**Summer**

14

Social Library

Siva Lalita Vissamsetty (00001032984) & Harsha Teja Kanikicherla (00001071900)

This document explains the application and the design patterns used to implement the application.

08

**Fall**

Table of Contents

1. Abstract 3

2. Scope 3

3. Tools, Servers & Programming Languages 3

4. Functional Requirements 3

6. Database Design 4

7. Design Patterns 7

7.1 Singleton Pattern 8

7.2 Factory Method 12

7.3 Command Pattern 16

7.4 Façade Pattern 28

7.5 Observer Pattern 31

7.6 Template Pattern 41

7.7 Strategy Pattern 45

7.8 Null Object Pattern 47

7.9 Decorator Pattern 49

7.10 Single Access Point 51

7.11 Front Controller Pattern 51

8. Class Diagram 52

9. Future Work 53

10. Take Away Points 53

# Abstract

This document explains about the application, ‘Social Library’, and the design patterns used during the implementation. It also gives a justification of why each of the patterns has been used and what were the advantages of using them.

# 2. Scope

Social Library is a web application that helps a small group of people share books among each other. The problem we solved with this application is that, it helps the users of the system keep track of the book. The user adds the book he wants to share, and once this book is being shared he can keep an eye on the book and stay updated with information like who holds the book currently and the list of users who have placed a request for the book. This would give him an idea of when he can expect the book to get back to him. This makes it easy for him to track and request back from the person who currently has it at any point of time.

# 3. Tools, Servers & Programming Languages

* Programming Languages: Java
* Web Server: Apache Tomcat (Version: 7.0)
* Database Server: MySql (Version: 5.6.11)
* Tools: Eclipse JEE (IDE, Version: Kepler Service Release 2), MySqlWorkbench AstahCommunity (UML)

# 4. Functional Requirements

* User registers with the system by filling his information along with the username and password.
* User should be able to create a group or join an existing group. He should be able to add a book and delete the books he added.
* User should be able to place a request for a book owned by the other members of the groups he belongs to.
* User should be able to view his profile information (his name, email etc.), the books he owns, he borrowed and he placed a request for.
* The members of the group(s) the user belongs to should be notified when a book is added or deleted.
* User should be able to search for members based on their first and last names, books based on its ISBN or Book name.
* User should be able to update the status of the book.
* When User clicks on Request/Delete for a book, the book has to be queued so that he/she should be able to view the list of books he wants to request for or delete and execute them all together.
* User should be able to buy, sell books.
* When purchasing a book the user should be able to select a payment method (Cash / Bit Coin) and the delivery method for the book (Self Pick Up / Deliver to Door). The order amount varies based on the options selected.

5. Assumptions

* No two users login at the same time.
* Only the status and the borrower of a book can be updated. Once the user adds a book, the basic information like the Book name etc. cannot be updated.
* When searching for a particular member the user is aware of and enters the first and last name of the member.

# 6. Database Design

MySql has been used as the Database Server. The Table 1.1 below gives the details of the database schema used for the application.

|  |
| --- |
| **Database Schema**: sociallibrary\_development |

|  |  |  |
| --- | --- | --- |
| **Table Name** | **Column** **Names** | **Description** |
| members | id  firstname  lastname  username  password  address  email | id- is the primary key of the table, which identifies each row uniquely.  The other columns store the information of the member. A record is created whenever a user registers with the system. |
| books | id  bookname  category\_id  rating  ISBN  price | id- is the primary key of the table, which identifies each row uniquely.  rating – holds the average of the rating given by each of the members who owns it.  If the books entered by the user does not exist already a record is created in this table as well as memberbooks. |
| bookcategories | id  category\_name | id- is the primary key of the table, which identifies each row uniquely.  category\_name- holds the different categories books can belong to  A record is created whenever a new category of book is created. |
| memberbooks | id  book\_id  owner\_id  borrower\_id memberrating last\_updated\_at availability  purchasable | id- is the primary key of the table, which identifies each row uniquely.  book\_id- is the foreign key, which corresponds to the primary key of the table books.  owner\_id - is the foreign key, which corresponds to the primary key of the table members. It contains the id of the member who is the owner of a book.  borrower\_id - is the foreign key, which corresponds to the primary key of the table members. It contains the id of the member who is the borrower of a book.  memberrating- holds the rating giving by the owner while adding the book.  availability- holds the information if the book is available to be delivered or borrowed by another member of the groups he belongs to.  A record is created in this table whenever a member adds a book that exists already or not. If the book exists already a record is created only here and not in books. |
| bookrequest | id  member\_id  member\_book\_id | id- is the primary key of the table, which identifies each row uniquely.  member\_id- is the foreign key, which corresponds to the primary key of the table members.  member\_book\_id- is the foreign key, which corresponds to the primary key of the table books.  A record is created whenever the user places a request for a book. |
| groups | id  groupname | id- is the primary key of the table, which identifies each row uniquely.  groupname- holds the name of a group.  A record is created whenever a group is created. |
| membergroups | group\_id  member\_id  id | id- is the primary key of the table, which identifies each row uniquely.  member\_id- is the foreign key, which corresponds to the primary key of the table members.  A record is created in this table whenever a member joins a particular group. A member can belong to multiple groups. |
| notifications | member\_id  notification | Member\_id- is the foreign key of the table, which corresponds to the primary key of the table ‘members’ (id).  Notification- holds the notification for a particular member. |
| nullreference | nullvalue | Holds the values to be displayed when objects return |

# 7. Design Patterns

The design patterns used for this project are as follows:

1. Singleton Pattern
2. Factory Method
3. Command Pattern
4. Façade Pattern
5. Observer Pattern
6. Template Pattern
7. Strategy Pattern
8. Null Object Pattern
9. Decorator
10. Single Access Point
11. Front Controller Pattern

The following sub-sections explain the usage of each of the patterns.

## 7.1 Singleton Pattern

Singleton Pattern is used when we want to restrict the number of instances to created for a particular class.

**Classes involved:**

* DatabaseConnection
* BookServiceController
* MemberServiceController
* CurrentSession

**Reason:**

This design pattern has been created since we needed to have a single instance for the above-specified classes.

Establishing a connection with the database every time the data has to be retrieved would be a costly approach. Hence it was necessary that we have a single instance for creating a connection and use the same instance whenever database has to be queried. The class DatabaseConnection has implemented the same.

The classes BookServiceController and MemberServiceController, controls all the operations, related to books (adding, deleting, updating books etc.) and members (login, sign out, joining groups etc.) that are to be performed. The client communicates only with these two classes to execute any actions. Hence, having a single instance of these classes would be less expensive, instead of having an instance created every time any user wants to do anything.

The class CurrentSession, maintains the information of the user/member who is currently logged in. The requirement is, once a user logs in the information should be maintained all through the session and no more instances should be created. Therefore, having a single instance of this class would solve this requirement.

**Class Diagram:**

The Figure 7.1 below is the class diagram for the Singleton class as explained in the previous subsections.

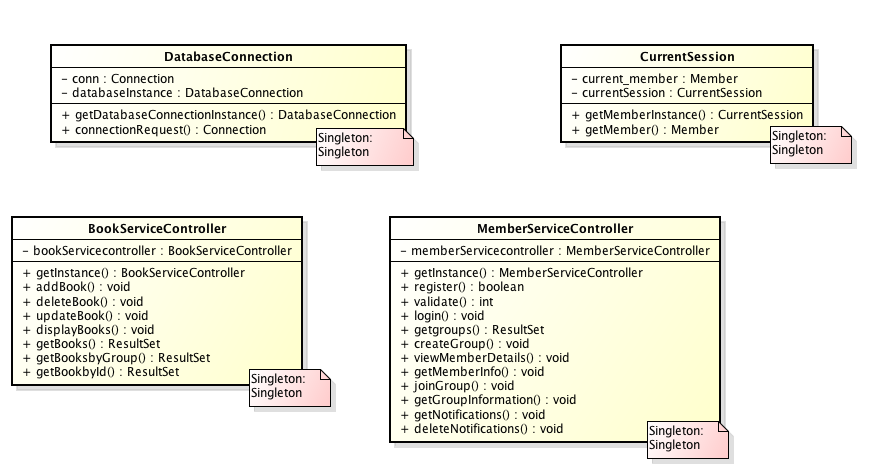
****

Figure 7.1

**Advantage:**

Having a singleton class will facilitate using the same instance to perform actions. We do not have to create a new instance every time you want to execute an action when it is not necessary.

