A2: Your Books Everywhere!

Analysis and Design Document

Student: Adrian Moldovan
Group: 30238

Table of Contents

1. Requirements Analysis

1.1Assignment 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.

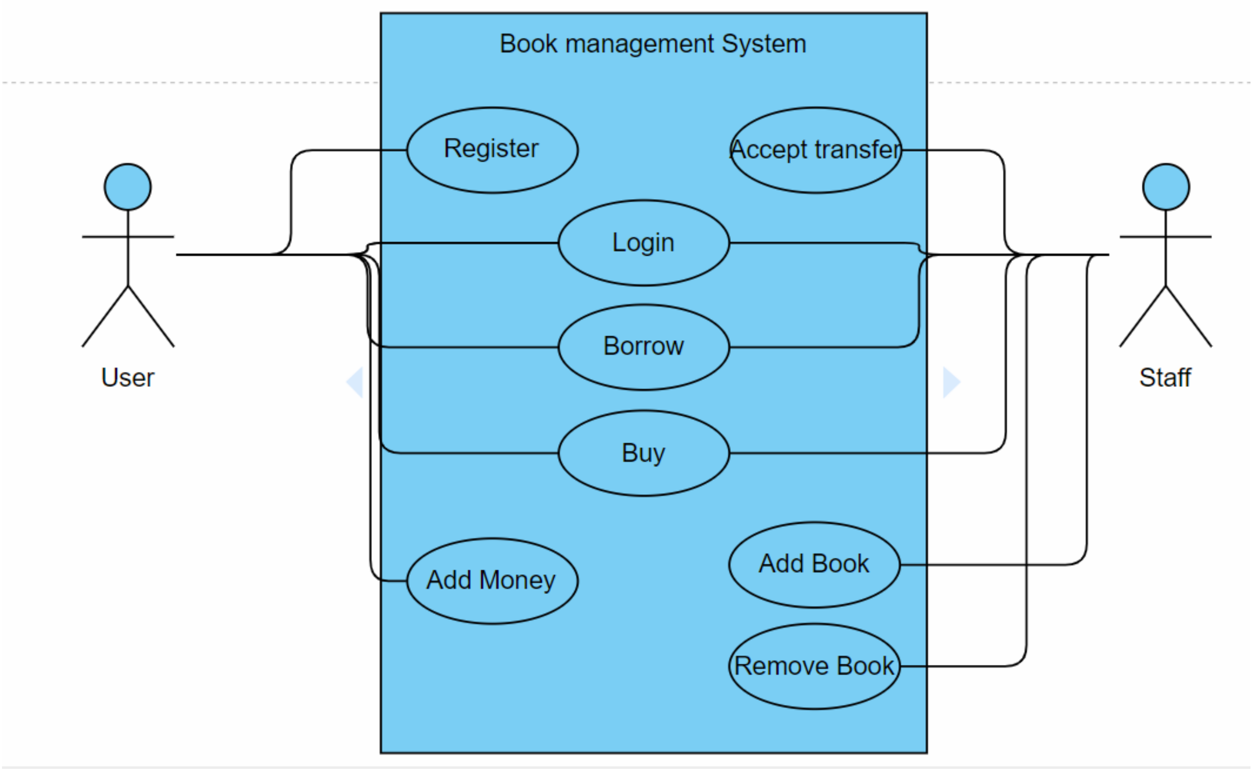
1.2 Functional Requirements

-User Registration
-Payment system
-Library management
-Dynamic recommendations
-Store data
-Input data has to be validated

1.3 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:

