**SOLID principles**

**Key Contents**

* **SOLID Introduction**
* **What is SOLID Acronym**
* **SOLID design principles**

**SOLID Introduction**

In Java, **SOLID principles** are an object-oriented approach.

These principles are applied to software structure design.

It is conceptualized by **Robert C. Martin**.

These five principles have changed the world of object-oriented programming and changed the way of writing software. It also ensures that the software is modular, easy to understand, debug, and refactor.

## ****SOLID Acronym****

**S** : Single Responsibility Principle (SRP)

**O** : Open closed Principle (OSP)

**L** : Liskov substitution Principle (LSP)

**I**: Interface Segregation Principle (ISP)

**D** : Dependency Inversion Principle (DIP)

## ****SOLID design principles****

**Single Responsibility Principle**

This principle states that “a class should have only one reason to change” which means every class should have a single responsibility or single purpose or single job.

Imagine that there is a class called **BankService**which performs following operations.

1. Withdraw
2. Deposit
3. Print Passbook
4. Get Loan Info
5. Send OTP

**package** com.javatechie.solid.srp;  
  
  
**public class** BankService {  
  
 **public long** deposit(**long** amount, String accountNo) {  
 //deposit amount **return** 0;  
 }  
  
 **public long** withDraw(**long** amount, String accountNo) {  
 //withdraw amount **return** 0;  
 }

**public void** printPassbook() {  
 //update transaction info in passbook }  
   
 **public void** getLoanInterestInfo(String loanType) {  
 **if** (loanType.equals(**"homeLoan"**)) {  
 //do some job }  
 **if** (loanType.equals(**"personalLoan"**)) {  
 //do some job }  
 **if** (loanType.equals(**"car"**)) {  
 //do some job }  
 }  
  
 **public void** sendOTP(String medium) {  
 **if** (medium.equals(**"email"**)) {  
 //write email related logic //use JavaMailSenderAPI }  
 }  
  
}

Here the class has multiple reasons to change.

For example, for **getLoanInterestInfo()** method, now **Bank** **Service** provides info only for **Personal Loan, Home Loan and Car Loan.** Let’s say in future **BankService** wants to provide some other loan features like **Gold** **Loan** and **Education Loan** then again you need to modify this class implementation, right?

similarly, for **sendOTP()** method, now **BankService** supports email as a medium to send OTP, but in the future **BankService** may have to introduce Phone as another medium to send OTP then again you need to change its implementation.

This class doesn’t follow single responsibility principle because it has many responsibilities or job to perform.

To achieve the goal of the single responsibility principle, we should implement multiple separate classes each performs only a single job or functionality.

For Example, we can move Print Passbook related job to **Printer Service**

public class PrinterService{  
 **public void** printPassbook () {  
 //update transaction info in passbook }  
}

Similarly, Loan Interest Information related job to **LoanService**

public class LoanService{  
**public void** getLoanInterestInfo(String loanType) {  
 **if** (loanType.equals(**"homeLoan"**)) {  
 *//do some job* }  
 **if** (loanType.equals(**"personalLoan"**)) {  
 *//do some job* }  
 **if** (loanType.equals(**"car"**)) {  
 *//do some job* }  
 }  
}

Similarly, Send OTP related Job to **NotificationService**

public class NotificationService {

**public void** sendOTP(String medium) {  
 **if** (medium.equals(**"email"**)) {  
 //write email related logic //use JavaMailSenderAPI }  
 }  
}

**Open closed Principle (OSP)**

This principle states that “software entities (modules, classes, functions, etc.) should be open for extension, but closed for modification” which means you should be able to extend a class behaviour, without modifying it.

let’s understand this principle with an example. let’s consider the same **NotificationService** which we just created.

public class NotificationService{

**public void** sendOTP(String medium) {  
 **if** (medium.equals(**"email"**)) {  
 //write email related logic //use JavaMailSenderAPI }  
 }  
}

Later if you want to introduce send OTP via **Mobile Phone number** or **WhatsApp number** then you need to modify source code in **NotificationService** right?

Here What OCP says, it is open for extension, but it is closed for modification hence it’s not recommended to modify source code in **NotificationService** for each OTP Feature as it will violate OCP.

So, to overcome this problem, you need to design your code in such a way that everyone can reuse your feature by just extending it.

And if they need any customization, they can extend it and add their customized feature on top of it like an abstraction.

