Your Books Everywhere!

Analysis and Design Document

Student: Bianca Elena Dondas

**Group: 30238**

Contents

[1. Requirements Analysis 3](#_Toc10189722)

[1.1 Assignment Specification 3](#_Toc10189723)

[1.2 Functional Requirements 3](#_Toc10189724)

[1.3 Non-functional Requirement 3](#_Toc10189725)

[2. Use-Case Model 3](#_Toc10189726)

[3. System Architectural Design 4](#_Toc10189727)

[3.1 Architectural Pattern Description 4](#_Toc10189728)

[4. UML Sequence Diagrams 5](#_Toc10189729)

[5. Class Design 6](#_Toc10189730)

[5.1 Design Patterns Description 6](#_Toc10189731)

[5.2 UML Class Diagram 7](#_Toc10189732)

[6. Data Model 8](#_Toc10189733)

[7. System Testing 9](#_Toc10189734)

[8. Bibliography 9](#_Toc10189735)

# 1. Requirements Analysis

## Assignment Specification

You are tasked to build a book management service.

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 and can be filtered by release date, author, title, genre. If a book is available a user can add it to your 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. This assigns the book to the next user in the waiting list after validation of the return by library staff. The service also provides users with dynamic recommendations based on latest trends (popular borrowed books) or user defined interests by genre or topic

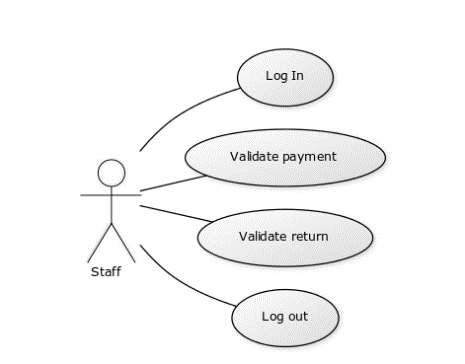
## Functional Requirements

* Log in and create account
* Choose payment plan
* Filter by: author, title, genre
* Borrow & return service
* Validate payments and validate return book by staff

## Non-functional Requirement

* Using OOP language (Java)
* Commit work to GitHub
* CQRS architecture
* Store data in MySql Database
* Use mediator and decorator pattern

# 2. Use-Case Model

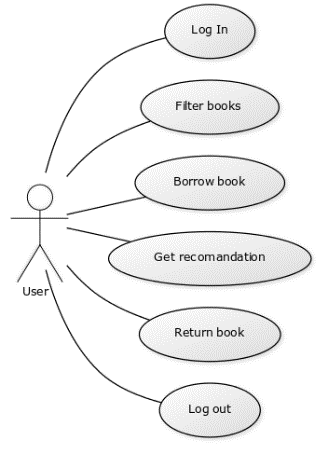


Use case: Validate return

Level: User- goal level

Primary actor: Staff

Main success scenario: The librarian connects to the app using the log in page. Then he check the return requests. He remove the person from return request table and the assign the book to the next person from the reading list.



Use case: Borrow book and return

Level: User- goal level

Primary actor: User

Main success scenario: The user connects to the app using the log in page. Then he can choose a book from the list. When he decides what he wants to read, he adds the book to his reading list. Once he finish the book he request the return of the book, which will be validated by staff.

# 3. System Architectural Design

## 3.1 Architectural Pattern Description

CQRS stands for **Command Query Responsibility Segregation**. At its heart is the notion that you can use a different model to update information than the model you use to read information. For some situations, this separation can be valuable, but beware that for most systems CQRS adds risky complexity.

The mainstream approach people use for interacting with an information system is to treat it as a CRUD datastore. By this I mean that we have mental model of some record structure where we can**c**reate new records, **r**ead records, **u**pdate existing records, and **d**elete records when we're done with them. In the simplest case, our interactions are all about storing and retrieving these records.

As our needs become more sophisticated we steadily move away from that model. We may want to look at the information in a different way to the record store, perhaps collapsing multiple records into one, or forming virtual records by combining information for different places. On the update side we may find validation rules that only allow certain combinations of data to be stored, or may even infer data to be stored that's different from that we provide.