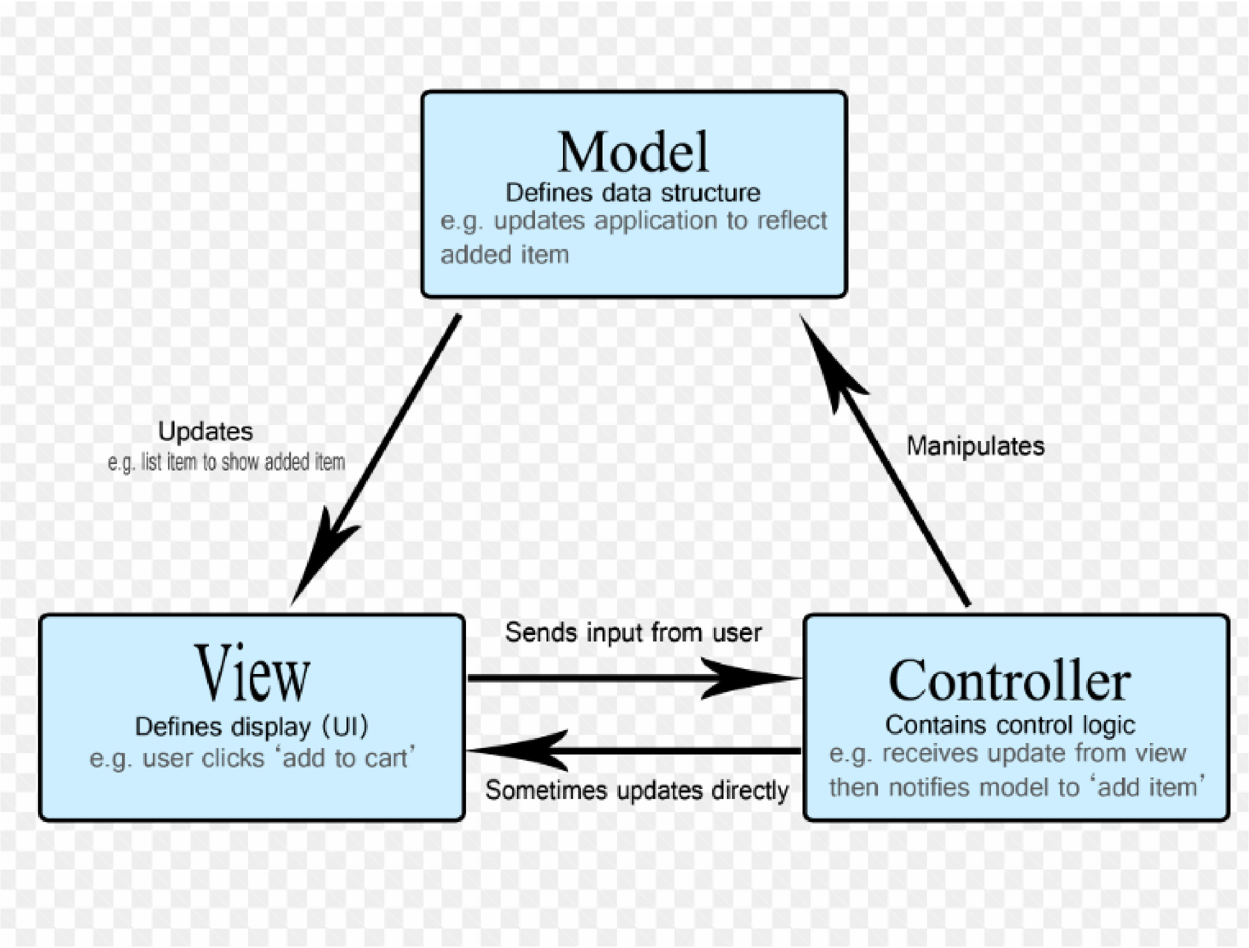
Model–View–Controller (usually known as MVC) is an architectural pattern commonly used for developing user interfaces that divides an application into three interconnected parts. This is done to separate internal representations of information from the ways information is presented to and accepted from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

Model
The central component of the pattern. It is the application's dynamic data structure, independent of the user interface.[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