**Source Code:**

**a. Class: DatabaseConnection**

**public** **class** DatabaseConnection {

**private** **static** Connection *conn*;

**private** **static** DatabaseConnection *databaseInstance*=**null**;

**private** DatabaseConnection(){

}

**public** **static** DatabaseConnection getDatabaseConnectionInstance(){

**if**(*databaseInstance*==**null**){

*databaseInstance*=**new** DatabaseConnection();

**return** *databaseInstance*;

}

**else**{

**return** *databaseInstance*;

}

}

**public** **static** Connection connectionRequest(){

*getDatabaseConnectionInstance*();

String url = "jdbc:mysql://localhost:3306/";

String dbName = "sociallibrary\_development";

String driver = "com.mysql.jdbc.Driver";

String userName = "root";

String password = "sridatta";

**if**(*conn*!=**null**){

**return** *conn*;

}

**else**{

**try** {

Class.*forName*(driver).newInstance();

*conn* = DriverManager.*getConnection*(url+dbName,userName,password);

**return** *conn*;

}**catch** (Exception e) {

e.printStackTrace();

}

}

**return** **null**;

}

}

**b. Class: MemberServiceController**

The following code snippet includes the code related to Singleton pattern only.

**public** **class** MemberServiceController {

**private** **static** MemberServiceController *memberServicecontroller*=**null**;

**private** MemberServiceController(){

}

//This method returns an instance for MemberServiceController and the same instance is used through out the application

**public** **static** MemberServiceController getInstance(){

**if**(*memberServicecontroller*==**null**){

*memberServicecontroller*=**new** MemberServiceController();

**return** *memberServicecontroller*;

}

**return** *memberServicecontroller*;

}

**}**

**c. Class: BookServiceController**

**public** **class** BookServiceController {

**private** **static** BookServiceController *bookServicecontroller*=**null**;

**private** BookServiceController(){

}

//This method returns an instance for BookServiceController and the same instance is used through out the application

**public** **static** BookServiceController getInstance(){

**if**(*bookServicecontroller*==**null**){

*bookServicecontroller*=**new** BookServiceController();

**return** *bookServicecontroller*;

}

**return** *bookServicecontroller*;

}

**}**

**d. Class: CurrentSession**

**public** **class** CurrentSession {

//current\_member holds the information of the member who is currently logged in

**private** **static** Member *current\_member*;

**private** **static** CurrentSession *cm*=**null**;

**private** CurrentSession(Member member){

*current\_member*=member;

}

// returns an instance of this class and the same instance is used through out

**public** **static** CurrentSession getMemberInstance(Member member){

**if**(*cm*==**null**){

*cm*=**new** CurrentSession(member);

**return** *cm*;

}

**else**{

*current\_member*=member;

**return** *cm*;

}

}

//current\_member variable is set once a user logs in. The same current\_member info is returned whenever this method is called

**public** **static** Member getMember(){

**if**(*cm*!=**null** && *current\_member*!=**null**){

**return** *current\_member*;

}

**else** **return** **null**;

}

//current\_member variable is reset to null so that it can be set to next user who logs in

**public** **static** **void** resetCurrentMember(){

*current\_member*=**null**;

}

}

## 7.2 Factory Method

The Factory Method pattern is a creational pattern that is used when the system has to take care of creating instances for classes.

**Classes involved:**

* GetBooksFactory- Factory Method
* GetUserRelatedBooks
* GetUserBorrowedBooks
* GetUserOwnedBooks
* GetUserRequestedBooks

**Reason:**

The classes GetUserBorrowedBooks, GetUserOwnedBooks, GetUserRequestedBooks have the implementation required to get the borrowed, owned and requested books of a particular user respectively. Initially, these classes were each a method in the ‘BookServiceController’. But as and when, features had to be added, a new method was added to the controller class, which violates the OCP (Open-Close Principle). Also since all the classes perform the same function of retrieving books of a particular kind, these methods have been converted to classes and abstracted with the class ‘GetUserRelatedBooks’. At this point of time, the client had to be aware of all these classes and create an instance when it required the books borrowed, owned or requested. In order to provide transparency to the client, a class making use of the Factory Method Pattern has been created which takes in the books the client needs as a parameter, and returns an instance of the respective class.

**Class Diagram:**

The Figure 7.2 shows the class diagram of how the Factory Method pattern has been implemented.

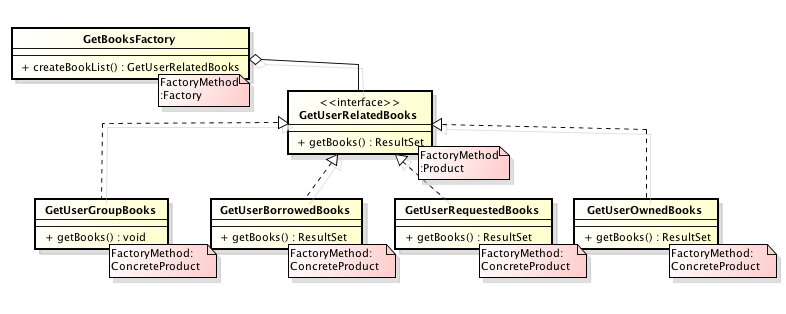


Figure 7.2

**Advantage:**

Making use of this pattern has provided transparency and if in future if we want to add further functionality to the application like retrieving purchased books or sold books, their respective classes can be created without having to violate Open-Closed Principle.

**Source Code:**

**Class: GetBooksFactory**

**public** **class** GetBooksFactory {

//This method takes care of creating an instance based on the bookType. This is a Factory method.

**public** GetUserRelatedBooks createBookList(String bookType){

**if**(bookType.equals("OwnedBooks")){

**return** **new** GetUserOwnedBooks();

}

**else** **if**(bookType.equals("BorrowedBooks")){

**return** **new** GetUserBorrowedBooks();

}

**else** **if**(bookType.equals("RequestedBooks")){

**return** **new** GetUserRequestedBooks();

}

**else** **if**(bookType.equals("GroupBooks")){

**return** **new** GetUserGroupBooks();

}

**else**{

**return** **null**;

}

}

}

**Class: GetUserRelatedBooks**

**public** **interface** GetUserRelatedBooks {

//This method is implemented by the subclasses. It retrieves the required type of books based on the parameter id.

**public** ResultSet getBooks(**int** id);

}

**Class: GetUserGroupBooks**

**public** **class** GetUserGroupBooks **implements** GetUserRelatedBooks{

//This method has an implementation which retrieves all the books belonging to members of the groups in which the user bearing the parameter id exists.

@Override

**public** ResultSet getBooks(**int** id) {

String sql="select \* "

+ "from memberbooks mb,books b "

+ "where mb.book\_id=b.id "

+ "and mb.owner\_id in (select mg.member\_id "

+ "from membergroups m,membergroups mg "

+ "where m.member\_id="+id+" "

+ "and m.group\_id=mg.group\_id)";

**return** DBHelper.*getQueryResult*(sql);

}

}

**Class: GetUserOwnedBooks**

**public** **class** GetUserOwnedBooks **implements** GetUserRelatedBooks {

@Override

**public** ResultSet getBooks(**int** id) {

String sql="Select mb.id memberbookid, b.id bookid, bookname "

+ "from memberbooks mb,books b "

+ "where mb.owner\_id="+id

+ " and mb.book\_id=b.id";

**return** DBHelper.*getQueryResult*(sql);

}

}

**Class: GetUserOwnedBooks**

**public** **class** GetUserOwnedBooks **implements** GetUserRelatedBooks {

//This method gives the implementation for retrieving the books of the member bearing the id ; parameter id.

@Override

**public** ResultSet getBooks(**int** id) {

String sql="Select mb.id memberbookid, b.id bookid, bookname "

+ "from memberbooks mb,books b "

+ "where mb.owner\_id="+id

+ " and mb.book\_id=b.id";

**return** DBHelper.*getQueryResult*(sql);

}

}

**Class: GetUserBorrowedBooks**

**public** **class** GetUserBorrowedBooks **implements** GetUserRelatedBooks{

//This method has an implementation to retrieve the books that have been borrowed by the member bearing the id; parameter id.

@Override

**public** ResultSet getBooks(**int** id) {

String sql="Select \* "

+ "from memberbooks mb,books b "

+ "where mb.borrower\_id="+id+" "

+ "and mb.book\_id=b.id";

**return** DBHelper.*getQueryResult*(sql);

}

}

**Class: GetUserRequestedBooks**

**public** **class** GetUserRequestedBooks **implements** GetUserRelatedBooks {

//This method has the implementation that retrieves the list of books requested by the member bearing the id; parameter id

@Override

**public** ResultSet getBooks(**int** id) {

String sql="Select \* "

+ "from bookrequest br, memberbooks mb, books b "

+ "where br.member\_book\_id=mb.id "

+ "and mb.book\_id=b.id "

+ "and br.member\_id="+id;

**return** DBHelper.*getQueryResult*(sql);

}

}

## 7.3 Command Pattern

The Command Pattern is used when methods have to be executed as a command or a macro, or execute a command at a later point of time.

