**SOLID Principles**

SOLID Principles

[Introduction 1](#_Toc40133571)

[Single Responsibility Principle 1](#_Toc40133572)

[Open-Close Principle 2](#_Toc40133573)

[Liskov Substitution Principle 4](#_Toc40133574)

[Interface Segregation Principle 6](#_Toc40133575)

[Dependency Inversion Principle 8](#_Toc40133576)

[Summary 12](#_Toc40133577)

# Introduction

The SOLID principles were first conceptualized by Robert C. Martin in his 2000 paper, [*Design Principles and Design Patterns*](https://fi.ort.edu.uy/innovaportal/file/2032/1/design_principles.pdf). These concepts were later built upon by Michael Feathers, who introduced us to the SOLID acronym.

SOLID design principles encourage us to create more maintainable, understandable, and flexible software. Consequently, as our applications grow in size, we can reduce their complexity.

S - Single Responsibility Principle  
O - Open-Closed Principle  
L- Liskov Substitution Principle  
I - Interface Segregation Principle  
D - Dependency Inversion Principle

These principles provide a valuable standard for guiding developers away from “Code Rot” and into applications that provide lasting value for customers and sanity for future developers working on the project.

# Single Responsibility Principle

The Single Responsibility Principle (SRP) states that there should never be more than one reason for a class to change. 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.

Example:

Customer{  
 public List<Item> addItems() {

}  
public long calculateBill(long tax) {

}  
public void generateReport(String reportType) {

}

}

In the above class

* If there is any change in the calculation of bill then we need to change Customer class.
* If you want to add one more report type to generate, then we need to change Customer class.

In order to achieve the Single Responsibility Principle

* Create a new class named BillCalculator and pass Customer object to it. This class will be responsible for calculation of the customer bill.

public class BillCalculator {

public long calculateBill(Customer customer, long tax) {

…

}

}

* Create a new class named Report and pass Customer object to it. This class will be responsible for generation of the customer report

public class ReportGenerator {

public void generateReport(Customer customer, String reportType) {

…

}

}

# Open-Close Principle

The Open-Closed Principle (OCP) states that classes should be open for extension but closed for modification. “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 have developed a class you should never modify it, except to correct bugs.

Ways of extending the class include:

* Inheriting from the class
* Overwriting the required behaviors from the class
* Extending certain behaviors of the class

Strategy Design Pattern is another example of Open Closed design Principle. Service class can use various strategies to perform certain tasks based on requirement so we will keep Service class closed but same time, System is open for extension, by introducing new Strategy which will implement Strategy interface. At runtime, you can call Service Class with any new Strategy, based upon your need

Example: This is to demonstrate the failure of Open-Close Principle

public void processCheckOut(Receipt receipt, String paymentMode) {

double total = 0.0;

List<Item> items = new ArrayList<>();

items.addAll(new Item().addItems());

for (Item item : items) {

total += item.getPrice();

}

if ("Card".equalsIgnoreCase(paymentMode)) {

Payment p = acceptCard(total);

receipt.addPayment(p);

} else {

Payment p = acceptCash(total);

receipt.addPayment(p);

}

}

In the above class if we want to add another payment mode, we should add an if() condition like below, but it violates the Open-Close Principle

public void processCheckOut(Receipt receipt, String paymentMode) {

double total = 0.0;

List<Item> items = new ArrayList<>();

items.addAll(new Item().addItems());

for (Item item : items) {

total += item.getPrice();

}

if ("Card".equalsIgnoreCase(paymentMode)) {

Payment p = acceptCard(total);

receipt.addPayment(p);

} else {

Payment p = acceptCash(total);

receipt.addPayment(p);

}

}

In order to achieve the Open Close Principle,

* Create an CheckoutStrategy interface

public interface CheckoutStrategy {

void processCheckout(Receipt receipt);

}

* Create CardTypeCheckoutStrategy and CashTypeCheckoutStrategy classes and implement CheckoutStrategy interface.

