AGILE SOLID PRINCIPLES

Semester 2 submitted to: Mr. Visham Hurbungs

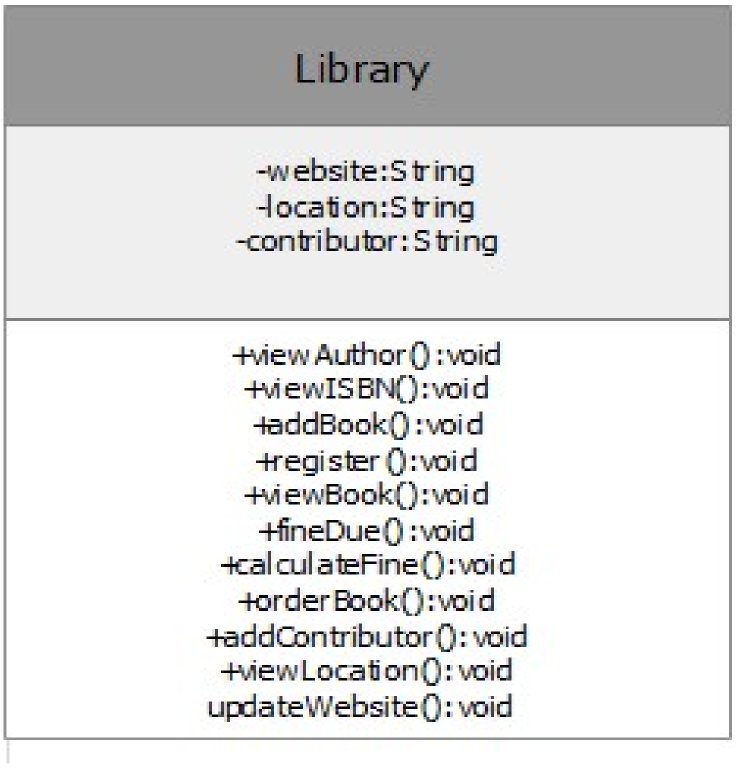
Dody Siddharth 1715790
Miniopoo Ryan 1700118
Joomun Afzaal 1710170
Hansley Kowlessur 1710275

SOLID Principles:

This assignment comprises of implementing the SOLID principles whereby it is suited in object-oriented programming. These principles make software design more understandable, easier to maintain and easier to extend. The assignment undertaken comprises of a library system written in java in which none of these principles were implemented. With further modifications, the code was modified having the SOLID principles implemented.

The SOLID design principles comprise of 5 principles namely:

- Single Responsibility Principle (SRP)
- Open Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)

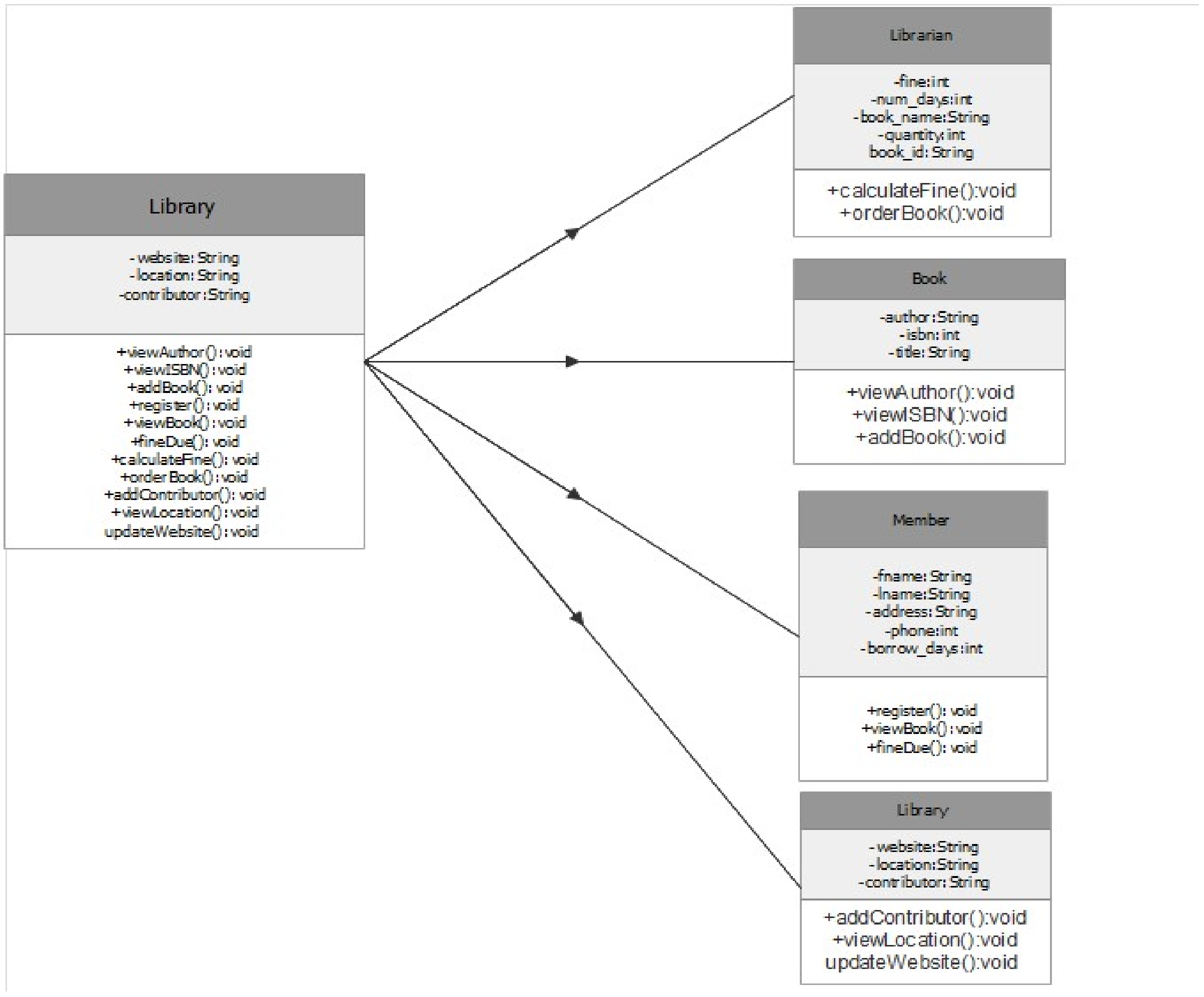


No application of the SOLID rules

Implementation of SRP rule:

The SRP rule states that a class must have a specific responsibility and nothing more. There must be a change for only a purpose and the responsibility should be encapsulated within the class thus having a strong cohesion within the class. Based on the class diagram without the rules implemented, it shows that there are functions and attributes not related to the task that the library should not perform. So as to tackle this problem, this is where SRP comes into action whereby the methods that do not make sense are moved to other classes. For example, it is not the task of the library to calculate the fine due.