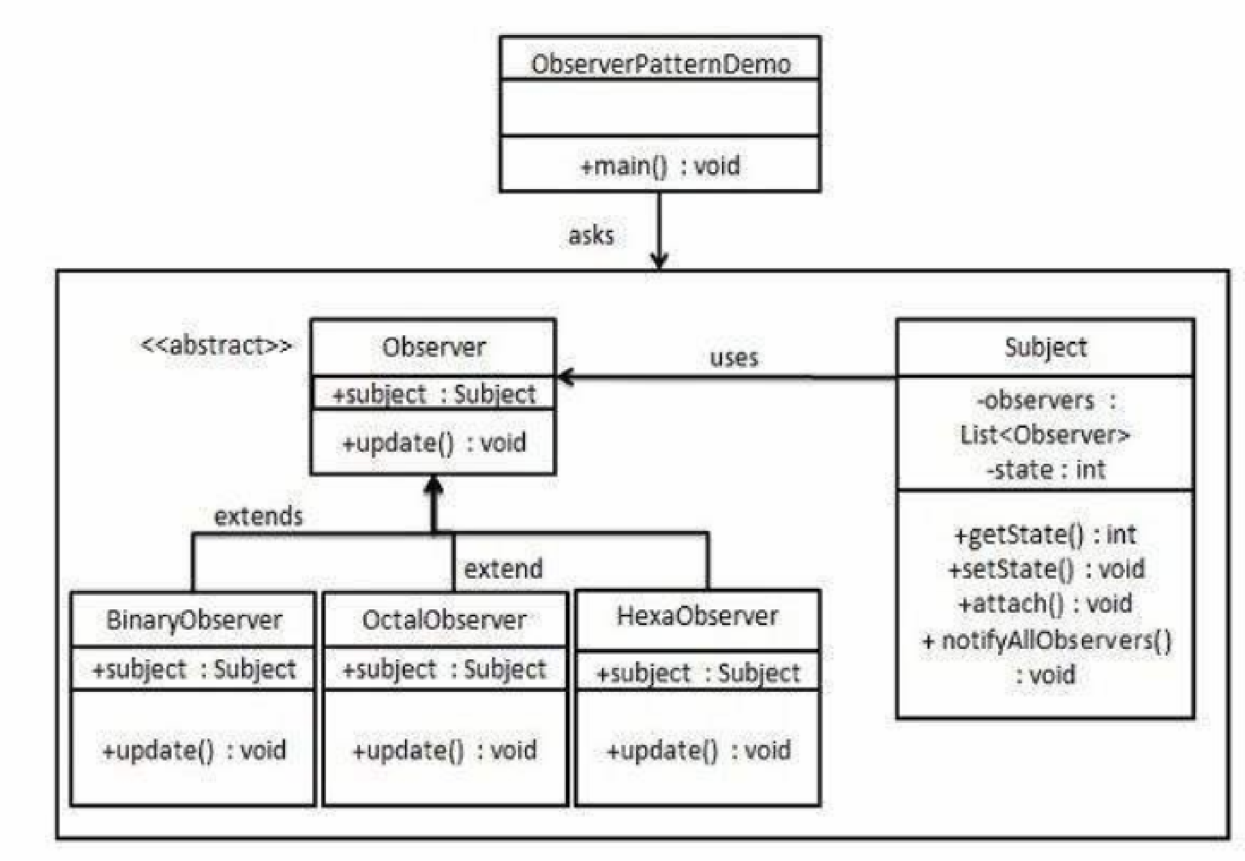
Accepts input and converts it to commands for the model or view.


Model
Defines data structure
e.g. updates application to reflect added item
Updates
e.g. list item to show added item
Manipulates
View
Defines display (UI)
e.g. user clicks 'add to cart'
Sends input from user
Sometimes updates directly
Controller
Contains control logic
e.g. receives update from view then notifies model to 'add item'

Observer Design Pattern:

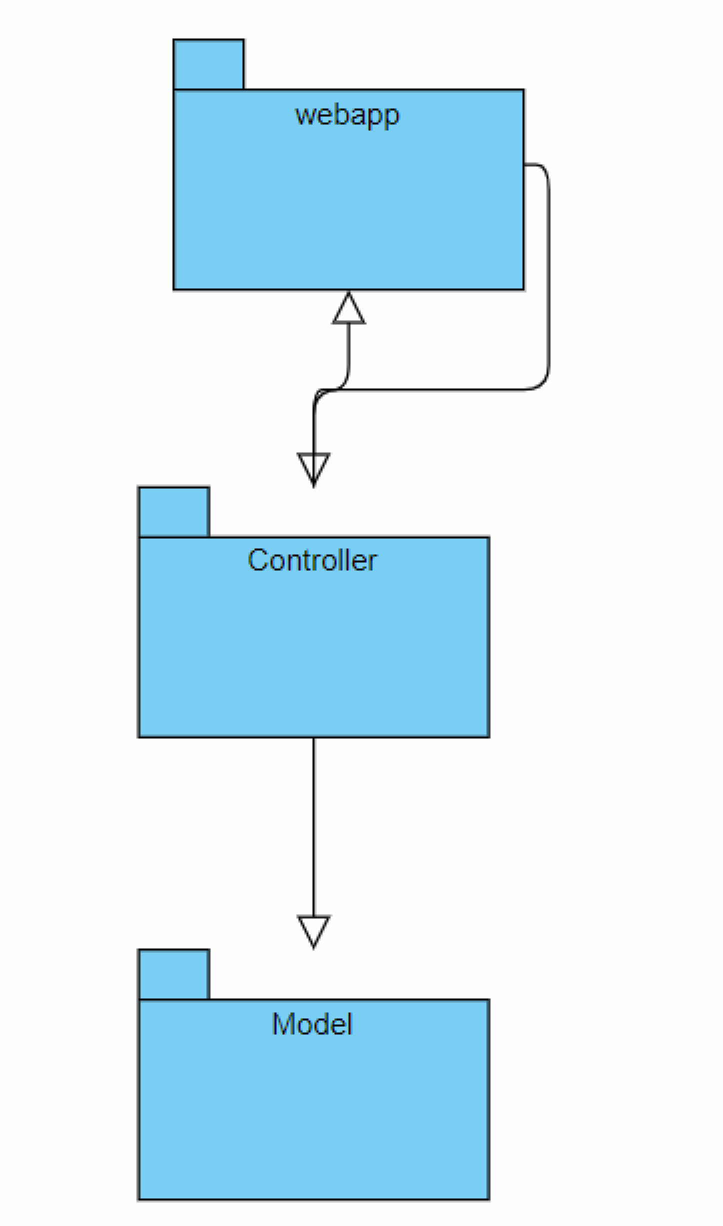
- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- Encapsulate the core (or common or engine) components in a Subject abstraction, and the variable (or optional or user interface) components in an Observer 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

