Refactoring techniques and Design patterns

Authors:

Sai Manikanta S Godavarthi

Viswa Chaitanya Kanakam

Abstract:

Our main aim of this project is to learn different refactoring techniques that can be applied on a project with bad smells and identify the type of refactoring techniques that can be implemented on the code. As a part of refactoring, we also want to implement some of the design patterns to make code more reusable for future updates and changes.

I. Introduction

As a part, of our coursework we have chosen a refactoring project to implement different refactoring techniques and a design patterns to the project with bad smells. Our main aim behind this project is to learn different bad smells that can exist in a project and how to tackle them using different refactoring and design patterns to make the code more reusable and easy to maintain. Initially we have chosen a project with bad smells and tried to design the code in MVC type of pattern and later identify different problems of code strategy and applied few refactoring techniques and design patterns whenever possible accordingly.

We have taken a library application which is written in JAVA and uses JAVA swing and collection API to display and execute the functionality of the programs. Here we are using collection API to store the data in the form of collection objects and are destroyed when we exit the program as it doesn't have a database to store all the data. Here we want to understand each of the refactoring techniques and design patterns accordingly. So as a part of learning we have taken this very sample application later in the future we want to connect a database to this application and make it more standard one. The front view of the applications is a frame objects where most of functionality is determined. The application has mainly two type of users, one is the normal user who uses the library application to choose a book among different options rent it if needed or if there is an existing book in his account, he can return the book when needed. The access fee is determined based on how long the user has book in his account. The other application user is administrator who has all the rights on the system. He can add new books, delete the existing books or update them. Manage the users, manage the categories of book i.e. he can add or delete the categories. He can manage the details of the book and users accordingly. We have initially taken this project determined some test cases on working of the project and later validated them on the refactored code to make sure the application still works as it needed to be. Here initially the project code is randomly generated and has not proper workflow. There are many bad smells initially and reusability of the project is very less. For a new update to be made, the developer must make changes at lot of places and when a part of the project code is removed the entire project breaks down. We want to tackle this problem apply some techniques to make code reusable and follow the standard coding principles of software engineering. For refactoring we have decided to choose to model the code as MVC (Model-View-Controller) type of pattern where we can separate the model, view and controller part of the part of code respectively and build a connection between them to communicate with each other accordingly and make the project to work as it is before. The MVC (Model-View-Controller) is helpful for separating the applications concern. Here the model is the one which updates the controller based on the logic it has inside its code. The controller can act both as model and view, it can update the view when required based on the events generated at the same time control the flow of data in model and apply specific login in model with changes happening. It acts a bridge between model and view. This type of pattern helps for loose coupling, better reusability of code, and enhanced flexibility, it is widely used in industries in recent days. We have followed the iteration process as described below, where we initially corrected few of the old code and applied few refactoring techniques later we introduced our MVC type pattern to structure the files and all functions accordingly.

II. Iterations

- **Iteration 1:** In the initial stage, we have concentrated on managing the old code and make it ready for our new pattern that we are planning to implement. We have made few changes in the code and implemented some of the refactoring techniques in our old code such as extract, move and other possible techniques. We have tried to separate each of the code to make them more independent so that it will easy for us to design as a pattern.
- **Iteration 2:** In the later stage, we have implemented the MVC type pattern to the code and at this stage we have made drastic changes to the code to make it function as it needed to be. Updated all the function calls, references, created new methods and objects if needed. And major refactoring techniques are implemented at this stage where we used composing methods, organizing data, simplifying conditional expressions, method calls, dealing with generalization, and feature between objects refactoring technique and selecting few options among them which are available as per the need. We tried to implement design patterns like observer, singleton pattern, strategy and composite patterns in our project.
- **Iteration 3:** At this stage, our major part of refactoring has been completed, here we tried to identify few bad smells in the project, such as we tried to rename the methods, objects to make them more sensible and meaningful. We have removed different types of comments and updated comments to meaningful context.