Diagram showing the implementation of the SRP rule

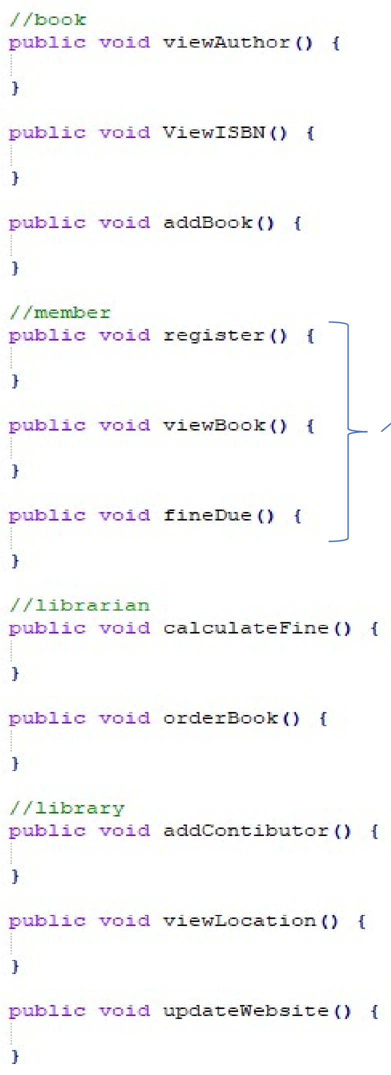


After the implementation of the SRP rule, classes were created so as to classify the methods to appropriate classes:

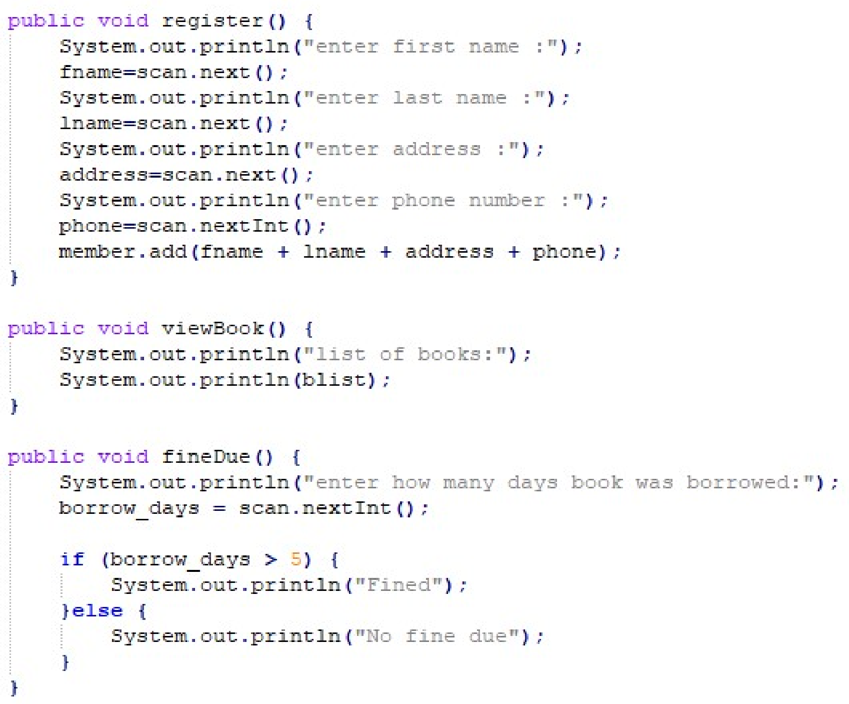
- Librarian
    - Performs the fine calculation
    - Makes order of books
- Book (for book identification)
    - View the author of the book
    - View the ISBN of the book
    - Add a new book
- Member
    - Has the ability to register
    - View the available books
    - Checks if the rented day has been surpassed
- Library
    - Add new contributors
    - View the location of the library
    - Update the website based on the commitments

Following the SRP approach, it enables ease of testing with few test cases. Moreover, the less functionality present leads to low dependencies to other classes. Proper code organisation is preserved and well-purposed classes are easier to search.

Sample code preview showing SRP:



```java
//book
public void viewAuthor() {

}

public void ViewISBN() {

}

public void addBook() {

}

//member
public void register() {

}

public void viewBook() {

}

public void fineDue() {

}

//librarian
public void calculateFine() {

}

public void orderBook() {

}

//library
public void addContibutor() {

}

public void viewLocation() {

}

public void updateWebsite() {

}
```



```java
public void register() {
    System.out.println("enter first name :");
    fname=scan.next();
    System.out.println("enter last name :");
    lname=scan.next();
    System.out.println("enter address :");
    address=scan.next();
    System.out.println("enter phone number :");
    phone=scan.nextInt();
    member.add(fname + lname + address + phone);
}

public void viewBook() {
    System.out.println("list of books:");
    System.out.println(blist);
}

public void fineDue() {
    System.out.println("enter how many days book was borrowed:");
    borrow_days = scan.nextInt();

    if (borrow_days > 5) {
        System.out.println("Fined");
    }else {
        System.out.println("No fine due");
    }
}
```

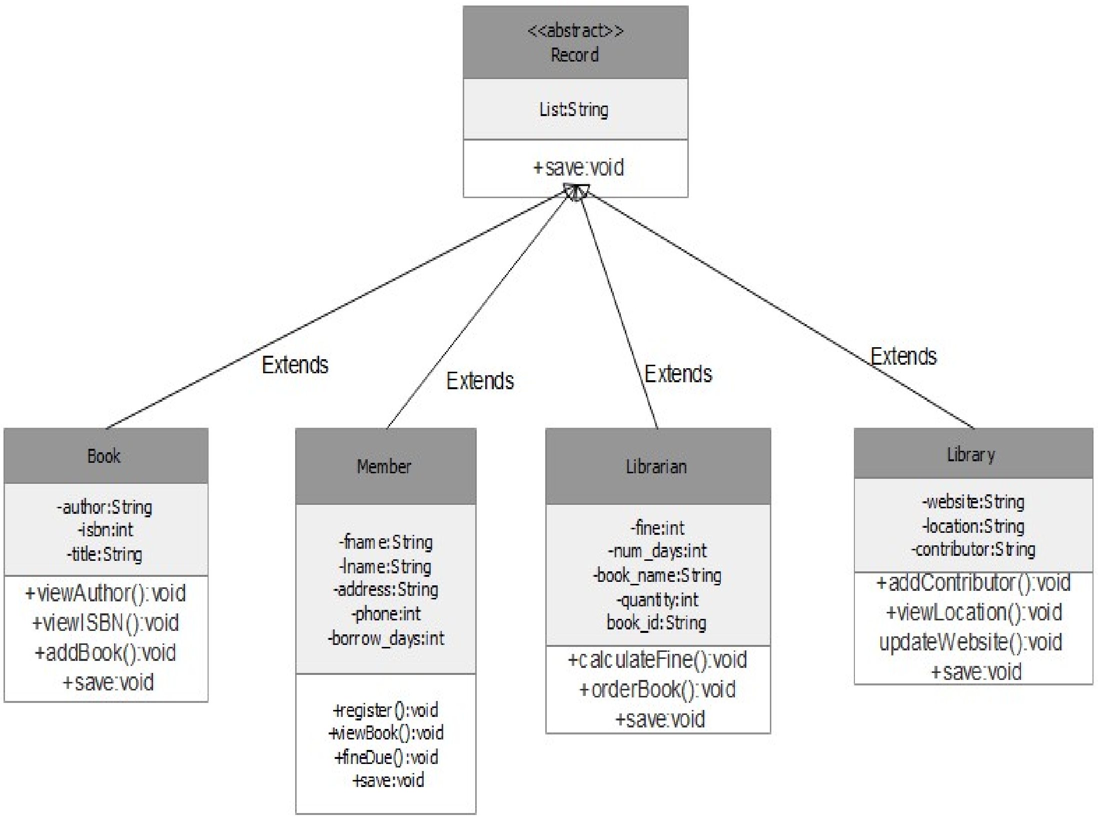
The function from the Library class is emerged into another class called Member so as to show SRP

Implementation of OCP rule:

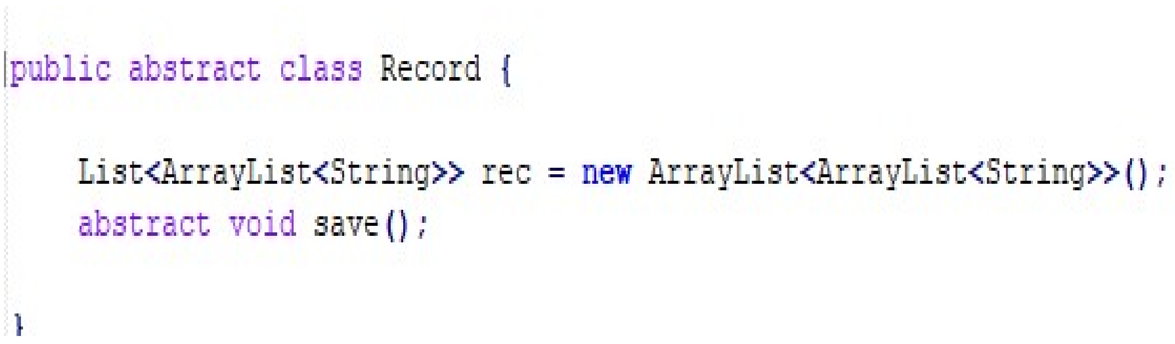
The OCP rule states that a module should be able to be extended having an additional functionality whereby the behaviour should not have an impact such as no modifications is to be deployed. Functions or base class should not be impacted with the details emanating from the subclass. A function which checks object types is a violation of OCP as when a new object is created, the function has to be modified in order to accommodate the new type. So as to maintain OCP, abstraction is well suited in this case and could be achieved by polymorphism or templates.

For example:

The library comprises of data to be stored, and this could be achieved by implementing a list whereby all the records derived can be stored into the list and to implement this approach. A class called Record is created so as the class such as Member for example could capture the credentials of the member into that list. So as to implement this, the class Record is extended. This has been combined with SRP.

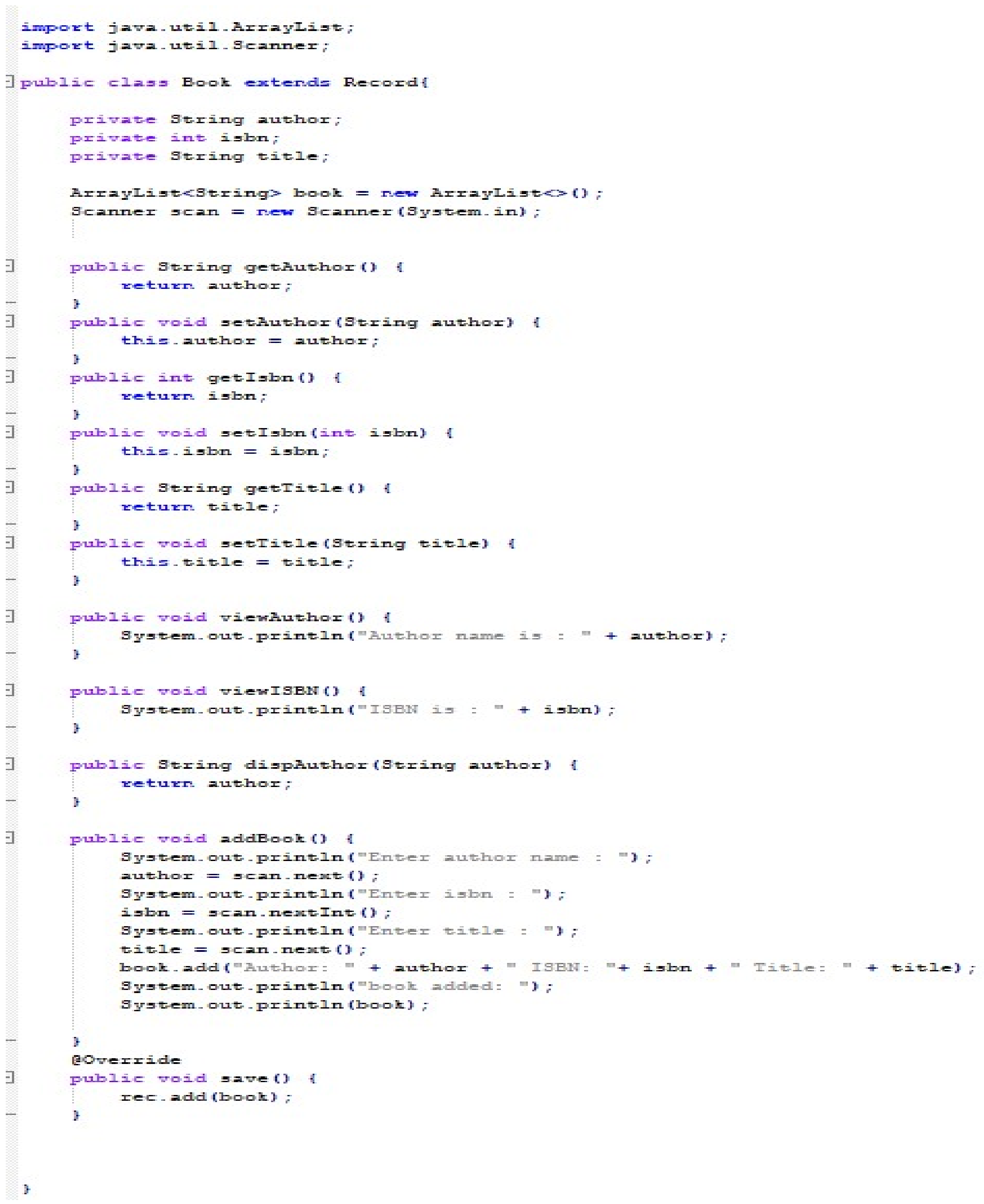


Here template pattern is implemented whereby it allows subclasses to provide implementation for the provided step. Abstraction is preferred in this approach and this promotes loose coupling.



```java
public abstract class Record {

    List<ArrayList<String>> rec = new ArrayList<ArrayList<String>>();
    abstract void save();

}
```

The class Record extends its functionality to its subclasses.

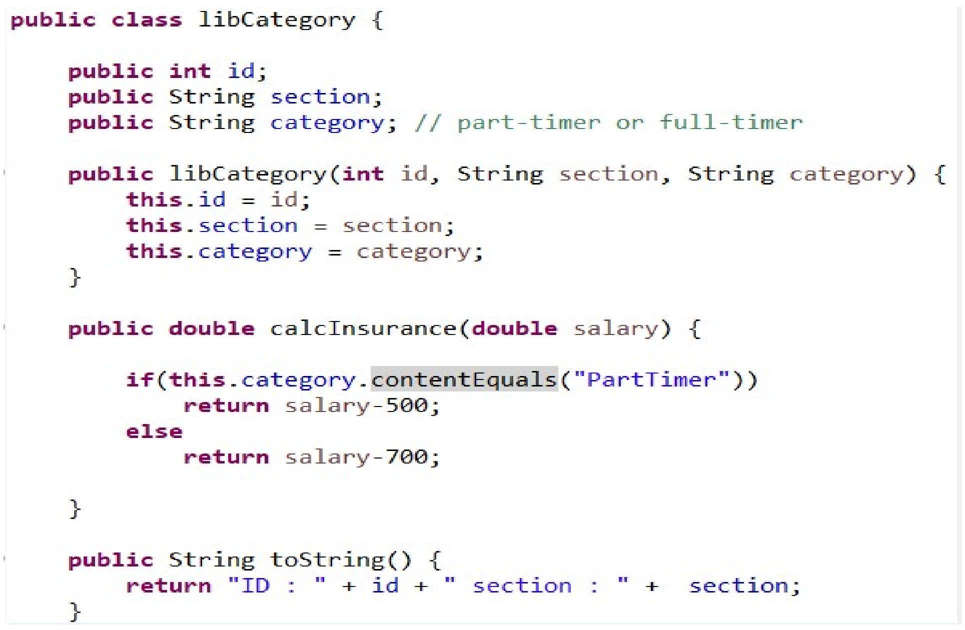


```java
import java.util.ArrayList;
import java.util.Scanner;

public class Book extends Record{

    private String author;
    private int isbn;
    private String title;

    ArrayList<String> book = new ArrayList<>();
    Scanner scan = new Scanner(System.in);

    public String getAuthor() {
        return author;
    }
    public void setAuthor(String author) {
        this.author = author;
    }
    public int getIsbn() {
        return isbn;
    }
    public void setIsbn(int isbn) {
        this.isbn = isbn;
    }
    public String getTitle() {
        return title;
    }
    public void setTitle(String title) {
        this.title = title;
    }

    public void viewAuthor() {
        System.out.println("Author name is : " + author);
    }

    public void viewISBN() {
        System.out.println("ISBN is : " + isbn);
    }

    public String dispAuthor(String author) {
        return author;
    }

    public void addBook() {
        System.out.println("Enter author name : ");
        author = scan.next();
        System.out.println("Enter isbn : ");
        isbn = scan.nextInt();
        System.out.println("Enter title : ");
        title = scan.next();
        book.add("Author: " + author + " ISBN: "+ isbn + " Title: " + title);
        System.out.println("book added: ");
        System.out.println(book);

    }
    @Override
    public void save() {
        rec.add(book);
    }


}
```

