REPORT

ASSIGNMENT – 2

Name: FAIZA AZIZ UMATIYA

Banner Id: B00899642

Course: CSCI – 5308 - Advanced Topics in Software Development

3 SOLID Principles with their Violation and Adherence is explained Below:

1. **Solid Responsibility Principle:**
   - A class should solely have one responsibility over a single part and should not be filled with excessive functionality. A class should have only one reason to change.
   - The class should have one responsibility but a responsibility often consists of methods to implement in a class.
   - Let's look at the below example:
   - The example is about Published paper that the student wants to buy for their research.

   Example of Violation:

   - This is the simple PublishedPaper class with some fields.

```java
public class PublishedPaper
{
    3 usages
    String paperTitle;
    1 usage
    String authorName;
    1 usage
    int yearOfPublish;
    4 usages
    int cost;
    1 usage
    int isbnNumber;
    //constructor with a parameter
    1 usage
    public PublishedPaper(String paperTitle, String authorName, int yearOfPublish, int cost, int isbnNumber)
    {
        //this keyword refers to current object in the constructor or method
        this.paperTitle = paperTitle;
        this.authorName = authorName;
        this.yearOfPublish = yearOfPublish;
        this.cost =  cost;
        this.isbnNumber = isbnNumber;
    }
}
```

- Now let's create the invoice class that will contain the logic of the invoice and calculating the total cost of the publication paper.

```java
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;

3 usages
public class Invoice {
    6 usages
    private PublishedPaper publishedPaper;
    3 usages
    private int quantity;
    3 usages
    private double discountRate;
    3 usages
    private double taxRate;
    3 usages
    private double totalCost;

    1 usage
    public void invoice(PublishedPaper publishedPaper, int quantity, double discountRate, double taxRate) {
        this.publishedPaper = publishedPaper;
        this.quantity = quantity;
        this.discountRate = discountRate;
        this.taxRate = taxRate;
        this.totalCost = this.calculateTotalCost();
    }
```

```java
    2 usages
    public double calculateTotalCost() {
        double price = (publishedPaper.cost - (publishedPaper.cost * discountRate/100))* this.quantity;
        double priceWithTaxes = price * (1 + (taxRate/100));
        return priceWithTaxes;
    }
```

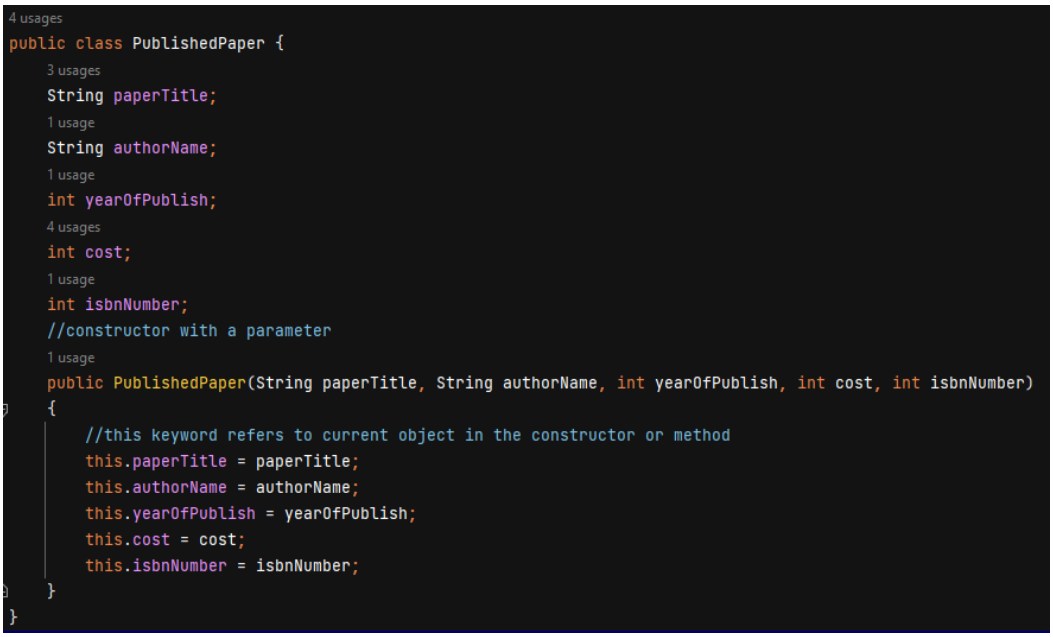
```java
    1 usage
    public void printInvoice() {
        System.out.println(quantity + " x " + publishedPaper.paperTitle + " " + "costs for $"+publishedPaper.cost);
        System.out.println("Discount Rate: " + discountRate+"%");
        System.out.println("TaxRate: " + taxRate+"%");
        System.out.println("Total Cost: $" + totalCost);
    }
```

```java
    1 usage
    public void saveToFile(String filename, Invoice invoice) {
        try
        {
            File file = new File(filename);
            FileWriter fileWriter = new FileWriter(file);
            PrintWriter printWriter = new PrintWriter(fileWriter);
            printWriter.println("Published Paper: "+ publishedPaper.paperTitle+"\n"+"Total cost is: $" +invoice.totalCost);
            printWriter.close();
        }
        catch (IOException e) {
            System.out.println("Error in saveToFile method");
        }
    }
}
```