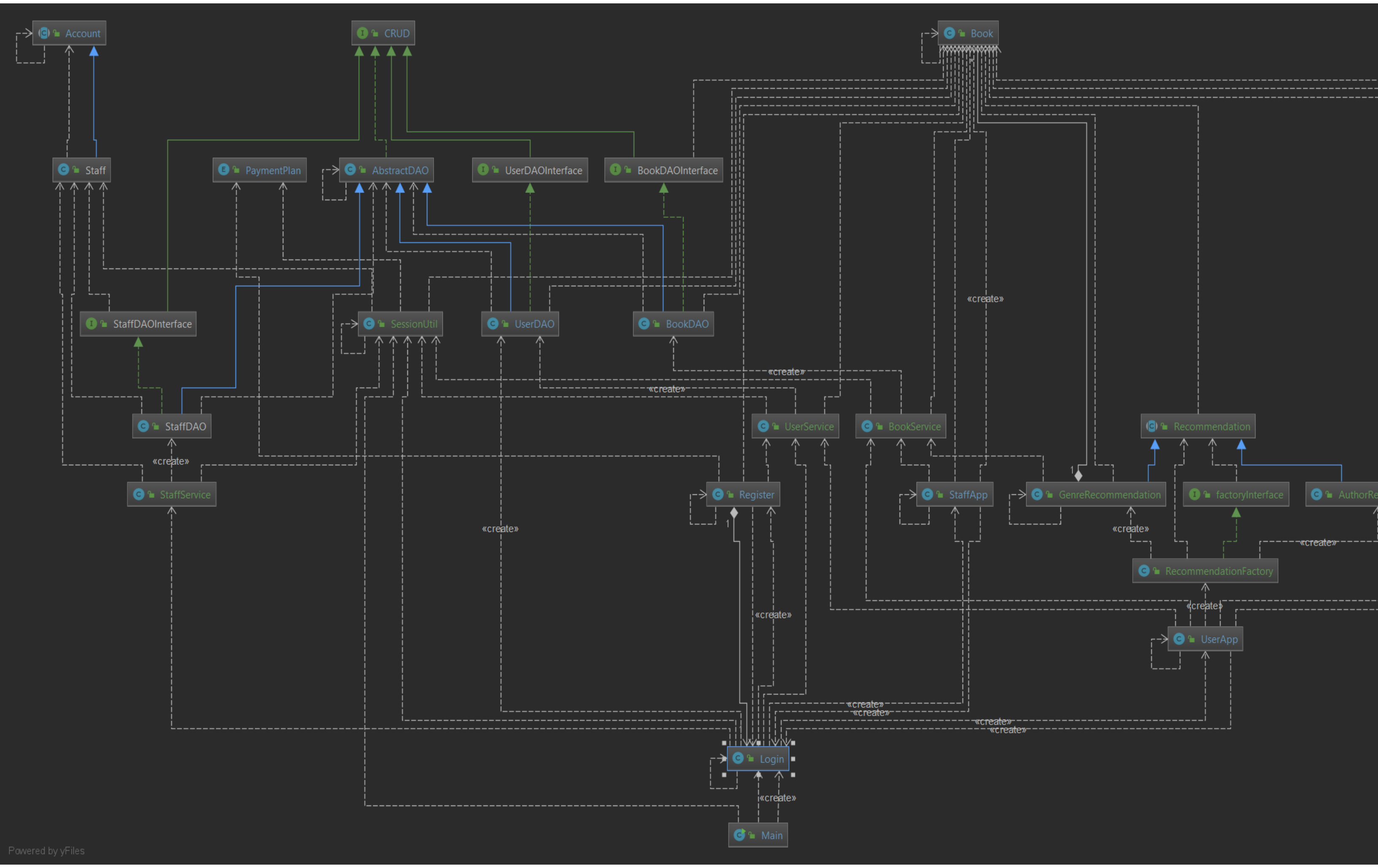
5. Class Design

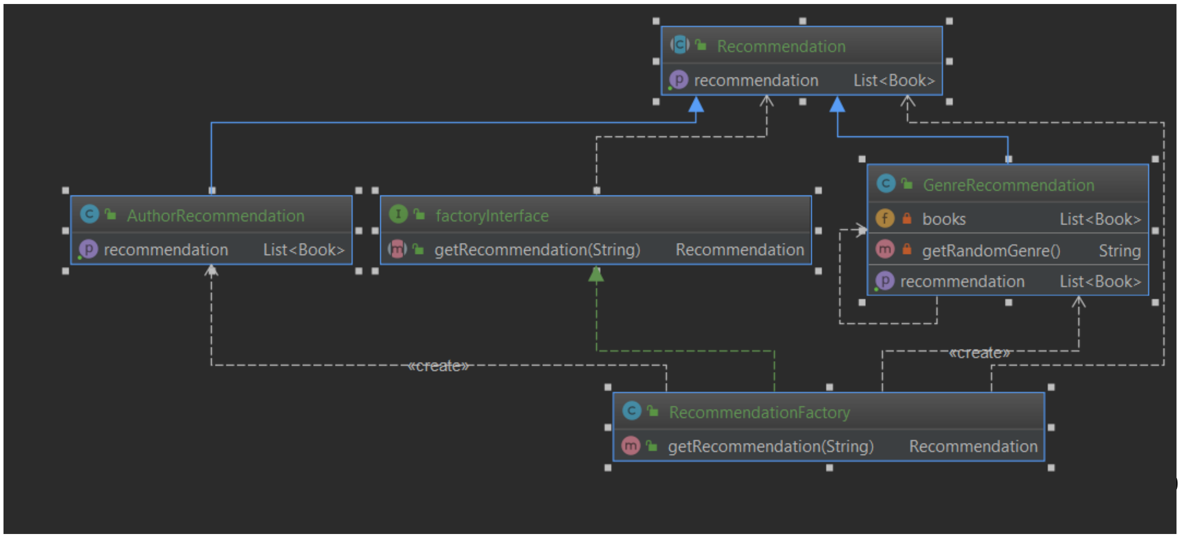
5.1 Design Patterns Description

