A2: Your Books Everywhere!

Analysis and Design Document

Student: Adrian Moldovan

**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 4

3. System Architectural Design 4

4. UML Sequence Diagrams 7

5. Class Design 7

6. Data Model 8

7. System Testing 10

8. Bibliography 10

1. Requirements Analysis

# Assignment Specification

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

-User Registration

-Payment system

-Library management

-Dynamic recommendations

-Store data

-Input data has to be validated

# Non-functional Requirements

-Payments:cash only polcy

-Waiting list for unavailable book

-Library is managed by staff.

-Transactions validated by staff

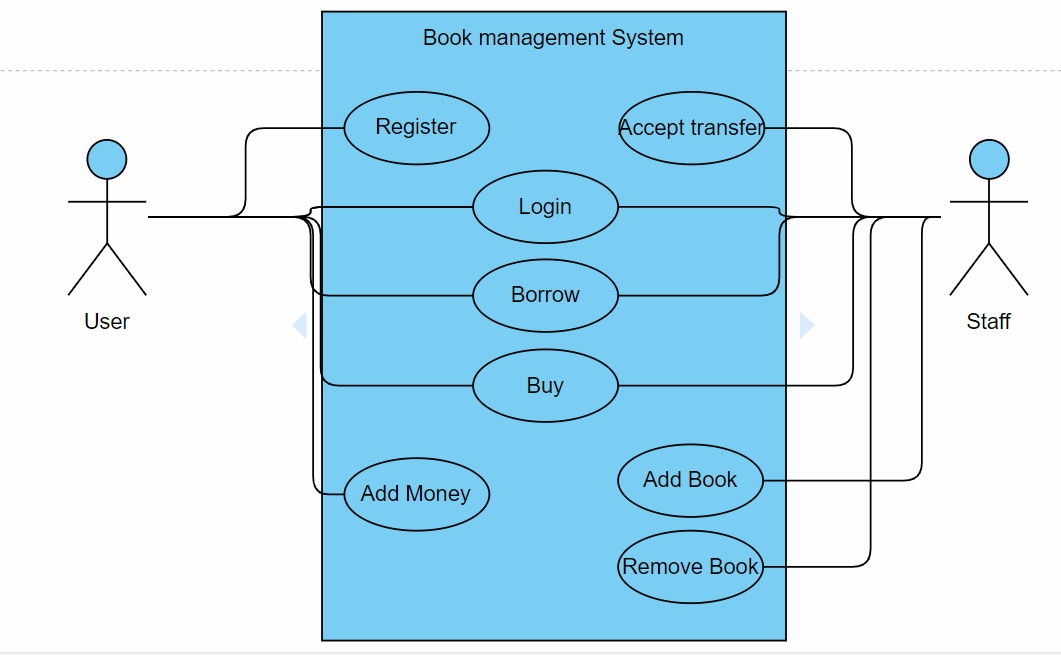
-Recommendations based on latest trends or user defined Interests by genre or topic

-Observer DP

-Factory method for building user recommendations

-Use Database for storage

2. Use-Case Model



3. System Architectural Design

**3.1 Architectural Pattern Description**

**Model-View-Controller:**

**Model–View–Controller** (usually known as MVC) is an [architectural pattern](https://en.wikipedia.org/wiki/Architectural_pattern) commonly used for developing [user interfaces](https://en.wikipedia.org/wiki/User_interface) that divides an application into three interconnected parts. This is done to separate internal representations of information from the ways information is presented to and accepted from the user. The MVC design pattern decouples these major components allowing for efficient [code reuse](https://en.wikipedia.org/wiki/Code_reuse) and parallel development.

