CS 780 Fall 2016 EECS/WSU

Advanced Software Engineering

Project report

Use of Design Patterns and Refactoring in Library System

Nimisha Niyogi , Todo Sidabalok, Michelle Thomas

Department of Electrical Engineering and Computer Science Wichita State University

Wichita, Kansas 6760, USA

Email:nbniyogi@shockers.wichita.edu

tododes3@gmail.com , maplaven@shockers.wichita.edu

Abstract: The aim of this project is to refactor the existing code and to improve the design of the program by implementing design patterns.

I. INTRODUCTION

In this project we picked an existing software system. 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. We chose a Library System program.

Library System is a Java program used to manage the inventory of a library. The original program had only books and magazines as inventory, and kept track of users in the system. We added an article class to the program. The original program allowed you to only add one inventory item at a time from keyboard input. We added the ability to add inventory from a file. The original program allowed you to view the inventory one class at a time. We added the ability to view all inventory at once.

Initially the project code was randomly generated and didn't have proper workflow. There were many bad smells, and the code was not closed for change but open for extension. The code was not modular either. We wanted to tackle this problem by applying some techniques to make code reusable and follow the standard coding principles of software engineering. To be more plug-and-play we decided to use the MVC (Model-View-Controller) design pattern so that we can separate the model, view, and controller parts of the code respectively and build a connection between them to communicate with each other accordingly. This type of pattern helps for loose coupling, better reusability of code, and enhanced flexibility, it is widely used in the software industry in recent days. We have followed the iteration process as described below, where we initially corrected some of the old code and applied a few refactoring techniques. Later we introduced our design patterns to make the program reusable and maintainable.

II. ITERATIONS

Iteration 1: In the initial stage, we concentrated on managing the old code and making it ready for our new patterns that we were planning to implement. We implemented some refactoring in the code such as extract method, move method and other techniques. We have tried to remove the dependencies in the code. We also implemented the singleton pattern and the factory pattern in our first iteration, and added the article class.

Iteration 2: In the second iteration, we made drastic changes to the code. Adding inventory with a file functionality was added. The major refactoring techniques implemented at this stage, were extract method, extract class, simplifying conditional expressions, and introduce parameter object. We also removed some unnecessary comments and updated comments to meaningful context and changed the name of classes, methods, and variables to more meaningful names. We implemented the Strategy

Pattern, the MVC Pattern, the Template Pattern, and the Null Object Pattern in our project.

A. Key Terms and Definition

This section contains the definition of different terms used in the report, e.g. *Design Patterns* and *Refactoring*.

Design Patterns: In software engineering, a design pattern is a general repeatable solution to a commonly occurring problem. It is a template for how to solve a problem that can be used in many different solutions.

Refactoring: Code refactoring is the process to restructure the code without changing its external behavior.

The following sections, give a detailed description about how design patterns and refactoring were implemented.

III. REFACTORINGS USED

1) Rename Variable/Method/Class: Names are the means through which the developers communicate with each other. The name of the variable or method or class is very crucial as it is primarily used to convey its purpose. Hence, it is imperative that the chosen name is meaningful and intent-revealing.

In this project there were many instances where we had to apply this type of refactoring as sometimes the variable name was bad, or the class name was bad, or the method name was bad. Fig.1 shows the original bad name and Fig.2 show the refactored name.

```
private String fName;
private String lName;
```

Fig. 1- Shows the original Name

```
private String firstName;
private String lastName;
```

Fig. 2- More meaningful name is given which is intent revealing.

In this project we also came across some encoded names. These names just made it difficult to read the program. So, we gave better names to such variables. Fig. 3 shows the original encoded name and Fig .4 shows the refactored name. Because of renaming variables, methods, and classes, the program becomes more readable and easy to understand. The name will clearly tell about the intent.

```
Option(int value, String desc) {
    this.value = value;
    this.description = desc;
}
```

Fig. 3- A code snippet showing the use of encoded names

```
Option(int value, String description) {
    this.value = value;
    this.description_magazine = description;
}
```

Fig. 4-the refactored names after applying Rename technique.

2)Extract Method/Class: A class may contain many fields/methods/lines of code. This makes the code less readable and hard to figure out if there is any error. Extract Method solves this problem of Long class. The Extract method refactoring technique is used to keep the methods cohesive (i.e.) to perform a single operation. There are many extract methods we have implemented in the project as a part of refactoring. We had a few methods which were grouped together and could be extracted by making a call to a new method where the extracted code is implemented.

One of the extract method examples from the project is as follows: Library.java was a long class. So, we have applied extract method to make it readable and remove the unnecessary code from the class.

