBookCommand(int id) {
        this.id = id;
    }

    @Override
    public void execute() {
        BookDAO.deleteBookById( bookId: id);
    }

}
```

Factory Pattern

The Factory pattern is used to create instances of BookDisplay, specifically to retrieve all loaned books.
By using a Factory class (BookDisplayFactory), the creation of display objects is encapsulated, allowing the client to use the created object without knowing how it is constructed.
This pattern enhances flexibility and scalability when creating new display options for books.



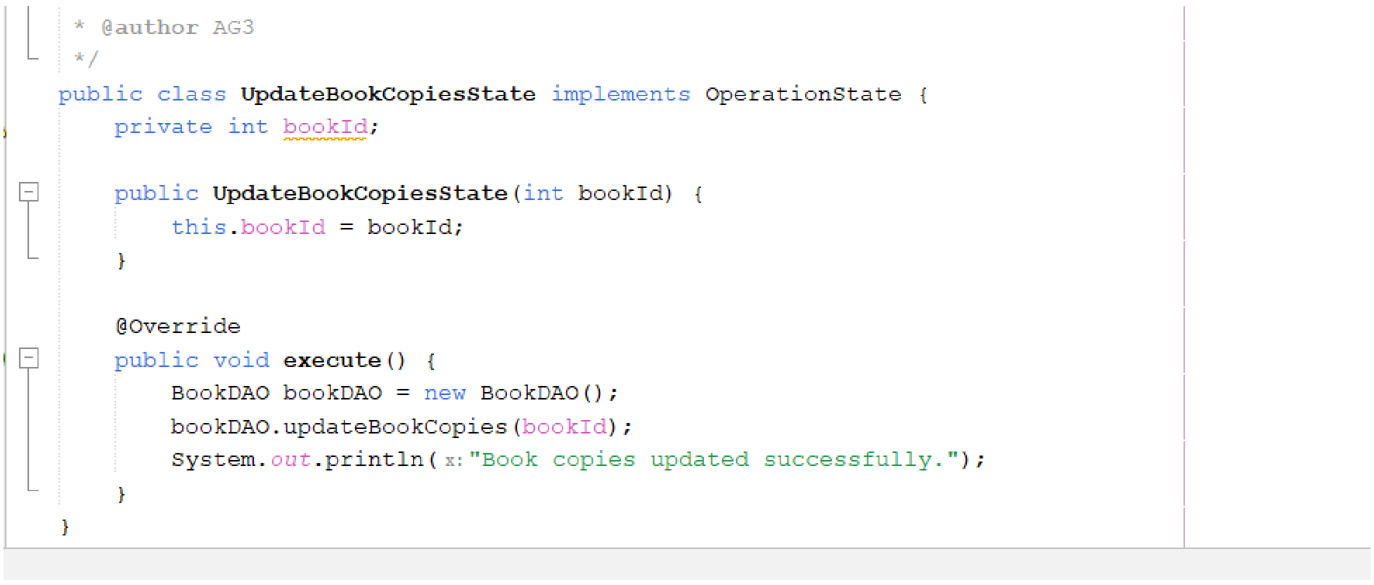
```java
import library.management.system.command.BorrowerBook;

/**
 *
 * @author AG3
 */
public class AllLoanedBooks implements BookDisplay {
    @Override
    public List<BorrowerBook> getBooks() {
        return BookDAO.getAllLoanedBooks();
    }
}
```



```java
 */
public class BookDisplayFactory {
    public static BookDisplay getBookDisplay() {
        return new AllLoanedBooks();
    }
}
```

State Pattern

The State pattern allows an object to alter its behavior when its internal state changes.
In this project, the OperationContext class uses different OperationState implementations (like UpdateBookCopiesState) to perform actions depending on the current state.
This pattern helps manage the flow of operations in a modular way, making it easier to add new states and modify behavior without changing existing code.



```java
 * @author AG3
 */
public class UpdateBookCopiesState implements OperationState {
    private int bookId;

    public UpdateBookCopiesState(int bookId) {
        this.bookId = bookId;
    }

    @Override
    public void execute() {
        BookDAO bookDAO = new BookDAO();
        bookDAO.updateBookCopies(bookId);
        System.out.println( x: "Book copies updated successfully.");
    }
}
```

Conclusion

The design patterns used in the Library Management System provide a flexible, maintainable, and extensible approach to solving common problems in application design.
By utilizing Singleton, Strategy, Command, Factory, and State patterns, the system achieves better code organization, separation of concerns, and improved scalability.
This modular approach will make future feature additions and modifications easier and more efficient.