III. Existing system

Existing application has following functionality and features, after refactoring we want to make sure that these features and functionality still exists and works as the same like before as they needed to be.

The system administrator has following basic features:

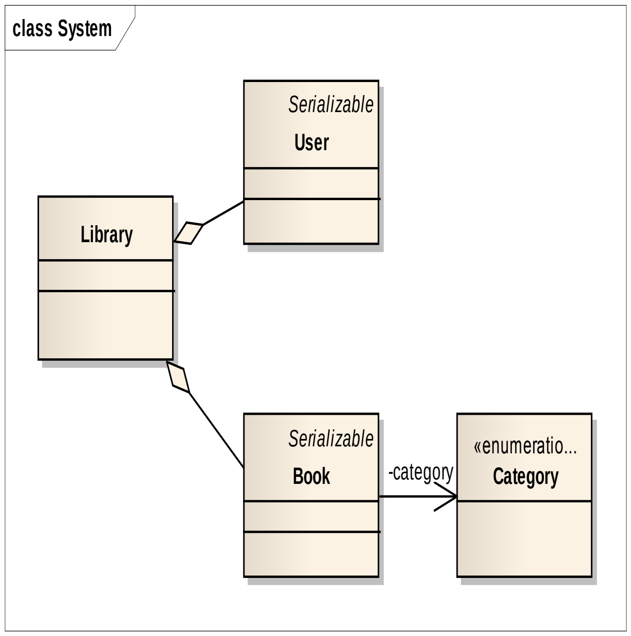
- Add, delete, and update the books
- Add, delete and update the customers
- Add, delete, and update the book categories
- List view of all the books, rented books, returned books recently, over due on the books, and list of all existing customers.

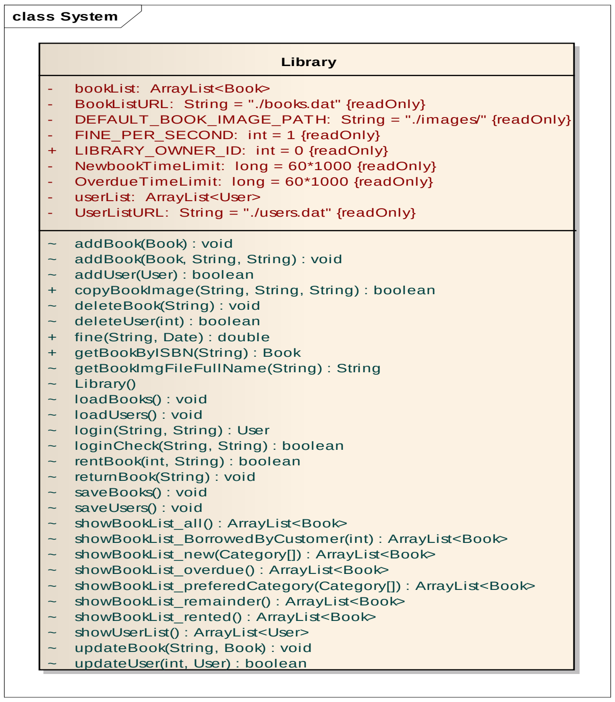
The normal library user has the following features

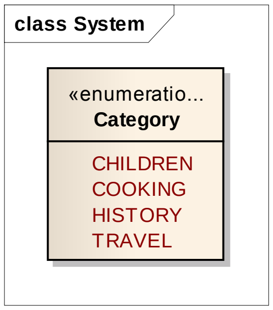
- List of available books with different options based on category or for listing all of them
- Options for user to rent a book, or return it and display the status of the book while choosing to rent it i.e. its availability.
- Track of the day’s user rented the books and all other details accordingly.