- If we notice, the invoice class has 3 responsibilities:
    1. printInvoice – To print the invoice.
    2. saveToFile – To save and write the invoice to a file.
- Because the class has 2 responsibilities, it has 2 reasons to change:
    1. The printInvoice method violates SRP. If we want to change the printing format, we would need to change the class which leads to breaking the code. So we should not mix the business logic in the same class.
    2. The another method that violates SRP is the saveToFile method. In my case its writing the invoice to a file. But there is a possibility it could be saving to a database or other persistence related things. So its important to keep persistence logic and business logic separate.
- Because the class has 2 responsibilities, we have to fix this like shown below.

Example of Adherence:

- We have this simple Published paper class and Invoice class.



```
4 usages
public class PublishedPaper {
    3 usages
    String paperTitle;
    1 usage
    String authorName;
    1 usage
    int yearOfPublish;
    4 usages
    int cost;
    1 usage
    int isbnNumber;
    //constructor with a parameter
    1 usage
    public PublishedPaper(String paperTitle, String authorName, int yearOfPublish, int cost, int isbnNumber)
    {
        //this keyword refers to current object in the constructor or method
        this.paperTitle = paperTitle;
        this.authorName = authorName;
        this.yearOfPublish = yearOfPublish;
        this.cost = cost;
        this.isbnNumber = isbnNumber;
    }
}
```

```java
public class Invoice {
    6 usages
    PublishedPaper publishedPaper;
    3 usages
    int quantity;
    3 usages
    double discountRate;
    3 usages
    double taxRate;
    3 usages
    double totalCost;

    1 usage
    public void invoice(PublishedPaper publishedPaper, int quantity, double discountRate, double taxRate) {
        this.publishedPaper = publishedPaper;
        this.quantity = quantity;
        this.discountRate = discountRate;
        this.taxRate = taxRate;
        this.totalCost = this.calculateTotalCost();
    }
    2 usages
    public double calculateTotalCost() {
        double price = (publishedPaper.cost - (publishedPaper.cost * discountRate/100))* this.quantity;
        double priceWithTaxes = price * (1 + (taxRate/100));
        return priceWithTaxes;
    }
}
```

- I created 2 classes InvoicePrinter and InvoicePersistance. Now these classes obeys the Single Responsibility Principle and every class is responsible for one particular aspects of that class. For instance, InvoicePrinter is only responsible for printing the invoice and InvoicePersistance is only responsible for writing the invoice to the file.

```java
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
2 usages
public class InvoicePersistenceLayer {
    1 usage
    Invoice invoice;

    1 usage
    public InvoicePersistenceLayer(Invoice invoice) { this.invoice = invoice; }

    1 usage
    public void saveToFile(String filename, Invoice invoice) {
        //create a file to save and write the invoice
        try {
            File file = new File(filename);
            FileWriter fileWriter = new FileWriter(file);
            PrintWriter printWriter = new PrintWriter(fileWriter);
            printWriter.println("Published Paper: "+ invoice.publishedPaper.paperTitle+"\n"+"Total cost: $" +invoice.totalCost);
            printWriter.close();
        } catch (IOException e) {
            System.out.println("Error in saveToFile method");
        }
    }
}
```

```
2 usages
public class InvoicePrinter {
    7 usages
    private Invoice invoice;

    1 usage
    public InvoicePrinter(Invoice invoice) {
        this.invoice = invoice;
    }

    1 usage
    public void printInvoice() {
        System.out.println(invoice.quantity + " x " + invoice.publishedPaper.paperTitle + "  $" + invoice.publishedPaper.cost);
        System.out.println("Discount Rate: " + invoice.discountRate+"%");
        System.out.println("Tax Rate: " + invoice.taxRate+"%");
        System.out.println("Total: $" + invoice.totalCost);
    }
}
```

- As you can see, I fix the SRP violation by moving all but a single responsibility to other classes or modules.
- Now in this solution if the logic for saving the file changes it will not break the other parts of the code.

