Your Books Everywhere!

Student: Alexandra Georgeana Czako

**Group: 30238**

Table of Contents

1. Requirements Analysis 3

1.1 Assignment Specification 3

1.2 Functional Requirements 3

1.3 Non-functional Requirements 3

2. Use-Case Model 3

3. System Architectural Design 3

4. UML Sequence Diagrams 3

5. Class Design 3

6. Data Model 3

7. System Testing 3

8. Bibliography 3

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

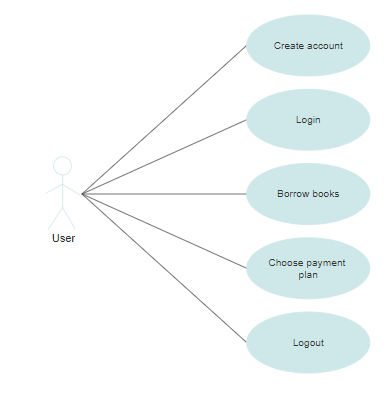
* Create account for user
* Choose payment plan for user
* Filter all the books by: author, title, genre; Show all the books available in the library
* Validate payments by staff

# Non-functional Requirements

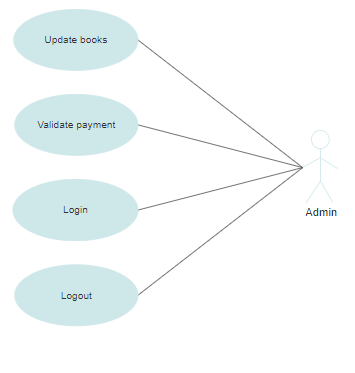
* Secured password
* Junit tests

2. Use-Case Model

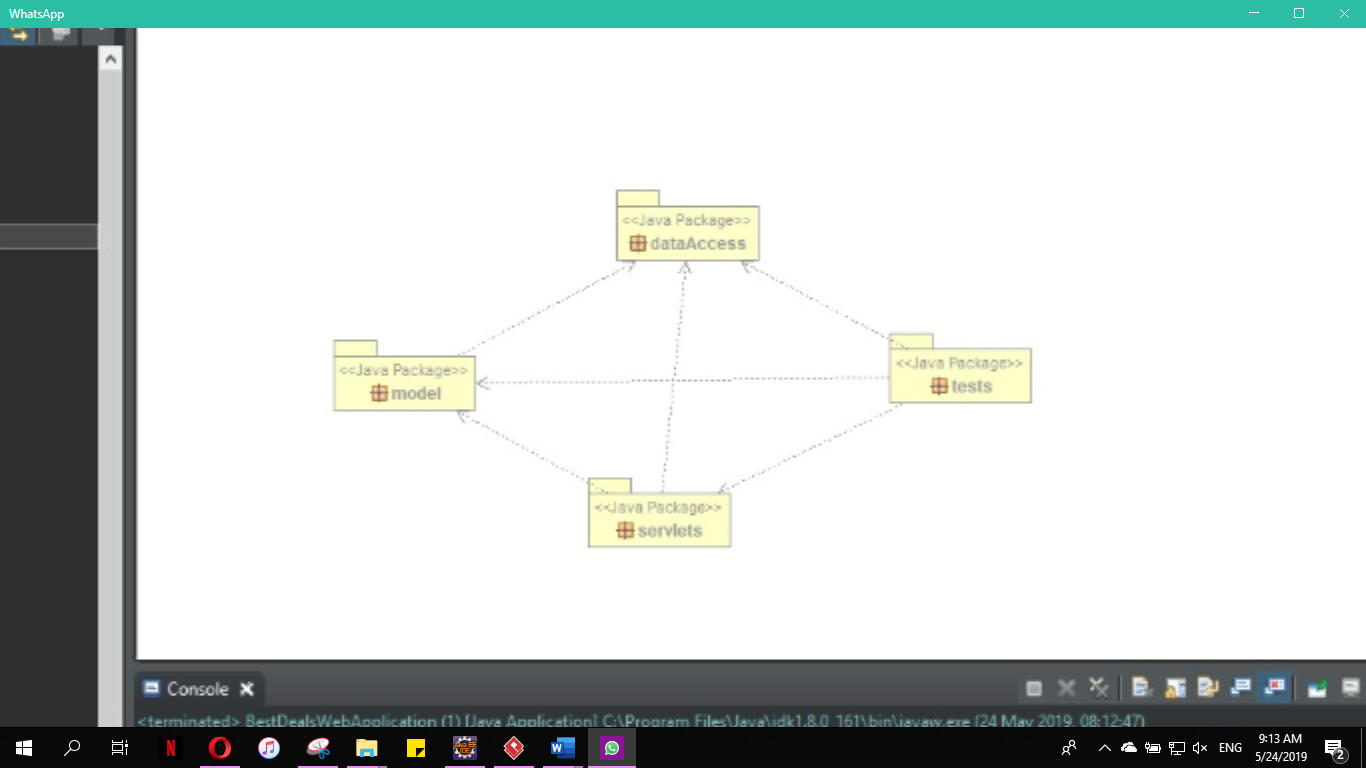
User’s use case model:



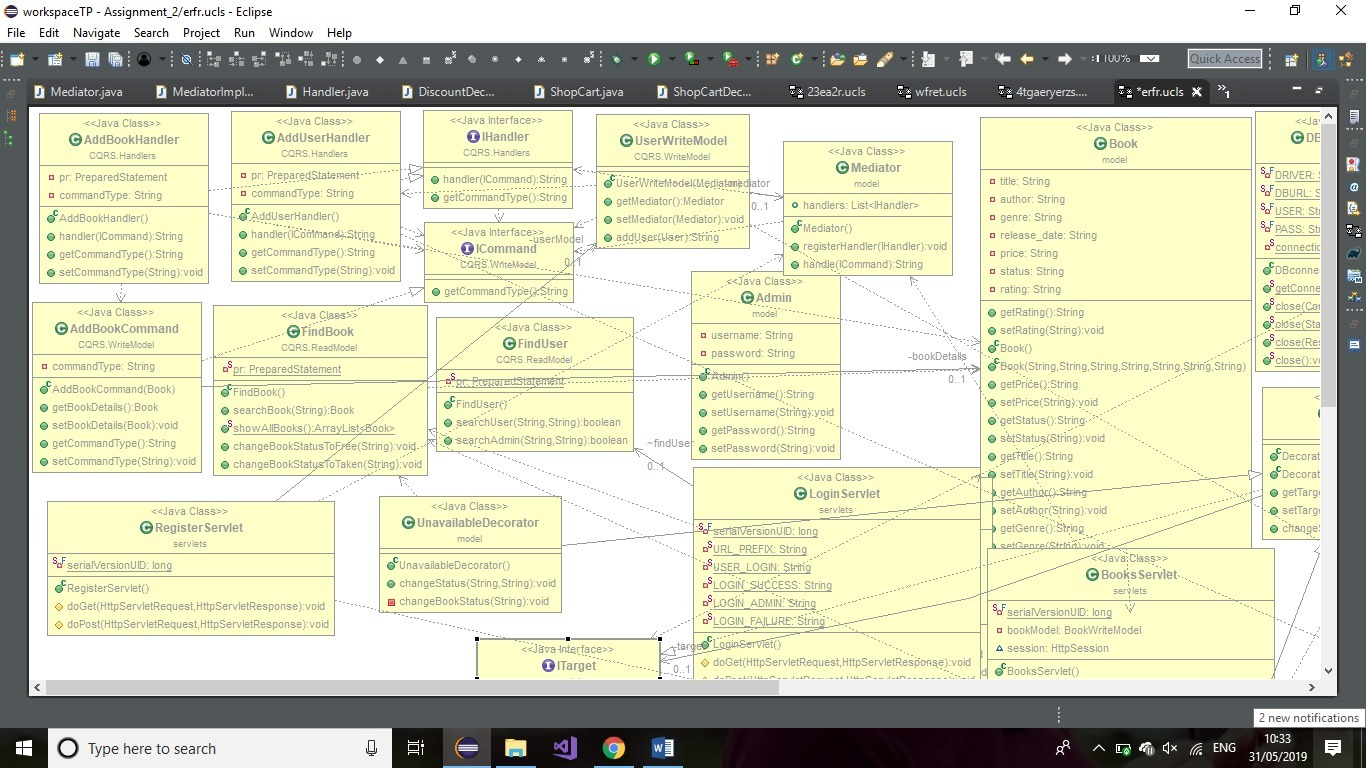
Admin’s use case model:



**Package diagram**



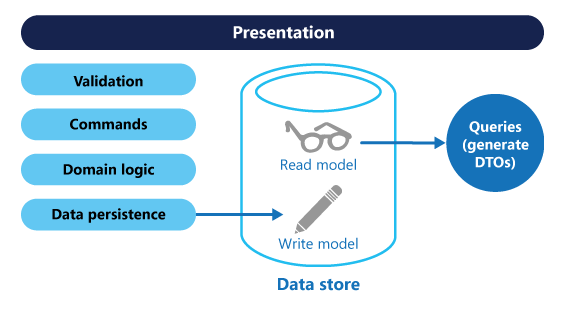
**Class diagram**



3. System Architectural Design

**3.1 Architectural Pattern Description**

**CQRS:** Segregate operations that read data from operations that update data by using separate interfaces. This can maximize performance, scalability, and security. Supports the evolution of the system over time through higher flexibility, and prevents update commands from causing merge conflicts at the domain level.

In traditional architectures, the same data model is used to query and update a database. That's simple and works well for basic CRUD operations. In more complex applications, however, this approach can become unwieldy. For example, on the read side, the application may perform many different queries, returning data transfer objects (DTOs) with different shapes. Object mapping can become complicated. On the write side, the model may implement complex validation and business logic. As a result, you can end up with an overly complex model that does too much.

Read and write workloads are often asymmetrical, with very different performance and scale requirements.

5. Class Design

**5.1 Design Patterns Description**

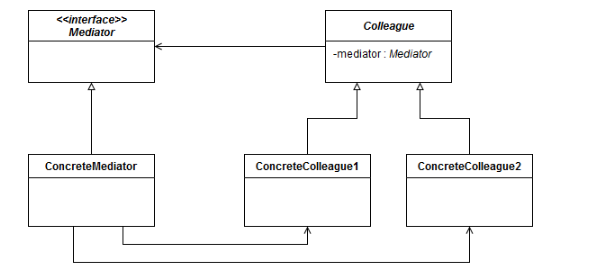
**Mediator pattern**

In object-oriented programming, we should always try to design the system in such a way that components are loosely coupled and reusable. This approach makes our code easier to maintain and test.

In real life, however, we often need to deal with a complex set of dependent objects. This is when the **Mediator Pattern** may come in handy.

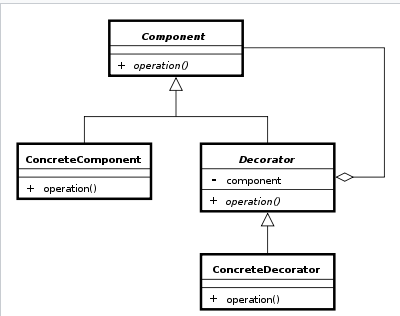
The intent of the **Mediator Pattern** is to reduce the complexity and dependencies between tightly coupled objects communicating directly with one another. This is achieved by creating a mediator object that takes care of the interaction between dependent objects. Consequently, all the communication goes through the mediator.

This promotes loose coupling, as a set of components working together no longer have to interact directly. Instead, they only refer to the single mediator object. This way, it is also easier to reuse these objects in other parts of the system.



**Decorator pattern**

In [object-oriented programming](https://en.wikipedia.org/wiki/Object-oriented_programming), the **decorator pattern** is a [design pattern](https://en.wikipedia.org/wiki/Design_pattern_(computer_science)) that allows behavior to be added to an individual [object](https://en.wikipedia.org/wiki/Object_(computer_science)), dynamically, without affecting the behavior of other objects from the same [class](https://en.wikipedia.org/wiki/Class_(computer_science)). The decorator pattern is often useful for adhering to the [Single Responsibility Principle](https://en.wikipedia.org/wiki/Single_responsibility_principle), as it allows functionality to be divided between classes with unique areas of concern. The decorator pattern is structurally nearly identical to the [chain of responsibility pattern](https://en.wikipedia.org/wiki/Chain_of_responsibility_pattern), the difference being that in a chain of responsibility, exactly one of the classes handles the request, while for the decorator, all classes handle the request.



8. Bibliography

* <https://www.sciencedirect.com/topics/computer-science/business-logic-layer>
* <https://objcsharp.wordpress.com/2013/07/22/what-is-a-business-logic-layer-anyway/>
* <https://www.tutorialspoint.com/design_pattern/factory_pattern.htm>