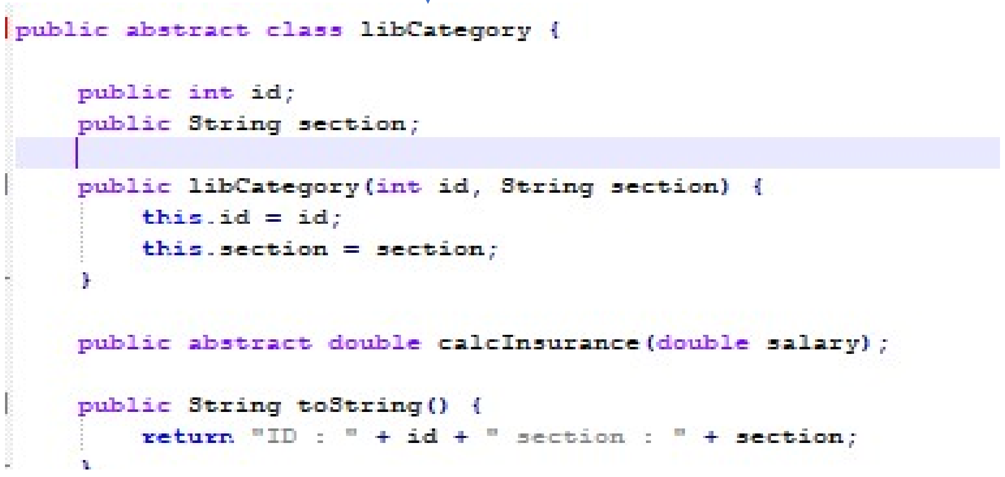
The class Book uses the function save whereby the information about a book is stored into the list.

Implementation of LSP rule:

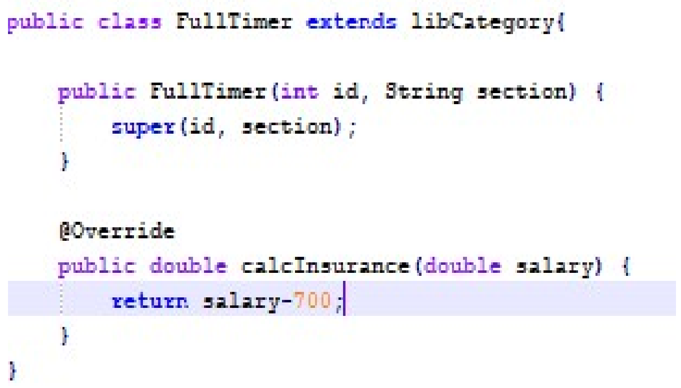
LSP implies that a subclass can be substituted for its base class without having to impact the behaviour. this approach helps in avoiding the mis usage of inheritance. Objects should be replaceable by instances of their subtypes without affecting the functions of the program. LSP ensures that abstractions are correct and helps with the reusability of the code along with the proper understanding of class hierarchies. Librarian could be perceived as part-timer or full-timer, following the OCP rule, SRP rule and the LSP rule which have been combined together generates class libCategory whereby it deducts the insurance amount from the salary depending on the type of the librarian. The classes FullTimer and PartTimer have been created as subclasses whereby the libCategory has the functionality of calculating the insurance function.



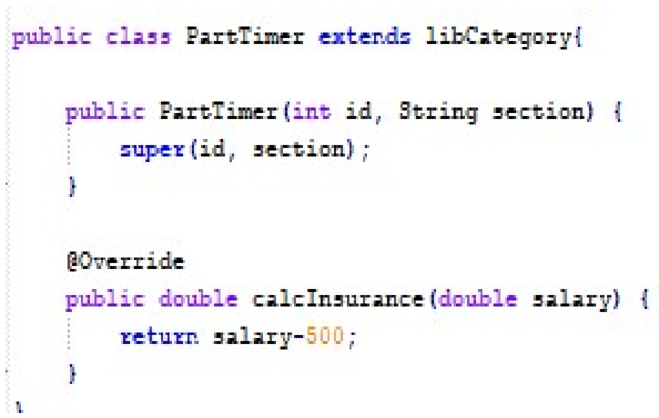
```
public class libCategory {

    public int id;
    public String section;
    public String category; // part-timer or full-timer

    public libCategory(int id, String section, String category) {
        this.id = id;
        this.section = section;
        this.category = category;
    }

    public double calcInsurance(double salary) {

        if(this.category.contentEquals("PartTimer"))
            return salary-500;
        else
            return salary-700;

    }

    public String toString() {
        return "ID : " + id + " section : " +  section;
    }
```



```
public abstract class libCategory {

    public int id;
    public String section;

    public libCategory(int id, String section) {
        this.id = id;
        this.section = section;
    }

    public abstract double calcInsurance(double salary);

    public String toString() {
        return "ID : " + id + " section : " + section;
    }
```

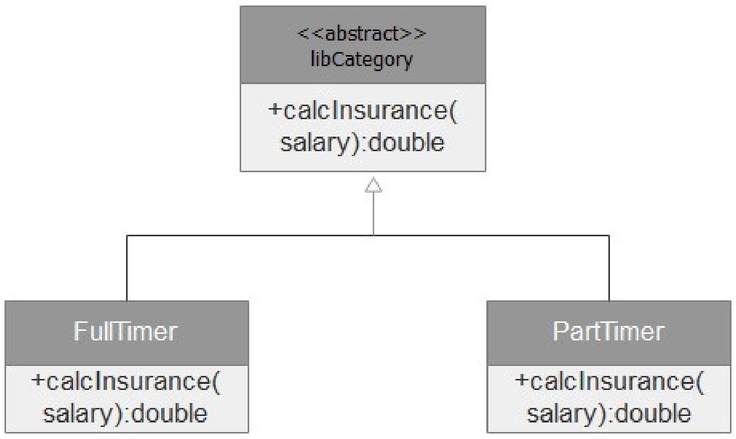


```java
public class FullTimer extends libCategory{

    public FullTimer(int id, String section) {
        super(id, section);
    }

    @Override
    public double calcInsurance(double salary) {
        return salary-700;
    }
}
```



```java
public class PartTimer extends libCategory{

    public PartTimer(int id, String section) {
        super(id, section);
    }

    @Override
    public double calcInsurance(double salary) {
        return salary-500;
    }
}
```

The class derived FullTimer and PartTimer are completely subtituable for their base type libCategory.