**Classes involved:**

* BookCRUDOperations - Receiver
* BookOperation - Command
* AddOperation - Concrete Command
* DeleteOperation - Concrete Command
* RequestOperation - Concrete Command
* UpdateOperation - Concrete Command
* OperationRequestor - Invoker
* OperationsFacade - Invoker

**Reason:**

BookCRUDOperations is a class, which contains the functionality required for adding, deleting a book and also updating the status of a book. Going further with extending the functionality of the system, a requirement which wanted the user to take a final look at the list of books he wanted to delete before he actually deletes the book, because it would not be possible to get back the book into the database once deleted. Similarly is the case with placing a request for a book. Implementing this feature would require queuing up the delete and request operations and Command Pattern is apt for such a requirement. Along with this, it was required that, a few actions be performed after the ‘add’, ‘update’, ‘request’ and ‘delete’ happens. Hence we have the AddOperation, UpdateOperation, RequestOperation and DeleteOperation classes executing a sequence of actions (macro).

**Class Diagram:**

The Figure 7.3 shows the class diagram of how the command pattern has been implemented.

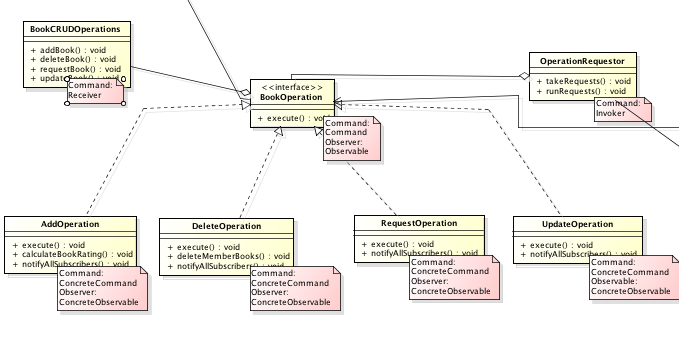


Figure 7.3

**Advantage:**

Using this pattern the operations can be parameterized and be executed whenever the user wants to execute (in case of update and request) at any point of time. Also, a series of actions can be performed using this pattern. Another advantage is that it decouples the receiver(BookCRUDOperations) and the invoker(OperationRequestor).

**Source Code:**

**Class: BookCRUDOperations**

**public** **class** BookCRUDOperations {

// Method to add create new Books for a group. If a book already exists a record is created only in 'memberbooks' table.

**public** **void** addBook(Book book){

**try** {

Statement st = DatabaseConnection.*connectionRequest*().createStatement();

ResultSet bookExistingQuery=st.executeQuery("Select \* "

+ "from books "

+ "where bookname='"+book.getBookName()+"'");

Calendar javaCalendar = **null**;

String currentDate = "";

javaCalendar = Calendar.*getInstance*();

currentDate = javaCalendar.get(Calendar.*YEAR*) + "/" + (javaCalendar.get(Calendar.*MONTH*) + 1) + "/" + javaCalendar.get(Calendar.*DATE*);

**int** bookCount=DBHelper.*getCount*(bookExistingQuery);

**if**(bookCount==1){

ResultSet newBookId=st.executeQuery("Select \* "

+ "from books "

+ "where ISBN='"+book.getBookISBN()+"'");

newBookId.next();

ResultSet checkExistingBook=st.executeQuery("Select \* "

+ "from memberbooks "

+ "where book\_id="+newBookId.getInt("id")+" "

+ "and owner\_id="+CurrentSession.*getMember*().getId());

**if**(DBHelper.*getCount*(checkExistingBook)==0){

st.executeUpdate("insert into "

+ "memberbooks (book\_id, owner\_id,borrower\_id, memberrating, last\_updated\_at ) "

+ "values ('"+newBookId.getInt("id")+"',"+CurrentSession.*getMember*().getId()+",null,"+book.getBookRating()+",'"+currentDate+"')");

}

}

**else**{

**int** categoryId=addCategory(book.getBookCategory());

**if**(categoryId>0){

st.executeUpdate("insert into books (bookname, category\_id,ISBN ) "

+ "values ('"+book.getBookName()+"',"+categoryId+",'"+book.getBookISBN()+"')");

ResultSet newBookId=st.executeQuery("Select \* from books where ISBN='"+book.getBookISBN()+"'");

newBookId.next();

st.executeUpdate("insert into memberbooks (book\_id, owner\_id,borrower\_id, memberrating, last\_updated\_at ) "

+ "values ('"+newBookId.getInt("id")+"',"+CurrentSession.*getMember*().getId()+","+CurrentSession.*getMember*().getId()+","+book.getBookRating()+",'"+currentDate+"')");

}

}

} **catch** (SQLException e) {

e.printStackTrace();

}

}

// Creates a category if it does not exist already

**public** **int** addCategory(String categoryName){

**try** {

Statement st = DatabaseConnection.*connectionRequest*().createStatement();

String sql="Select \* "

+ "from bookcategories "

+ "where categoryname='"+categoryName+"'";

ResultSet categoryExistingQuery=st.executeQuery(sql);

**int** categoryCount=DBHelper.*getCount*(categoryExistingQuery);

**if**(categoryCount==1){

**while** (categoryExistingQuery.last()){

**return** categoryExistingQuery.getInt("id");

}

}

**else**{

st.executeUpdate("insert into bookcategories (categoryname) values ('"+categoryName+"')");

ResultSet newCategory=st.executeQuery(sql);

**while** (newCategory.last()){

**return** newCategory.getInt("id");

}

**return** 0;

}

} **catch** (SQLException e) {

e.printStackTrace();

}

**return** 0;

}

//This method is to delete book based on the id

**public** **void** deleteBook(**int** memberBookid){

Statement st;

**try** {

st = DatabaseConnection.*connectionRequest*().createStatement();

String sql="Delete from memberbooks "

+ "where id="+memberBookid;

st.executeUpdate(sql);

} **catch** (SQLException e) {

e.printStackTrace();

}

}

//This method is to place a request based on the id

**public** **boolean** requestBook(**int** memberBookId){

Statement st;

**try** {

st = DatabaseConnection.*connectionRequest*().createStatement();

String sql="Select \* from memberbooks where id="+memberBookId;

ResultSet bookInfo=st.executeQuery(sql);

bookInfo.first();

**int** id=CurrentSession.*getMember*().getId();

**if**(((bookInfo.getInt("borrower\_id")==id))){

**return** **false**;

}

**else** **if**((bookInfo.getInt("borrower\_id")!=id)){

String validateQuery="Select \* from bookrequest where member\_id="+id+" and member\_book\_id="+memberBookId;

ResultSet request=st.executeQuery(validateQuery);

**if**(DBHelper.*getCount*(request)==0){

String insertQuery="insert into bookrequest (member\_book\_id,member\_id) values("+memberBookId+","+id+")";

st.executeUpdate(insertQuery);

**return** **true**;

}

**return** **false**;

}

} **catch** (SQLException e) {

e.printStackTrace();

}

**return** **false**;

}

//This method delegates the update request based on what is to be requested.

**public** **void** updateBook(String updateAction,**int** id){

**if**(updateAction.equals("availability")){

IUpdateTemplate update=**new** UpdateBookAvailability();

update.executeUpdate(id);

}

**if**(updateAction.equals("borrower")){

IUpdateTemplate update=**new** UpdateBookBorrower();

update.executeUpdate(id);

}

}

}

**Class: BookOperation**

**public** **interface** BookOperation **extends** IObservable {

// the variable bo is the receiver

BookCRUDOperations *bo*=**new** BookCRUDOperations();

**public** **void** execute();

}

**Class: AddOperation**

//This class acts as a Concrete Command of the Command Pattern and Observable of Observer Pattern

**public** **class** AddOperation **implements** BookOperation, IObservable{

Book book;

**private** ArrayList<Member> members=**new** ArrayList<Member>();

**public** AddOperation(Book book){

**this**.book=book;

}

@Override

**public** **void** execute() {

//Macro- these 4 lines of code give the sequence of actions to be performed. Hence command pattern was used.

*bo*.addBook(book);

updateRating();

addSubscribers();

notifyAllSubscribers();

}

// This method computes list of members to be notified of the addition of a book and add them to the observers list (variable members here) as a part of the Observer Pattern.