Extract Method was implemented, by moving AlphabeticalComparator and DateComparator out of their current class and made into their own class to remove noise.

Because of this extract method implementation there is more readable code, less duplication, and we have isolated independent parts of the code

```
package data;

// Extract Refactoring from library class
import java.util.Comparator;

/**
 *
 * @author Nimisha
 */
public class AlphabeticalComparator implements Comparator<Publication> {
        @Override
        public int compare(Publication o1, Publication o2) {
            if (o1 == null && o2 == null) {
                return 0;
            }
```

Fig. 5 – Shows the extract method AlphabeticalComparator

```java
package data;
// Extract Refactoring from library class
import java.util.Comparator;

/**
 *
 * @author Nimisha
 */
 public class DateComparator implements Comparator<Publication> {
       @Override
       public int compare(Publication o1, Publication o2) {
           if (o1 == null && o2 == null) {
               return 0;
           }
```

Fig. 6 – Shows the extract method DateComparator

3) Code Cleanup: The commented code in the source file was cleaned. Commented code makes the code look cluttered and can also be misleading at times for other developers. They often leave an impression that the piece of code, although commented, is important and hence has not been removed. There is a good chance that the developers might spend time in trying to understand the commented-out code in the process of trying to understand the whole application.

Fig.7. Shows the bad comments in one of the java files. This comment was just noise as the names were self-explanatory.

```java
//variable for communication with the user
private DataReader dataReader;
private FileManager fileManager;

//"library" that stores data
Library library;
```

Fig. 7 - Comment in the source code which was not required.

4)Removing Implementation Specific terms: In the code there were implementation specific variable names. Such a name is redundant as the type of variable is evident from the type declaration. Also, when the implementation is changed the name could be misleading. Such names were removed from the code and meaningful names were given.

5) Move Method: This can be used when a method is used more in another class than in its own class. To solve this problem, we create a new method in the class that uses the method the most, then move code from the old method to there. We then turn the code of the original method into a reference to the new method in the other class or else remove it entirely

6) Reducing cluttered code and improving readability: If there are multiple catch blocks it is difficult for the programmer to read and understand the program. So, it is important to remove this kind of cluttering from the code and improve readability. Fig. 8 and Fig. 9 show the cluttered code and removing of the cluttered code**.**

```java
public static Library getInstanceFromSaveFile(){
        if(instance == null){
                if(fileManager == null)
                        fileManager = LibraryFileManager.getInstance();
                try {
                        instance = fileManager.readLibraryFromFile();
                        System.out.println("Data loaded from the file");
                } catch (FileNotFoundException e) {
                        instance = getInstance();
                        System.out.println("New library database created.");
                } catch (ClassNotFoundException e) {
                        instance = getInstance();
                        System.out.println("New library database created.");
                } catch (IOException e) {
                        instance = getInstance();
                        System.out.println("New library database created.");
                }
        }
        return instance;
}
```

Fig. 8- Shows clutter code

```java
try {
        instance = fileManager.readLibraryFromFile();
        System.out.println("Data loaded from the file");
} catch (FileNotFoundException e) {
        instance = getInstance();
        System.out.println("New library database created.");
} catch (ClassNotFoundException | IOException e) {
        instance = getInstance();
        System.out.println("New library database created.");
}
```

Fig. 9 – Refactored Code

7)Replace Nested Conditional with Guard Clauses: Used when we have a group of nested conditionals and it is hard to determine the normal flow of code execution. In this case we isolate all special checks and edge cases into separate clauses and place them before the main checks. Ideally, there should be a "flat" list of conditionals, one after the other.

In this project we encountered this condition.

```
if (borrowedPublications == null) {
    if (other.borrowedPublications != null)
        return false;
} else if (!borrowedPublications.equals(other.borrowedPublications))
    return false;
if (publicationHistory == null) {
    if (other.publicationHistory != null)
        return false;
} else if (!publicationHistory.equals(other.publicationHistory))
    return false;
return true;
```

Fig. 10 – Shows nested if.

```
if (borrowedPublications == null && other.borrowedPublications
    return false;
if (!borrowedPublications.equals(other.borrowedPublications))
    return false;
if (publicationHistory == null && other.publicationHistory != r
        return false;
```

Fig. 11- Shows the refactoring done on nested if's

8) Switch Statements: In the project we encountered a complex switch statement. To remove this complex structure we need to isolate switch and put it in the right class, which may need extract method and then move method . OR if a switch case is there which is based on type code, such as when the program's runtime mode is switched, we use replace type code with subclasses or replace type code with State/Strategy. After specifying the inheritance structure, we use replace conditional with polymorphism.

