SOLID PRINCIPLES

Submitted by:

Md. Shihab Shahriar
170041016
8th Semester, CSE

Islamic University of Technology
Course: CSE 4851

The SOLID Principles are five principles of Object-Oriented class design. They are a set of rules and best practices to follow while designing a class structure. These five principles help us understand the need for certain design patterns and software architecture in general. They all serve the same purpose:

"To create understandable, readable, and testable code that many developers can collaboratively work on."

The principles are, following the SOLID acronym:

- The **S**ingle Responsibility Principle
- The **O**pen-Closed Principle
- The **L**iskov Substitution Principle
- The **I**nterface Segregation Principle
- The **D**ependency Inversion Principle

The Single Responsibility Principle

The Single Responsibility Principle states that a class should do one thing and therefore it should have only a single reason to change.

Suppose, **Student** is a class having three methods namely **printDetails(), calculatePercentage(),** and **addStudent().** Hence, the Student class has three responsibilities to print the details of students, calculate percentages, and database.

```
public class Student
{
    public void printDetails();
    {
        //functionality of the method
    }
    public void calculatePercentage();
    {
        //functionality of the method
    }
    public void addStudent();
    {
        //functionality of the method
    }
}
```

The above code snippet violates the single responsibility principle. To achieve the goal of the principle, we should implement a separate class that performs a single functionality only.

```
public class Student
{
    public void addStudent();
    {
        //functionality of the method
    }
}

public class PrintStudentDetails
{
    public void printDetails();
    {
        //functionality of the method
    }
}

public class Percentage
{
    public void calculatePercentage();
    {
        //functionality of the method
    }
}
```

Hence, we have achieved the goal of the single responsibility principle by separating the functionality into three separate classes.

Open-Closed Principle

The application or module entities the methods, functions, variables, etc. The open-closed principle states that according to new requirements **the module should be open for extension but closed for modification.**

Suppose, **VehicleInfo** is a class and it has the method **vehicleNumber()** that returns the vehicle number.

```
public class VehicleInfo
{
    public double vehicleNumber(Vehicle vcl)
    {
        if (vcl instanceof Car)
        {
            return vcl.getNumber();
        if (vcl instanceof Bike)
        {
            return vcl.getNumber();
        }
    }
}
```

If we want to add another subclass named Truck, simply, we add one more if statement that violates the open-closed principle. The only way to add the subclass and achieve the goal of principle by overriding the **vehicleNumber()** method, as shown below.

```
public class VehicleInfo
{
    public double vehicleNumber()
    {
    //functionality
    }
}
public class Car extends VehicleInfo
{
    public double vehicleNumber()
    {
        return this.getValue();
    }
}
public class Car extends Truck
{
    public double vehicleNumber()
    {
        return this.getValue();
    }
}
```

Here, we can add more vehicles by making another subclass extending from the vehicle class. the approach would not affect the existing application.

Liskov Substitution Principle

The Liskov Substitution Principle (LSP) was introduced by **Barbara Liskov**. It applies to inheritance in such a way that the **derived classes must be completely substitutable for their base classes**. In other words, if class A is a subtype of class B, then we should be able to replace B with A without interrupting the behavior of the program.

An example for understanding the principle is shown below

```
public class Student
{
    private double height;
    private double weight;
    public void setHeight(double h)
    {
        height = h;
    }
    public void setWeight(double w)
    {
        weight= w;
    }
}
public class StudentBMI extends Student
{
    public void setHeight(double h)
    {
        super.setHeight(h);
        super.setWeight(w);
    }
    public void setWeight(double h)
    {
        super.setHeight(h);
        super.setWeight(w);
    }
}
```

The above classes violated the Liskov substitution principle because the StudentBMI class has extra constraints i.e. height and weight that must be the same. Therefore, the Student class (base class) cannot be replaced by StudentBMI class (derived class).

Interface Segregation Principle

The principle states that the larger interfaces split into smaller ones. Because the implementation classes use only the methods that are required.

Suppose, we have created an interface named **Conversion** having three methods **intToDouble(), intToChar(),** and **charToString()**.

```
public interface Conversion
{
    public void intToDouble();
    public void intToChar();
    public void charToString();
}
```

The above interface has three methods. If we want to use only a method intToChar(), we have no choice to implement the single method. To overcome the problem, the principle allows us to split the interface into three separate ones.

```
public interface ConvertIntToDouble
{
    public void intToDouble();
}
public interface ConvertIntToChar
{
    public void intToChar();
}
public interface ConvertCharToString
{
    public void charToString();
}
```

Now we can use only the method that is required. Suppose, we want to convert the integer to double and character to string then, we will use only the methods **intToDouble()** and **charToString().**

```
public class DataTypeConversion implements ConvertIntToDouble,
ConvertCharToString
{
    public void intToDouble()
    {
        //conversion logic
    }
    public void charToString()
    {
        //conversion logic
    }
}
```

Dependency Inversion Principle

The principle states that we must use abstraction (abstract classes and interfaces) instead of concrete implementations. High-level modules should not depend on the low-level module but both should depend on the abstraction.

An example is shown below to understand the principle

```
public class App {
    private ExcelExporter exporter = new ExcelExporter();

    public File export(){
        return exporter.toFile();
    }
}
```

Here, the main App uses directly the ExcelExporter class. we have that the high-level module (App class) depends on low-level component (ExcelExporter class). The problem is that if we need to produce a different File (e.g. a PDF file instead of an Excel) we have to modify the code.

A better design adopts the Dependency Inversion as follows:

```
public interface Exporter {
    public File toFile();
}

public class ExcelExporter implements Exporter {
    public File toFile(){
        //Arrange contents in Excel File
    }
}

public class PDFExporter implements Exporter {
    public File toFile(){
        //Arrange contents in PDF File
    }
}

public class App{
    private Exporter exporter;

    public App(Exporter exporter){
        this.exporter = exporter;
    }

    public File export(){
        return exporter.toFile();
    }
}
```

The specific exporter implementation can be provided through the constructor as follows:

```
App app = new App(new ExcelExporter());
```

Or,

```
App app = new App(new PDFExporter());
```