CardTypeCheckoutStrategy.java

public class CardTypeCheckoutStrategy implements CheckoutStrategy {

@Override

public void processCheckout(Receipt receipt) {

…

Payment p = acceptCard(total);

}

private Payment acceptCard(double total) {

Payment payment = new Payment();

payment.setTotalAmount(total);

return payment;

}

}

CashTypeCheckoutStrategy.java

public class CashTypeCheckoutStrategy implements CheckoutStrategy {

@Override

public void processCheckout(Receipt receipt) {

…

Payment p = acceptCash(total);

}

private Payment acceptCash(double total) {

Payment payment = new Payment();

payment.setTotalAmount(total);

return payment;

}

}

# Liskov Substitution Principle

Liskov Substitution Principle stating that every subclass / derived class should be substitutable for their Parent class.

The parent class should be able to refer child objects seamlessly during runtime polymorphism

The Liskov Substitution Principle (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.

This avoids misusing inheritance. It helps us conform to the “is-a” relationship. We can also say that subclasses must fulfill a contract defined by the base class. In this sense, it’s related to Design by Contract that was first described by Bertrand Meyer. For example, it’s tempting to say that a circle is a type of ellipse but circles don’t have two foci or major/minor axes.

A typical example that violates LSP is a Square class that derives from a Rectangle class. The Square class always assumes that the width is equal with the height. If a Square object is used in a context where a Rectangle is expected, unexpected behaviour may occur because the dimensions of a Square cannot (or rather should not) be modified independently.

This problem cannot be easily fixed: if we can modify the setter methods in the Square class so that they preserve the Square invariant (i.e., keep the dimensions equal), then these methods will weaken (violate) the post-conditions for the Rectangle setters, which state that dimensions can be modified independently.

Rectangle.java

public class Rectangle

{

private int length;

private int breadth;

public int getLength()

{

return length;

}

public void setLength(int length)

{

this.length = length;

}

public int getBreadth()

{

return breadth;

}

public void setBreadth(int breadth)

{

this.breadth = breadth;

}

public int getArea()

{

return this.length \* this.breadth;

}

}

}

Square.java

public class Square extends Rectangle

{

public void setBreadth(int breadth)

{

super.setBreadth(breadth);

super.setLength(breadth);

}

public void setLength(int length)

{

setBreadth(length);

}

}

The Square class does not need methods like setBreadth or setLength. The LSPDemo class would need to know the details of derived classes of Rectangle (such as Square) to code appropriately to avoid throwing error. The change in the existing code breaks the open-closed principle in the first place.

In order to solve LSP violation problem,

Create a class called Shape with length and breadth properties and extends Shape class in Rectangle and Square.

Shape.java

public class Shape

{

private int length;

private int breadth;

public int getArea()

{

return this.length \* this.breadth;

}

}

Rectangle.java

public class Rectangle extends Shape

{

public void setLength(int length)

{

this.length = length;

}

public void setBreadth(int breadth)

{

this.breadth = breadth;

}

}

}

Square.java

public class Square extends Shape

{

public void setBreadth(int breadth)

{

this.breadth = breadth;

this.length = length;

}

public void setLength(int length)

{

setBreadth(length);

}

}

# Interface Segregation Principle

The Interface Segregation Principle (ISP) states that clients should not be forced to depend upon interface members they do not use. When we have non-cohesive interfaces, the ISP guides us to create multiple, smaller, cohesive interfaces.

Simply, clients should not be forced to implement unnecessary methods which they will not use.