9)Replace Conditional with Polymorphism: This refactoring can be applied when we have a conditional that chooses different behavior depending on the type of an object. In this project we implemented this refactoring and combine it with Strategy pattern.

```
public void controlLoop() {
    app.GetOptions.Option option = null;
    while (option != app.GetOptions.Option.EXIT) {
        try {
            printOptions();
            option = app.GetOptions.Option.createFromInt(dataReader.getInt());
            switch (option) {
            case ADD_BOOK:
                addBook();
                break;
            case ADD_PERIODICAL:
                addPeriodical();
                break;
            case PRINT_BOOKS:
                printBooks();
                break;
            case PRINT_PERIODICALS:
                printPeriodicals();
                break;
            case ADD_USER:
                addUser();
                break;
            case PRINT_USERS:
                printUsers();
                break;
            case EXIT:
                exit();
            }
        } catch (InputMismatchException exception) {
            System.out.println("Incorrect data entered, no publication added."+exception);
        } catch (NumberFormatException | NoSuchElementException e) {
            System.out.println("The selected option does not exist, select again:");
        }
    }

    dataReader.close();
}
```

Fig. 12- Shows the code with switch cases

```
public void controlLoop() {
    MenuItem currentMenuItem = new NullMenuItem();
    while (currentMenuItem.getIndex() != EXIT_NUMBER){
        try {
            printOptions();
            currentMenuItem = dataReader.getMenuItem();
            currentMenuItem.OnSelected();
        } catch (InputMismatchException exception) {
            System.out.println("Incorrect data entered, no publication added."+exception);
        } catch (NumberFormatException | NoSuchElementException e) {
            System.out.println("The selected option does not exist, select again:");
        }
    }

    dataReader.close();
}
```

```
package Menu;

import LibraryActions.LibraryBehaviour;

public abstract class MenuItem {

        protected int index;
        protected String description;
        protected LibraryBehaviour myBehaviour;

        public MenuItem(int index, String description) {
                super();
                this.index = index;
                this.description = description;
        }

        public void OnSelected(){
                myBehaviour.execute();
        }

        public int getIndex() {
                return index;
        }

        public String getDescription() {
                return description;
        }

}
```

Fig. 13- The code after replacing conditional polymorphism

```
public class AddBookMenuItem extends MenuItem {

        public AddBookMenuItem(int index, String description) {
                super(index, description);
                myBehaviour = new AddBookBehaviour();
        }

}
```

Fig. 14- Control loop after using replace conditional with polymorphism.

So, we replaced the long switch case statement with MenuItem class and declare it as abstract. Each MenuItem subclass will have the behaviours for each of those switch cases (AddBookMenuItem will call addBook method, AddUserMenuItem will call addUser, etc), and each of those calls will be executed in OnSelected method. The MenuItem class will be further explained in Strategy pattern section.

10)Replace constructor with factory method: We use this type refactoring when there is a constructor that does something more than just setting parameter values in object fields. To solve this problem, we create a factory method and use it to replace constructor calls.

```
 public class AddArticle extends Menu {
+
+    public static AddArticle AddFactory() {
+        return new AddArticle();
+    }
```

Fig. 15 – Shows the constructor being replaced by factory

IV. DESIGN PATTERNS IMPLEMENTED

1)Simple Factory Method: This is a creational design pattern wherein the task of creating objects is dedicated to a Factory class. In the future, when the application expands, more objects may have to be created. The factory pattern uses factory methods to deal with the creation of objects without having to specify the exact class of the object that will be created.

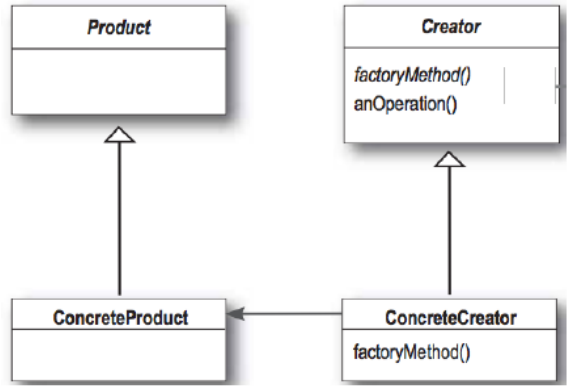
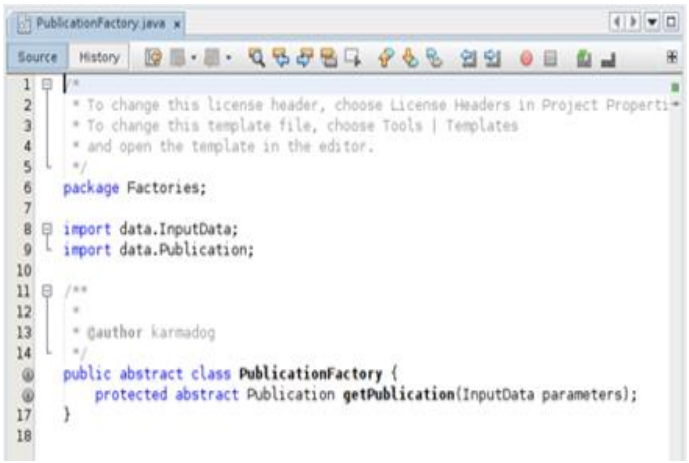