2. Interface Segregation Principle (ISP):

- Definition: “No client should be forced to depend on methods it does not use. Many client-specific interfaces are better than one general-purpose interface” [2].
- To any changes in code, we should prevent our code from dependency creation. This can be achieved by keeping interfaces small. This results in reduced coupling.
- Interface Segregation Principle helps avoiding the dependency hell. For example, module A depends on module B, module B depends on module C and module C depends on module A [2].
- Violation of ISP increases dependency hell. That is, we cannot detach a module from a cycle because of the loop. We do not want classes that implement the interface have to implement do nothing methods. If an interface is very large, it becomes more and more difficult for every class implementing the interface to have a need for every method defined in the interface.
- Let’s look examples of ISP Violation and Adherence:
- The example is about different social media apps having different features like watch videos, subscribe, like and share, watch stories etc.

Example of Violation:

- In this example there is one large interface: ISocialMedia. This interface describes the features that are available in different social media apps capable of adding posts, watching videos, doing subscriptions, watching stories and hitting like and share button.

```
3 usages  3 implementations
public interface ISocialMedia {
    1 usage  3 implementations
    void addPost(int posts);
    1 usage  3 implementations
    void watchVideos(int videos);
    1 usage  3 implementations
    void doSubscriptions(int subscribes);
    1 usage  3 implementations
    void watchStories(int stories);
    1 usage  3 implementations
    void doLikeandShare(int likes, int shares);
}
```

- One of the app named “Youtube” class meets the expectations of the interface because it supports all 5 functions.

```
public class Youtube implements ISocialMedia{
    1 usage
    @Override
    public void addPost(int posts) {
        System.out.println("Adding "+posts+" post!");
    }
    1 usage
    @Override
    public void watchVideos(int videos) {
        System.out.println("Currently binging "+videos+" videos!");
    }
    1 usage
    @Override
    public void doSubscriptions(int subscribes) {
        System.out.println("Subscribed to "+subscribes+" channel!");
    }
    1 usage
    @Override
    public void watchStories(int stories) {
        System.out.println("watching "+stories+ "short stories!");
    }
    1 usage
    @Override
    public void doLikeandShare(int likes, int shares) {
        System.out.println("Gave "+likes+" like  and "+shares+" shared!");
    }
}
```

- The “WhatsApp” class though is only capable of only adding posts, watching videos and watching stories and therefore has to implement do nothing methods for doSubscriptions() and doLikeandShare() so that it can be included in generic programming algorithms that deal with ISocialMedia objects.

```java
public class WhatsApp implements ISocialMedia{
    1 usage
    @Override
    public void addPost(int posts) {
        System.out.println("Adding "+posts+" post!");
    }
    1 usage
    @Override
    public void watchVideos(int videos) {
        System.out.println("Currently binginig "+videos+"  videos!");
    }
    1 usage
    @Override
    public void watchStories(int stories) {
        System.out.println("Watching "+stories+" short stories!");
    }
    1 usage
    @Override
    public void doSubscriptions(int subscribes) {
        // do nothing method
    }
    1 usage
    @Override
    public void doLikeandShare(int likes, int shares) {
        // do nothing method
    }
}
```

- Similarly, the app named “Facebook” has to implement do nothing methods for doSubscriptions() so that it can be included in generic programmimg algorithms that deal with ISocialMedia objects.

```java
public class Facebook implements ISocialMedia{
    1 usage
    @Override
    public void addPost(int posts) {
        System.out.println("Adding "+posts+" posts!");
    }
    1 usage
    @Override
    public void watchVideos(int videos) {
        System.out.println("Currently binging "+videos+" videos!");
    }
    1 usage
    @Override
    public void doSubscriptions(int subscribes) {
        // do nothing method
    }
    1 usage
    @Override
    public void watchStories(int stories) {
        System.out.println("watching "+stories+" short stories!");
    }
    1 usage
    @Override
    public void doLikeandShare(int likes, int shares) {
        System.out.println("Gave "+likes+" like  and "+shares+" shared!");
    }
}
```

- Therefore, the interface is violating ISP.