You can design **NotificationService** like below and let different OTP feature implementors to provide their own customized implementation.

public interface NotificationService{

**public void** sendOTP(String medium);  
 **public void** sendTransactionNotification(String medium);  
}

**EmailNotification** service implementation

public class EmailNotification implements NotificationService{  
 **public void** sendOTP(String medium){  
 // write Logic using JavaMailSenderAPI   
 }  
 **public void** sendTransactionNotification(String medium){  
 }  
}

**MobileNotification** service implementation

public class MobileNotification implements NotificationService{  
 **public void** sendOTP(String medium){  
 // write Logic using [Twilio SMS AP](https://www.twilio.com/docs/sms/api)I  
 }  
 **public void** sendTransactionNotification(String medium){  
 }  
}

similarly, you can add implementation for **WhatsAppNotification** service.

public class WhatsAppNotification implements NotificationService {  
 **public void** sendOTP(String medium){  
 // write Logic using WhatsApp API  
 }  
 **public void** sendTransactionNotification(String medium){  
 }  
}

**Liskov substitution Principle (LSP)**

This principle states that “Derived or child classes must be substitutable for their base or parent 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 behaviour of the program.

This principle is bit tricky and interesting. And it is designed based on **Inheritance** concepts, so let’s better understand this with an example.

Let’s consider I have an abstract class called **SocialMedia** which supports all social media activities for the user to entertain them like below.

**package** com.javatechie.solid.lsp;  
  
**public abstract class** SocialMedia {  
   
 **public abstract void** chatWithFriend();  
   
 **public abstract void** publishPost(Object post);  
   
 **public abstract void** sendPhotosAndVideos();  
   
 **public abstract void** groupVideoCall(String... users);  
}

Social media can have multiple implementations (Child Classes) like **Facebook, WhatsApp, Instagram, and Twitter** etc...

now let’s assume **Facebook** want to use these features or functionalities.

**package** com.javatechie.solid.lsp;  
  
**public class** Facebook **extends** SocialMedia {  
  
 **public void** chatWithFriend() {  
 //logic  }  
  
 **public void** publishPost(Object post) {  
 //logic  }  
  
 **public void** sendPhotosAndVideos() {  
 //logic  }  
  
 **public void** groupVideoCall(String... users) {  
 //logic  }  
}

As all the above-mentioned features available in **Facebook**, here we can consider **Facebook** is complete substitute of **SocialMedia** class.

Now let’s discuss WhatsApp class.

**package** com.javatechie.solid.lsp;  
  
**public class** WhatsApp **extends** SocialMedia {  
 **public void** chatWithFriend() {  
 //logic }  
  
 **public void** publishPost(Object post) {  
 //Not Applicable }  
  
 **public void** sendPhotosAndVideos() {  
 //logic }  
  
 **public void** groupVideoCall(String... users) {  
 //logic }  
}

due to publishPost() method **WhatsApp** child is not substitute of its parent **SocialMedia**,so it doesn’t follow **LSP.**

Similarly, **Instagram** doesn’t support groupVideoCall() feature, so we say **Instagram** child is not substitute of its parent **SocialMedia.**

How to overcome this issue so that my code can follow **LSP?**

**Solution**

create a **SocialMedia** interface.

public interface SocialMedia {   
 **public void** chatWithFriend();  
 **public void** sendPhotosAndVideos()  
}

Similarly, another one called **SocialPostAndMediaManager**.

public interface SocialPostAndMediaManager {   
 **public void** publishPost(Object post);  
}

Similarly, another interface called **VideoCallManager.**

public interface VideoCallManager{   
 **public void** groupVideoCall(String... users);  
}

Now if you observe we segregate specific functionality to separate interface to follow **LSP.**

Now it’s up to implementation class, based on their supported features, they can implement respective interfaces. For example, **Instagram** doesn’t support group video call feature so **Instagram** implementation can be designed something like this.

public class Instagram implements SocialMedia, SocialPostAndMediaManager {

**public void** chatWithFriend() {  
 //logic  
 }  
 **public void** sendPhotosAndVideos() {  
 //logic  
 }  
 **public void** publishPost(Object post){  
 //logic  
 }

}

This is how you can design **LSP.**

**Interface Segregation Principle (ISP)**

This principle is the first principle that applies to **Interfaces** instead of **Classes** in SOLID and it is similar to the **Single Responsibility Principle**.