**public** **void** addSubscribers(){

**try** {

Statement st = DatabaseConnection.*connectionRequest*().createStatement();

String sql="Select \* from groups g, membergroups mg, members m "

+ "where g.id=mg.group\_id "

+ "and mg.member\_id=m.id "

+ "and mg.member\_id="+CurrentSession.*getMember*().getId();

ResultSet member=st.executeQuery(sql);

**if**(DBHelper.*getCount*(member)>0){

member.beforeFirst();

**while**(member.next()){

Member m=**new** Member(member.getString("firstname"),member.getString("lastname"),member.getString("username"),member.getString("password"),member.getString("address"),member.getString("email"));

**int** id=member.getInt("member\_id");

m.setId(member.getInt("member\_id"));

System.*out*.println(id);

members.add(m);

}

}

} **catch** (SQLException e) {

e.printStackTrace();

}

}

//This method is one of the actions to be performed after the addition of a book happens

**public** **void** updateRating(){

String sql= "Select memberrating "

+ "from memberbooks mb,books b "

+ "where mb.book\_id=b.id "

+ "and b.ISBN='"+book.getBookISBN()+"'";

**int** rating=0;

**int** count=0;

Statement st;

**try** {

st = DatabaseConnection.*connectionRequest*().createStatement();

ResultSet ratingList=st.executeQuery(sql);

**while**(ratingList.next()){

count=count+1;

rating =rating+ratingList.getInt("memberrating");

}**if**(count>0){

String updateRatingQuery="Update books "

+ "set rating="+(rating/count)+" "

+ "where isbn='"+book.getBookISBN()+"'";

st.executeUpdate(updateRatingQuery);

}

**else**{

String updateRatingQuery="Update books "

+ "set rating="+(rating)+" "

+ "where isbn='"+book.getBookISBN()+"'";

st.executeUpdate(updateRatingQuery);

}

} **catch** (SQLException e) {

e.printStackTrace();

}

}

//This method calls notify method on all the observers as a part of the Observer Pattern

**public** **void** notifyAllSubscribers(){

**for**(**int** i=0;i<members.size();i++){

members.get(i).notify(book.getBookName(),"add");

}

}

}

**Class: DeleteOperation**

//This class acts as a Concrete Command of the Command Pattern and Observable of Observer Pattern

**public** **class** DeleteOperation **implements** BookOperation, IObservable {

**private** **int** id;

**private** ArrayList<Member> members=**new** ArrayList<Member>();

**public** DeleteOperation(**int** id){

**this**.id=id;

}

@Override

**public** **void** execute() {

//Macro- these 4 lines of code give the sequence of actions to be performed. Hence command pattern was used.

addSubscribers();

notifyAllSubscribers();

deleteBookRequests();

*bo*.deleteBook(id);

}

//This method is one of the actions to be performed after the addition of a book happens

**public** **void** deleteBookRequests(){

String sql ="Delete from bookrequest where member\_book\_id="+id;

Statement st;

**try** {

st = DatabaseConnection.*connectionRequest*().createStatement();

st.executeUpdate(sql);

} **catch** (SQLException e) {

e.printStackTrace();

}

}

// This method computes list of members to be notified of the addition of a book and add them to the observers list (variable members here) as a part of the Observer Pattern.

**public** **void** addSubscribers(){

**try** {

Statement st = DatabaseConnection.*connectionRequest*().createStatement();

String sql="Select \* from groups g, membergroups mg, members m "

+ "where g.id=mg.group\_id "

+ "and mg.member\_id=m.id "

+ "and mg.member\_id="+CurrentSession.*getMember*().getId();

ResultSet member=st.executeQuery(sql);

**if**(DBHelper.*getCount*(member)>0){

member.beforeFirst();

**while**(member.next()){

Member m=**new** Member(member.getString("firstname"),member.getString("lastname"),member.getString("username"),member.getString("password"),member.getString("address"),member.getString("email"));

m.setId(member.getInt("member\_id"));

members.add(m);

}

}

} **catch** (SQLException e) {

e.printStackTrace();

}

}

//This method calls notify method on all the observers as a part of the Observer Pattern

@Override

**public** **void** notifyAllSubscribers() {

**for**(**int** i=0;i<members.size();i++){

ResultSet book=BookServiceController.*getInstance*().getBookbyId(id);

**try** {

book.first();

members.get(i).notify(book.getString("bookname"),"delete");

} **catch** (SQLException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

}

**Class: UpdateOperation**

//This class acts as a Concrete Command of the Command Pattern and Observable of Observer Pattern

**public** **class** UpdateOperation **implements** BookOperation, IObservable {

**private** ArrayList<Member> members=**new** ArrayList<Member>();

**private** String updateAction;

**private** **int** memberBookId;

**public** UpdateOperation(String updateAction,**int** memberBookId){

**this**.updateAction=updateAction;

**this**.memberBookId=memberBookId;

}

@Override

**public** **void** execute() {

//Macro- these 4 lines of code give the sequence of actions to be performed. Hence command pattern was used.

*bo*.updateBook(updateAction, memberBookId);

addSubscribers();

notifyAllSubscribers();

}

//Macro- these 4 lines of code give the sequence of actions to be performed. Hence command pattern was used.

**public** **void** addSubscribers(){

String sql="Select \* "

+ "from members m, bookrequest br "

+ "where br.member\_id=m.id "

+ "and br.member\_book\_id="+memberBookId;

ResultSet bookRequestor=DBHelper.*getQueryResult*(sql);

**if**(bookRequestor!=**null**){

**try** {

**while**(bookRequestor.next()){

Member m=**new** Member(bookRequestor.getString("firstname"), bookRequestor.getString("lastname"), bookRequestor.getString("username"), bookRequestor.getString("password"), bookRequestor.getString("address"), bookRequestor.getString("email"));

m.setId(bookRequestor.getInt("member\_id"));

members.add(m);

}

} **catch** (SQLException e) {

e.printStackTrace();

}

}

}

//This method calls notify method on all the observers as a part of the Observer Pattern

**public** **void** notifyAllSubscribers(){

**for**(**int** i=0;i<members.size();i++){

ResultSet book=BookServiceController.*getInstance*().getBookbyId(memberBookId);

**try** {

book.first();

members.get(i).notify(book.getString("bookname"),"update");

} **catch** (SQLException e) {

e.printStackTrace();

}

}

}

}

**Class: RequestOperation**

//This class acts as a Concrete Command of the Command Pattern and Observable of Observer Pattern

**public** **class** RequestOperation **implements** BookOperation {

**private** **int** id;

**private** ArrayList<Member> members=**new** ArrayList<Member>();

**public** RequestOperation(**int** id){

**this**.id=id;

}

@Override

**public** **void** execute() {

//Macro- these 3 lines of code give the sequence of actions to be performed. Hence command pattern was used.

*bo*.requestBook(id);

addSubscribers();

notifyAllSubscribers();

}

// This method computes list of members to be notified of the addition of a book and add them to the observers list (variable members here) as a part of the Observer Pattern.

**public** **void** addSubscribers(){

String sql ="Select \* "

+ "from memberbooks mb, members m "

+ "where m.id=mb.borrower\_id"

+ " and mb.id="+id;

**try** {

Statement st=DatabaseConnection.*connectionRequest*().createStatement();

ResultSet member=st.executeQuery(sql);

member.first();

Member m=**new** Member(member.getString("firstname"),member.getString("lastname"),member.getString("username"),member.getString("password"),member.getString("address"),member.getString("email"));

**int** id=member.getInt("member\_id");

m.setId(member.getInt("member\_id"));

members.add(m);

} **catch** (SQLException e) {

e.printStackTrace();

}

}

//This method calls notify method on all the observers as a part of the Observer Pattern

@Override

**public** **void** notifyAllSubscribers() {

**for**(**int** i=0;i<members.size();i++){

ResultSet book=BookServiceController.*getInstance*().getBookbyId(id);

**try** {

book.first();

members.get(i).notify(book.getString("bookname"),"delete");

} **catch** (SQLException e) {

e.printStackTrace();

}

}

}

}

**Class: OperationRequestor**

//This class acts as an invoker

**public** **class** OperationRequestor {

//OperationsList maintains the list of operations that are to be queued up.

**private** ArrayList<BookOperation> operationsList=**new** ArrayList<BookOperation>();

//These static variables are maintained just to store and display the details of the books that are to be deleted and requested to the user.Does not play much role in Command Pattern.

**public** **static** ArrayList<Book> *addBooks*=**new** ArrayList<Book>();

**public** **static** ArrayList<Integer> *deleteBooks*=**new** ArrayList<Integer>();

**public** **static** ArrayList<Integer> *requestBooks*=**new** ArrayList<Integer>();

//Method to queue delete and request operations through operationsList

**public** **void** addOperation(BookOperation operation){

operationsList.add(operation);

}

//Method to remove all the operations if the user wants to cancel all the deletes or requests.

**public** **void** removeOperations(){

**for**(**int** i=0;i<operationsList.size();i++){

operationsList.remove(i);

}

}

//Calls an execute over the operations which are queued up

**public** **void** runRequests(){

**for**(**int** i=0;i<operationsList.size();i++){

operationsList.get(i).execute();

}

}

}

## 7.4 Façade Pattern

The Façade Pattern is a structural pattern and is used when the complexities of a subsystem has to be hidden.