**Model**

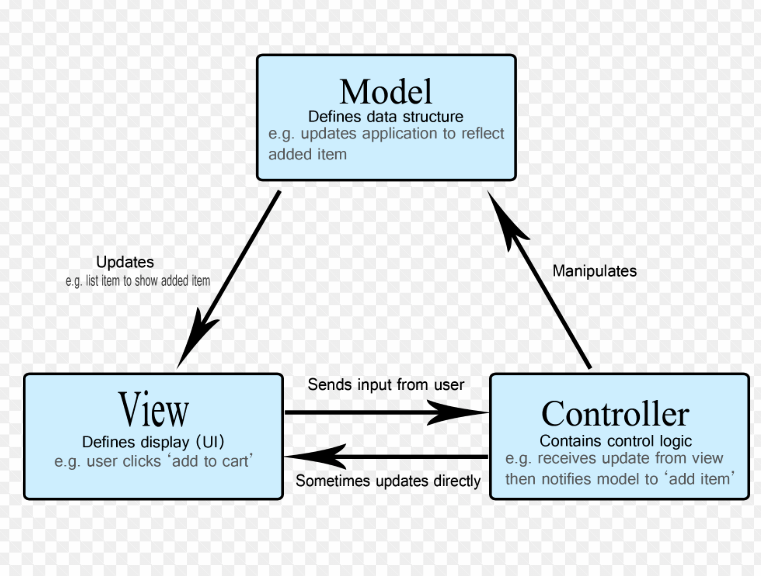
The central component of the pattern. It is the application's dynamic data structure, independent of the user interface.[[4]](https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller#cite_note-4) It directly manages the data, logic and rules of the application.

**View**

Any representation of information such as a chart, diagram or table. Multiple views of the same information are possible, such as a bar chart for management and a tabular view for accountants.

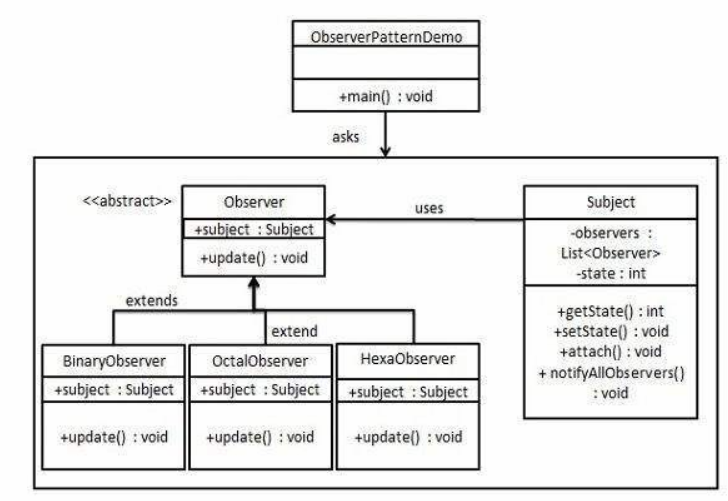
**Controller**

Accepts input and converts it to commands for the model or view.



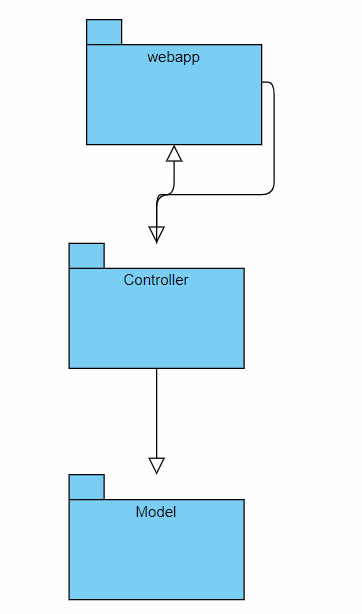
**Observer Design Pattern:**

* Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
* Encapsulate the core (or common or engine) components in a Subject abstraction, and the variable (or optional or user interface) components in an Observer hierarchy.
* The "View" part of Model-View-Controller.
* Define an object that is the "keeper" of the data model and/or business logic (the Subject). Delegate all "view" functionality to decoupled and distinct Observer objects. Observers register themselves with the Subject as they are created. Whenever the Subject changes, it broadcasts to all registered Observers that it has changed, and each Observer queries the Subject for that subset of the Subject's state that it is responsible for monitoring.
* This allows the number and "type" of "view" objects to be configured dynamically, instead of being statically specified at compile-time.
* The protocol described above specifies a "pull" interaction model. Instead of the Subject "pushing" what has changed to all Observers, each Observer is responsible for "pulling" its particular "window of interest" from the Subject. The "push" model compromises reuse, while the "pull" model is less efficient.
* Issues that are discussed, but left to the discretion of the designer, include: implementing event compression (only sending a single change broadcast after a series of consecutive changes has occurred), having a single Observer monitoring multiple Subjects, and ensuring that a Subject notify its Observers when it is about to go away.
* The Observer pattern captures the lion's share of the Model-View-Controller architecture that has been a part of the Smalltalk community for years.