Fig. 16- Basic UML structure for Simple Factory Pattern

To implement the factory method, first we rename the Magazine class to Periodical, since it is storing both magazines and newspapers. The Book and Periodical classes already inherited from the Publication class. We then created the PublicationFactory class.



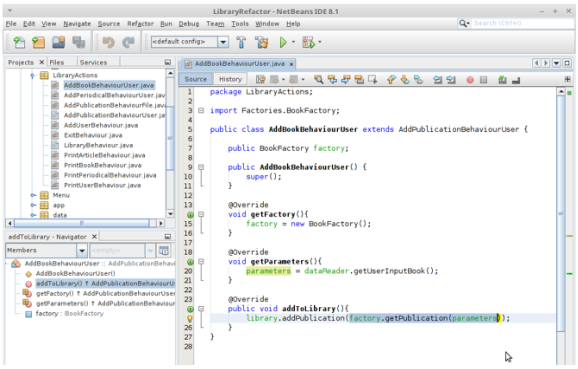
The derived classes are BookFactory, ArticleFactory, and PeriodicalFactory, all override the getPublication() method the same way except for which type of publication they return. BookFactory is shown as an example.

```
public class BookFactory extends PublicationFactory {


    @Override
    public Publication getPublication(InputData parameters){
        return new Book(parameters);
    }

}
```

Fig. 17– BookFactory implemented



The factory is used with getPublication() instead of using new Book()

The program now uses a factory to create the Publication objects. While doing this we saw the need to change an extensive list of parameters to an object. This is when the InputData class was created.

2)Separating the Model and the View Layer: The previous version of Library System, used the" Main Activity" class (i.e. View) to both fetch and display the words on the screen. To preserve MVC in the application, a big refactoring was applied. Initially the project files were not in proper packages and folders. We implemented the refactoring and changed the class name accordingly and tried to put in proper project structure.