**Classes involved:**

* OperationsFacade -Facade
* OperationRequestor

**Reason:**

As explained in 7.3, each of the operations had its own of being executed. The RequestOperation and DeleteOperation had to be queued up and then all the operations had to be executed. The AddOperation, RequestOperation, UpdateOperation dint require to be queued and executed immediately. For the later operations Command Pattern was being used since they were to be executed as macros. The client had to be aware of this and in order to avoid these complexities; the class, OperationsFacade implementing the Façade Pattern, was created which takes care of how to execute the operations. It also takes care of how to add the delete and request operations to the queue and then call for execution. Since it calls for execution immediately, in case of Add and Update Operation, it acts as a Invoker as well in the Command Pattern.

**Class Diagram:**

The Figure 7.4 shows how the Façade Pattern has been implemented.

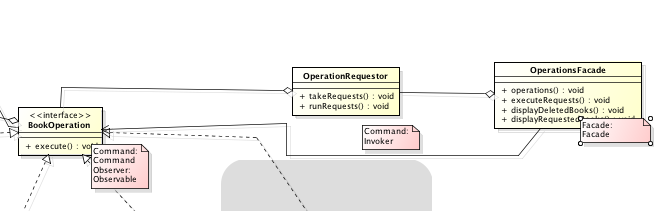


Figure 7.4

**Advantage:**

The client does not have to worry about how to execute the operations. The OperationsFacade takes care of it and makes it easy for the client.

**Source Code:**

**Class: OperationsFacade**

**public** **class** OperationsFacade {

OperationRequestor or=**new** OperationRequestor();

// This method takes care of what operation is to be called based on the parameters. It hides the complexity of how to execute a command since each of the operations have a different way of being executed.

**public** **void** operations(String operation,Integer parameter,Book book){

**if**(operation.equals("Add")){

//Add Operation needs to be executed immediately. It need not be queued up. Hence execute is called from here itself. This also acts as an invoker of the command pattern.

//Add Operation requires a sequence of steps to be executed. Hence Command is used.

**new** AddOperation(book).execute();

}

**else** **if**(operation.equals("Delete")){

//Delete Operation needs to be just queued up here

OperationRequestor.*deleteBooks*.add(parameter);

or.addOperation(**new** DeleteOperation(parameter));

}

**else** **if**(operation.equals("Request")){

//Request Operation needs to be just queued up here

or.addOperation(**new** RequestOperation(parameter));

OperationRequestor.*requestBooks*.add(parameter);

}

**else** **if**((operation.equals("availability")||(operation.equals("borrower")))){

//Update Operation needs to be executed immediately. It need not be queued up. Hence execute is called from here itself. This also acts as an invoker of the command pattern.

//Update Operation requires a sequence of steps to be executed. Hence Command is used.

**new** UpdateOperation(operation,parameter).execute();

}

}

//This method is to execute the queued up operations.

**public** **void** executeRequests(){

or.runRequests();

}

//This method cancels all the operations that are queued up.

**public** **void** removeOperations(){

**for**(**int** i=0;i< OperationRequestor.*deleteBooks*.size();i++){

OperationRequestor.*deleteBooks*.remove(i);

}

**for**(**int** i=0;i< OperationRequestor.*requestBooks*.size();i++){

OperationRequestor.*requestBooks*.remove(i);

}

or.removeOperations();

}

//This method is used for displaying the list of books to be deleted to current user. Does not play much role in Command Pattern

**public** ArrayList<String> displayDeletedBooks(){

**return** getListofBooks(OperationRequestor.*deleteBooks*);

}

//This method is used for displaying the list of books to be deleted to current user. Does not play much role in Command Pattern

**public** ArrayList<String> displayRequestedBooks(){

**return** getListofBooks(OperationRequestor.*requestBooks*);

}

//This method is used for getting the list of books through the Arraylists.

**private** ArrayList<String> getListofBooks(ArrayList<Integer> book){

ArrayList<String> returnBooks=**new** ArrayList<String>();

**for**(**int** i=0;i<book.size();i++){

**int** id=book.get(i);

ResultSet books=BookServiceController.*getInstance*().getBookbyId(id);

//String category=bo.getCategory(id);

**try** {

books.first();

String book1=books.getString("bookname");

returnBooks.add(book1);

} **catch** (SQLException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

**return** returnBooks;

}

}

**Class: OperationsRequestor**

As in 7.4

## 7.5 Observer Pattern

The Observer pattern is a behavioral pattern, which is used when anything has to be notified of a change that occurs in another class/object.

**Classes involved:**

* IObservable -Observable
* IObserver –Observer
* Member- Observer
* BookOperation - Observer
* AddOperation –Concrete Observer
* DeleteOperation - Concrete Observer
* RequestOperation - Concrete Observer
* UpdateOperation - Concrete Observer

**Reason:**

As the functionality of the system was being extended, a requirement was being added, which wanted the users to be notified based on the operation being executed. For example, when a book has been added by a user/member, and this user/member belongs to a group, all the members who belong to this group had to be notified that the book had been added. Similar is the case with deleting a book. When a book is requested, the owner of the book has to be notified. When a book status is updated, the owner of the book and the first member of the list of requestors of the book have to be notified. Observing this requirement at a higher level, it is understood that the AddOperation, DeleteOperation, RequestOperation, UpdateOperation classes have to observed and whenever they are executed, the Members have to be notified. Hence Observer Pattern has been implemented for this requirement.

**Class Diagram:**

The Figure 7.5 shows the implementation of Observer Pattern.

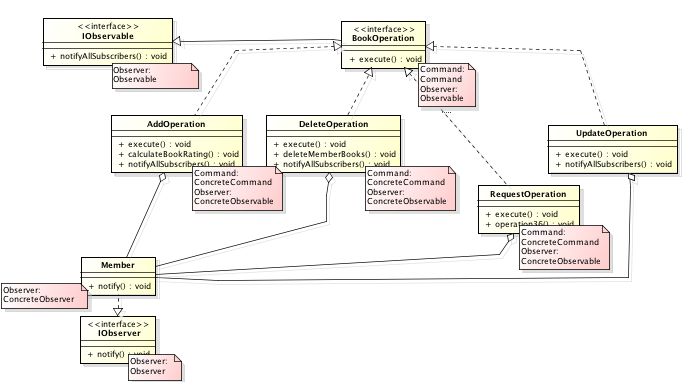


Figure 7.5

**Advantage:**

Using the Observer Pattern, made it easy to notify the members of the operation that was performed.

**Source Code:**

**Class: IObserver**

//This class acts as a Observable of the Observer Pattern

**public** **interface** IObserver {

// The actual observer provide an implementation for this method

**public** **void** notify(String bookname,String action);

}

**Class: Member**

//This class acts as a Concrete Observer of the observer Pattern

**public** **class** Member **implements** IObserver {

**private** String firstName;

**private** String lastName;

**private** String username;

**private** String password;

**private** String address;

**private** String Email;

**int** id;

**public** Member(String firstName,String lastName,String username,String password,String address,String Email){

**this**.firstName=firstName;

**this**.lastName=lastName;

**this**.username=username;

**this**.password=password;

**this**.address=address;

**this**.Email=Email;

}

**public** **void** setId(**int** id){

**this**.id=id;

}

**public** **int** getId(){

**return** **this**.id;

}

**public** String getFirstName(){

**return** **this**.firstName;

}

**public** String getLastName(){

**return** **this**.lastName;

}

**public** String getUsername(){

**return** **this**.username;

}

**public** String getPassword(){

**return** **this**.password;

}

**public** String getAddress(){

**return** **this**.address;

}

**public** String getEmail(){

**return** **this**.Email;

}

//This method provides functionality of what shud happen when notified.

**public** **void** notify(String bookname,String action){

**try** {

Statement st = DatabaseConnection.*connectionRequest*().createStatement();

String notification="";

**if**(action.equals("add")){

notification=bookname+"has been added by "+CurrentSession.*getMember*().firstName;

}

**else** **if**(action.equals("delete")){

notification=bookname+"has been deleted by "+CurrentSession.*getMember*().firstName;

}

**else** **if**(action.equals("update")){

notification=bookname+"has been updated by "+CurrentSession.*getMember*().firstName;

}

String sql="Insert into "

+ "notifications (member\_id,notification) "

+ "values ("+**this**.id+", '"+notification+"')";

st.executeUpdate(sql);

} **catch** (SQLException e) {

e.printStackTrace();

}

}

}

**Class: IObservable**

//This class acts as a Observable of the Observer Pattern

**public** **interface** IObservable {

// The subjects that are being observed provide the implementation of the method.

**public** **void** notifyAllSubscribers();

}

**Class: IObservable**

//This class acts as a Observable of the Observer Pattern

**public** **interface** IObservable {

// The subjects that are being observed provide the implementation of the method.