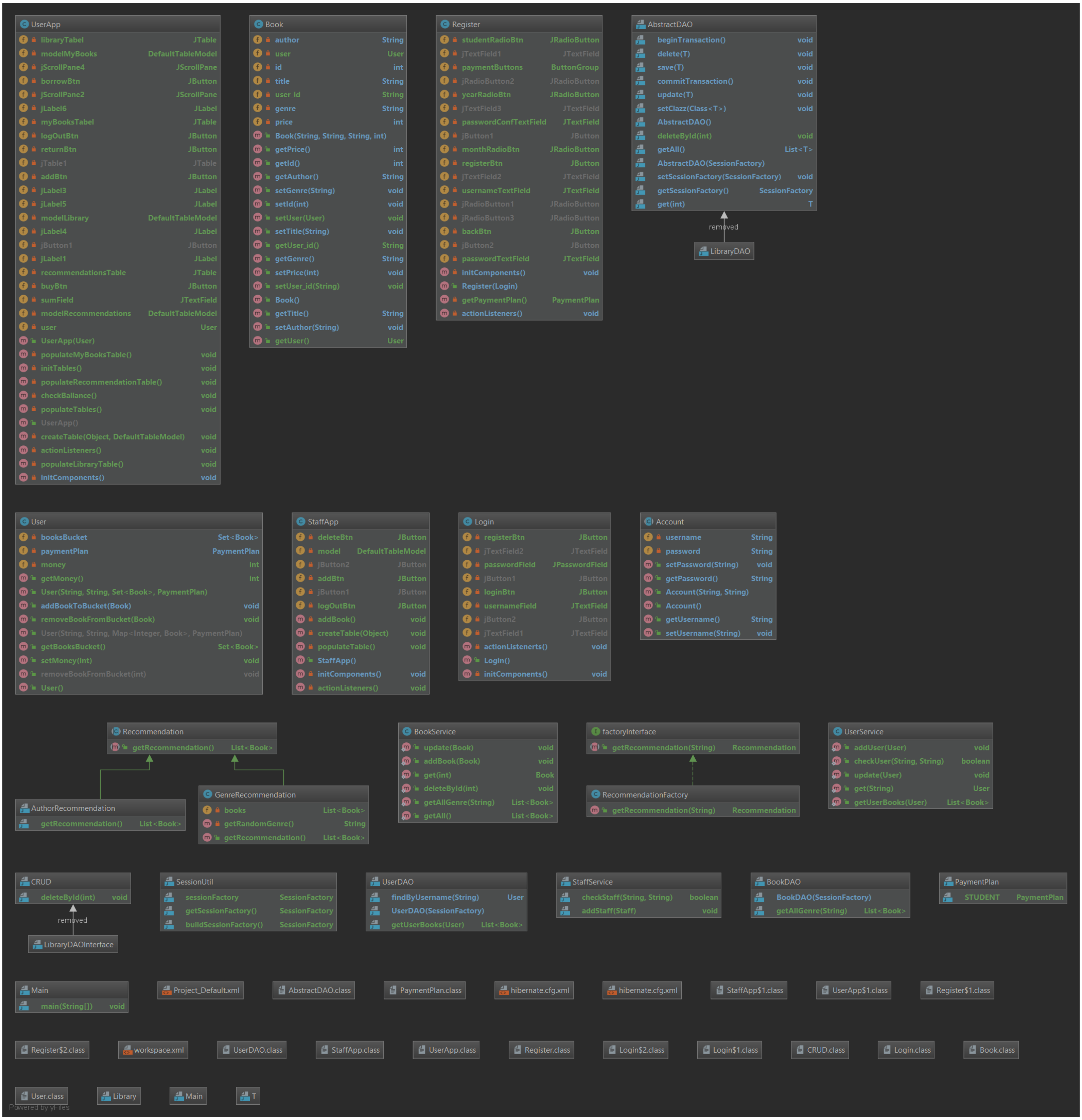
Factory method pattern:

The **factory method pattern** is a creational 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.