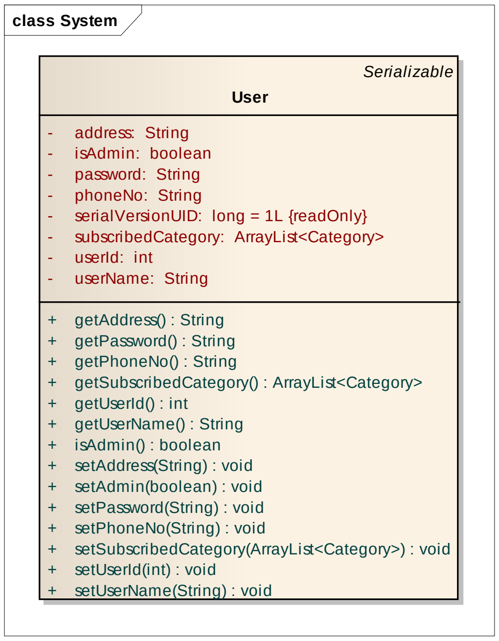
The test cases are chosen based on the above determined points, making sure that system works as it needed to be even after refactoring.

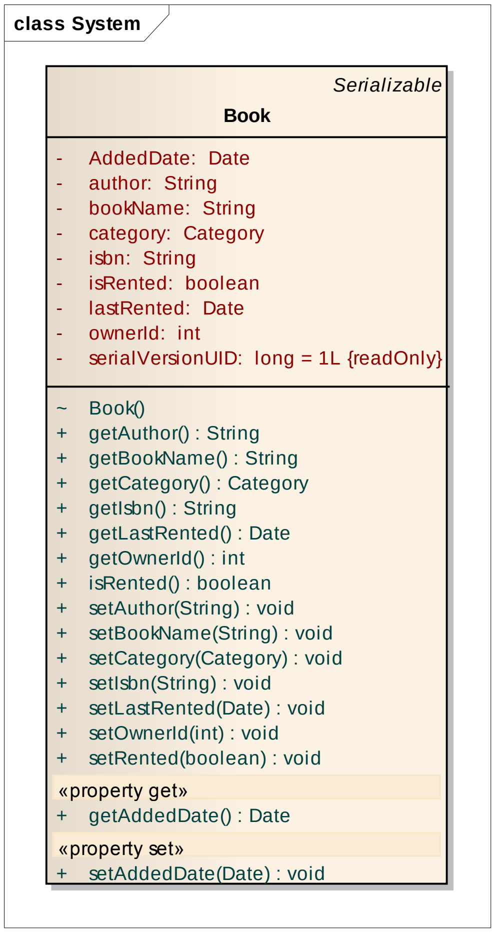
IV. Class Diagrams:




class System
Library
- bookList: ArrayList<Book>
- BookListURL: String = "./books.dat" {readOnly}
- DEFAULT_BOOK_IMAGE_PATH: String = "./images/" {readOnly}
- FINE_PER_SECOND: int = 1 {readOnly}
+ LIBRARY_OWNER_ID: int = 0 {readOnly}
- NewbookTimeLimit: long = 60*1000 {readOnly}
- OverdueTimeLimit: long = 60*1000 {readOnly}
- userList: ArrayList<User>
- UserListURL: String = "./users.dat" {readOnly}
~ addBook(Book) : void
~ addBook(Book, String, String) : void
~ addUser(User) : boolean
+ copyBookImage(String, String, String) : boolean
~ deleteBook(String) : void
~ deleteUser(int) : boolean
+ fine(String, Date) : double
+ getBookByISBN(String) : Book
~ getBookImgFileFullName(String) : String
~ Library()
~ loadBooks() : void
~ loadUsers() : void
~ login(String, String) : User
~ loginCheck(String, String) : boolean
~ rentBook(int, String) : boolean
~ returnBook(String) : void
~ saveBooks() : void
~ saveUsers() : void
~ showBookList_all() : ArrayList<Book>
~ showBookList_BorrowedByCustomer(int) : ArrayList<Book>
~ showBookList_new(Category[]) : ArrayList<Book>
~ showBookList_overdue() : ArrayList<Book>
~ showBookList_preferedCategory(Category[]) : ArrayList<Book>
~ showBookList_remainder() : ArrayList<Book>
~ showBookList_rented() : ArrayList<Book>
~ showUserList() : ArrayList<User>
~ updateBook(String, Book) : void
~ updateUser(int, User) : boolean




