SOLID principles are a set of five design principles. These principles help software developers design robust, testable, extensible, and maintainable object-oriented software systems.

Each of these five design principles solves a particular problem that might arise while developing the software systems.

SOLID is an acronym that stands for:

* Single Responsibility Principle (SRP)
* Open-Closed Principle (OCP)
* Liskov Substitution Principle (LSP)
* Interface Segregation Principle (ISP)
* Dependency Inversion Principle (DIP)

Following these principles can result in a very large codebase for a software system. But in the long run, the main aim of the principles is never defeated. That is, helping software developers make changes to their code without causing any major issues.

What are the benefits of using SOLID Design Principles?

* Reduced code complexity: SOLID principles help to break down complex code into smaller, more manageable pieces.
* Improved code quality: SOLID principles help to create code that is easy to read, understand, and maintain. Thus, spend less time figuring out what it does.
* Improved testability: SOLID principles help to create code that is easy to test, reducing the likelihood of bugs and errors.
* Enhanced flexibility: SOLID principles help to create code that can be easily modified and extended.

## SOLID Principles-The Single Responsibility Principle

## This Principle says that there should never be more than one reason for a class to change. A class should be focused on a single functionality, address a specific concern. This means that every class, or similar structure, in your code should have only one job to do. Everything in the class should be related to that single purpose, i.e., be cohesive. It does not mean that your classes should only contain one method or property.

The more responsibilities your class has, the more often you need to change it. If your class implements multiple responsibilities, they are no longer independent of each other. Therefore, you will need to change your class more often. Consequently, it will have more side-effects, and requires a lot more work than it should have. So, it’s better to avoid these problems by making sure that each class has only one responsibility.

Suppose you built your software over a longer period and you need to adapt it to on-going change requests. You might feel the easiest and fastest approach is adding a method or functionality to your existing code instead of writing a new class or component. But that often results in classes with more than responsibility and makes it more and more difficult to maintain the software. You can avoid these problems by asking a simple question before you make any changes.

What is the responsibility of your class/component/microservice?

If your answer includes the word “and”, you’re most likely breaking the Single Responsibility Principle.

Then it’s better to take a step back and rethink your current approach. There is most likely a better way to implement it. For example, let’s understand ‘SOLID Principles-The Single Responsibility Principle’ with the help of codes.

Example#1: Code that violates Single Responsibility Principle

This example demonstrates the code that violates ‘SOLID Principles-The Single Responsibility Principle’.

Suppose we are writing a java application for a hiring consultancy. We will create a Resume class that lets consultant get & set the technology & years of experiences in each resume, and search the resume from the repository.

**public class Resume {**

**String technology;**

**Integer yearsOfExperience;**

**public String getTechnology() {**

**return technology;**

**}**

**public void setTechnology(String technology) {**

**this.technology = technology;**

**}**

**public Integer getYearsOfExperience() {**

**return yearsOfExperience;**

**}**

**public void setYearsOfExperience(Integer yearsOfExperience) {**

**this.yearsOfExperience = yearsOfExperience;**

**}**

**public void searchResume() {**

**//logic to search resume goes here**

**}**

**}**

However, the above code violates the Single Responsibility Principle, as the Resume class has two responsibilities. First, it sets the properties (technology and yearsofExperience) related to the Resume. Second, it searches for the resume in the repository. The setter methods change the Resume object, which might cause problems when we want to search the same resume in the repository.

### ****Example#1: Code that follows the Single Responsibility Principle****

This example demonstrates the code that follows ‘SOLID Principles-The Single Responsibility Principle’.

In order to apply the ‘SOLID Principles-The Single Responsibility Principle’, we need to decouple the two responsibilities. In the refactored code, the Resume class will only be responsible for getting and setting the properties of the Resume object. For example, below code demonstrates the concept.

**public class Resume {**

**String technology;**

**Integer yearsOfExperience;**

**public String getTechnology() {**

**return technology;**

**}**

**public void setTechnology(String technology) {**

**this.technology = technology;**

**}**

**public Integer getYearsOfExperience() {**

**return yearsOfExperience;**

**}**

**public void setYearsOfExperience(Integer yearsOfExperience) {**

**this.yearsOfExperience = yearsOfExperience;**

**}**

**}**

Then, we create another class called RepositoryView that will be responsible for searching the resume. We move the searchResume() method here and reference the Resume class in the constructor.

