Your Books Everywhere

Analysis and Design Document

Student: Borsa Iulian-Laurentiu

**Group: 30238**

Table of Contents

Your Books Everywhere 1

Analysis and Design Document 1

Student: Borsa Iulian-Laurentiu 1

Table of Contents 2

1. Requirements Analysis 4

1.1 Assignment Specification 4

1.2 Functional Requirements 4

1.3 Non-functional Requirements 4

2. Use-Case Model 4

3. System Architectural Design 5

4. Class Design 6

5. Data Model 7

6. System Testing 8

7. Bibliography 8

1. Requirements Analysis

# Assignment Specification

Design and implementation of an application for a book management service using OOP language Java. It will represent a library management with two types of users: normal user and staff user.

A user should be able to create an account, choose a payment plan and login to search the book library. Payments can be done via a cash only policy and need to be validated by library staff. The library is managed by staff.

If a book is available a user can add it to library. If not the user can join a waiting list. Once a book has been read by a user it can be returned via the online library return function.

The service also provides users with dynamic recommendations based on latest trends (popular borrowed books) or user defined interests by genre or topic.

# Functional Requirements

* The data will be stored in a relationship database.
* The application should have layered architecture
* A part of inputs will be validated against invalid data
* It contains a factory method to send recommendations
* Payment can be done via a cash only policy and need to be validated by staff
* Deals are managed by staff

# Non-functional Requirements

* Accessibility – the application is for all kind of users
* Security and privacy – the application is just for registered users
* Ease of development – The application is not complex to implement

2. Use-Case Model

Use case goal: return a book to library

Primary actor: an actor named “user”

Main success scenario:

1. User hasn’t got an account, so User is creating a new account

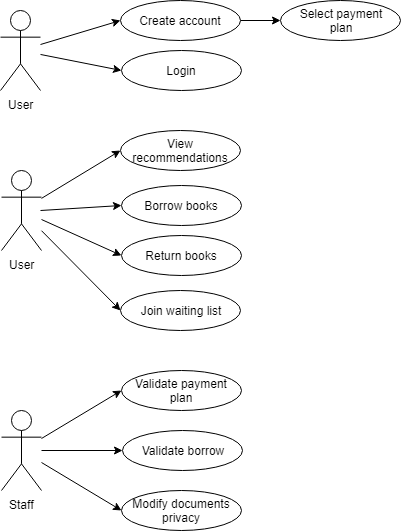
2. User is waiting to be validated by Staff

3. User login

4. User choose a book which is available

5. User borrow the book

6. User read the book and return it



3. System Architectural Design

**3.1 Architectural Pattern Description**

CQRS, which means Command Query Responsibility Segregation, comes from CQS (Command Query Separation) introduced by Bertrand Meyer in Object Oriented Software Construction. Meyer states that every method should be either a query or a command.

The difference between CQS and CQRS is that every CQRS object is divided in two objects: one for the query and one for the command.

A command is defined as a method that changes state. On the contrary, a query only returns a value.

The following schema shows a basic implementation of the CQRS pattern inside an application. All messages are sent through commands and events. Let’s take a closer look at this.

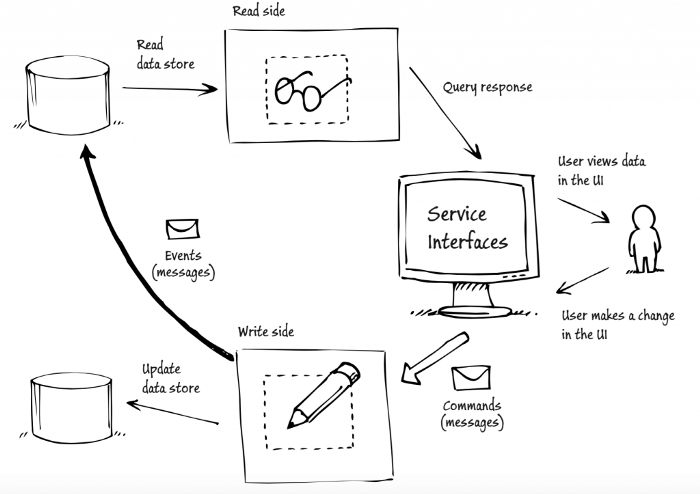


Figure 3.1 - Architectural Pattern

We can clearly see the separation between writing parts and reading ones: the user does a modification on his page, resulting in a command being executed. Once this command is fully handled, an event is dispatched to signal a modification.