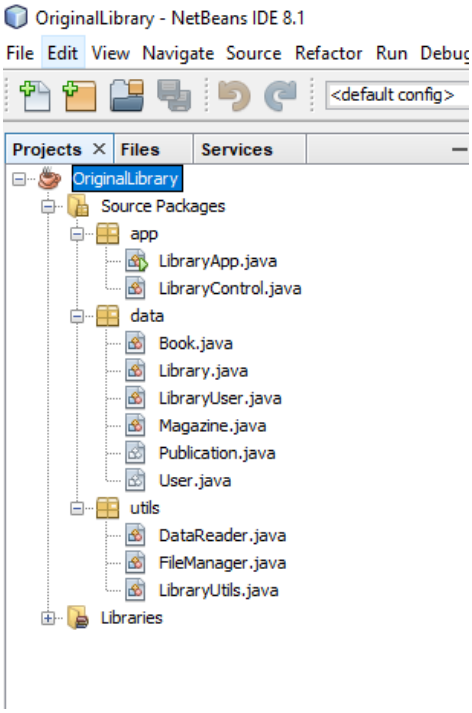


Fig. 18 – Shows the initial bad structure of project

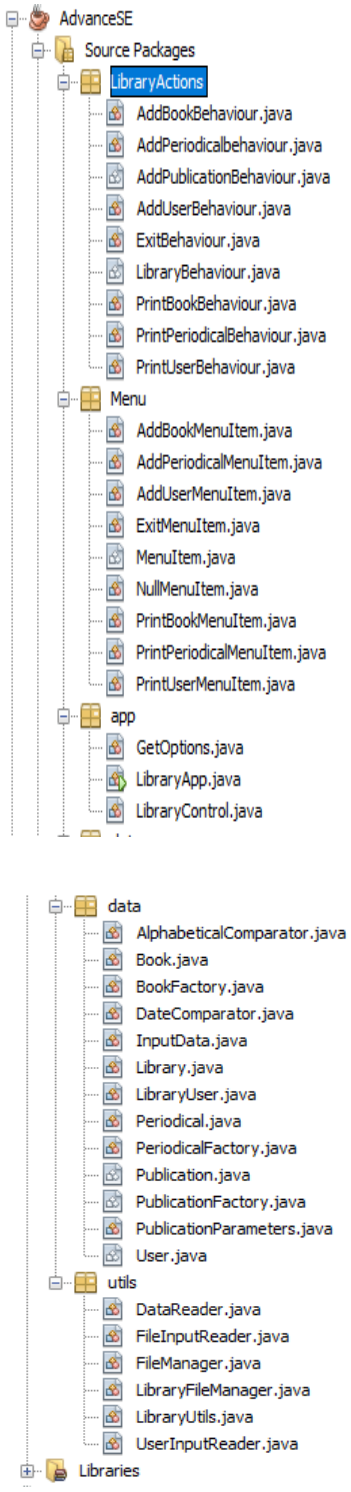


Fig. 19– Proper Structure of project (According- to MVC)

3)Singleton Pattern: **Singleton** is a creational design pattern that lets you ensure that a class has

only one instance and provide a global access point to this instance.

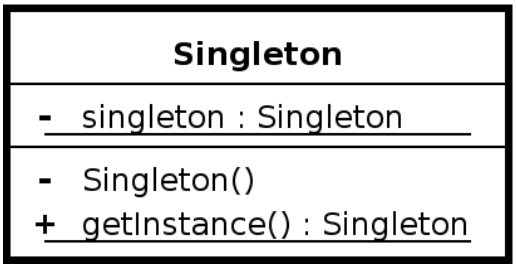


Fig. 20- Basic UML structure for Singleton Pattern

After reviewing the code, we saw that the LibraryControl class is a class that will create one instance of it and that instance will control the entire library program. So, we decided it was better to implement a singleton pattern to prevent multiple instantiations that may cost a lot of garbage collection.

To do that, we created a private static instance of LibraryControl itself, and a private constructor so that we can prevent other classes from creating another instance of it. Next, we create the public static getInstance() method as the only way to get/create an instance of this object. If the singleton instance is null, then it creates new one. Otherwise, it returns the existing one.

```
private static LibraryControl instance;

private LibraryControl() {
    dataReader = new DataReader();
    fileManager = new FileManager();
    library = Library.getInstanceFromSaveFile();
    System.out.println("Data loaded from the file ");
}

public static LibraryControl getInstance(){
    if(instance == null)
        instance = new LibraryControl();
    return instance;
}
```

Fig. 21- Instance being created

And this is how we get the LibraryControl instance in LibraryApp class, by calling the getInstance method.

```
public static void main(String[] args) {
    final String application_Name = "Library System version-1.0 ";
    System.out.println(application_Name);
    LibraryControl libraryControl = LibraryControl.getInstance();
    libraryControl.controlLoop();
}
```

Fig. 22 – The client or the main class

The same thing also goes for the Library class. But because Library can also be instantiated by loading the save file which is done by the FileManager class, we made two getInstance class, the first one is the standard singleton getInstance() methods, and the second one is getInstanceFromSaveFile(), that uses a private static FileManager object inside the Library class, and then from that static FileManager object we can instantiate the Library instance.

```
private static Library instance;
private static FileManager fileManager;

private Library() {
    publications = new HashMap<>();
    users = new HashMap<>();
    fileManager = new FileManager();
}

public static Library getInstance(){
    if(instance == null)
        instance = new Library();
    return instance;
}

public static Library getInstanceFromSaveFile(){
    if(instance == null){
        if(fileManager == null)
            fileManager = new FileManager();
        try {
            instance = fileManager.readLibraryFromFile();
        } catch (FileNotFoundException e) {
            instance = getInstance();
        } catch (ClassNotFoundException e) {
            instance = getInstance();
        } catch (IOException e) {
            instance = getInstance();
        }
    }
    return instance;
}
```

Fig. 23- Shows the Changes in code in Library.java file

Finally, we call the getInstanceFromSaveFile() inside the LibraryControl constructor to get/create the Library object.

```
private LibraryControl() {
    dataReader = new DataReader();
    fileManager = new FileManager();
    library = Library.getInstanceFromSaveFile();
    System.out.println("Data loaded from the file ")
}
```

Fig. 24 – Instance is called in LibraryControl Class

4)Strategy Pattern: Strategy is a behavioral design pattern that lets you define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.

