**SOLID**

**S :** single responsibility principle.(a class should have only one reason to change)

**O :** open closed principle.(open for extension but closed for modification)

**L** : liskov substitution principle.(objects of a superclass should be replaceable with objects of its subclasses without breaking the application)

**I :** interface segregation principle.(Segregating a interface into multiple interfaces)

**D :** dependency inversion principle. **(**high level modules should not depend on low level modules; both should depend on abstractions.)

**Advantages**

* Help us to write better code.
* Avoid duplicate code
* Easy to maintain & understand
* Reduce Complexity

**Single Responsibility Principle :**

**public class** Pen {

**private** String **brand**;

**private** String **color**;

**private int price**;

**public** Pen(String brand, String color, **int** price) {

**this**.**brand** = brand;

**this**.**color** = color;

**this**.**price** = price;

}

**public** String getBrand() {

**return brand**;

}

**public void** setBrand(String brand) {

**this**.**brand** = brand;

}

**public** String getColor() {

**return color**;

}

**public void** setColor(String color) {

**this**.**color** = color;

}

**public int** getPrice() {

**return price**;

}

**public void** setPrice(**int** price) {

**this**.**price** = price;

}

}

**Here Invoice class is doing multiple stuffs.**

**Such as calculating price, downloading invoice, saving the pen object to DB.**

**Here this Invoice class is violating the SRP. Because SRP states that class should have only 1 reason to change.**

**public class** Invoice {

Pen **pen**;

**int quantity**;

**public** Invoice(Pen pen, **int** quantity){

**this**.**pen** = pen;

**this**.**quantity** = quantity;

}

**public int** calculatePrice(Pen pen, **int** quantity){

**int** price = pen.getPrice() \* quantity;

**return** price;

}

**public void** downloadInvoice(Pen pen){

System.***out***.println(**"Downloading Invoice "** + pen);

}

**public void** saveToDB(Pen pen){

System.***out***.println(**"Saving to DB "**+pen);

}

}

**So to support SRP we can introduce 3 classes instead of Invoice class which can perform single responsibility.**

**public class** InvoiceDAO {

Pen **pen**;

**int quantity**;

**public** InvoiceDAO(Pen pen, **int** quantity) {

**this**.**pen** = pen;

**this**.**quantity** = quantity;

}

**public void** saveToDB(Pen pen) {

System.***out***.println(**"Saving to DB "** + pen);

}

}

**public class** InvoiceDownload {

Pen **pen**;

**int quantity**;

**public** InvoiceDownload(Pen pen, **int** quantity) {

**this**.**pen** = pen;

**this**.**quantity** = quantity;

}

**public void** downloadInvoice(Pen pen) {

System.***out***.println(**"Downloading Invoice "** + pen);

}

}

**public class** InvoicePriceCalc {

Pen **pen**;

**int quantity**;

**public** InvoicePriceCalc(Pen pen, **int** quantity){

**this**.**pen** = pen;

**this**.**quantity** = quantity;

}

**public int** calculatePrice(Pen pen, **int** quantity){

**int** price = pen.getPrice() \* quantity;

**return** price;

}

}

**Here every class has only single responsibility.Now this is supporting SRP.**

**OPEN / CLOSED PRINCIPLE**

**public class** InvoiceDAO {

Pen **pen**;

**int quantity**;

**public** InvoiceDAO(Pen pen, **int** quantity) {

**this**.**pen** = pen;

**this**.**quantity** = quantity;

}

**public void** saveToDB(Pen pen) {

System.***out***.println(**"Saving to DB "** + pen);

}

}

**There is a new requirement where we have to save the pen object to file. So we have introduced a new method to save it into a file .**

**public class** InvoiceDAO {

Pen **pen**;

**int quantity**;

**public** InvoiceDAO(Pen pen, **int** quantity) {

**this**.**pen** = pen;

**this**.**quantity** = quantity;

}

**public void** saveToDB(Pen pen) {

System.***out***.println(**"Saving to DB "** + pen);

}

**public void** saveToFile(Pen pen){

System.***out***.println(**"Saving to File "** + pen);

}

}

**As OCP states that it’s open for extension but closed for modification but here we have modified the old class so it’s violating the OCP.**

**public interface** InvoiceDAO {

**void** save(Pen pen);

}

**public class** InvoiceFileSaver **implements** InvoiceDAO{

@Override

**public void** save(Pen pen) {

System.***out***.println(**"Saving to File"**+pen);

}

}

**public class** InvoiceDBSaver **implements** InvoiceDAO {

@Override

**public void** save(Pen pen) {

System.***out***.println(**"Saving to DB"** + pen);

}

}

**Now here OCP stands without any violation.**

**Here, we can extend multiple classes with InvoiceDAO without modifying any class.**

**If in future there is further requirement we can do it by extending InvoiceDAO without violating OCP.**

**Liskov Substitution Principle**

**public interface Birl {**

**void eat();**

**void walk();**

**void fly();**

**}**

**public class Crow implements Birl{**

**@Override**

**public void eat() {**

**System.*out*.println("Eating");**

**}**

**@Override**

**public void walk() {**

**System.*out*.println("Walking");**

**}**

**@Override**

**public void fly() {**

**System.*out*.println("Flying");**

**}**

**}**

**public class Penguin implements Birl{**

**@Override**

**public void eat() {**

**System.*out*.println("Eating");**

**}**

**@Override**

**public void walk() {**

**System.*out*.println("Walking");**

**}**

**@Override**

**public void fly() {**

**}**

**}**

**In this example, Bird can eat , walk & fly. Crow can do all these things because crow is a bird.**

**But Penguin is also a bird but can’t fly.Penguin class is a subtype of class Bird, but it shouldn't be able to use the fly method, that means we are breaking the LSP principle.**

