**Solid Principle in Java**

1. **S 🡪** Single Responsibility Principle
2. **O 🡪**Open-closed Principle
3. **L 🡪**Liskov Substitution Principle
4. **I 🡪**Interface Segregation Principle
5. **D 🡪** Dependency Inversion Principle

**Single Responsibility Principle**

Single Responsibility principle state that every java must perform single functionality. Implementation of multiple functionalities in a single class mashup the code and if there is any modification in the class it hampers the functionality of the class.

Advantages

Easy to maintain

Easy to understand

**Open-closed Principle**

The Open closed principle states that each class should be open for extension but closed for modification. Suppose there is area class which is finding the area of circle and the code is working properly, now suddenly there is requirement of finding the area of rectangle. Now if we try to modify the area class it will create problem in the future so what we will do is we will simply create an interface of area class and implements that area class with circle and rectangle so that it will follow open-closed Principle.

**Liskov Substitution Principle**

If class B is the subclass of class A then we can replace the object of A with the object of B without breaking the behaviour of the class.

Subclass should extends the capability of the class not narrow the capability of the parent class.

Interface **vehicle**{

Void engineOn();

Void Accelarates();

}

Class bike implements Vehicle{

Public static void engineOn {

Return engine on;

}

Public static void accelareates(){

Return Accelarates+10;

}

Class bicycles implements vehicle{

Public static void engineOn {

Throws engineException(“error in engine”);

}

Public static void accelareates(){

Return Accelarates+10;

}

}

Class bicycle narrowing the behaviour of the vehicle so it is not following linskon substitution principle

**Interface Segregation Principle**

Interface should be such that clients shouldn’t implements unnecessary functionality that they don’t need.

Like suppose there is restaurant interface

Interface restaurant{

Void serve();

Void cook();

Void washUtensils();

}

Class waiter implements restaurant{

}

It has to implements all the the functionality, but his major work is to only serve the customer;

**Dependency Inversion Principle**

Class should depend on interface rather than concrete class.

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. Because the abstraction does not depend on detail but the detail depends on abstraction. It decouples the software.

**Design Pattern in java**

* **Factory Method Design Pattern**
* **Singleton method Design Pattern**
* **Prototype method Design Pattern**
* **Observer Design Pattern**

**Factory Method Design Pattern:-**

Factory design pattern in java is creation design pattern that talks about the creation of the object. The factory design pattern say to design the interface for the creation of objects and let the subclass decide to which class to instantiate.

Now suppose any employer needs an 100 employees then it will implement factory design Pattern. Simply they will hire a recruiting agency and tells the requirement of the candidates.

The agency will handle all the jobs , it will accumulate the data ,schedule the interview , hire the candidate and provide the candidate to the company. Company don’t have to accumulate the detail and search for the details.

Like :-

//product

Interface company{

Void Candidate();

}

//Concrete Products

Class agency1 implements company(){

Public static candidate(){

Sysout (devopsCandidate);}

}

Class agency2 implements company(){

Public static candidate(){

Sysout (javaCandidate);}

}

//factory Interface

Interface Factory{

Company FactoryMethod

}

//Concrete factories

Class ConcreteFactoryA implements factory{

Public void company FactoryMethod(){

Return new agency1();

}

Class ConcreteFactoryB implements factory{

Public void company FactoryMethod(){

Return new agency2();

}

Class FactoryMethodname{

Public static void main(String [] args)

Factory fact1 = new ConcreteFactoryA();

Company cmp =fact1. FactoryMethod ();

Cmp.candidate();

Factory fact2 = new ConcreteFactoryA();

Company cmp2 =fact2. FactoryMethod ();

Cmp2.candidate();

}