Example:

Assume we have created simple Set interface as below

public interface Set<E> {

   boolean add(E e);

   boolean contains(Object o);

   E ceiling(E e);

   E floor(E e);

}

Create a class TreeSet.java as below

public class TreeSet implements Set{

@Override

public boolean add(Object e) {

// Implement this method

return false;

}

@Override

public boolean contains(Object o) {

// Implement this method

return false;

}

@Override

public Object ceiling(Object e) {

// Implement this method

return null;

}

@Override

public Object floor(Object e) {

// Implement this method

return null;

}

}

Create another class HashSet.java as below.

public class HashSet implements Set{

@Override

public boolean add(Object e) {

return false;

}

@Override

public boolean contains(Object o) {

return false;

}

@Override

public Object ceiling(Object e) {

// This method is not applicable for HashSet

return null;

}

@Override

public Object floor(Object e) {

// This method is not applicable for HashSet

return null;

}

}

**Even though you do not require ceiling and floor method in HashSet, we have to implement them.**  
The correct solution for above problem will be:  
Create another interface called NavigableSet which will have ceiling and floor method.

public interface NavigableSet<E> {

   E ceiling(E e);

   E floor(E e);

}

and Set interface will be changed as below

public interface Set<E> {

   boolean add(E e);

   boolean contains(Object o);

}

Now TreeSet.java will be going to implement two interfaces Set and NavigableSet. Change TreeSet.java as below

public class TreeSet implements Set,NaviagableSet{

@Override

public boolean add(Object e) {

// Implement this method

return false;

}

@Override

public boolean contains(Object o) {

// Implement this method

return false;

}

@Override

public Object ceiling(Object e) {

// Implement this method

return null;

}

@Override

public Object floor(Object e) {

// Implement this method

return null;

}

}

HashSet will be going to implement only Set as it does not require ceiling and floor methods

public class HashSet implements Set{

@Override

public boolean add(Object e) {

return false;

}

@Override

public boolean contains(Object o) {

return false;

}

}

HashSet does not implement any method which it does not require.

# Dependency Inversion Principle

The Dependency Inversion Principle (DIP) states that high-level modules should not depend upon low-level modules; they should depend on abstractions. Secondly, abstractions should not depend upon details; details should depend upon abstractions. The idea is that we isolate our class behind a boundary formed by the abstractions it depends on. If all the details behind those abstractions change, then our class is still safe. This helps keep coupling low and makes our design easier to change.

Let us again understand it through another practical example.

You go to a local store to buy something, and you decide to pay for it by using your debit 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. Even if you have given a Visa card, he will not put out a Visa machine for swiping your card. The type of credit card or debit card that you have for paying does not even matter; they will simply swipe it. So, in this example, you can see that both you and the clerk are dependent on the credit card abstraction and you are not worried about the specifics of the card.

It allows a programmer to remove hardcoded dependencies so that the application becomes loosely coupled and extendable.

Consider below example without Dependency Inversion Principle

LightBulb.java

public class LightBulb {

public void turnOn() {

System.out.println("LightBulb: Bulb turned on...");

}

public void turnOff() {

System.out.println("LightBulb: Bulb turned off...");

}

}

In the LightBulb class above, we wrote the turnOn() and turnOff() methods to turn a bulb on and off.

ElectricPowerSwitch.java

public class ElectricPowerSwitch {

public LightBulb lightBulb;

public boolean on;

public ElectricPowerSwitch(LightBulb lightBulb) {

this.lightBulb = lightBulb;

this.on = false;

}

public boolean isOn() {

return this.on;

}

public void press(){

boolean checkOn = isOn();

if (checkOn) {

lightBulb.turnOff();

this.on = false;

} else {

lightBulb.turnOn();

this.on = true;

}

}

}

In the example above, we wrote the ElectricPowerSwitch class with a field referencing LightBulb. In the constructor, we created a LightBulb object and assigned it to the field. We then wrote a isOn() method that returns the state of ElectricPowerSwitch as a boolean value. In the press() method, based on the state, we called the turnOn() and turnOff() methods.