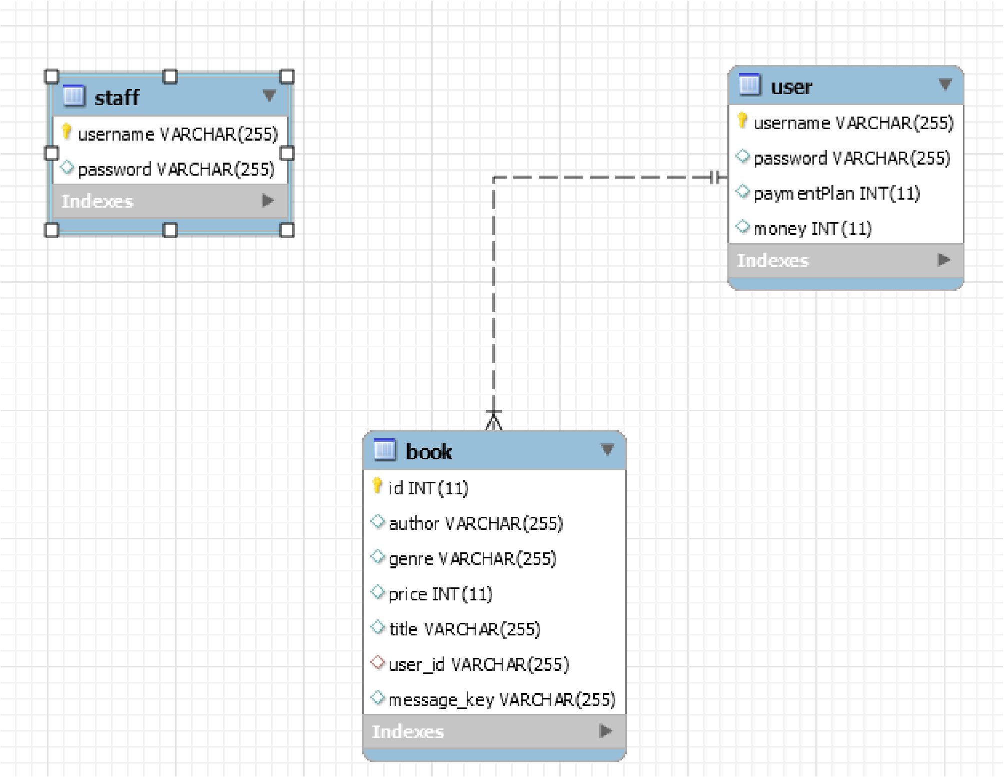
5.2 UML Class Diagram




Recommendation
recommendation List<Book>
AuthorRecommendation
recommendation List<Book>
factoryInterface
getRecommendation(String) Recommendation
GenreRecommendation
books List<Book>
getRandomGenre() String
recommendation List<Book>
«create»
«create»
RecommendationFactory
getRecommendation(String) Recommendation