**public** **void** notifyAllSubscribers();

}

**Class: BookOperation**

//This interface and the classes implementing this class are observalbles of the observer pattern

**public** **interface** BookOperation **extends** IObservable {

// the variable bo is the receiver

BookCRUDOperations *bo*=**new** BookCRUDOperations();

**public** **void** execute();

}

**Class: AddOperation**

//This class acts as a Concrete Command of the Command Pattern and Observable of Observer Pattern

**public** **class** AddOperation **implements** BookOperation{

Book book;

**private** ArrayList<Member> members=**new** ArrayList<Member>();

**public** AddOperation(Book book){

**this**.book=book;

}

@Override

**public** **void** execute() {

//Macro- these 4 lines of code give the sequence of actions to be performed. Hence command pattern was used.

*bo*.addBook(book);

updateRating();

addSubscribers();

notifyAllSubscribers();

}

// This method computes list of members to be notified of the addition of a book and add them to the observers list (variable members here) as a part of the Observer Pattern.

**public** **void** addSubscribers(){

**try** {

Statement st = DatabaseConnection.*connectionRequest*().createStatement();

String sql="Select \* from groups g, membergroups mg, members m "

+ "where g.id=mg.group\_id "

+ "and mg.member\_id=m.id "

+ "and mg.member\_id="+CurrentSession.*getMember*().getId();

ResultSet member=st.executeQuery(sql);

**if**(DBHelper.*getCount*(member)>0){

member.beforeFirst();

**while**(member.next()){

Member m=**new** Member(member.getString("firstname"),member.getString("lastname"),member.getString("username"),member.getString("password"),member.getString("address"),member.getString("email"));

**int** id=member.getInt("member\_id");

m.setId(member.getInt("member\_id"));

System.*out*.println(id);

members.add(m);

}

}

} **catch** (SQLException e) {

e.printStackTrace();

}

}

//This method is one of the actions to be performed after the addition of a book happens

**public** **void** updateRating(){

String sql= "Select memberrating "

+ "from memberbooks mb,books b "

+ "where mb.book\_id=b.id "

+ "and b.ISBN='"+book.getBookISBN()+"'";

**int** rating=0;

**int** count=0;

Statement st;

**try** {

st = DatabaseConnection.*connectionRequest*().createStatement();

ResultSet ratingList=st.executeQuery(sql);

**while**(ratingList.next()){

count=count+1;

rating =rating+ratingList.getInt("memberrating");

}**if**(count>0){

String updateRatingQuery="Update books "

+ "set rating="+(rating/count)+" "

+ "where isbn='"+book.getBookISBN()+"'";

st.executeUpdate(updateRatingQuery);

}

**else**{

String updateRatingQuery="Update books "

+ "set rating="+(rating)+" "

+ "where isbn='"+book.getBookISBN()+"'";

st.executeUpdate(updateRatingQuery);

}

} **catch** (SQLException e) {

e.printStackTrace();

}

}

//This method calls notify method on all the observers as a part of the Observer Pattern

**public** **void** notifyAllSubscribers(){

**for**(**int** i=0;i<members.size();i++){

members.get(i).notify(book.getBookName(),"add");

}

}

}

**Class: DeleteOperation**

//This class acts as a Concrete Command of the Command Pattern and Observable of Observer Pattern

**public** **class** DeleteOperation **implements** BookOperation{

**private** **int** id;

**private** ArrayList<Member> members=**new** ArrayList<Member>();

**public** DeleteOperation(**int** id){

**this**.id=id;

}

@Override

**public** **void** execute() {

//Macro- these 4 lines of code give the sequence of actions to be performed. Hence command pattern was used.

addSubscribers();

notifyAllSubscribers();

deleteBookRequests();

*bo*.deleteBook(id);

}

//This method is one of the actions to be performed after the addition of a book happens

**public** **void** deleteBookRequests(){

String sql ="Delete from bookrequest where member\_book\_id="+id;

Statement st;

**try** {

st = DatabaseConnection.*connectionRequest*().createStatement();

st.executeUpdate(sql);

} **catch** (SQLException e) {

e.printStackTrace();

}

}

// This method computes list of members to be notified of the addition of a book and add them to the observers list (variable members here) as a part of the Observer Pattern.

**public** **void** addSubscribers(){

**try** {

Statement st = DatabaseConnection.*connectionRequest*().createStatement();

String sql="Select \* from groups g, membergroups mg, members m "

+ "where g.id=mg.group\_id "

+ "and mg.member\_id=m.id "

+ "and mg.member\_id="+CurrentSession.*getMember*().getId();

ResultSet member=st.executeQuery(sql);

**if**(DBHelper.*getCount*(member)>0){

member.beforeFirst();

**while**(member.next()){

Member m=**new** Member(member.getString("firstname"),member.getString("lastname"),member.getString("username"),member.getString("password"),member.getString("address"),member.getString("email"));

m.setId(member.getInt("member\_id"));

members.add(m);

}

}

} **catch** (SQLException e) {

e.printStackTrace();

}

}

//This method calls notify method on all the observers as a part of the Observer Pattern

@Override

**public** **void** notifyAllSubscribers() {

**for**(**int** i=0;i<members.size();i++){

ResultSet book=BookServiceController.*getInstance*().getBookbyId(id);

**try** {

book.first();

members.get(i).notify(book.getString("bookname"),"delete");

} **catch** (SQLException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

}

**Class: RequestOperation**

//This class acts as a Concrete Command of the Command Pattern and Observable of Observer Pattern

**public** **class** RequestOperation **implements** BookOperation {

**private** **int** id;

**private** ArrayList<Member> members=**new** ArrayList<Member>();

**public** RequestOperation(**int** id){

**this**.id=id;

}

@Override

**public** **void** execute() {

//Macro- these 3 lines of code give the sequence of actions to be performed. Hence command pattern was used.

*bo*.requestBook(id);

addSubscribers();

notifyAllSubscribers();

}

// This method computes list of members to be notified of the addition of a book and add them to the observers list (variable members here) as a part of the Observer Pattern.

**public** **void** addSubscribers(){

String sql ="Select \* "

+ "from memberbooks mb, members m "

+ "where m.id=mb.borrower\_id"

+ " and mb.id="+id;

**try** {

Statement st=DatabaseConnection.*connectionRequest*().createStatement();

ResultSet member=st.executeQuery(sql);

member.first();

Member m=**new** Member(member.getString("firstname"),member.getString("lastname"),member.getString("username"),member.getString("password"),member.getString("address"),member.getString("email"));

**int** id=member.getInt("member\_id");

m.setId(member.getInt("member\_id"));

members.add(m);

} **catch** (SQLException e) {

e.printStackTrace();

}

}

//This method calls notify method on all the observers as a part of the Observer Pattern

@Override

**public** **void** notifyAllSubscribers() {

**for**(**int** i=0;i<members.size();i++){

ResultSet book=BookServiceController.*getInstance*().getBookbyId(id);

**try** {

book.first();

members.get(i).notify(book.getString("bookname"),"delete");

} **catch** (SQLException e) {

e.printStackTrace();

}

}

}

}

**Class: UpdateOperation**

//This class acts as a Concrete Command of the Command Pattern and Observable of Observer Pattern

**public** **class** UpdateOperation **implements** BookOperation{

**private** ArrayList<Member> members=**new** ArrayList<Member>();

**private** String updateAction;

**private** **int** memberBookId;

**public** UpdateOperation(String updateAction,**int** memberBookId){

**this**.updateAction=updateAction;

**this**.memberBookId=memberBookId;

}

@Override

**public** **void** execute() {

//Macro- these 4 lines of code give the sequence of actions to be performed. Hence command pattern was used.

*bo*.updateBook(updateAction, memberBookId);

addSubscribers();

notifyAllSubscribers();

}

//Macro- these 4 lines of code give the sequence of actions to be performed. Hence command pattern was used.

**public** **void** addSubscribers(){

String sql="Select \* "

+ "from members m, bookrequest br "

+ "where br.member\_id=m.id "

+ "and br.member\_book\_id="+memberBookId;

ResultSet bookRequestor=DBHelper.*getQueryResult*(sql);

**if**(bookRequestor!=**null**){

**try** {

**while**(bookRequestor.next()){

Member m=**new** Member(bookRequestor.getString("firstname"), bookRequestor.getString("lastname"), bookRequestor.getString("username"), bookRequestor.getString("password"), bookRequestor.getString("address"), bookRequestor.getString("email"));

m.setId(bookRequestor.getInt("member\_id"));

members.add(m);

}

} **catch** (SQLException e) {

e.printStackTrace();

}

}

}

//This method calls notify method on all the observers as a part of the Observer Pattern

**public** **void** notifyAllSubscribers(){

**for**(**int** i=0;i<members.size();i++){

ResultSet book=BookServiceController.*getInstance*().getBookbyId(memberBookId);

**try** {

book.first();

members.get(i).notify(book.getString("bookname"),"update");

} **catch** (SQLException e) {

e.printStackTrace();

}

}

}

}

## 7.6 Template Pattern

Template Pattern is a behavioral pattern, which is used when a set of actions have to be performed (these actions are implemented by subclasses).