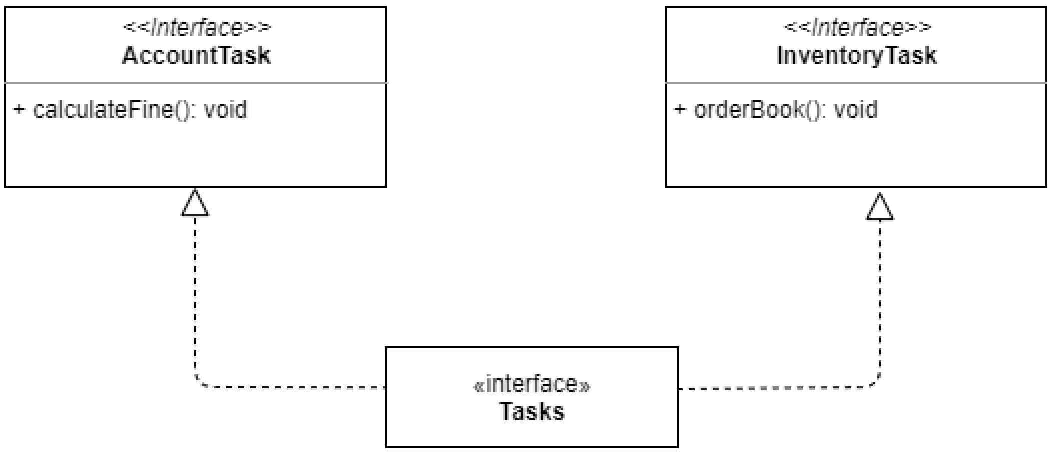
<<abstract>>
libCategory
+calcInsurance(salary):double
FullTimer
+calcInsurance(salary):double
PartTimer
+calcInsurance(salary):double

<u>Implementation of ISP rule:</u>

Interface Segregation Principle (ISP), part of the SOLID design principle, is about business logic. That is a client should never be forced to implement an interface that it doesn't use or shouldn't be forced to depend on methods they do not use. Interfaces should be specific to the needs of the object which must use the interface.

In the Librarian class from the SRP, the class implemented the interface Tasks which consisted of 2 methods namely, calculateFine() and orderBook(). These methods are not related to each other.

If we consider a librarian in the accounting department, he will only use the calculatingFine() method and will not consider the orderBook() method. Thus, the Tasks interface has been separated into AccountTask and InventoryTask respectively. The AccountTask will consider only the calculateFine() method whereas the InventoryTask will consider the orderBook() method only.



Implementation of DIP rule:

In the Dependency Inversion Principle (DIP) it states that:

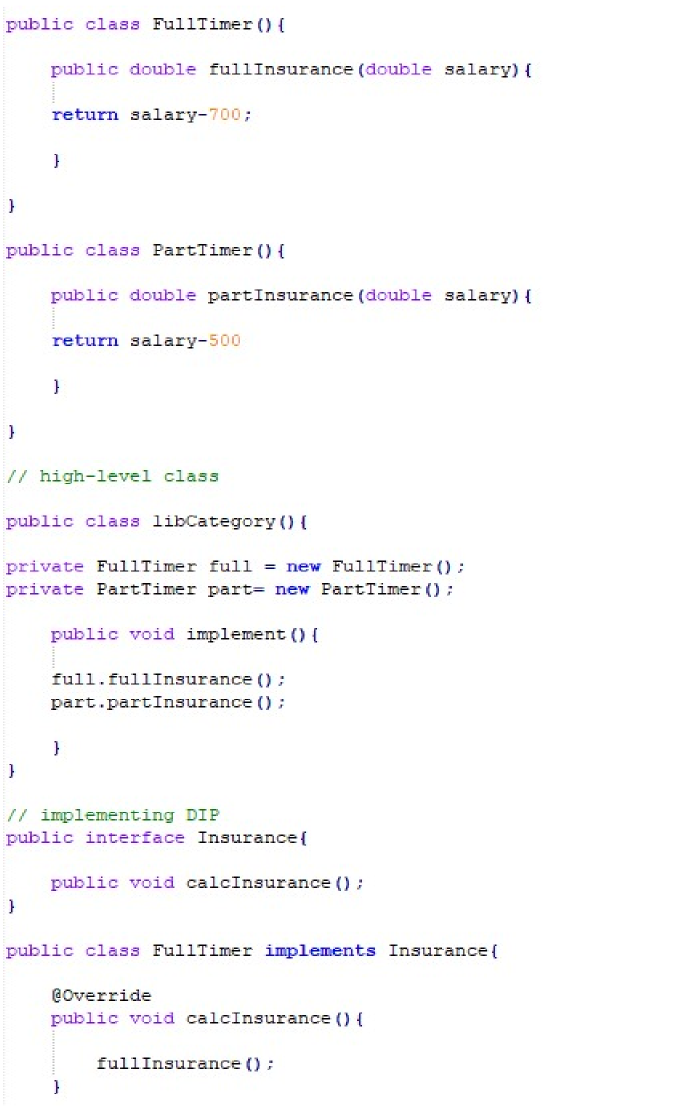
1. High-level modules should not depend on low-level modules. Both should depend on abstraction.

2. Abstraction should not depend on details. Details should depend on abstraction.

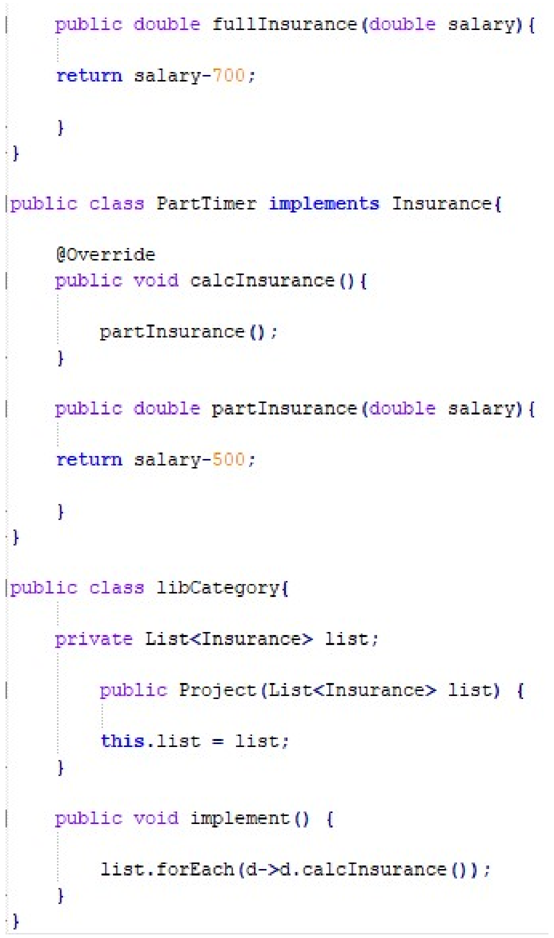
In the context of the libCategory class which is the high-level class, it can be observed that it depends on low-level classes such as FullTimer and PartTimer as well. Moreover, methods fullInsurance and partInsurance are bounding to the corresponding classes thus, violating the 2nd rule.

An Insurance interface has been created and an abstraction was introduced in class FullTimer and PartTimer.