**public class RepositoryView {**

**Resume resume;**

**public RepositoryView(Resume resume) {**

**this.resume = resume;**

**}**

**public void searchResume() {**

**//logic to search resume goes here**

**}**

**}**

### ****Example#2: Code that violates the Single Responsibility Principle****

This example demonstrates the code that violates ‘SOLID Principles-The Single Responsibility Principle’.

Let’s consider another example to understand it more as in the below code:

**public class Employee {**

**public Double calculatePay() {...}**

**public void saveEmployee() {...}**

**public void getEmployeeReport() {...}**

**}**

How many responsibilities?

The correct answer is three.

Here we have 1) calculation logic 2) database logic and 3) reporting logic. All mixed up within one class. If you have multiple responsibilities combined into one class, it might be difficult to change one part without breaking others. Mixing responsibilities also makes the class harder to understand and harder to test. It will also decrease the cohesion. The easiest way to fix this is to split the class into three different classes, with each having only one responsibility: database access, calculating payment and reporting, all separated.

Next question, how many reasons to change?

Again, the correct answer is three. ;-).

There are 3 reasons for this class to change. 1) Any change in payment calculation. 2) Any change in logic of saving employee into the database. 3) Any change in reporting logic. If you think in a broad way, in order to make changes in one part, we may also have to change all parts unnecessarily.

### ****Example#2: Code that follows the Single Responsibility Principle****

This example demonstrates the code that follows ‘SOLID Principles-The Single Responsibility Principle’.

We will split the Employee class into three classes in order to follow the ‘SOLID Principles-The Single Responsibility Principle’. For example, below code demonstrates the concept.

**public class EmployeePayment {**

**public Double calculatePay() {**

**// logic to calculate employee payment**

**}**

**}**

**public class EmployeeRepository {**

**public void saveEmployee() {**

**// logic to save employee object**

**}**

**}**

**public class EmployeeReport {**

**public void getEmployeeReport() {**

**// logic to generate employee report**

**}**

**}**

## ****What is SOLID Principles-The Open Closed Principle?****

The principle says, “Software components should be open for extension, but closed for modification”. Simply put, Software components like classes, modules, and functions should be open for extension but closed for modifications.

If we consider our software component as a class, then Classes should be open for extension, but closed for modification. In doing so, we stop ourselves from modifying existing code.

Of course, the one exception to the rule is when fixing bugs in existing code. So, we should modify our class only at the time of bug fixing.

“Open to extension” means that you should design your classes so that new functionality can be added as new requirements are generated. “Closed for modification” means that once you developed a class you should never modify it, except to correct bugs.

These two parts of the principle appear to be contradictory. However, if you correctly structure your classes and their dependencies, you can add functionality without editing existing source code.

Design and code should be done in a way that new functionality should be added with minimum or no changes in the existing code. When needs to extend functionality – avoid tight coupling, don’t use if-else/switch-case logic, do code refactoring as required.

Generally, you achieve this by referring to abstractions for dependencies, such as interfaces or abstract classes, rather than using concrete classes. We can add the functionality by creating new classes that implement the interfaces. This reduces the risk of introducing new bugs to existing code, leading to more robust software.

### ****Example: Code that violates OCP****

Let’s assume that we have to write a program that calculates area of various shapes. We start with creating a class for our first shape, let’s say Rectangle which has 2 attributes length & width

**public class Rectangle {**

**public Double length;**

**public Double width;**

**}**

Further, we create a class to calculate the area of this Rectangle which has a method calculateRectangleArea() which takes the Rectangle as an input parameter and calculates its area.

**public class AreaCalculator {**

**public Double calculateRectangleArea(Rectangle rectangle) {**

**return rectangle.length \* rectangle.width;**

**}**

**}**

So far so good. Now let’s assume we have to write a program for our second shape circle. So, we promptly create a new class Circle with a single attribute radius.

**public class Circle {**

**public Double radius;**

**}**

Then we modify AreaCalculator class to add circle calculations through a new method calculateCircleArea().