**Classes involved:**

* IUpdateTemplate –Template Interface
* UpdateBookAvailability
* UpdateBookBorrower

**Reason:**

The UpdateBookAvailability and the UpdateBookBorrower classes were implemented as separate methods in BookServiceController class initially. Later, it was observed that a validation check has to be done to make sure that only the owner or the borrower can update the status of the book depending on the details, which are being updated. For example, once the user finishes reading the book, he can set the availability of the book, which tells that the book is available for other members. Also if the owner of the book observes that a particular user had the book with him for a very long time, he can set the availability to true. Similarly, when the book currently held by a user delivers the book to the next requestor of the book, he updates the borrower of the book. Since whenever an update happens we need to perform two steps. One is checking if the user is authorized to perform this action. If he is the second step is updating the action. Thus it is clear that a template of actions have to be performed. Hence the Template pattern has been implemented.

**Class Diagram:**

The Figure 7.6 shows how the Template Pattern has been implemented.

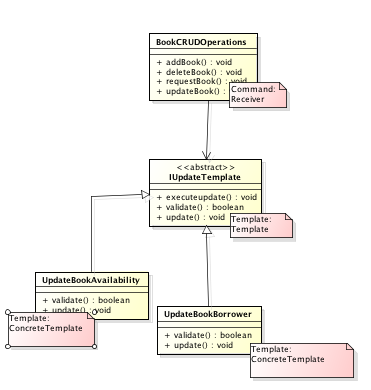


Figure 7.6

**Advantage:**

In the future, if any update functionalities are included, they have to extend the IUpdateTemplate class. This makes sure a validation definitely happens before actually update the information of the book.

**Source Code:**

**Class: IUpdateTemplate**

//This class provides a template method as a part of Template Pattern

**public** **abstract** **class** IUpdateTemplate{

**private** **int** memberBookId;

//This is a template method which gives a skeleton of what is to be done in a sequence.When an update happens a validation needs to be done to make sure the user is authorized to make changes and then update.

**public** **void** executeUpdate(**int** memberBookId){

**this**.memberBookId=memberBookId;

**if**(validate()){

update();

}

}

**public** **int** getId(){

**return** memberBookId;

}

**abstract** **boolean** validate();

**abstract** **boolean** update();

}

**Class: UpdateBookBorrower**

//This class provides an implementation for the abstract methods of IUpdateTemplate

**public** **class** UpdateBookBorrower **extends** IUpdateTemplate{

//This method validates if the user updating the book is an owner or a borrower of the book

**public** **boolean** validate(){

Statement st;

**try** {

st = DatabaseConnection.*connectionRequest*().createStatement();

String sql="Select \* from memberbooks where id="+getId();

ResultSet bookInfo=st.executeQuery(sql);

bookInfo.first();

**int** id=CurrentSession.*getMember*().getId();

**if**(((bookInfo.getInt("owner\_id")==id)||(bookInfo.getInt("borrower\_id")==id))){

bookInfo.first();

**return** **true**;

}

} **catch** (SQLException e) {

e.printStackTrace();

}

**return** **false**;

}

//Updates the borrower of the book

@Override

**public** **boolean** update() {

**try** {

Statement st = DatabaseConnection.*connectionRequest*().createStatement();

String sql="Select \* "

+ "from bookrequest "

+ "where member\_book\_id="+getId();

ResultSet bookRequest=st.executeQuery(sql);

**if**(DBHelper.*getCount*(bookRequest)>0){

bookRequest.first();

String updateQuery="Update memberbooks set borrower\_id="+bookRequest.getInt("member\_id") +" where id="+getId();

st.executeUpdate("Delete from bookrequest where member\_id="+bookRequest.getInt("member\_id"));

st.executeUpdate(updateQuery);

**return** **true**;

}

} **catch** (SQLException e) {

e.printStackTrace();

}

**return** **false**;

}

}

**Class: UpdateBookAvailability**

//This class provides an implementation for the abstract methods of IUpdateTemplate

**public** **class** UpdateBookAvailability **extends** IUpdateTemplate{

//This method validates if the user updating the book is a borrower of the book

**public** **boolean** validate(){

Statement st;

**try** {

st = DatabaseConnection.*connectionRequest*().createStatement();

String sql="Select \* from memberbooks where id="+getId();

ResultSet bookInfo=st.executeQuery(sql);

bookInfo.first();

**int** id=CurrentSession.*getMember*().getId();

**if**(((bookInfo.getInt("borrower\_id")==id))){

bookInfo.first();

**return** **true**;

}

} **catch** (SQLException e) {

e.printStackTrace();

}

**return** **false**;

}

@Override

**public** **boolean** update() {

**try** {

Statement st = DatabaseConnection.*connectionRequest*().createStatement();

String sql="Select \* from memberbooks where id="+getId();

ResultSet bookInfo=st.executeQuery(sql);

bookInfo.first();

**if**(bookInfo.getBoolean("availability")==**true**){

String updateQuery="update memberbooks set availability=false where id="+getId();

st.executeUpdate(updateQuery);

}

**if**(bookInfo.getBoolean("availability")==**false**){

String updateQuery="update memberbooks set availability=true where id="+getId();

st.executeUpdate(updateQuery);

}

} **catch** (SQLException e) {

e.printStackTrace();

}

**return** **false**;

}

}

## 7.7 Strategy Pattern

The strategy pattern is a behavioral object design pattern where different algorithms are represented as Concrete Strategy classes and they share the common interface.

**Classes Involved:**

SearchContext - Context

SearchOperation – Strategy Interface

BookSearch – Concrete Strategy

IsbnSearch - Concrete Strategy

NameSearch - Concrete Strategy

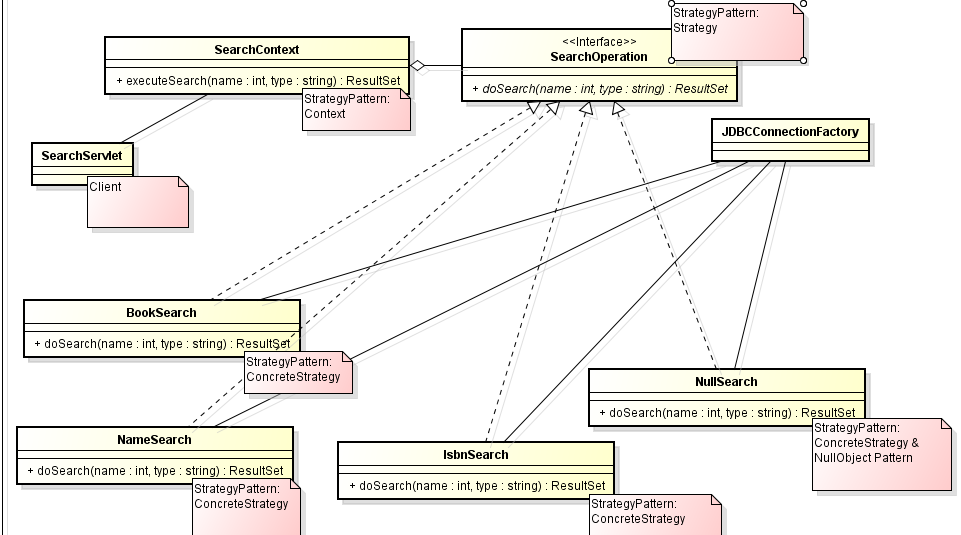
**Reason:**

As our Search implementation required different data to be populated with different queries, and also we know the fact that at any point in future with the growth of data, we had to add more queries to filter the data. In order to be in line with Open-Closed principle we used strategy pattern.

At a point we had a doubt if it’s the correct implementation as the every implementation we found was either from the point of formula or the algorithm. But we justified ourselves saying that it’s the “Behavior” that matters, in our case every concrete strategy class behaves differently i.e. have different queries and gives out different result set.

We were also a bit confused between “State” and “Strategy” as both of their implementations was similar. This gave us an opportunity to differentiate both and pick the apt one.

**Class Diagram:**



**Advantages:**

As said above, if we had to add more queries or methods to retrieve the data it will be easy to do so without breaking the ocp.

**Source Code:**

Context class:

**public** **class** SearchContext {

**private** SearchOperation soperation;

**public** SearchContext(SearchOperation soperation) {

**this**.soperation = soperation;

}

//calls the concrete classes to get the desired results

**public** ResultSet executeSearch(String name, String type) {

**return** soperation.doSearch(name, type);

}

}

Interface:

// Interface for concrete Strategy classes

**public** **interface** SearchOperation {

**public** ResultSet doSearch( String name, String type );

}

Concrete Strategy classes:

public class BookNameSearch implements SearchOperation {

// class having the algorithm to get various data using a book name

@Override

public ResultSet doSearch(String name) {

String sql = "SELECT \* "

+ "FROM books,bookcategories "

+ "where bookname= '"+ name + "' "

+ "AND books.id = bookcategories.id";

return DBHelper.getQueryResult(sql);

}

}

public class IsbnSearch implements SearchOperation {

// class having the algorithm to get data using a isbn number

public ResultSet doSearch(String isbn) {

String sql = "Select \* "

+ "from books "

+ "where ISBN = '"+ isbn +"'";

return DBHelper.getQueryResult(sql); }

}

public class MemberNameSearch implements SearchOperation {

// class having the algorithm to get various data using a member name

@Override

public ResultSet doSearch(String name) {

String sql = "SELECT \* "

+ "FROM members where concat(firstname, concat(' ',lastname))='"+name+"'";

return DBHelper.getQueryResult(sql);

}

}

## 7.8 Null Object Pattern

This Behavioral design pattern is used to provide “Do Nothing” behavior.