To tackle the 1st rule violated, the libCategory has been refactored so that it does not depend on FullTimer and PartTimer.



```
public class FullTimer(){

    public double fullInsurance(double salary){

    return salary-700;

    }

}

public class PartTimer(){

    public double partInsurance(double salary){

    return salary-500

    }

}

// high-level class

public class libCategory(){

private FullTimer full = new FullTimer();
private PartTimer part= new PartTimer();

    public void implement(){

    full.fullInsurance();
    part.partInsurance();

    }

}

// implementing DIP
public interface Insurance{

    public void calcInsurance();

}

public class FullTimer implements Insurance{

    @Override
    public void calcInsurance(){

        fullInsurance();
    }
```



```
    public double fullInsurance(double salary){

    return salary-700;

    }
}

public class PartTimer implements Insurance{

    @Override
    public void calcInsurance(){

        partInsurance();
    }

    public double partInsurance(double salary){

    return salary-500;

    }
}

public class libCategory{

    private List<Insurance> list;

        public Project(List<Insurance> list) {

        this.list = list;
    }

    public void implement() {

        list.forEach(d->d.calcInsurance());
    }
}
```

Metrics

The Most Relevant Metrics used in agile:

Abstractness

The ratio of the number of abstract classes (and interfaces) in the analyzed package to the total number of classes in the analyzed package. The range for this metric is 0 to 1, with A=0 indicating a completely concrete package and A=1 indicating a completely abstract package.

Afferent coupling

The number of classes in other packages that depend upon classes within the package is an indicator of the package's responsibility. Afferent couplings signal inward.

Distance

The perpendicular distance of a package from the idealized line A + I = 1. D is calculated as D = | A + I - 1 |. This metric is an indicator of the package's balance between abstractness and stability. A package squarely on the main sequence is optimally balanced with respect to its abstractness and stability. Ideal packages are either completely abstract and stable (I=0, A=1) or completely concrete and unstable (I=1, A=0). The range for this metric is 0 to 1, with D=0 indicating a package that is coincident with the main sequence and D=1 indicating a package that is as far from the main sequence as possible.

Efferent Coupling

The number of classes in other packages that the classes in a package depend upon is an indicator of the package's dependence on externalities. Efferent couplings signal outward.

Number of Classes in package

This metric is related to the OCP. The value of this metric will give us the number of concrete and abstract classes and interfaces in the package and is indicative of the extensibility of the package. The higher the extensibility, the more it is in line with the OCP/LSP. This metric must be used together with the number of interfaces in the package.

Number of interfaces in package

This metric will help to ensure the OCP. This metric must be used together with the number of classes in the package.

Loose class coupling

This metric is useful to know the degree to which the DIP has been applied since modules should interact with another module through simple and stable interfaces.

Lack of Cohesion Methods 1

The SRP principle states that a module should have a single responsibility instead of being responsible for different functionalities. The LOCM1 will indicate to which extent this has been respected.

Lack of Cohesion Methods 3

Similar to the LOCM1, this metric will also indicate the degree of cohesion in the system. This may indicate whether the SRP has been respected or not.

Lack of Cohesion Methods 4

LOCM4 also indicates whether a class should be split into smaller classes or not, which means that it indicates the degree to which SRP has been respected.

Number of Children

This metric will allow the testers to check the degree to which LSP has been respected. The more children that we have, the more testing should be done to ensure that a subclass can act as a substitute for its base class.

Tight Class Coupling

The tighter the coupling between the classes, the more the classes are dependent on one another. This means that it will indicate whether DIP has been respected since the classes should not be dependent on other concrete classes and should rather depend on abstractions

Number of parameters

This metric is directly linked to the maintainability of the system since the simplification of a group of parameters allows the formation of rich abstractions that encapsulate a coherent set of operations. Since Agile methodologies encourage maintainability, this metric is considered useful. This can be used together with the Number of methods since a ratio can be computed with these

Number of Methods

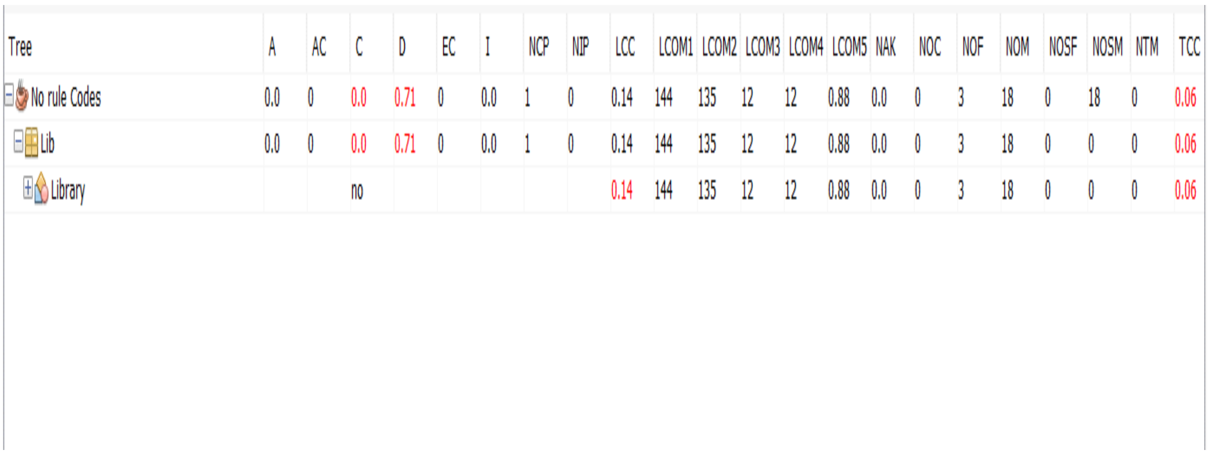
As explained above, the ratio computed with these 2 metrics can be used to check if the code is maintainable and simple. This can be used to ensure the formation of rich abstractions.

Number of Assertions per KLOC

Metric that returns the total number of assertions per KLOC.

<u>Metrics results:</u>

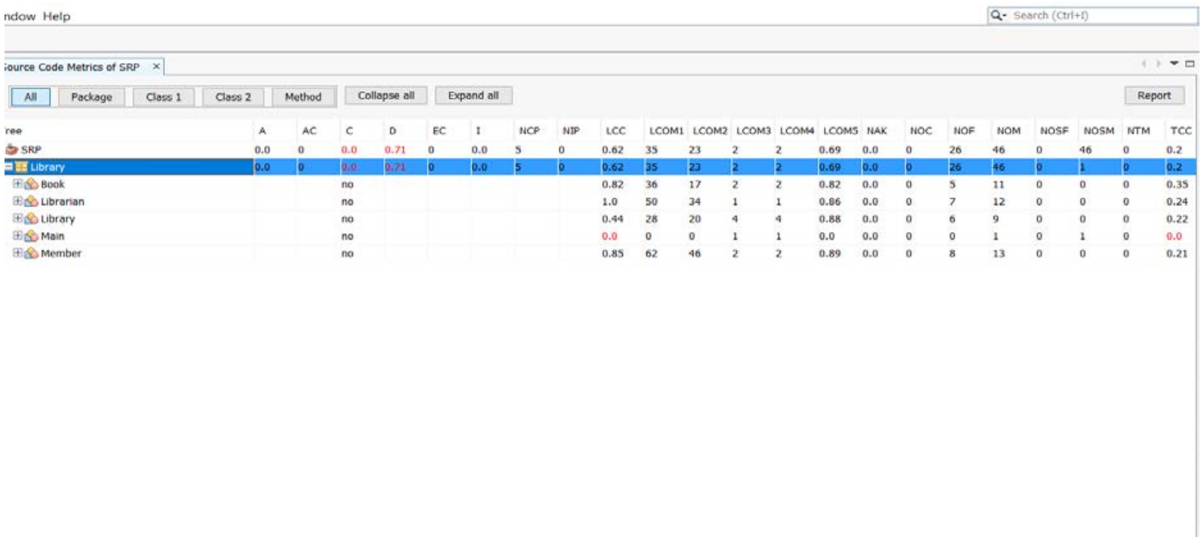
Metrics for No Rule Code



Tree	A	AC	C	D	EC	I	NCP	NIP	LCC	LCOM1	LCOM2	LCOM3	LCOM4	LCOM5	NAK	NOC	NOF	NOM	NOSF	NOSM	NTM	TCC
No rule Codes	0.0	0	0.0	0.71	0	0.0	1	0	0.14	144	135	12	12	0.88	0.0	0	3	18	0	18	0	0.06
Lib	0.0	0	0.0	0.71	0	0.0	1	0	0.14	144	135	12	12	0.88	0.0	0	3	18	0	0	0	0.06
Library			no						0.14	144	135	12	12	0.88	0.0	0	3	18	0	0	0	0.06

In the above diagram we can see that no rule is being applied to the codes and we can see that the cohesion is very low.