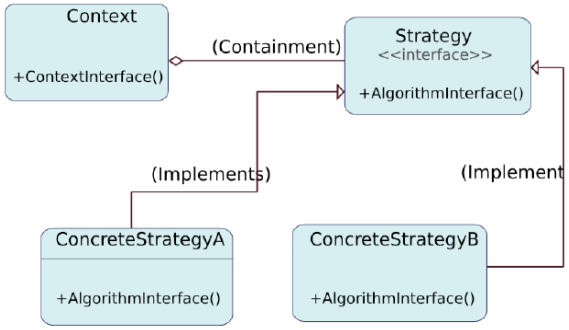


Fig. 25- Basic UML structure for Strategy Pattern

```
package LibraryActions;

import data.Library;

public abstract class LibraryBehaviour {

        protected Library library;

        public LibraryBehaviour(){
                library = Library.getInstance();
        }

        public abstract void execute();

}
```

```
package LibraryActions;

import Factories.ArticleFactory;
import View.AddArticle;

public class AddArticleBehaviour extends LibraryBehaviour {

    public AddArticleBehaviour() {
        super();
    }

    public void execute() {
        new AddArticle();
    }
}
```

```
package LibraryActions;

import utils.LibraryFileManager;

public class ExitBehaviour extends LibraryBehaviour {

        private LibraryFileManager fileManager;

        public ExitBehaviour() {
                super();
                fileManager = LibraryFileManager.getInstance();
        }

        @Override
        public void execute() {
                fileManager.writeLibraryToFile(library);
        }

}
```

```
package LibraryActions;

import View.BookDisplay;

public class PrintBookBehaviour extends LibraryBehaviour {

        public PrintBookBehaviour() {
                super();
        }

        @Override
        public void execute() {
                new BookDisplay(library);
        }

}
```

```
package LibraryActions;

import View.UserDisplay;

public class PrintUserBehaviour extends LibraryBehaviour {

        public PrintUserBehaviour() {
                super();
        }

        @Override
        public void execute() {
            new UserDisplay(library);
        }

}
```

```
package LibraryActions;

import View.AddUser;
import data.LibraryUser;

public class AddUserBehaviour extends LibraryBehaviour {

        public AddUserBehaviour(){
                super();
        }

        @Override
        public void execute() {
            new AddUser();
        }

}
```

Fig. 26- The library strategies class

The strategy pattern will be implemented for the library behaviour, which is the behaviour that will be executed when we choose one of the menu item.

```
package Menu;

import LibraryActions.LibraryBehaviour;
import java.util.Scanner;

public abstract class MenuItem {

        protected int index;
        protected String description;
        protected LibraryBehaviour myBehaviour;
        private final Scanner scanner;

        public MenuItem(int index, String description) {
                super();
                this.index = index;
                this.description = description;
                scanner = new Scanner(System.in);
        }

        public void OnSelected(){
                myBehaviour.execute();
        }

        public int getIndex() {
                return index;
        }

        public String getDescription() {
                return description;
        }


}
```

Fig. 27 – Shows Menu Items being implemented

The MenuItem class will have LibraryBehaviour object in it, so every subclass of Menu will be able to dynamically choose what kind of library behaviour it wants to have.

```
package Menu;

import LibraryActions.AddUserBehaviour;

public class AddUserMenuItem extends MenuItem {

        public AddUserMenuItem(int index, String description) {
                super(index, description);
                myBehaviour = new AddUserBehaviour();
        }
}
package Menu;

import LibraryActions.PrintBookBehaviour;

public class PrintBookMenuItem extends MenuItem {

        public PrintBookMenuItem(int index, String description) {
                super(index, description);
                myBehaviour = new PrintBookBehaviour();
        }

}
```

Fig. 28 – Shows the different behavior one can choose from Menu Items

5)Template Pattern: In the Template pattern, an abstract class defines way(s)/template(s) to execute its methods. Its subclasses can override the method implementation as per need, but the invocation is to be in the same way as defined by the abstract class. This pattern comes under the behavior pattern category. The motivation for implementing the Template Pattern came from seeing that the same four methods had to be executed every time a new inventory item was added, regardless of class type. Defining a template for the addition followed from that observation**.**

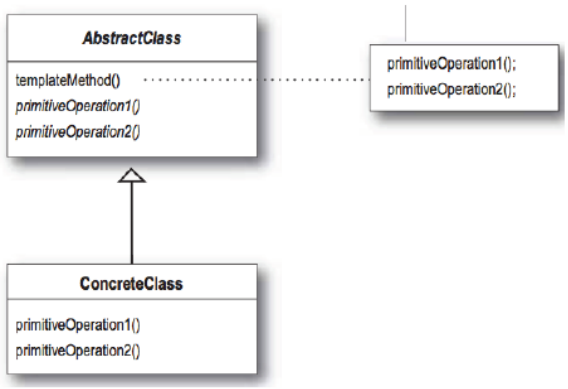


Fig. 29- Basic UML structure for Template Method.

```
package LibraryActions;

import data.InputData;
import utils.UserInputReader;


public abstract class AddPublicationBehaviour extends LibraryBehaviour {

        public UserInputReader dataReader;
        public InputData parameters;

        public AddPublicationBehaviour(){
                super();

        }

        @Override
        public final void execute(){
            getDataReader();
            getFactory();
            getParameters();
            addToLibrary();
        }

       void getDataReader(){
            dataReader = UserInputReader.getInstance();
        }

        abstract void getFactory();
```