class System
Serializable
User
- address: String
- isAdmin: boolean
- password: String
- phoneNo: String
- serialVersionUID: long = 1L {readOnly}
- subscribedCategory: ArrayList<Category>
- userId: int
- userName: String
+ getAddress() : String
+ getPassword() : String
+ getPhoneNo() : String
+ getSubscribedCategory() : ArrayList<Category>
+ getUserId() : int
+ getUserName() : String
+ isAdmin() : boolean
+ setAddress(String) : void
+ setAdmin(boolean) : void
+ setPassword(String) : void
+ setPhoneNo(String) : void
+ setSubscribedCategory(ArrayList<Category>) : void
+ setUserId(int) : void
+ setUserName(String) : void



Files:

users.dat - Store all the users
books.dat - Store all the books

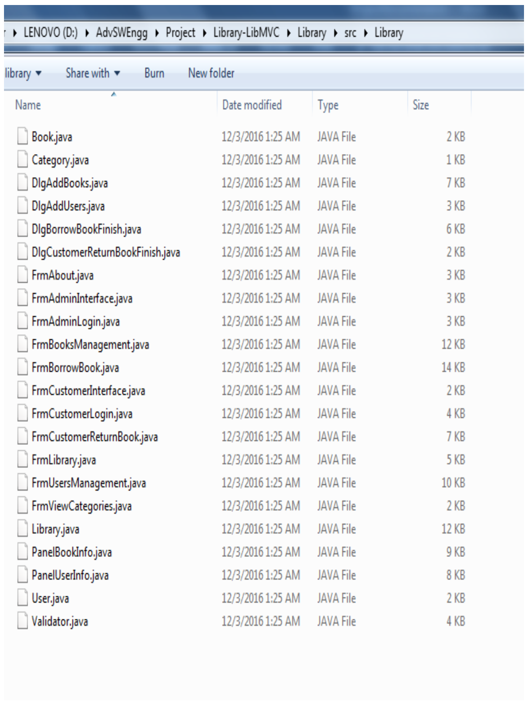
Book's image files:
All the book image files put into the folder "bookimg/"
bookimg/12345.jpg
stands for the image of the book which has ISBN=12345

V. Refactoring techniques and design patterns

A. MVC type of pattern:

Initially as we have mentioned before, the project was generated on random code with many bad smells. We have refactored the project as many techniques and patterns that we can in the available time. initially we have decided to choose to represent the project in MVC (Model-View-Controller) pattern as discussed in the introduction part. We have separated the code into separate parts based on the functionality determined in the code and we have obtained project structure as follows in Fig 1:

Fig1: Initial file structure of the project



› LENOVO (D:) › AdvSWEngg › Project › Library-LibMVC › Library › src › Library

library ▾ Share with ▾ Burn New folder

Name	Date modified	Type	Size
Book.java	12/3/2016 1:25 AM	JAVA File	2 KB
Category.java	12/3/2016 1:25 AM	JAVA File	1 KB
DlgAddBooks.java	12/3/2016 1:25 AM	JAVA File	7 KB
DlgAddUsers.java	12/3/2016 1:25 AM	JAVA File	3 KB
DlgBorrowBookFinish.java	12/3/2016 1:25 AM	JAVA File	6 KB
DlgCustomerReturnBookFinish.java	12/3/2016 1:25 AM	JAVA File	2 KB
FrmAbout.java	12/3/2016 1:25 AM	JAVA File	3 KB
FrmAdminInterface.java	12/3/2016 1:25 AM	JAVA File	3 KB
FrmAdminLogin.java	12/3/2016 1:25 AM	JAVA File	3 KB
FrmBooksManagement.java	12/3/2016 1:25 AM	JAVA File	12 KB
FrmBorrowBook.java	12/3/2016 1:25 AM	JAVA File	14 KB
FrmCustomerInterface.java	12/3/2016 1:25 AM	JAVA File	2 KB
FrmCustomerLogin.java	12/3/2016 1:25 AM	JAVA File	4 KB
FrmCustomerReturnBook.java	12/3/2016 1:25 AM	JAVA File	7 KB
FrmLibrary.java	12/3/2016 1:25 AM	JAVA File	5 KB
FrmUsersManagement.java	12/3/2016 1:25 AM	JAVA File	10 KB
FrmViewCategories.java	12/3/2016 1:25 AM	JAVA File	2 KB
Library.java	12/3/2016 1:25 AM	JAVA File	12 KB
PanelBookInfo.java	12/3/2016 1:25 AM	JAVA File	9 KB
PanelUserInfo.java	12/3/2016 1:25 AM	JAVA File	8 KB
User.java	12/3/2016 1:25 AM	JAVA File	2 KB
Validator.java	12/3/2016 1:25 AM	JAVA File	4 KB

Later we have separated each of the files to form a structure of MVC i.e. making model, view and controller files separately based on the logic determined in the programs. For this purpose, we have taken three packages, each of them are represented as follows:

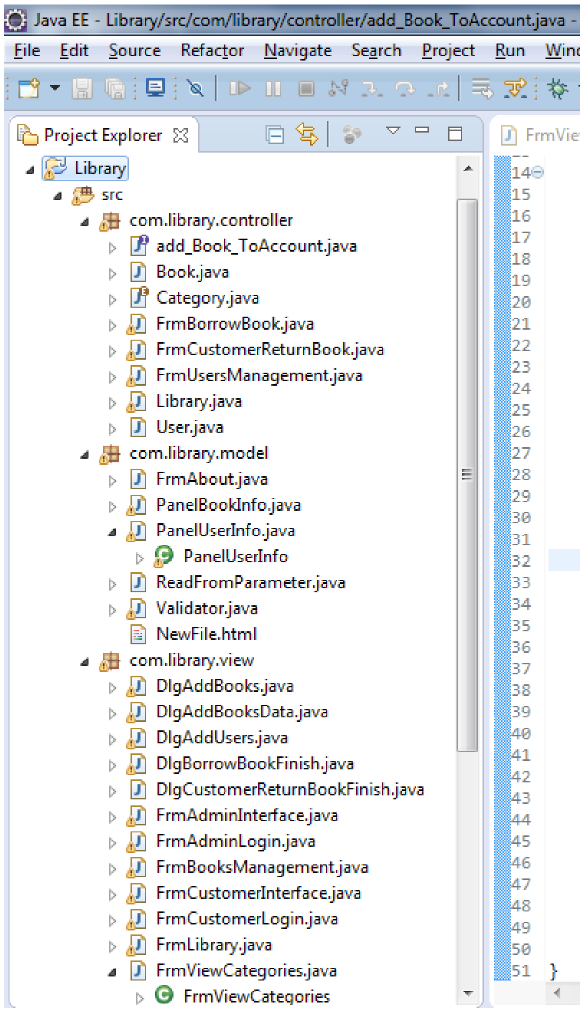
Com.library.controller → acts as a controller part

Com.library.model → acts as a Model part

Com.library.view → acts as a view part

All these together forms our MVC type of pattern and we an extra package named test which is used to test the complete functionality of the project by calling the main files to execute the code and start the application. The structure can be viewed in below Fig.2.