Example of Adherence:

- I solved the violation by breaking the interface up into smaller interfaces. This allows classes to pick and choose the functionality they are willing to commit to.
- Below are the interfaces that I break down:

```
3 usages  3 implementations
public interface ISocialMedia {
    1 usage  3 implementations
    void addPost(int posts);
    1 usage  3 implementations
    void watchVideos(int videos);
    1 usage  3 implementations
    void watchStories(int stories);
}
```

```
2 usages  2 implementations
public interface ILikeandShare {
    1 usage  2 implementations
    void doLikeandShare(int likes, int shares);
}
```

```
1 usage  1 implementation
public interface ISubscription {
    1 usage  1 implementation
    void doSubscriptions(int subscribes);
}
```

- So each class can implement the specific interface with the only useful method. Like in the below illustration, Youtube class consists of all features, so it implemented the interfaces that have useful method.
- In case of Youtube class, it supports all functions so it implemented all interfaces.

```java
public class Youtube implements ISocialMedia, ILikeandShare, ISubscription{
    1 usage
    @Override
    public void doLikeandShare(int likes, int shares) {
        System.out.println("Gave "+likes+" like  and "+shares+" shared!");
    }
    1 usage
    @Override
    public void addPost(int posts) {
        System.out.println("Adding "+posts+" post!");
    }
    1 usage
    @Override
    public void watchVideos(int videos) {
        System.out.println("Currently binging "+videos+" videos!");
    }
    1 usage
    @Override
    public void watchStories(int stories) {
        System.out.println("watching "+stories+" short stories!");
    }
    1 usage
    @Override
    public void doSubscriptions(int subscribes) {
        System.out.println("Subscribe to my channel!");
    }
}
```

- On the other hand, Facebook that only have a feature like add post, watching videos, watching stories, subscribing had implemented methods like addPost(), watchVideos(), watchStories() and doSubscriptions().
- Now it does not have to implement do nothing methods like doLikeandShare() because of the interface segregation principle.
- Because the interface is segregated, now the classes are less likely coupled which avoids dependency hell.

```
2 usages
public class Facebook implements ISocialMedia, ILikeandShare {
    1 usage
    @Override
    public void doLikeandShare(int likes, int shares) {
        System.out.println("Gave "+likes+" like  and "+shares+" shared!");
    }
    1 usage
    @Override
    public void addPost(int posts) {
        System.out.println("Adding "+posts+" posts!");
    }
    1 usage
    @Override
    public void watchVideos(int videos) {
        System.out.println("Currently binging "+videos+" videos!");
    }
    1 usage
    @Override
    public void watchStories(int stories) {
        System.out.println("watching "+stories+" short stories!");
    }
}
```

- Similarly, now WhatsApp class don’t have to implement do nothing methods by just implementing the interfaces that consists of the methods it want to use.

```
2 usages
public class WhatsApp implements ISocialMedia{
    1 usage
    @Override
    public void addPost(int posts) {
        System.out.println("Adding "+posts+" post!");
    }
    1 usage
    @Override
    public void watchVideos(int videos) {
        System.out.println("Currently binging "+videos+" videos!");
    }
    1 usage
    @Override
    public void watchStories(int stories) {
        System.out.println("watching "+stories+" short stories!");
    }
}
```

- So I make small interfaces out of one large interface which reduced the coupling between modules. i.e. now the modules can function independently.

3. Dependency Inversion Principle (DIP):

- The Dependency inversion states that our class should depend on abstractions and not concretions.
- Dependency inversion principle helps to achieve flexibility in our software by depending on the abstractions and not concrete classes and functions. One should depend on interfaces instead of classes that depend on interface. Dependency injection is like dependency inversion principle.
- The below example is about the customer doing purchase from a grocery shop with different card payments.
- To understand DIP, let's see the example:

Example of Violation:

- The below illustration is shows that the class GroceryShop has 2 methods doPurchaseDividendCredit() and doPurchaseDirectDebit(). Its up to the customer if the customer wants to do the payment using debit card or credit card.
- Also, we have one DirectDebit class and DividendCredit class that allows user to do the transactions using that particular payment method.

```
2 usages
public class DividenedCredit {
    1 usage
    public void Transaction(long amountToPay)
    {
        System.out.println("Payment using Credit Card");
    }
}
```

```
2 usages
public class DirectDebit {
    1 usage
    public void Transaction(long amountToPay)
    {
        System.out.println("Payment using Debit Card");
    }
}
```

- But here the GroceryShop class depends on concrete classes doPurchaseDividendCredit() and doPurchaseDirectDebit() which creates high coupling and makes it difficult to swap out concrete objects. This results in decreased flexibility.

```
2 usages
public class GroceryShop {
    2 usages
    private DirectDebit debitCard;
    2 usages
    private DividenedCredit creditCard;
    1 usage
    public void doPurchaseDirectDebit(long amountToPay)
    {
        debitCard = new DirectDebit();
        debitCard.Transaction(amountToPay);
        System.out.println("Amount to pay: $" +amountToPay);
    }
    1 usage
    public void doPurchaseDividendCredit(long amountToPay)
    {
        creditCard = new DividenedCredit();
        creditCard.Transaction(amountToPay);
        System.out.println("Amount to pay: $" +amountToPay);
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        GroceryShop groceryShop = new GroceryShop();
        //groceryShop.doPurchaseDirectDebit(5000);
        groceryShop.doPurchaseDividendCredit( amountToPay: 6000);
    }
}
```

- Let's see how we can solve this problem.

Example of Adherence:

- Simply by making interface IBankCard, I inject interface IBankCard in the GroceryShop() method which makes the class depend on abstraction rather than concrete objects.
- So now our code is flexible which means if there is any payment method that we need to add can be added without breaking the code.

```java
5 usages  2 implementations  Faiza Aziz Umatiya
public interface IBankCardPayment
{
    1 usage  2 implementations  Faiza Aziz Umatiya
    public void Transaction(long amountToPay);
}
```

```java
Faiza Aziz Umatiya
public class DirectDebitCard implements IBankCardPayment {
    1 usage  Faiza Aziz Umatiya
    public void Transaction(long amountToPay)
    {
        System.out.println("Payment using Debit Card");
    }

}
```

```java
1 usage  Faiza Aziz Umatiya
public class DividenedCreditCard implements IBankCardPayment {
    1 usage  Faiza Aziz Umatiya
    public void Transaction(long amountToPay)
    {
        System.out.println("Payment using Credit Card");
    }

}
```

```java
2 usages  Faiza Aziz Umatiya
public class GroceryShop {
    2 usages
    private IBankCardPayment bankCard ;
    1 usage  Faiza Aziz Umatiya
    public GroceryShop(IBankCardPayment bankCard)
    {
        this.bankCard = bankCard;
    }
    1 usage  Faiza Aziz Umatiya
    public void doPurchase(long amountToPay)
    {
        bankCard.Transaction(amountToPay);
        System.out.println("Amount to pay: $" +amountToPay);
    }
}
```

```java
Faiza Aziz Umatiya
public class Main {
    Faiza Aziz Umatiya
    public static void main(String[] args) {
        IBankCardPayment bankCard = new DividenedCreditCard();
        GroceryShop groceryShop= new GroceryShop(bankCard);
        groceryShop.doPurchase( amountToPay: 80000);
    }
}
```

- Now if suppose customer's card is not working and wants to use different card, we can just add another card that implements the interface without changing the whole code.

References:

[1] Brightspace “Single Responsibility Principle,” *Brightspace*. [Online]. Available: https://dal.brightspace.com/d2l/le/content/221748/viewContent/3006810/View. [Accessed: 19-Jun-2022].

[2] Brightspace “Interface Segregation Principle,” *Brightspace*. [Online]. Available: https://dal.brightspace.com/d2l/le/content/221748/viewContent/3006813/View. [Accessed: 18-Jun-2022].

[3] Brightspace “Dependency Inversion Principle,” *Brightspace*. [Online]. Available: https://dal.brightspace.com/d2l/le/content/221748/viewContent/3006814/View. [Accessed: 20-Jun-2022].

[4] W3schools “Java Create and Write to Files,” *w3schools*. [Online]. Available: https://www.w3schools.com/java/java_files_create.asp. [Accessed: 20-Jun-2022].

Git Link:

https://git.cs.dal.ca/courses/2022-summer/csci-5308/umatiya