Fig. 30- The base class AddPublicationBehaviourUser

The template is the execute method. We implemented getDataReader here in the base class, since all the derived classes will be using the same dataReader.

```
public class AddBookBehaviour extends AddPublicationBehaviour {
        public BookFactory factory;

        public AddBookBehaviour() {
                super();
        }

        @Override
        void getFactory(){
            factory = new BookFactory();
        }

        @Override
        void getParameters(){
            parameters = dataReader.getUserInputBook();
        }
        @Override
        public void addToLibrary(){
             library.addPublication(factory.getPublication(parame
        }
}
```

Fig. 31- First derived class

```
public class AddPeriodicalbehaviour extends AddPublicationBehaviour {
    public PeriodicalFactory factory;


       public AddPeriodicalbehaviour() {
              super();
       }
```

Fig. 32- Second Derive class

6)MVC Pattern: MVC Pattern stands for Model-View-Controller Pattern. This pattern is used to separate application's concerns.

Model - Model represents an object or JAVA POJO carrying data. It can also have logic to update controller if its data changes.

View - View represents the visualization of the data that model contains.

Controller - Controller acts on both model and view. It controls the data flow into model object and updates the view whenever data changes. It keeps view and model separate**.**

MVC is patterns of patterns. It is a compound pattern. It can implement strategy, observer and composite pattern in it.

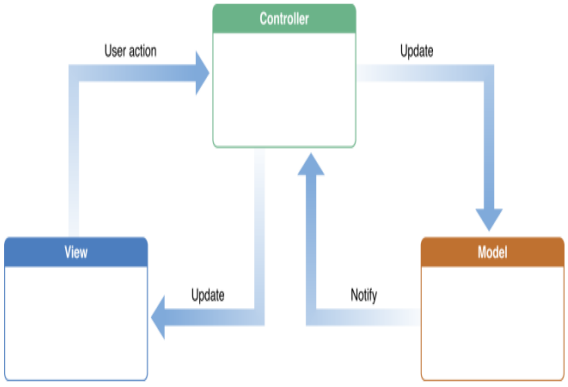


Fig. 33- General Structure for MVC

The original user interface for the program was a simple menu printed to the screen for the user to enter a number, and had all the menu choices on the same screen. To implement MVC, we first created a new interface for the program that was GUI, with submenus

```
run:
Library System version-1.0
File not found Library.o
New library database created.
Select an option:
0 - Exit program
1 - Add a book
2 - Add a magazine / newspaper
3 - Add an article
4 - View available books
5 - View available magazines / newspapers
6 - View available articles
7 - Add a new user
8 - View the list of users
9 - Upload inventory file
```

Fig. 34- The user interface before the change.

A main menu, with submenus, made of button.

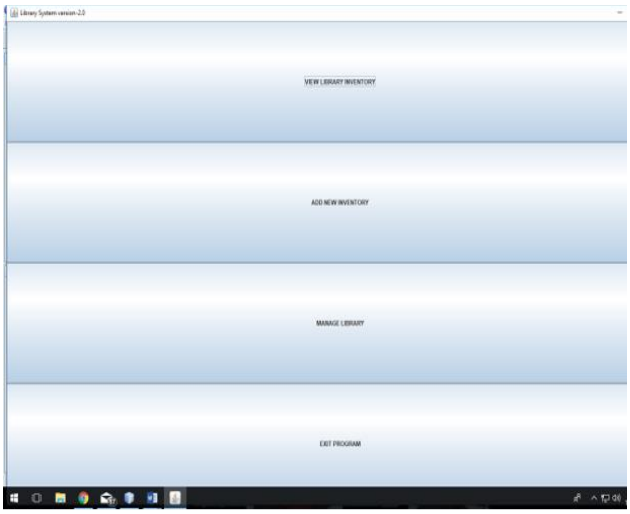


Fig. 35- Buttons with submenus

We then connected the GUI menus to the behavior classes of the Strategy pattern. This allowed us to decouple the model from the view in our program. The Strategy pattern acts as the controller.