Fig2: MVC type of file structure for the project



Java EE - Library/src/com/library/controller/add_Book_ToAccount.java -

```
Library
  src
    com.library.controller
      add_Book_ToAccount.java
      Book.java
      Category.java
      FrmBorrowBook.java
      FrmCustomerReturnBook.java
      FrmUsersManagement.java
      Library.java
      User.java
    com.library.model
      FrmAbout.java
      PanelBookInfo.java
      PanelUserInfo.java
        PanelUserInfo
      ReadFromParameter.java
      Validator.java
      NewFile.html
    com.library.view
      DlgAddBooks.java
      DlgAddBooksData.java
      DlgAddUsers.java
      DlgBorrowBookFinish.java
      DlgCustomerReturnBookFinish.java
      FrmAdminInterface.java
      FrmAdminLogin.java
      FrmBooksManagement.java
      FrmCustomerInterface.java
      FrmCustomerLogin.java
      FrmLibrary.java
      FrmViewCategories.java
        FrmViewCategories
```

Here we want to show more about segregating of classes based on their functionality and application rather than just file structure differentiating on the type of file. The MVC pattern we implemented is more of the class implementation and logic written in the code.

B. Extract Method

We have used extract method refactoring whenever needed in the project. There are many extract methods we have implemented in project as a part of refactoring. We have few methods which are grouped together and can be extracted by making a call to new method where the extract code is implemented. This make code more readable, less duplication, and isolated independent part of the code. A simple example is given below that we have implemented:

Before:
File name: DlgBorrowBookFinish.java
Code snippet:

```
public class DlgBorrowBookFinish extends JDialog{

// All other code goes here

Public void setSuccess(DlgAddBooks dlgAddBook)
{
// All other code goes here
}
Public void setFailed(DlgAddBooks dlgAddBook)
{
// All other code goes here
}
}
```

File name: BorrowBookFinish.java
Code snippet:

```
public class BorrowBookFinish extends JDialog{

// All other code goes here
}
Public void setSuccess(Book book)
{
// All other code goes here
}
Public void setFailed(Book book)
{
// All other code goes here
}
```

Similarly, we have many other extract method refactoring techniques implemented in the project.

C. Inline method

We have composing method refactoring techniques in our project, where we have used inline method refactoring at times when needed. When a method simply delegated to another method, and there are many such type of methods and becoming confusing to tackle. So, we have used inline method to tackle such problems

D. Introduce parameter object

We have introduced a parameter object in PanelUserInfo.java class, since we have repeating group of parameters being used we have replaced them with a parameter object called from other class.

E. Observer pattern

We have tried to implement an observer pattern in our project, where we created an interface named add_Book_ToAccount.java
It takes the updates made from the user and forward to the method class where the data need to be updated according the changes and options selected by the user.

F. Composite pattern

We tried to implement a composite pattern to one of our modules of customer login, where all the details groups of customer are taken and are treated as a single object later in the project. We have admin and customer users which are treated as a tree but their usage is different in the project implementation. Main aim of composite pattern is such that it creates a class that contains group of its own objects and it provides ways to modify its group of some objects. Composite pattern is used where we need to treat a group of objects in similar way as a single object.