**public class AreaCalculator{**

**public Double calculateRectangleArea(Rectangle rectangle){**

**return rectangle.length \* rectangle.width;**

**}**

**public Double calculateCircleArea(Circle circle){**

**return (22 / 7) \* circle.radius \* circle.radius;**

**}**

**}**

So here we are having more extensions to the functionality but in order to accommodate the extension we are modifying the code for each extension which is against the principle.

However, please note that there are flaws in the way of designing our solution above.

Let’s say we have a new shape pentagon. In that case, we will again end up modifying the AreaCalculator class. As the types of shapes grows this becomes messier as AreaCalculator keeps on changing and any consumers of this class will have to keep on updating their libraries which contain AreaCalculator. As a result, AreaCalculator class will not be baselined(finalized) with surety as every time a new shape comes, it will be modified. So, this design is not closed for modification.

AreaCalculator will need to keep on adding their computation logic in newer methods. We are not really expanding the scope of shapes; rather we are simply doing piece-meal(bit-by-bit) solution for every shape that is added.

### ****Example: Code that follows OCP****

Let’s now see a more elegant design which solves the flaws in the above design by adhering to the Open/Closed Principle. First of all, we will make the design extensible. For this we need to first define a base type Shape and have Circle & Rectangle implement Shape interface. For example, below code demonstrates the concept.

**public interface Shape {**

**public Double calculateArea();**

**}**

**public class Rectangle implements Shape {**

**Double length;**

**Double width;**

**public double calculateArea() {**

**return length \* width;**

**}**

**}**

**public class Circle implements Shape {**

**public Double radius;**

**public Double calculateArea() {**

**return (22 / 7) \* radius \* radius;**

**}**

**}**

As aforementioned, there is a base interface Shape. All shapes now implement the base interface Shape. Shape interface has an abstract method calculateArea(). Both circle & rectangle provide their own overridden implementation of calculateArea() method using their own attributes. If in future we want to calculate area of other shapes like triangle, square etc., we can implement the Shape interface without changing any class.

We have brought in a degree of extensibility as shapes are now an instance of Shape interfaces. This allows us to use Shape instead of individual classes.

The last point is the consumer of these shapes. The consumer will be the AreaCalculator class which would now look like this.

**public class AreaCalculator {**

**public Double calculateShapeArea(Shape shape) {**

**return shape.calculateArea();**

**}**

**}**

This AreaCalculator class now fully removes our design flaws noted above and provides a clean solution which adheres to the Open-Closed Principle.

## ****What is SOLID Principles-The Liskov Substitution Principle?****

Simply put, if class A is a subtype of class B, then we should be able to replace objects of B with objects of A (i.e., objects of type A may substitute objects of type B) without changing the behaviour (correctness, functionality, etc.) of our program.

“Derived types must be completely substitutable for their base types”

In Layman’s terms, it states that an object of a superclass should be replaceable by objects of its subclasses without causing issues in the application. Therefore, a child class should never change the characteristics of its parent class (such as the argument list and return types). Basically, derived classes should never do less than their base class.

LSP applies to inheritance hierarchies, specifying that you should design your classes so that client dependencies can be substituted with subclasses without the client knowing about the change

All subclasses must, therefore, operate in the same manner as their base classes. The specific functionality of the subclass may be different, but must conform to the expected behavior of the base class. To be a true behavioral subtype, the subclass must not only implement the base class’s methods and properties, but also stick to its implied behavior.

Why is it required to follow Liskov Substitution Principle?

This avoids overuse/misuse of inheritance. It helps us conform to the “is-a” relationship. We can also say that subclasses must fulfil a contract defined by the base class.

## Bad example

public class Bird{

public void fly(){}

}

public class Duck extends Bird{}

The duck can fly because it is a bird, but what about this:

public class Ostrich extends Bird{}

Ostrich is a bird, but it can't fly, Ostrich 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.

## Good example

public class Bird{}

public class FlyingBird extends Bird{

public void fly(){}

}

public class Duck extends FlyingBird{}

public class Ostrich extends Bird{}

## What is ****SOLID Principles-The Interface Segregation Principle (ISP)?****

If you review your project code thoroughly, you will find that this principle is the most violated just because we don’t focus on the good design while coding.

The Interface Segregation principle says that “Clients should not be forced to depend upon interfaces that they do not use.”