**public interface Birl {**

**void eat();**

**void walk();**

**}**

**public interface FlyingBird extends Birl{**

**void fly();**

**}**

**public class Crow implements FlyingBird{**

**@Override**

**public void eat() {**

**System.*out*.println("Eating");**

**}**

**@Override**

**public void walk() {**

**System.*out*.println("Walking");**

**}**

**@Override**

**public void fly() {**

**System.*out*.println("Flying");**

**}**

**}**

**public class Penguin implements Birl{**

**@Override**

**public void eat() {**

**System.*out*.println("Eating");**

**}**

**@Override**

**public void walk() {**

**System.*out*.println("Walking");**

**}**

**}**

**Here now we can use object of Bird class with Penguin class without violating LSP. As we can cast the super class with subclass without breaking the application.**

**Interface Segregation Principle**

**public interface ITWorker {**

**void develop();**

**void test();**

**void infra();**

**void manage();**

**void hire();**

**}**

**public class Developer implements ITWorker{**

**@Override**

**public void develop() {**

**System.*out*.println("Can develop application");**

**}**

**@Override**

**public void test() {**

**System.*out*.println("can test application");**

**}**

**@Override**

**public void infra() {**

**}**

**@Override**

**public void manage() {**

**}**

**@Override**

**public void hire() {**

**}**

**}**

**public class HR implements ITWorker{**

**@Override**

**public void develop() {**

**}**

**@Override**

**public void test() {**

**}**

**@Override**

**public void infra() {**

**}**

**@Override**

**public void manage() {**

**System.*out*.println("can manage");**

**}**

**@Override**

**public void hire() {**

**System.*out*.println("Can hire");**

**}**

**}**

**public class Manager implements ITWorker{**

**@Override**

**public void develop() {**

**}**

**@Override**

**public void test() {**

**}**

**@Override**

**public void infra() {**

**}**

**@Override**

**public void manage() {**

**System.*out*.println("can manage");**

**}**

**@Override**

**public void hire() {**

**}**

**}**

**public class Tester implements ITWorker{**

**@Override**

**public void develop() {**

**}**

**@Override**

**public void test() {**

**System.*out*.println("can do testing");**

**}**

**@Override**

**public void infra() {**

**}**

**@Override**

**public void manage() {**

**}**

**@Override**

**public void hire() {**

**}**

**}**

**Here every one is a IT worker but a tester can only test the application , developer can develop & test the app, manager can manage the team, HR can manage & hire new member. But rest of the method we have to implement without any business logic. So it’s violating the ISP. Instead of having one interface we can have multiple interface .**

**public interface IDeveloper {**

**void develop();**

**}**

**public interface IHire {**

**void hire();**

**}**

**public interface IManage {**

**void manage();**

**}**

**public interface ITester {**

**void test();**

**}**

**public class Developer implements IDeveloper, ITester{**

**@Override**

**public void develop() {**

**System.*out*.println("Can develop application");**

**}**

**@Override**

**public void test() {**

**System.*out*.println("can test application");**

**}**

**}**

**public class HR implements IManage, IHire{**

**@Override**

**public void manage() {**

**System.*out*.println("can manage");**

**}**

**@Override**

**public void hire() {**

**System.*out*.println("Can hire");**

**}**

**}**

**public class Manager implements IManage{**

**@Override**

**public void manage() {**

**System.*out*.println("can manage");**

**}**

**}**

**public class Tester implements ITester{**

**@Override**

**public void test() {**

**System.*out*.println("can do testing");**

**}**

**}**

**Now in this example we can se multiple interfaces having specific methods.**

**So each class can do their specific tasks without implementing non required methods.This is supporting ISP.**

**Dependency Inversion Principle**

**public class Shop {**

**CreditCard creditCard = new CreditCard();**

**public void shopping(){**

**creditCard.pay();**

**}**

**}**

**public class CreditCard {**

**public void pay(){**

**System.*out*.println("Paying with CC");**

**}**

**}**

**public class DebitCard {**

**public void pay(){**

**System.*out*.println("paying with DC");**

**}**

**}**

**Here High level module (Shop) is directly depending on low level module(Credit card) which shows it’s violating DIP.**

**Instead of directly depending on each other both module should depend on abstraction.**

**public interface Payment {**

**void pay();**

**}**

**public class Shop {**

**Payment payment;**

**public Shop(Payment payment) {**

**this.payment = payment;**

**}**

**public void shopping() {**

**payment.pay();**

**}**

**}**

**public class DebitCard implements Payment {**

**@Override**

**public void pay() {**

**System.*out*.println("paying with DC");**

**}**

**}**

**public class CreditCard implements Payment {**

**@Override**

**public void pay(){**

**System.*out*.println("Paying with CC");**

**}**

**}**

**Now here both the modules are depending on abstraction insteading of depending on each other. Supporting DIP .**