G. Singleton pattern

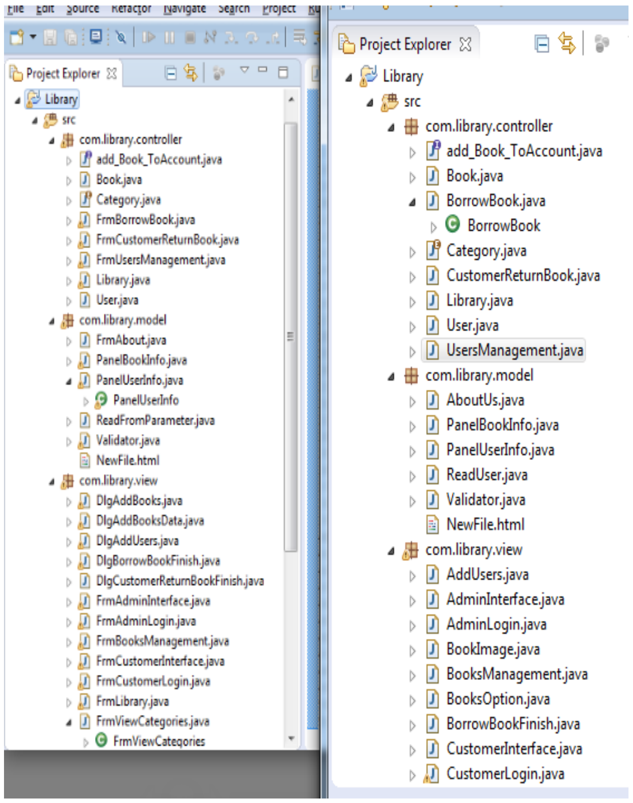
We have created a singleton pattern class that holds all the common objects being used in the project which can be seen in validator.java file that contains a defined condition for validation of Regex objects required in the project and this pattern is used at couple of times when needed in the project which can be seen in different files being implemented.

H. Given good meaningful names to all classes.

I. Pronounceable names for all the classes.

After applying MVC pattern and refactoring techniques as needed to most of the code, we started updating the names of the classes i.e. file that are meaningful and pronounceable. Just to differentiate, the before and after snapshots of name of few of the files is as shown below in Fig 3:

Fig 3: Comparing name of files before and after



J. Comments which are mandated and informative are implemented

K. Removed disinformation in naming

L. Removed scary comments:

for example,

Before:

File name: Library.java

```
    while(userItr.hasNext()){
        User tempUser = userItr.next
        if(userName.equals(tempUser.
            return true;
        }//end if
    }//end while
    return false;

}//login check  - old version
```

After:

File name: Library.java

```
    while(userItr.hasNext()){
        User tempUser = userItr.next();
        index++;
        if(tempUser.getUserId()==userId){
            userList.set(index, user);
            return true;
        }
    }

    return false;
}
```

M. Explained intention of code when needed

N. Self-encapsulated field

O. Move method refactoring:

Move method refactoring technique has been applied at many places as needed when required. For example, userManagement method has been moved out of main class context as it not needed at that place making more sense by separating it from the place where it is originally available.

P. Introduced enum methods:

We have introduced enum method named as category that can be find in Category.java file.

VI. After refactoring

After applying all kinds of refactoring techniques and design patterns, we have tested the system to make sure its functionality is not changed and is working as it is before and needed to be.

On the initial systems, we build some test cases to know the working of the system and applied them on the system after refactoring and it works as it needed to be there by concluding the functionality of the system remained same even after refactoring and it was success. Couple of test cases included as follows:

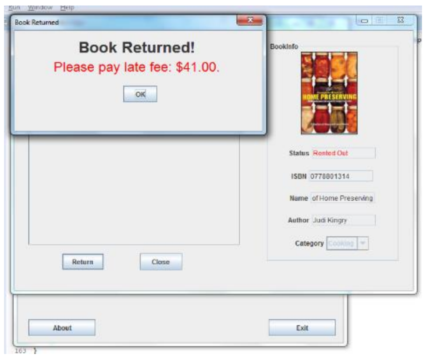
System start:

The system starts and work as it needed before and after refactoring

All functionalities are working in the same manner as they are before

The overdue fee is calculated in the same way as it is before, for example after refactoring we created an overdue book and the result fee is same as it needed to be as shown in figure 4.