When we have non-cohesive interfaces, the ISP guides us to create multiple, smaller, cohesive interfaces. Larger interfaces should be split into smaller ones. By doing so, we can ensure that implementing classes only needs to be concerned about the methods that are of interest to them.

A client, no matter what, should never be forced to implement an interface that it does not use or the client should never be forced to depend on any method, which is not used by them. So basically, the interface segregation principles as you prefer the interfaces, which are small but client specific instead of monolithic and bigger interface. In short, it would be bad for you to force the client to depend on a certain thing, which they don’t need.

Basically, the lesson here is **“Don’t depend on things you don’t need”**.

### ****Example#1: Code that violates ISP****

Let’s assume that there is a Restaurant interface which contains methods for accepting orders from online customers, telephone customers and walk-in customers. It also contains methods for handling online payments (for online customers) and in-person payments. In-person payments deal with the walk-in customers as well as telephone customers. Moreover, telephone customers pay in-person at the time of order delivery. Now let’s create a Java Interface for Restaurant and name it as RestaurantInterface.java.

**public interface RestaurantInterface {**

**public void acceptOnlineOrder();**

**public void acceptTelephoneOrder();**

**public void acceptWalkInCustomerOrder();**

**public void payOnline();**

**public void payInPerson();**

**}**

We have 5 methods declaration in RestaurantInterface. They are for accepting online order, taking a telephonic order, accepting orders from a walk-in customer in order to place the order. Similarly, accepting online payment and accepting payment in person in order to make the payments.

Let’s start by implementing the RestaurantInterface for online customers as OnlineCustomerImpl.java.

**public class OnlineCustomerImpl implements RestaurantInterface {**

**public void acceptOnlineOrder() {**

**//logic for placing online order**

**}**

**public void acceptTelephoneOrder() {**

**//Not Applicable for Online Order**

**throw new UnsupportedOperationException();**

**}**

**public void payOnline() {**

**//logic for paying online**

**}**

**public void acceptWalkInCustomerOrder() {**

**//Not Applicable for Online Order**

**throw new UnsupportedOperationException();**

**}**

**public void payInPerson() {**

**//Not Applicable for Online Order**

**throw new UnsupportedOperationException();**

**}**

**}**

nce OnlineCustomerImpl.java is for online customers, we will have to throw UnsupportedOperationException for the methods which are not applicable for online customers. This is also termed as ‘Interface Pollution’. Here we can observe clear violation of Interface Segregation Principle.

The implementation classes for Telephonic customer and Walk-in customer will have unsupported methods. Since the 5 methods are part of the RestaurantInterface, the implementation classes have to implement all 5 of them. Any change in any of the methods of the RestaurantInterface will propagate to all implementation classes. Maintenance of code then starts becoming really cumbersome and regression effects of changes will keep increasing.

RestaurantInterface.java also breaks Single Responsibility Principle because the logic for payments as well as that for order placement is grouped together in a single interface

### ****Example#1: Code that follows ISP****

In order to overcome the aforementioned problems, we will apply Interface Segregation Principle to refactor the above design.

Separate out payment and order placement functionalities into two separate lean interfaces, PaymentInterface.java and OrderInterface.java.

**public interface OrderInterface{**

**public void placeOrder();**

**}**

**public interface PaymentInterface{**

**public void payForOrder();**

**}**

Each customer will now implement both interfaces like below:

**public class OnlineCustomerImpl implements OrderInterface, PaymentInterface {**

**@Override**

**public void placeOrder() {**

**// logic to place online order**

**}**

**@Override**

**public void payForOrder() {**

**// logic to do online payment**

**}**

**}**

**public class WalkInCustomerImpl implements OrderInterface, PaymentInterface {**

**@Override**

**public void placeOrder() {**

**// logic to place in-person order**

**}**

**@Override**

**public void payForOrder() {**

**// logic to do in-person payment**

**}**

**}**

**public class TelephoneCustomerImpl implements OrderInterface, PaymentInterface {**

**@Override**

**public void placeOrder() {**

**// logic to place telephonic order**

**}**

**@Override**

**public void payForOrder() {**

**// logic to do online payment**

**}**

**}**

In case of a change in any one of the order, interfaces does not affect the other. Similarly, In case of a change in any one of the payment, interfaces does not affect the other. They are independent now. There will be no need to do any dummy implementation. Even there is no need to throw an UnsupportedOperationException as each interface has only methods that it will always use.