* 1. **Diagrams**

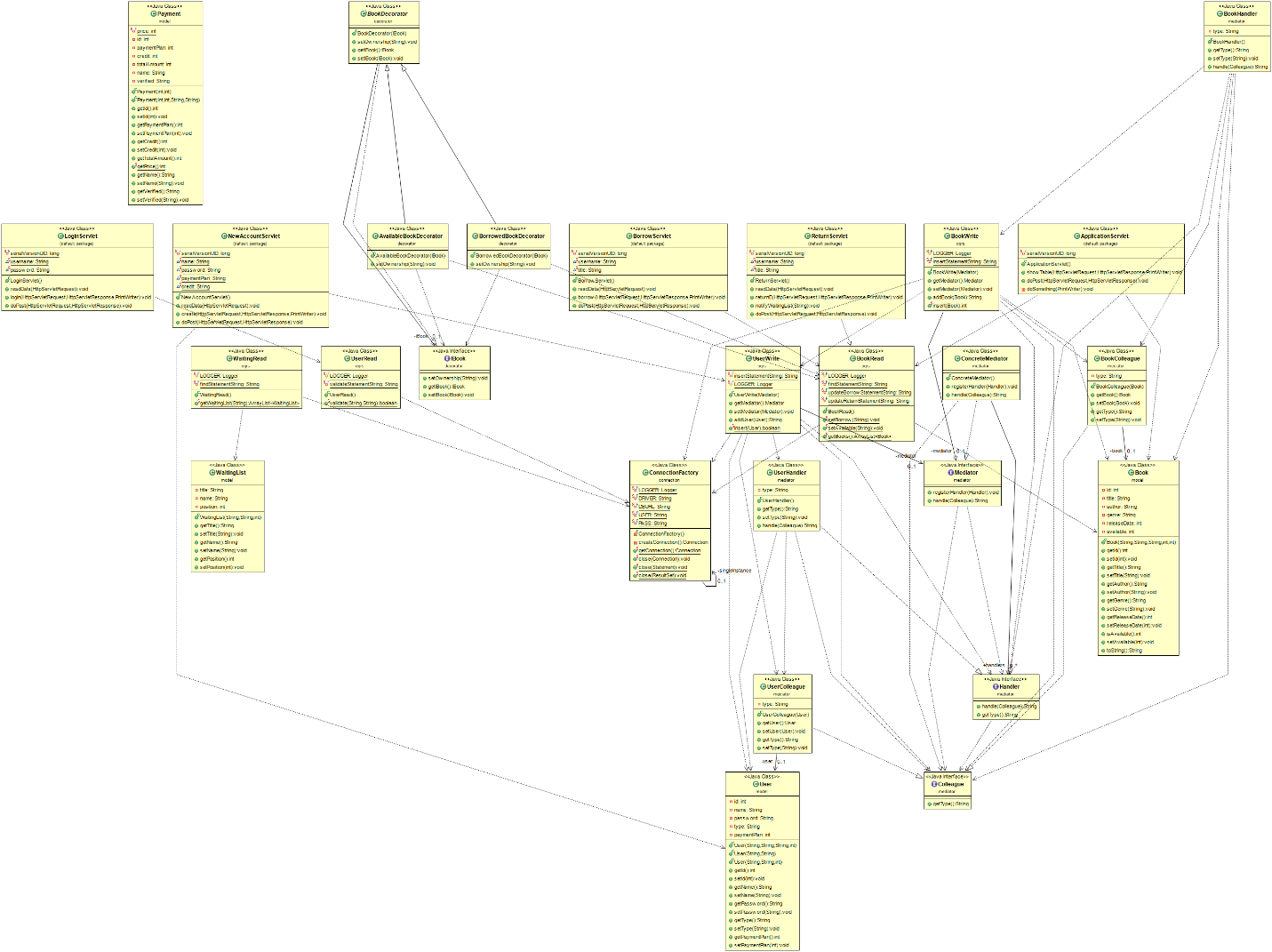


Figure 3.2 - Package Diagram

4. Class Design

**4.1 Design Patterns Description**

Mediator design pattern is one of the important and widely used behavioral design pattern. Mediator enables decoupling of objects by introducing a layer in between so that the interaction between objects happen via the layer. If the objects interact with each other directly, the system components are tightly-coupled with each other that makes higher maintainability cost and not hard to extend. Mediator pattern focuses on providing a mediator between objects for communication and help in implementing lose-coupling between objects.