Fig 4: Overdue cost notification when returning book



VII. File execution

This is a JAVA core project. We need to compile and execute to run the code. The steps for compiling and execution are as follows:

- Extract the Zip file
- Open command prompt and go to the folder where the project is residing using following command in command prompt:

>Cd <file-path>

Now we go into source folder using the command

>cd src

And now we compile the java files to class files using the following command, please note that file names are case sensitive:

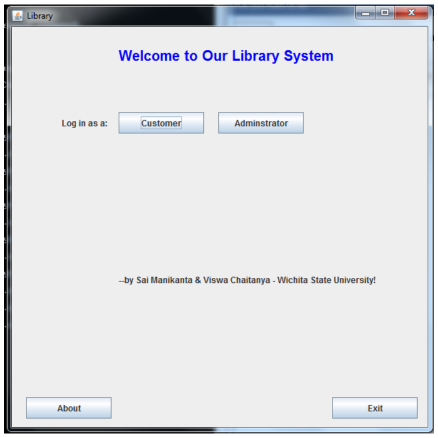
>**javac com\library\view\LibraryApplication.java**

Later we execute the project using following command:

>java com.library.view.LibraryApplication

Once the above code is entered in command prompt and press enter, we will see the welcome screen of our application. Sample welcome screen is as below in the Fig 5.

Fig 5: Sample welcome screen of our application


Library
Welcome to Our Library System
Log in as a:
Customer
Adminstrator
--by Sai Manikanta & Viswa Chaitanya - Wichita State University!
About
Exit

VIII. Conclusion

Finally, we have completed the project applying refactoring’s and implementing few of the design patterns. More than a project we have learnt different techniques of refactoring that can be applied and the way design patterns are implemented. We worked parallelly on all changes and updated the changes to GitHub which also helped us to learn git version control and project management with git. We learned all the basic and high level concepts of the techniques that we have applied and those that are taught in the class. We shared our work equally and contributions of each of us are shown in the below table 1.

Time spent: we started our project in October and started identifying the bad smells and iteration planning of the project and finally we have completed with the changes by December first week and started documenting all those we have implemented in the project.

Challenges: we faced lot of challenges while applying refactoring and implementing design patterns. At times the project starting breaking due to improper references to and relation with other classes and other challenges include choosing a bad refactoring technique or design pattern to specific part of the code. We identified each of the problems while implementing the project and learnt different things on refactoring and design patterns and their applications with the different situations and better choice of application according the situations that we faced.

Table 1: Contributions

Task Performed	Sai Manikanta	Chaitanya

Iteration planning	50%	50%
UML diagrams	50%	50%
Code	50%	50%
Model classes	100%	0
View classes	0	100%
Controller classes	50%	50%
Refactoring's	50%	50%
Design patterns	50%	50%
Testing	50%	50%
Documentation	50%	50%

Design patterns:
MVC pattern – Sai Manikanta, and Chaitanya
Observer – Sai Manikanta
Composite – Chaitanya
Other patterns – Sai Manikanta, and Chitanya

Refactoring:
All the refactoring's are equally divided based on the module we worked on later, we combined and together implemented them if any needed.
Removing, Updating comments – Sai Manikanta
Renaming classes – Chaitanya.

IX. Furtue directions

Although we made most of the changes and implementations of different refactoring's and design patterns, still there is lot of scope for improving the project and identifying bad smells. In future, we want to further refine the project to understand and identify different problems and solutions for each of them. As a part of project expansion, we can connect a database server instead of using collection objects for data storage and we can further implement other design patterns to specific parts of code.

X. References

Refactoring tecniques:

- Class notes slides
- Refactoring guru website, https://refactoring.guru/

Design Patterns:

- Class slides
- Source making website, https://sourcemaking.com/design_patterns
- Tutorial point: design patterns, https://www.tutorialspoint.com/design_pattern/design_pattern_overview.htm
- OOdesign website, http://www.oodesign.com/
- Headfirst design patterns book

Project management:
Eclipse
Rational rose
Github:
https://github.com/SaiManikanta23/Library/tree/LibMVC

Note: Library master branch has original files with few refactored source code; LibMVC is the branch where we implemented all refactoring and design patterns on the code. We haven't merged them because of conflict of file structure and to differentiate before and after process of the project.