Like any pattern, CQRS is useful in some places, but not in others. Many systems do fit a CRUD mental model, and so should be done in that style. CQRS is a significant mental leap for all concerned, so shouldn't be tackled unless the benefit is worth the jump. While I have come across successful uses of CQRS, so far the majority of cases I've run into have not been so good, with CQRS seen as a significant force for getting a software system into serious difficulties. In particular CQRS should only be used on specific portions of a system (a [BoundedContext](https://martinfowler.com/bliki/BoundedContext.html) in DDD lingo) and not the system as a whole. In this way of thinking, each Bounded Context needs its own decisions on how it should be modeled. So far I see benefits in two directions. Firstly is that a few complex domains may be easier to tackle by using CQRS. I must stress, however, that such suitability for CQRS is very much the minority case. Usually there's enough overlap between the command and query sides that sharing a model is easier. Using CQRS on a domain that doesn't match it will add complexity, thus reducing productivity and increasing risk.

The other main benefit is in handling high performance applications. CQRS allows you to separate the load from reads and writes allowing you to scale each independently. If your application sees a big disparity between reads and writes this is very handy. Even without that, you can apply different optimization strategies to the two sides. An example of this is using different database access techniques for read and update.

If your domain isn't suited to CQRS, but you have demanding queries that add complexity or performance problems, remember that you can still use a [ReportingDatabase](https://martinfowler.com/bliki/ReportingDatabase.html). CQRS uses a separate model for all queries. With a reporting database you still use your main system for most queries, but offload the more demanding ones to the reporting database.

# 5. Class Design

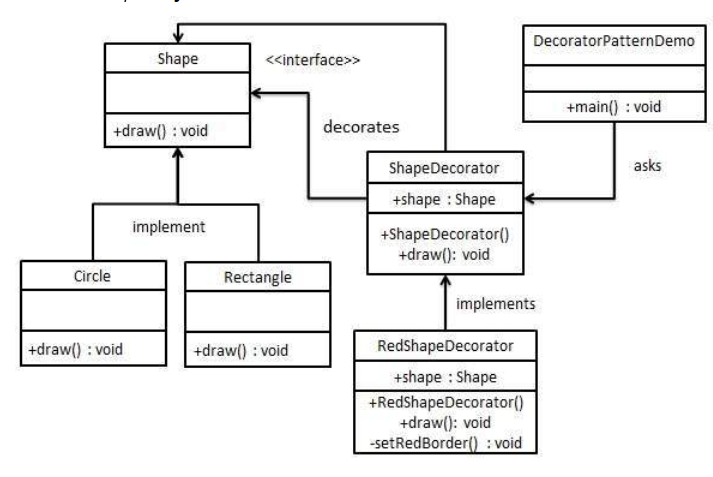
## 5.1 Design Patterns Description

**Decorator Pattern**

Decorator pattern allows a user to add new functionality to an existing object without altering its structure. This type of design pattern comes under structural pattern as this pattern acts as a wrapper to existing class.

This pattern creates a decorator class which wraps the original class and provides additional functionality keeping class methods signature intact.

We are demonstrating the use of decorator pattern via following example in which we will decorate a shape with some color without alter shape class.



**Mediator Pattern**

In software engineering, the mediator pattern defines an object that encapsulates how a set of objects interact. This pattern is considered to be a behavioral patterndue to the way it can alter the program's running behavior.

Usually a program is made up of a large number of classes. Logic and computation are distributed among these classes. However, as more classes are added to a program, especially during maintenance and/or refactoring, the problem of communication between these classes may become more complex. This makes the program harder to read and maintain. Furthermore, it can become difficult to change the program, since any change may affect code in several other classes.

With the mediator pattern, communication between objects is encapsulated within a mediator object. Objects no longer communicate directly with each other, but instead communicate through the mediator. This reduces the dependencies between communicating objects, thereby reducing coupling.

What problems can the Mediator design pattern solve?

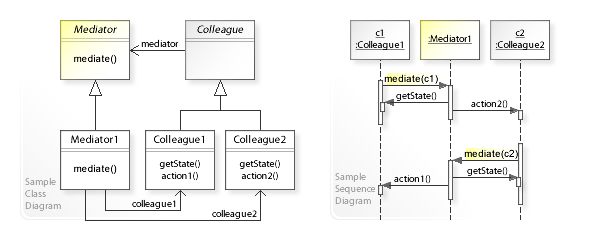
* Tight coupling between a set of interacting objects should be avoided.
* It should be possible to change the interaction between a set of objects independently.