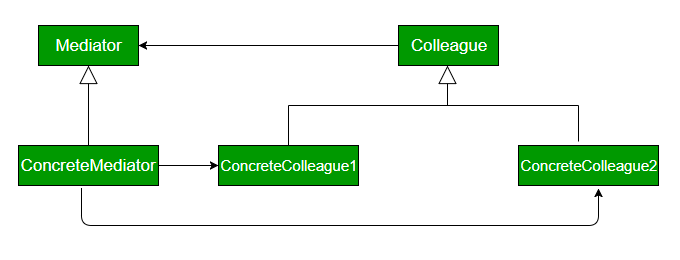


Figure 5.1 – Mediator Design Pattern

A Decorator pattern can be used to attach additional responsibilities to an object either statically or dynamically. A Decorator provides an enhanced interface to the original object.

In the implementation of this pattern, we prefer composition over an inheritance – so that we can reduce the overhead of subclassing again and again for each decorating element. The recursion involved with this design can be used to decorate our object as many times as we require.

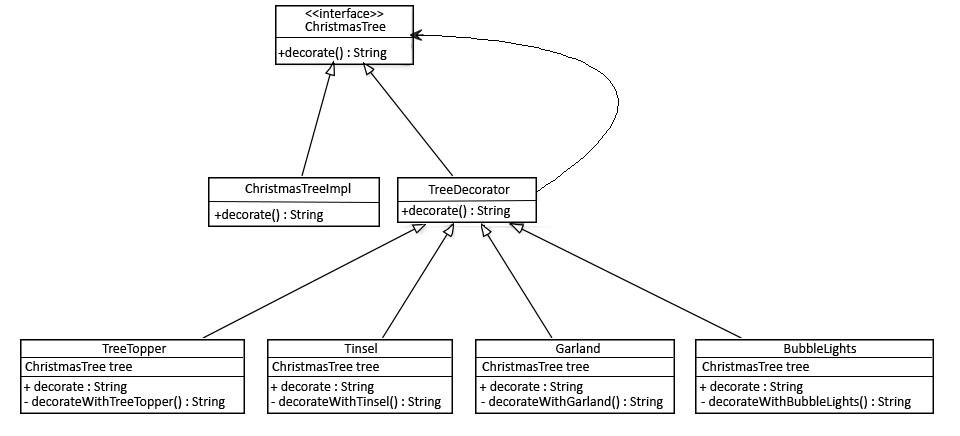
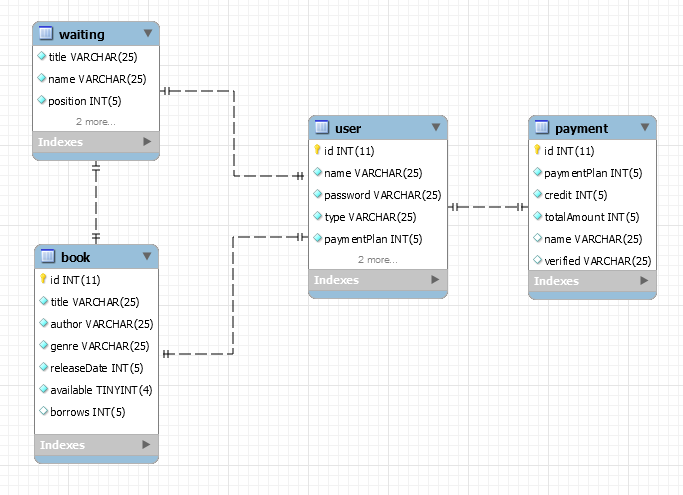


Figure 5.2 – Decorator Design Pattern

5. Data Model



6. System Testing

I’m using **Graphical User Interface Testing**, who is the process of testing a product graphical interface. This is done through the use of a set of test case scenarios. To generate a set of test cases, I will try to cover all the functionality of the system and verify entire GUI system, first as normal user, and than as guest.

7. Bibliography

<http://www.bredemeyer.com/ArchitectingProcess/ConceptualArchitecture.htm>

<https://www.tutorialspoint.com/design_pattern/factory_pattern.htm>