UserApp
libraryTabel JTable
modelMyBooks DefaultTableModel
jScrollPane4 JScrollPane
borrowBtn JButton
jScrollPane2 JScrollPane
jLabel6 JLabel
myBooksTabel JTable
logOutBtn JButton
returnBtn JButton
jTable1 JTable
addBtn JButton
jLabel3 JLabel
jLabel5 JLabel
modelLibrary DefaultTableModel
jLabel4 JLabel
jButton1 JButton
jLabel1 JLabel
recommendationsTable JTable
buyBtn JButton
sumField JTextField
modelRecommendations DefaultTableModel
user User
UserApp(User)
populateMyBooksTable() void
initTables() void
populateRecommendationTable() void
checkBallance() void
populateTables() void
UserApp()
createTable(Object, DefaultTableModel) void
actionListeners() void
populateLibraryTable() void
initComponents() void
Book
author String
user User
id int
title String
user_id String
genre String
price int
Book(String, String, String, int)
getPrice() int
getId() int
getAuthor() String
setGenre(String) void
setId(int) void
setUser(User) void
setTitle(String) void
getUser_id() String
getGenre() String
setPrice(int) void
setUser_id(String) void
Book()
getTitle() String
setAuthor(String) void
getUser() User
Register
studentRadioBtn JRadioButton
jTextField1 JTextField
paymentButtons ButtonGroup
jRadioButton2 JRadioButton
yearRadioBtn JRadioButton
jTextField3 JTextField
passwordConfTextField JTextField
jButton1 JButton
monthRadioBtn JRadioButton
registerBtn JButton
jTextField2 JTextField
usernameTextField JTextField
jRadioButton1 JRadioButton
jRadioButton3 JRadioButton
backBtn JButton
jButton2 JButton
passwordTextField JTextField
initComponents() void
Register(Login)
getPaymentPlan() PaymentPlan
actionListeners() void
AbstractDAO
beginTransaction() void
delete(T) void
save(T) void
commitTransaction() void
update(T) void
setClazz(Class<T>) void
AbstractDAO()
deleteById(int) void
getAll() List<T>
AbstractDAO(SessionFactory)
setSessionFactory(SessionFactory) void
getSessionFactory() SessionFactory
get(int) T
removed
LibraryDAO
User
booksBucket Set<Book>
paymentPlan PaymentPlan
money int
getMoney() int
User(String, String, Set<Book>, PaymentPlan)
addBookToBucket(Book) void
removeBookFromBucket(Book) void
User(String, String, Map<Integer, Book>, PaymentPlan)
getBooksBucket() Set<Book>
setMoney(int) void
removeBookFromBucket(int) void
User()
StaffApp
deleteBtn JButton
model DefaultTableModel
jButton2 JButton
addBtn JButton
jButton1 JButton
logOutBtn JButton
addBook() void
createTable(Object) void
populateTable() void
StaffApp()
initComponents() void
actionListeners() void
Login
registerBtn JButton
jTextField2 JTextField
passwordField JPasswordField
jButton1 JButton
loginBtn JButton
usernameField JTextField
jButton2 JButton
jTextField1 JTextField
actionListenerts() void
Login()
initComponents() void
Account
username String
password String
setPassword(String) void
getPassword() String
Account(String, String)
Account()
getUsername() String
setUsername(String) void
Recommendation
getRecommendation() List<Book>
AuthorRecommendation
getRecommendation() List<Book>
GenreRecommendation
books List<Book>
getRandomGenre() String
getRecommendation() List<Book>
BookService
update(Book) void
addBook(Book) void
get(int) Book
deleteById(int) void
getAllGenre(String) List<Book>
getAll() List<Book>
factoryInterface
getRecommendation(String) Recommendation
RecommendationFactory
getRecommendation(String) Recommendation
UserService
addUser(User) void
checkUser(String, String) boolean
update(User) void
get(String) User
getUserBooks(User) List<Book>
CRUD
deleteById(int) void
removed
LibraryDAOInterface
SessionUtil
sessionFactory SessionFactory
getSessionFactory() SessionFactory
buildSessionFactory() SessionFactory
UserDAO
findByUsername(String) User
UserDAO(SessionFactory)
getUserBooks(User) List<Book>
StaffService
checkStaff(String, String) boolean
addStaff(Staff) void
BookDAO
BookDAO(SessionFactory)
getAllGenre(String) List<Book>
PaymentPlan
STUDENT PaymentPlan
Main
main(String[]) void
Project_Default.xml
AbstractDAO.class
PaymentPlan.class
hibernate.cfg.xml
hibernate.cfg.xml
StaffApp$1.class
UserApp$1.class
Register$1.class
Register$2.class
workspace.xml
UserDAO.class
StaffApp.class
UserApp.class
Register.class
Login$2.class
Login$1.class
CRUD.class
Login.class
Book.class
User.class
Library
Main
T
Powered by yFiles

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