**Classes Involved:**

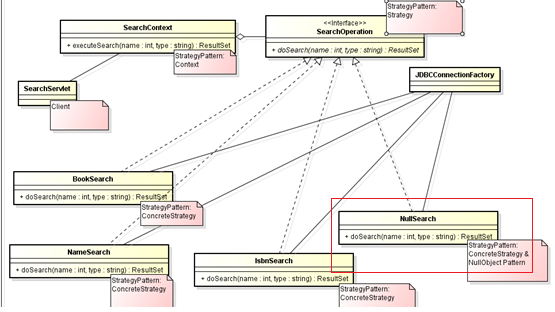
SearchOperation – (Existing Interface)

NullSearch – (Concrete Null Object)

**Reason:**

I personally feel this can be most used pattern as every situations needs to handle “Null”. When we were testing the search functionality, we had an exception for null which we forgot to handle. Then we got this idea of using it instead of having special null check which will provide us with “Transparency”. There is one thing that has to be kept in mind is that this pattern only works in collaboration with other patterns and implements the same interface. As our interface was defined to return the result set, we had to also get the default value from the database to bring in compliance.

**Class Diagram:**



**Advantages:**

- Transparency

- Can be used to avoid the special blocks of code

**Code:**

Interface:

//Interface for concrete Strategy classes

**public** **interface** SearchOperation {

**public** ResultSet doSearch( String name, String type );

}

Null Object Class:

public class NullSearch implements SearchOperation {

// class implementing the default behavior for null object

@Override

public ResultSet doSearch(String name) {

String sql = "SELECT \* "

+ "FROM nullsearch";

return DBHelper.getQueryResult(sql);

}

}

## 7.9 Decorator Pattern

This pattern can be used to modify or extend the behavior of an instance at runtime.

**Classes Involved:**

OrderPriceCalculator – (Interface)

ActualPrice – (The object to be decorated)

BitCoinTotal – (Abstract Decorator)

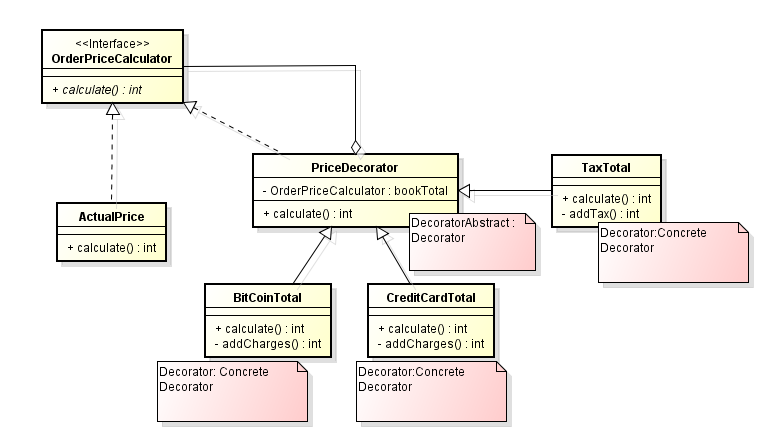
CreditCardTotal – (Concrete Decorator)

TaxTotal-(Concrete Decorator)

**Reason:**

Initially we have used abstraction to calculate the prices depending on the type of delivery (We have 2 types – Delivery by owner which he charges and pick up by customer which is free), but later we felt that we are reusing the same code and were looking for an alternative. We realized that we can wrap up (an addition) the actual price with the delivery charges and taxes by using decorator pattern by reducing the code redundancy.

**Class Diagram:**



**Advantages:**

* Decoupling

**Code:**

**Interface:**

public interface OrderPriceCalculator {

int calculate(int bookTotal) }

**Class that should be decorated :**

public class ActualPrice implements OrderPriceCalculator {

// wrapped component

int bookTotal;

@Override

public int calculate(int bookTotal) {

// TODO Auto-generated method stub

this.bookTotal=bookTotal;

return bookTotal;

}

**Abstract Decorator:**

public abstract class PriceDecorator implements OrderPriceCalculator {

public OrderPriceCalculator calculator;

public PriceDecorator(OrderPriceCalculator calculator) {

this.calculator = calculator;

}

public int calculate(int bookTotal) {

return calculator.calculate(bookTotal);

}

}

**Concrete Decorator:**

public class BitCoinTotal extends PriceDecorator {

int total;

public BitCoinTotal(OrderPriceCalculator calculator) {

super(calculator);

}

public int calculate(int bookTotal) {

total = calculator.calculate(bookTotal);

return addCharges();

}

private int addCharges() {

return total + 2;

}

}

**Concrete Decorator:**

public class CreditCardTotal extends PriceDecorator {

public CreditCardTotal(OrderPriceCalculator bookTotal){

super(bookTotal);

}

int total;

public int calculate(int bookTotal) {

total = calculator.calculate(bookTotal);

return addCharges();

}

private int addCharges() {

return total + 10;

}

}

## 7.10 Single Access Point

Provides one and only access point to the system

**Classes and files involved:**

Main.jsp (Login)

MainPage (Servlet)

**Reason:**

The main reason was to provide a single point of access to the entire user community, which holds the personal details. In order make it secured we have used this pattern.



**Advantages:**

* Makes system secured as it may prevent attacks at the root by having proper checkpoints (Locking the user for invalid attempts)
* Makes life easier for different users
* Allows loading of various user data into session in reference to username.

## 7.11 Front Controller Pattern

Provides a centralized request handling mechanism so that all requests were handled by a single handler.

**Classes and files Involved:**

MainPage

Register.jsp

Main.jsp

**Reason:**

The whole idea behind using this is to redirect user to the register page after a handful of unsuccessful attempts. The implementation basically requires a front controller where the validation is done and then the request is forwarded to the dispatcher, which handles it by redirecting it to appropriate views. In our case as we have used jsp and servlets we had to make mainpage.java to act as both controller and dispatcher and the two jsp pages being the views.

**Source Code :**

**if**(MemberServiceController.*memberServicecontroller*.login(request.getParameter("username"),request.getParameter("password"))){

nextPage="/home.jsp";

}

**else**{

nextPage="/register.jsp";

}

RequestDispatcher dispatcher =

getServletContext().getRequestDispatcher(nextPage);

dispatcher.forward(request, response);

}

# 8. Class Diagram

The Figure 8.1 shows the over all class diagram with all the classes involved in implementing Design Patterns.

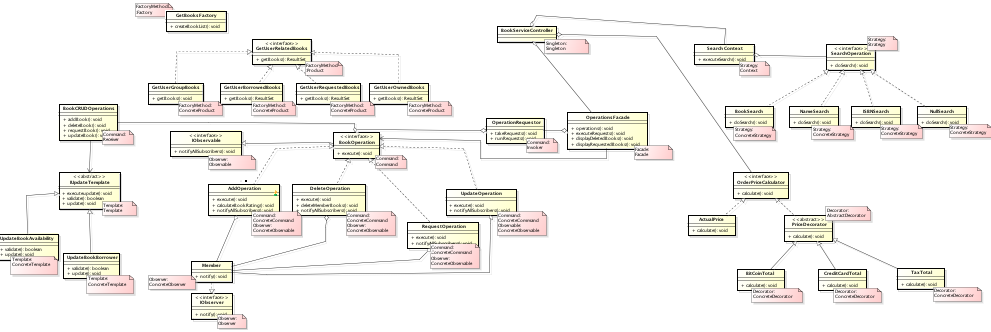


Figure 8.1

# 9. Future Work

* Check Points or filters at various points (Intercepting Filter Pattern)
* Adding various “roles” like admin , guest etc with different privileges
* Friend suggestions for the users within a limited radius of their location(Observer Pattern)
* Comments updates for the users (Observer Pattern)

# 10. Take Away Points

Making use of Design Patterns helps us build a better application and makes it more scalable and can be extended easily. Through this project, we have practically learnt the advantages of using Design Patterns. The main take away from this project are :

* Learned the importance of writing the code with reusability
* Commonality variability analysis was of great help
* Have to cultivate thinking from the whole perspective instead of from point of view of the patterns you love