## ****What is SOLID Principles-The Dependency Inversion Principle (DIP) ?****

Dependency Inversion is the strategy of depending upon interfaces or abstract functions and classes rather than upon concrete functions and classes.

This way, instead of high-level modules depending on low-level modules, both will depend on abstractions. Every dependency in the design should target an interface or an abstract class. No dependency should target a concrete class.

### ****Example: Code that violates Dependency Inversion Principle****

Suppose a book store asked us to build a new feature that enables customers to put their favorite books on a shelf.

In order to implement the new functionality, we create a lower-level Book(dependent class as in spring) class and a higher-level Shelf class(target class). The Book class will allow users to see reviews and read a sample of each book they store on their shelves. The Shelf class will let them add a book to their shelf and customize the shelf. For example, observe the below code.

**public class Book {**

**void seeReviews() {**

**...**

**}**

**void readSample() {**

**...**

**}**

**}**

**public class Shelf {**

**Book book;**

**void addBook(Book book) {**

**...**

**}**

**void customizeShelf() {**

**...**

**}**

**}**

Everything looks fine, but as the high-level Shelf class depends on the low-level Book, the above code violates the Dependency Inversion Principle. This becomes clear when the store asks us to enable customers to add DVDs to their shelves, too. In order to fulfil the demand, we create a new DVD class:

**public class DVD {**

**void seeReviews() {**

**...**

**}**

**void watchSample() {**

**...**

**}**

**}**

Now, we should modify the Shelf class so that it can accept DVDs, too. However, this would clearly break the Open/Closed Principle too. we need to inject DVD class also to the shelf class.

### ****Example: Code that follows Dependency Inversion Principle****

The solution is to create an abstraction layer for the lower-level classes (Book and DVD). We’ll do so by introducing the Product interface, both classes will implement it. For example, below code demonstrates the concept.

**public interface Product {**

**void seeReviews();**

**void getSample();**

**}**

**public class Book implements Product {**

**@Override**

**public void seeReviews() {**

**...**

**}**

**@Override**

**public void getSample() {**

**...**

**}**

**}**

**public class DVD implements Product {**

**@Override**

**public void seeReviews() {**

**...**

**}**

**@Override**

**public void getSample() {**

**...**

**}**

**}**

Now, Shelf can reference the Product interface instead of its implementations (Book and DVD). The refactored code also allows us to later introduce new product types (for instance, Magazine) that customers can put on their shelves, too..

**public class Shelf {**

**Product product;**

**void addProduct(Product product) {**

**...**

**}**

**void customizeShelf() {**

**...**

**}**

**}**

The above code also follows the Liskov Substitution Principle, as the Product type can be substituted with both of its subtypes (Book and DVD) without breaking the program. At the same time, we have also implemented the Dependency Inversion Principle, as in the refactored code, high-level classes don’t depend on low-level classes, either. Let’s check with the class diagram in both the cases.

## How is DIP related to Dependency Injection of Spring Framework?

It would be correct if you think that Dependency Inversion Principle is related to [Dependency Injection](https://javatechonline.com/spring-dependency-injection/) as it applies to the Spring Framework. Uncle Bob Martin introduced the concept of Dependency Inversion before Martin Fowler introduced the term Dependency Injection. These both concepts are extremely related. Dependency Inversion is more concentrated on the structure of your code. Moreover, its focus is keeping your code loosely coupled. On the other hand, Dependency Injection is about how the code functionally works.

Dependency Inversion Principle has been very well implemented in Spring framework, the beauty of this design principle is that any class which is injected by DI framework is easy to test with the mock object and easier to maintain because object creation code is centralized in the framework and client code is not messed up with that.

## What is the benefit of Dependency Inversion Principle?

Below are some of the benefits when we apply this principle in our code.

1) Keeps your code loosely coupled  
2) Easier Maintenance  
3) Better Code Reusability

## When should we not use Dependency Inversion Principle?

If our java class has functionality that is more likely to remain unchanged in the future, there is no need to apply this principle as it will not provide us any benefit.