**It states that** **“do not force any client to implement an interface which is irrelevant to them “.**

For example, let’s say you have an interface called **UPIPayments** like below.

**public interface** UPIPayments {  
   
 **public void** payMoney();  
   
 **public void** getScratchCard();  
   
 **public void** getCashBackAsCreditBalance();  
}

Now let’s talk about 2 implementations for **UPIPayments** like **Google Pay** and **Paytm.**

**Google Pay** supports all these features so it can directly implement this **UPIPayments,** but **Paytm** doesn’t support getCashBackAsCreditBalance() feature so here we shouldn’t force it to override this method by implementing **UPIPayments.**

we need to segregate interface based on client’s need, so to support this **ISP**, so we can design something like below.

create a separate interface called **CashBackManager** which will deal with Cashback.

public interface CashbackManager{  
 **public void** getCashBackAsCreditBalance();  
}

Now we can remove getCashBackAsCreditBalance from **UPIPayments** interface.

Now **Google Pay** can implement both the interfaces **UPIPayments** and **CashBackManager** as it supports all the features, But **Paytm** can implement only **UPIPayments** then as a client we are not forcing it for anything to use, hence the code follows **ISP.**

**Dependency Inversion Principle (DIP)**

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.

let’s consider a best use case.

You go to a local store to buy something, and you decide to pay for it by using your card. So, when you give your card to the clerk for making the payment, the clerk doesn’t bother to check what kind of card you have given.

Whether you have given a debit card or credit card it does not even matter; they will simply swipe it. This is what the abstraction between clerk and you to rely on for Card processing.

now let’s replace this example in code to understand it better.

let’s assume you have two options to do payments **Debit card** and **Credit card.**

Debit Card

public class DebitCard{  
 public void doTransaction(int amount){  
 System.out.println("tx done with DebitCard");  
 }  
}

Credit Card

public class CreditCard{  
 public void doTransaction(int amount){  
 System.out.println("tx done with CreditCard");  
 }  
}

Now with these two cards you went to shopping mall and purchased some order and decided to pay using **CreditCard**.

public class ShoppingMall {

private DebitCard debitCard;

public ShoppingMall (DebitCard debitCard) {  
 this.debitCard = debitCard;  
}

public void doPayment(Object order, int amount){

debitCard.doTransaction(amount);

}

public static void main(String[] args) {  
 DebitCard debitCard = new DebitCard();  
 ShoppingMall shoppingMall=new ShoppingMall(debitCard);  
 shoppingMall.doPayment("some order",5000);  
 }  
}

if you observe this is wrong design of coding, now ShoppingMall class tightly coupled with **DebitCard**.

Now there is some error in your **DebitCard,** and you want to go with Credit card then this won’t be possible because ShoppingMall is tightly couple with Debit Card

We can do that by remove **DebitCard** from constructor and inject **CreditCard,** but this is not good approach to write code.  
Because to follow **DIP,**we need to design our application in such a way that my **ShoppingMall** payment system should accept any type of ATM Card (it shouldn’t care whether it is **DebitCard** or **CreditCard**)

To follow DIP, we can create an interface called **Bankcard** like bellow.

public interface BankCard {  
 public void doTransaction(int amount);  
}

Now both **DebitCard** and **CreditCard** will use this **BankCard** as an abstraction.

CreditCard

public class CreditCard implements BankCard {  
 public void doTransaction(int amount) {

System.out.println("tx done with CreditCard");  
 }  
}

similarly, DebitCard

public class DebitCard implements BankCard {  
 public void doTransaction(int amount) {  
 System.out.println("tx done with DebitCard");  
 }  
}

Now you need to redesign ShoppingMall implementation.

public class ShoppingMall {

private BankCard bankCard;

public ShoppingMall(BankCard bankCard) {  
 this.bankCard = bankCard;  
 }

public void doPayment(Object order, int amount) {  
 bankCard.doTransaction(amount);  
 }

public static void main(String[] args) {  
 BankCard bankCard=new CreditCard();  
 ShoppingMall shoppingMall1=new ShoppingMall(bankCard);  
 shoppingMall1.doPayment("do some order", 10000);  
 }  
}

Now if you observe shopping mall is loosely coupled with BankCard, so any type of card can be used to process the payment without any impact of code change, hence the design of code implementation proves **DIP.**