**3.2 Diagrams**

**Package Diagram:**



4. UML Sequence Diagrams

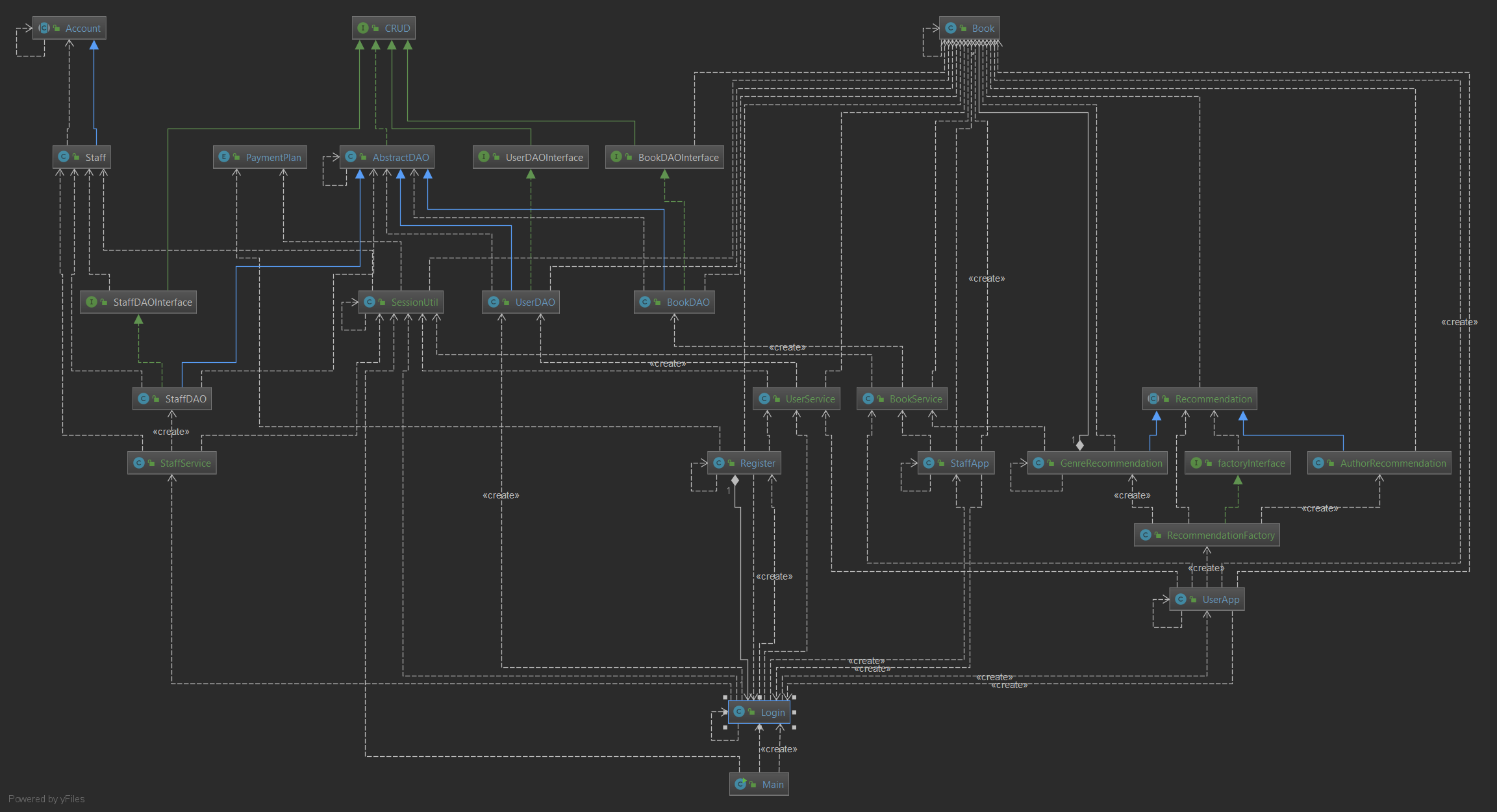
5. Class Design

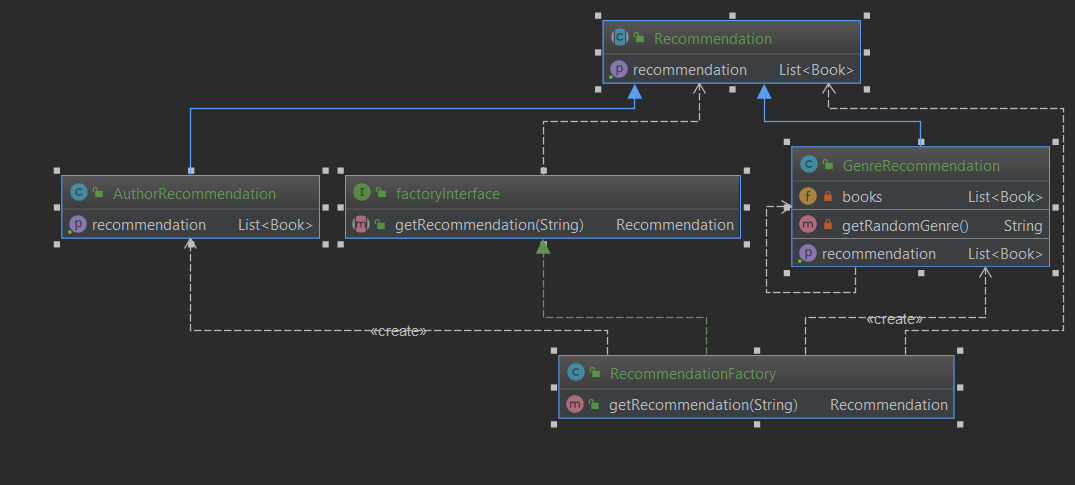
**5.1 Design Patterns Description**

**Factory method pattern:**

The **factory method pattern** is a [creational](https://en.wikipedia.org/wiki/Creational_pattern) pattern that uses factory methods to deal with the problem of creating objects without having to specify the exact class of the object that will be created. This is done by creating objects by calling a