Metrics for SRP



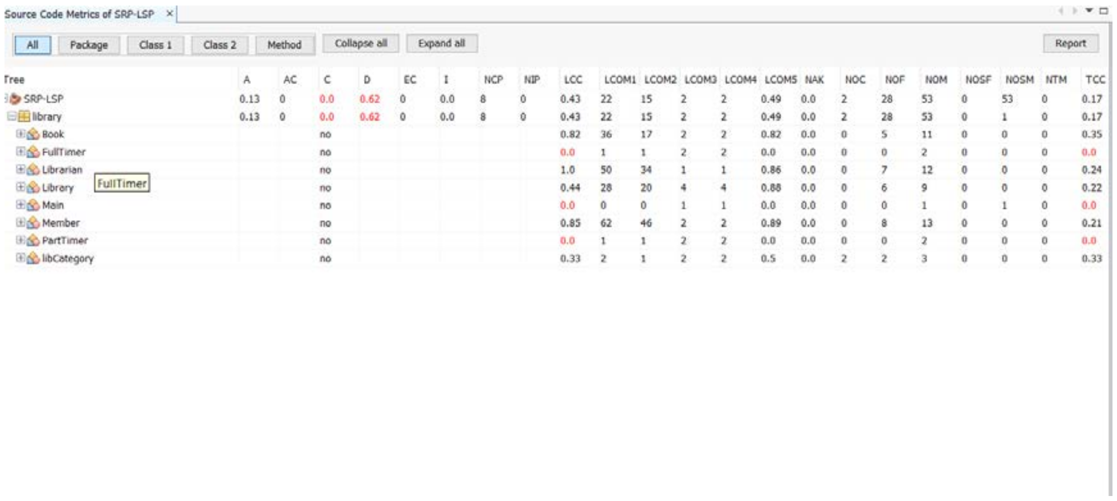
Tree	A	AC	C	D	EC	I	NCP	NIP	LCC	LCOM1	LCOM2	LCOM3	LCOM4	LCOM5	NAK	NOC	NOF	NOM	NOSF	NOSM	NTM	TCC
SRP	0.0	0	0.0	0.71	0	0.0	5	0	0.62	35	23	2	2	0.69	0.0	0	26	46	0	46	0	0.2
Library	0.0	0	0.0	0.71	0	0.0	5	0	0.62	35	23	2	2	0.69	0.0	0	26	46	0	1	0	0.2
Book			no						0.82	36	17	2	2	0.82	0.0	0	5	11	0	0	0	0.35
Librarian			no						1.0	50	34	1	1	0.86	0.0	0	7	12	0	0	0	0.24
Library			no						0.44	28	20	4	4	0.88	0.0	0	6	9	0	0	0	0.22
Main			no						0.0	0	0	1	1	0.0	0.0	0	0	1	0	1	0	0.0
Member			no						0.85	62	46	2	2	0.89	0.0	0	8	13	0	0	0	0.21

The single Responsibility principle states that one class does only one functionality. For instance, different class was implemented and each class has different methods to keep track of the data related to the class.

For e.g.: Class books have auhtor, ibsn and title.

Upon splitting the classes, we can see that the LCOM have decreases and the TTC have increases which reduces the level of dependency.

Metrics for SRP-LSP

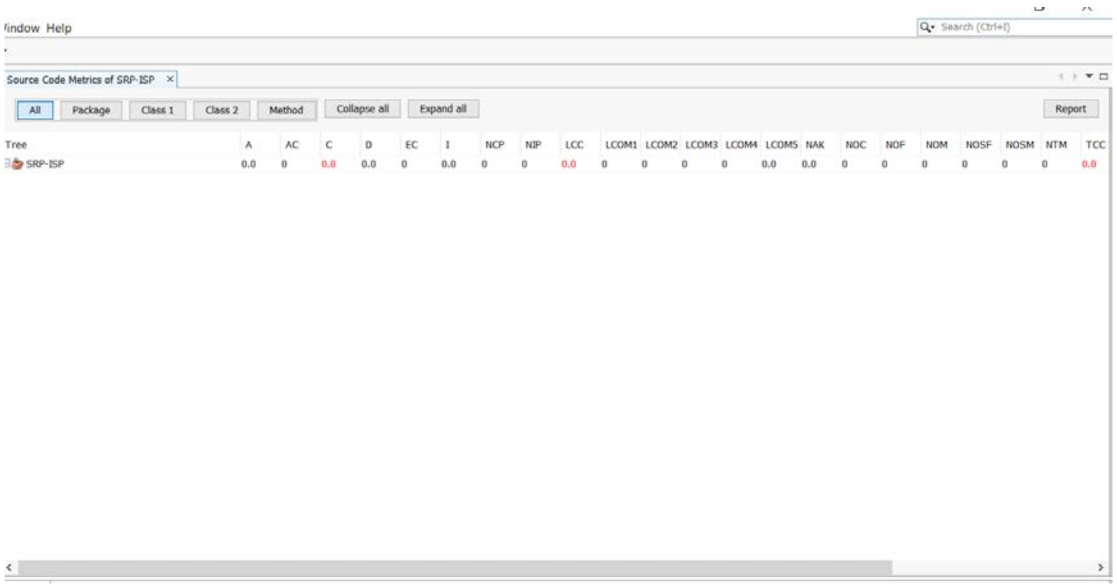

Source Code Metrics of SRP-LSP

Tree	A	AC	C	D	EC	I	NCP	NIP	LCC	LCOM1	LCOM2	LCOM3	LCOM4	LCOM5	NAK	NOC	NOF	NOM	NOSF	NOSM	NTM	TCC
SRP-LSP	0.13	0	0.0	0.62	0	0.0	8	0	0.43	22	15	2	2	0.49	0.0	2	28	53	0	53	0	0.17
library	0.13	0	0.0	0.62	0	0.0	8	0	0.43	22	15	2	2	0.49	0.0	2	28	53	0	1	0	0.17
Book			no						0.82	36	17	2	2	0.82	0.0	0	5	11	0	0	0	0.35
FullTimer			no						0.0	1	1	2	2	0.0	0.0	0	0	2	0	0	0	0.0
Librarian			no						1.0	50	34	1	1	0.86	0.0	0	7	12	0	0	0	0.24
Library			no						0.44	28	20	4	4	0.88	0.0	0	6	9	0	0	0	0.22
Main			no						0.0	0	0	1	1	0.0	0.0	0	0	1	0	1	0	0.0
Member			no						0.85	62	46	2	2	0.89	0.0	0	8	13	0	0	0	0.21
PartTimer			no						0.0	1	1	2	2	0.0	0.0	0	0	2	0	0	0	0.0
libCategory			no						0.33	2	1	2	2	0.5	0.0	2	2	3	0	0	0	0.33