Defining a set of interacting objects by accessing and updating each other directly is inflexible because it tightly couples the objects to each other and makes it impossible to change the interaction independently from (without having to change) the objects. And it stops the objects from being reusable and makes them hard to test.   
*Tightly coupled objects* are hard to implement, change, test, and reuse because they refer to and know about many different objects.

What solution does the Mediator design pattern describe?

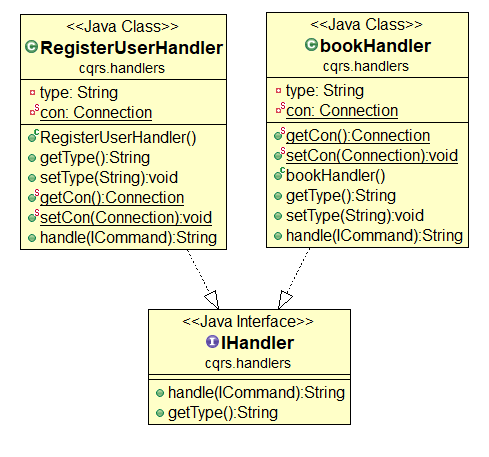
* Define a separate (mediator) object that encapsulates the interaction between a set of objects.
* Objects delegate their interaction to a mediator object instead of interacting with each other directly.

The objects interact with each other indirectly through a mediator object that controls and coordinates the interaction.   
This makes the objects *loosely coupled*. They only refer to and know about their mediator object and have no explicit knowledge of each other.

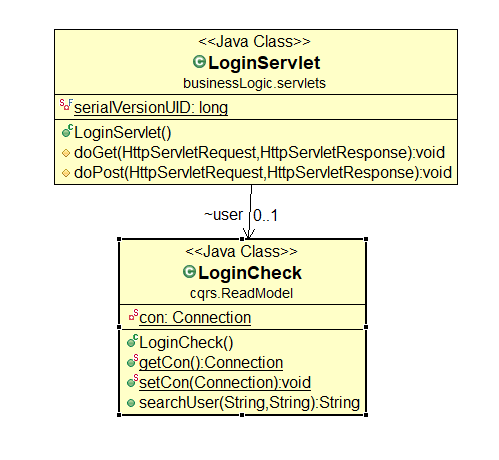


## 5.2 UML Class Diagram

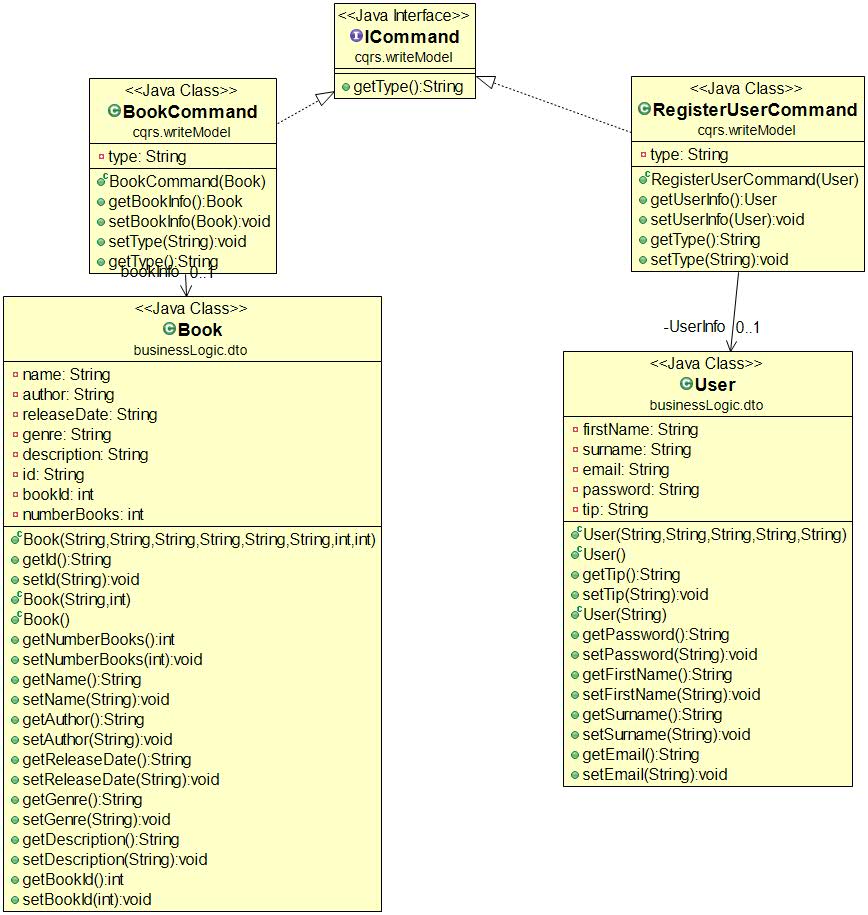
Class diagram for Handler package:



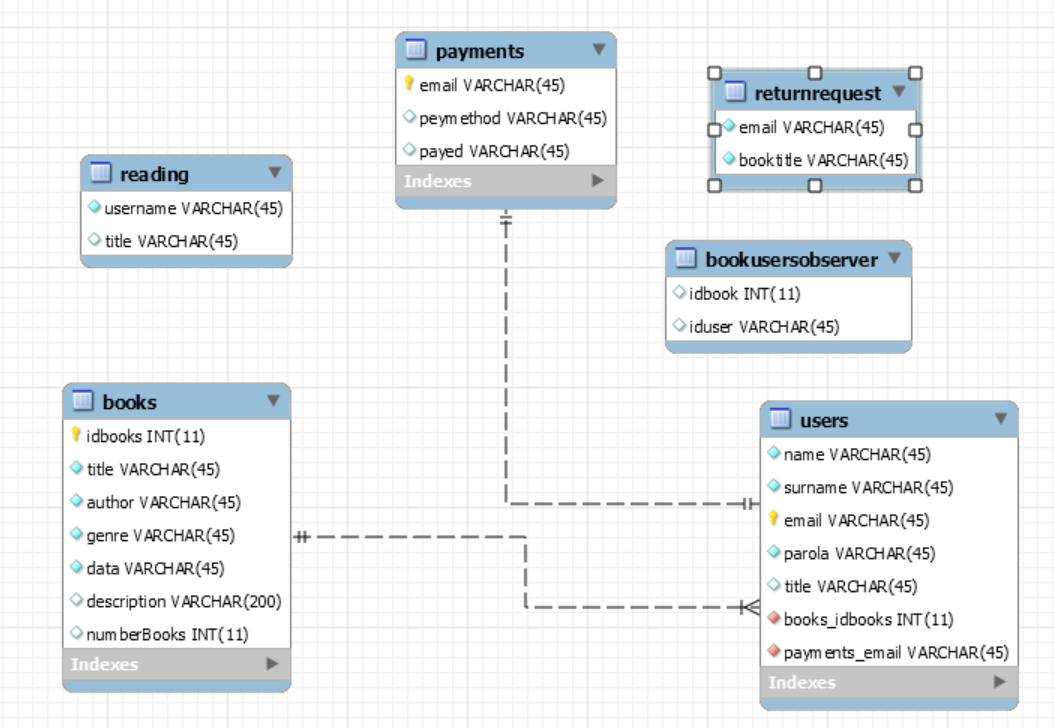
Class diagram for classes that implements login



Class Diagram for classes that implements ICommand



# 6. Data Model



# 7. System Testing

*[Present the used testing strategies (unit testing, integration testing, validation testing) and testing methods (data-flow, partitioning, boundary analysis, etc.).]*

# 8. Bibliography

* <https://dzone.com/articles/design-patterns-mediator>
* <https://www.tutorialspoint.com/design_pattern/decorator_pattern.htm>
* <https://www.baeldung.com/java-decorator-pattern>
* <https://en.wikipedia.org/wiki/Mediator_pattern>
* <https://martinfowler.com/bliki/CQRS.html>