7) Null Object Pattern: The null object pattern is a design pattern that simplifies the use of dependencies that can be undefined. This is achieved by using instances of a concrete class that implements a known interface, instead of null references. We create an abstract class specifying various operations to be done, concrete classes extending this class and a null object class providing do nothing implementation of this class that will be used seamlessly where we need to check for a null value

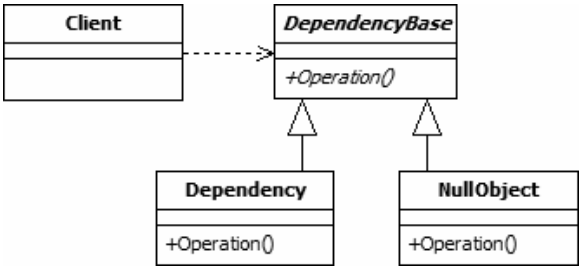


Fig. 36- Basic UML Structure for Null Object Pattern

```
package Menu;

public class NullMenuItem extends MenuItem {

        public NullMenuItem() {
                super(-1, "");
                // TODO Auto-generated constructor stub
        }

}
```

```
    private static final int EXIT_NUMBER = 0;
    private static LibraryControl instance;

    private DataReader dataReader;
    private FileManager fileManager;

    private Library library;
    private GetOptions options;

    private LibraryControl() {
    dataReader = DataReader.getInstance();
    fileManager = FileManager.getInstance();
    library = Library.getInstanceFromSaveFile();
    options = GetOptions.getInstance();
}

public static LibraryControl getInstance(){
    if(instance == null)
            instance = new LibraryControl();
    return instance;
}

public void controlLoop() {
    MenuItem currentMenuItem = new NullMenuItem();
    while (currentMenuItem.getIndex() != EXIT_NUMBER){
        try {
            printOptions();
            currentMenuItem = dataReader.getMenuItem();
            currentMenuItem.OnSelected();
        } catch (InputMismatchException exception) {
            System.out.println("Incorrect data entered, no publication added."+exception);
        } catch (NumberFormatException | NoSuchElementException e) {
            System.out.println("The selected option does not exist, select again:");
        }
    }

    dataReader.close();
}
```

Fig. 37- implementation of Null Object Pattern

Instead of initializing the MenuItem to null, this pattern enables us to initialize it with a new NullMenuItem, so that we can check the index of the current menu item (in the while loop) without being terminated by null pointer exception. NullMenuItem's index will be -1, so it will not terminate the while loop.

V-FUTURE WORK

Refactoring is an ongoing process. Like in any application, there is scope for further improvement/refining of the code in this application as well. We would like to add search functionality, subscription functionality, and, when there is no user in the list, it should display no user. As the application extends, the scope for implementing distinctive design patterns in the code also increases.

VI. CONCLUSION

Since the application is java based, internally it uses a lot of design patterns like Singleton, Strategy etc., the scope of implementing any pattern externally through code was limited. But, for this reason, the project was challenging too. Overall, this project gave us a good understanding of how to identify an appropriate design pattern for a problem and how to use different refactoring techniques to make the code better. 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.

While applying design patterns and refactoring we have faced a lot of challenges. Many times, it happened that the project starting breaking due to improper references and relations with other classes. Other challenges included choosing a bad refactoring technique or design pattern to a specific part of the code. We identified each of the problems while implementing the project and learned different things on refactoring and design patterns and their applications with the different situations and better choice of application according to the situations that we faced.

VII- CONTRIBUTIONS

Michelle Thomas

- Factory Pattern
- Template Pattern
- GUI interface (all View classes)
- Rename Class method
- Introduce parameter object
- Added file reader functionality
- Collapse Hierarchy refactor.
- Added the Article class
- Extract Class refactor
- Extract Method refactor
- Move Method refactor
- Added view all inventory functionality.

Nimisha Niyogi

- Most of the refactoring.
- Extract method.
- Rename variable method
- Remove comments
- Remove unused imports
- Remove code which is no longer required
- Reducing clutter code and improving readability
- Removing Implementation Specific terms
- Replace constructor with factory method
- Replace Nested Conditional with Guard Clauses
- Move Method
- Documentation for the iteration meetings and final report.

Todo Sidabalok

- Singleton pattern
- Null Object Pattern
- Strategy pattern, part of MVC
- Refactoring: Replace Conditional with Polymorphism
- Switch Statements

VIII- REFERENCES

[1]. Class slides.

[2]. Head first Design Patterns

[3]. Refactoring guru

Link for our GitHub repository-
https://github.com/karmadog/ClassProject780