The Liskov Substitution Principle, often considered as an extension of the OCP states that, if S is a subtype of T, then objects of type T in a program may be replaced with objects of type S without altering any of the desirable properties of that program

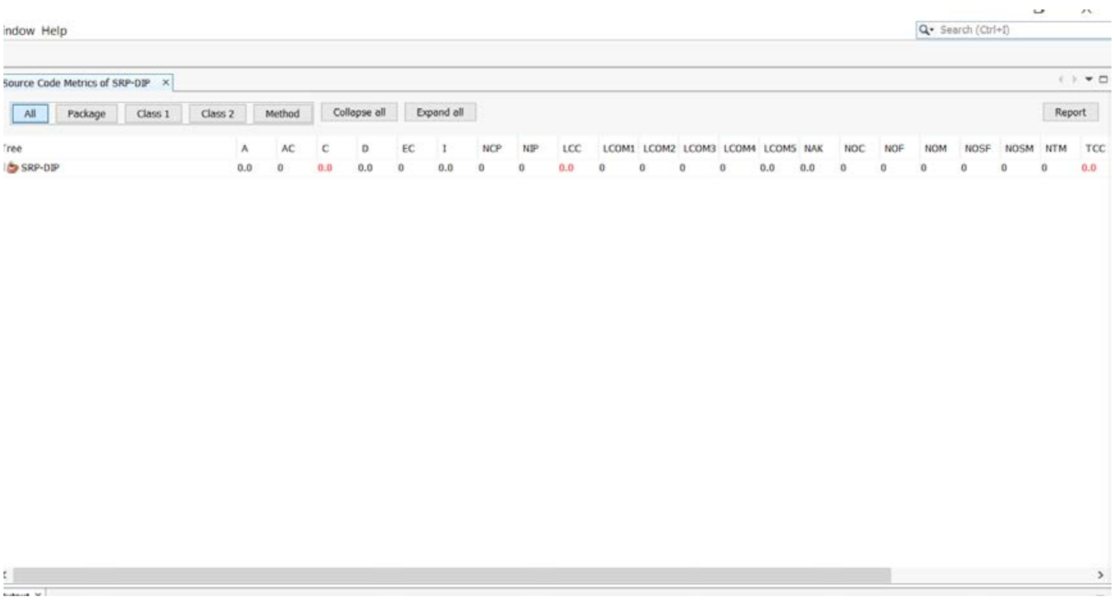
The figure is pretty the same but there is lower abstractness when the LSP principle is implemented.

Metrics for SRP-ISP


Source Code Metrics of SRP-ISP

Tree	A	AC	C	D	EC	I	NCP	NIP	LCC	LCOM1	LCOM2	LCOM3	LCOM4	LCOM5	NAK	NOC	NOF	NOM	NOSF	NOSM	NTM	TCC
SRP-ISP	0.0	0	0.0	0.0	0	0.0	0	0	0.0	0	0	0	0	0.0	0.0	0	0	0	0	0	0	0.0

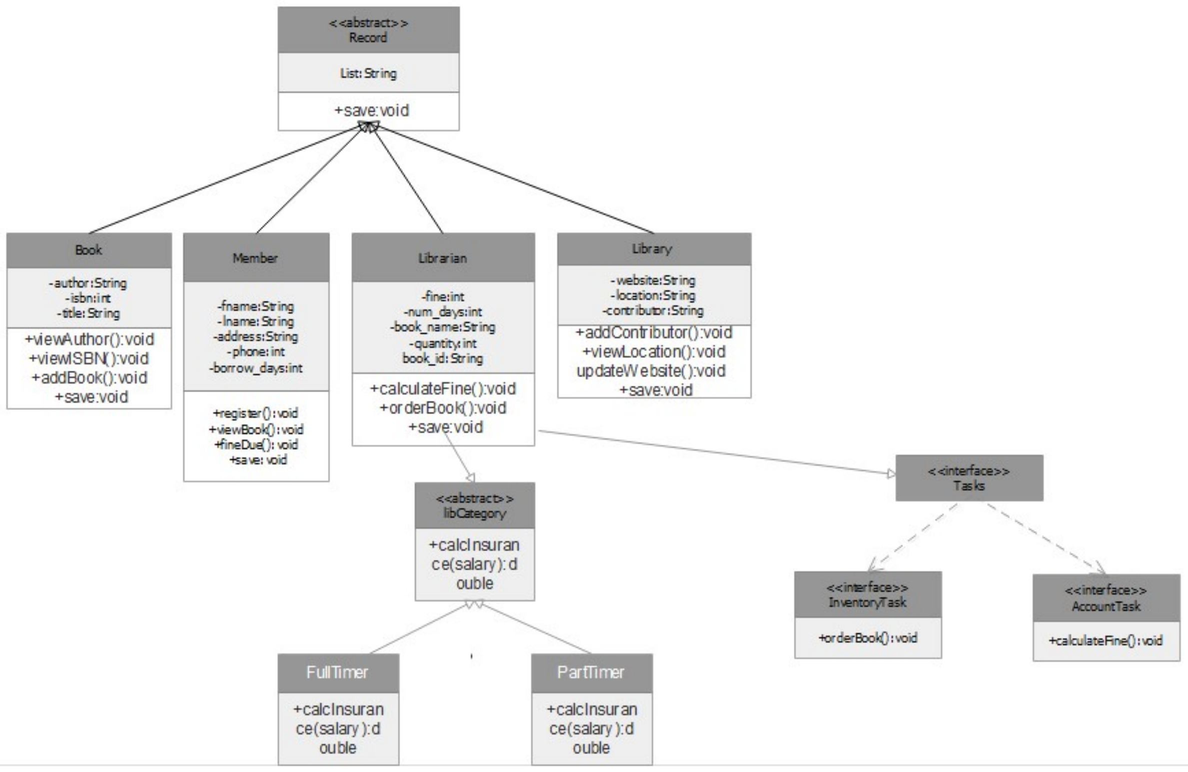
Metrics for SRP-DIP



Tree	A	AC	C	D	EC	I	NCP	NIP	LCC	LCOM1	LCOM2	LCOM3	LCOM4	LCOM5	NAK	NOC	NOF	NOM	NOSF	NOSM	NTM	TCC
SRP-DIP	0.0	0	0.0	0.0	0	0.0	0	0	0.0	0	0	0	0	0.0	0.0	0	0	0	0	0	0	0.0

Both the ISP and the DIP have the same result, all are zero. This is because in both cases, the codes are shorts and clear so there is neither coupling nor cohesion and does not to split more. They are just an interface of other classes.

Rules implemented:



<<abstract>> Record
List: String
+save:void
Book
-author:String
-isbn:int
-title: String
+viewAuthor():void
+viewISBN():void
+addBook():void
+save:void
Member
-fname:String
-lname: String
-address:String
-phone: int
-borrow_days:int
+register():void
+viewBook():void
+fineDue(): void
+save: void
Librarian
-fine:int
-num_days:int
-book_name:String
-quantity:int
book_id: String
+calculateFine():void
+orderBook():void
+save:void
Library
-website:String
-location:String
-contributor:String
+addContributor():void
+viewLocation():void
updateWebsite():void
+save:void
<<interface>> Tasks
<<abstract>> libCategory
+calcInsurance(salary):double
<<interface>> InventoryTask
+orderBook():void
<<interface>> AccountTask
+calculateFine():void
FullTimer
+calcInsurance(salary):double
PartTimer
+calcInsurance(salary):double