factory method—either specified in an interface and implemented by child classes, or implemented in a base class and optionally overridden by derived classes rather than by calling a constructor.

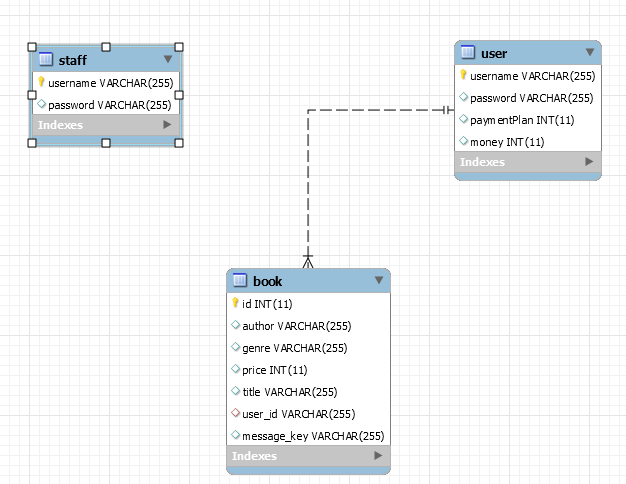
**5.2 UML Class Diagram**

**





6. Data Model



7. System Testing

8. Bibliography

<https://www.baeldung.com/simplifying-the-data-access-layer-with-spring-and-java-generics>

<https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller>

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