**SOLID Principle**

SOLID principles are a set of design principles that help make code more adaptable, scalable, and easier to maintain.

S: Single Responsibility Principle

O: Open for extension and Closed for Modification

L: Liskov Substitution Principle

I: Interface Segment Principle

D: Dependency Inversion Principle

**Advantages of following these Principles**

1. Avoid Duplicate Code
2. Easy to Maintain
3. Easy to Understand
4. Flexible Software
5. Reduce Complexity
6. **S: Single Responsibility Principle**

Single Responsibility Principle (SRP): A class should have a single, well-defined responsibility or job. This makes code more modular and easier to maintain and extend.

package com.automation.solidPrinciple.SRP;  
  
public class Marker {  
  
 String name;  
 String color;  
 int year;  
 int price;  
  
 public Marker(String name, String color, int year, int price) {  
 this.name = name;  
 this.color = color;  
 this.year = year;  
 this.price = price;  
 }  
}

package com.automation.solidPrinciple.SRP;  
  
public class Invoice {  
  
 private Marker marker;  
 private int quantity;  
  
 public Invoice(Marker marker, int quantity) {  
 this.marker = marker;  
 this.quantity = quantity;  
 }  
  
 public int calculateTotal() {  
 return marker.price\*this.quantity;  
 }  
  
// public void printInvoice(){  
//  
// }  
//  
// public void saveToDB(){  
//  
// }  
}

package com.automation.solidPrinciple.SRP;  
  
public class InvoicePrinter {  
  
 private Marker marker;  
  
 public InvoicePrinter(Marker marker) {  
 this.marker = marker;  
 }  
  
 public void printInvoice(){  
 System.*out*.println("print the invoice");  
 }  
}

package com.automation.solidPrinciple.SRP;  
  
public class InvoiceSave {  
  
 private Marker marker;  
  
 public InvoiceSave(Marker marker) {  
 this.marker = marker;  
 }  
  
 public void saveToDB(){  
  
 System.*out*.println("save invoice");  
  
 }  
}

1. **Open for Extension and Close for Modification**

A class should be open for extension but closed for modification. This helps make code more maintainable and flexible.

In this Principle, let us suppose if we need to add new method in the class which is already live and running in the production, then we should add new method in the existing class like below example

package com.automation.solidPrinciple.OCP;  
  
public class OCP {  
  
 private Invoice invoice;  
  
 public OCP(Invoice invoice) {  
 this.invoice = invoice;  
 }  
  
 public void save(Invoice invoice){  
// save to DB  
 }  
  
// public void saveToFile(Invoice invoice){  
//// save to file  
// }  
  
}

Instead of using this approach we should create an interface and create new classes to implement it

package com.automation.solidPrinciple.OCP;  
  
public interface OcpInterface {  
  
 public void save(Invoice invoice);  
}

package com.automation.solidPrinciple.OCP;  
  
public class SaveToDB implements OcpInterface{  
 @Override  
 public void save(Invoice invoice) {  
 //save to DB  
 }  
}

package com.automation.solidPrinciple.OCP;  
  
public class SaveToFile implements OcpInterface{  
 @Override  
 public void save(Invoice invoice) {  
   
 //Save to File  
 }

public void saveTOfile(Invoice invoice) {  
   
 //Save to File  
 }  
}

1. **Liskov Substitution Principle**

If class B is subtype of class A, then we should be able to replace object of A with B without breaking the behaviour of the program.

public interface Car {

void turnOnEngine();

void accelerate();

}

Above, we define a simple Car interface with a couple of methods that all cars should be able to fulfill: turning on the engine and accelerating forward.

Let’s implement our interface and provide some code for the methods:

public class MotorCar implements Car {

private Engine engine;

//Constructors, getters + setters

public void turnOnEngine() {

//turn on the engine!

engine.on();

}

public void accelerate() {

//move forward!

engine.powerOn(1000);

}

}

But wait — we are now living in the era of electric cars:

public class ElectricCar implements Car {

public void turnOnEngine() {

throw new AssertionError("I don't have an engine!");

}

public void accelerate() {

//this acceleration is crazy!

}

}

By throwing a car without an engine into the mix, we are inherently changing the behavior of our program. This is**a blatant violation of Liskov substitution and is a bit harder to fix than our previous two principles.**

One possible solution would be to rework our model into interfaces that take into account the engine-less state of our Car.

**Liskov Substitution Principle with Solution**

package com.automation.solidPrinciple.LSP;  
  
public class Vehicle {  
  
 public Integer numberOfVehicle() {  
 return 2;  
 }  
}

package com.automation.solidPrinciple.LSP;  
  
public class Bycycle extends Vehicle{  
  
 public boolean hasEngine(){  
 return false;  
 }  
}

package com.automation.solidPrinciple.LSP;  
  
public class Car extends Vehicle{  
  
 public boolean hasEngine(){  
 return true;  
 }  
}

package com.automation.solidPrinciple.LSP;  
  
import org.checkerframework.checker.units.qual.C;  
  
import java.util.ArrayList;  
import java.util.List;  
  
public class MainClass {  
  
 public static void main(String[] args) {  
  
 List<Vehicle> vehicleList = new ArrayList<>();  
 vehicleList.add(new Bycycle());  
 vehicleList.add(new Car());  
  
 for(Vehicle vehicle:vehicleList){  
 System.*out*.println(vehicle.numberOfVehicle().toString());  
  
 }  
  
 }  
}

1. **Interface Segregation**

It simply means that **larger interfaces should be split into smaller ones. By doing so, we can ensure that implementing classes only need to be concerned about the methods that are of interest to them.**

For this example, we’re going to try our hands as zookeepers. And more specifically, we’ll be working in the bear enclosure.

Let’s start with an interface that outlines our roles as a bear keeper:

public interface BearKeeper {

void washTheBear();

void feedTheBear();

void petTheBear();

}

As avid zookeepers, we’re more than happy to wash and feed our beloved bears. But we’re all too aware of the dangers of petting them. Unfortunately, our interface is rather large, and we have no choice but to implement the code to pet the bear.

Let’s **fix this by splitting our large interface into three separate ones**:

public interface BearCleaner {

void washTheBear();

}

public interface BearFeeder {

void feedTheBear();

}

public interface BearPetter {

void petTheBear();

}

Now, thanks to interface segregation, we’re free to implement only the methods that matter to us:

public class BearCarer implements BearCleaner, BearFeeder {

public void washTheBear() {

//I think we missed a spot...

}

public void feedTheBear() {

//Tuna Tuesdays...

}

}

And finally, we can leave the dangerous stuff to the reckless people:

public class CrazyPerson implements BearPetter {

public void petTheBear() {

//Good luck with that!

}

}