Our switch is now ready for use to turn on and off the light bulb. But the mistake we did is apparent. Our high-level ElectricPowerSwitch class is directly dependent on the low-level LightBulb class. if you see in the code, the LightBulb class is hardcoded in ElectricPowerSwitch. But, a switch should not be tied to a bulb. It should be able to turn on and off other appliances and devices too, say a fan, an AC, or the entire lightning system of an amusement park. Now, imagine the modifications we will require in the ElectricPowerSwitch class each time we add a new appliance or device. We can conclude that our design is flawed and we need to revisit it by following the Dependency Inversion Principle

**Solution:**

To follow the Dependency Inversion Principle in our example, we will need an abstraction that both the ElectricPowerSwitch and LightBulb classes will depend on. But, before creating it, let’s create an interface for switches.

Switch.java

public interface Switch {

boolean isOn();

void press();

}

We wrote an interface for switches with the isOn() and press() methods. This interface will give us the flexibility to plug in other types of switches, say a remote control switch later on, if required. Next, we will write the abstraction in the form of an interface, which we will call Switchable.

Switchable.java

public interface Switchable {

void turnOn();

void turnOff();

}

In the example above, we wrote the Switchable interface with the turnOn() and turnoff() methods. From now on, any switchable devices in the application can implement this interface and provide their own functionality. Our ElectricPowerSwitch class will also depend on this interface, as shown below:

ElectricPowerSwitch.java

public class ElectricPowerSwitch implements Switch {

public Switchable client;

public boolean on;

public ElectricPowerSwitch(Switchable client) {

this.client = client;

this.on = false;

}

public boolean isOn() {

return this.on;

}

public void press(){

boolean checkOn = isOn();

if (checkOn) {

client.turnOff();

this.on = false;

} else {

client.turnOn();

this.on = true;

}

}

}

In the ElectricPowerSwitch class we implemented the Switch interface and referred the Switchable interface instead of any concrete class in a field. We then called the turnOn() and turnoff() methods on the interface, which at run time will get invoked on the object passed to the constructor. Now, we can add low-level switchable classes without worrying about modifying the ElectricPowerSwitch class.

We will add two such classes: LightBulb and Fan.

LightBulb.java

public class LightBulb implements Switchable {

@Override

public void turnOn() {

System.out.println("LightBulb: Bulb turned on...");

}

@Override

public void turnOff() {

System.out.println("LightBulb: Bulb turned off...");

}

}

Fan.java

public class Fan implements Switchable {

@Override

public void turnOn() {

System.out.println("Fan: Fan turned on...");

}

@Override

public void turnOff() {

System.out.println("Fan: Fan turned off...");

}

}

In both the LightBulb and Fan classes that we wrote, we implemented the Switchable interface to provide their own functionality for turning on and off. While writing the classes, if you have missed how we arranged them in packages, notice that we kept the Switchable interface in a different package from the low-level electric device classes. Although, this did not make any difference from coding perspective, except for an import statement, by doing so we have made our intentions clear- We want the low-level classes to depend (inversely) on our abstraction. This will also help us if we later decide to release the high-level package as a public API that other applications can use for their devices.

ElectricPowerSwitchMain.java

public class ElectricPowerSwitchMain {

Public static void main() throws Exception {

Switchable switchableBulb=new LightBulb();

Switch bulbPowerSwitch=new ElectricPowerSwitch(switchableBulb);

bulbPowerSwitch.press();

bulbPowerSwitch.press();

Switchable switchableFan=new Fan();

Switch fanPowerSwitch=new ElectricPowerSwitch(switchableFan);

fanPowerSwitch.press();

fanPowerSwitch.press();

}

}

# Summary

|  |  |  |
| --- | --- | --- |
| SRP | [The Single Responsibility Principle](https://docs.google.com/open?id=0ByOwmqah_nuGNHEtcU5OekdDMkk) | A class should have one, and only one, reason to change. |
| OCP | [The Open Closed Principle](http://docs.google.com/a/cleancoder.com/viewer?a=v&pid=explorer&chrome=true&srcid=0BwhCYaYDn8EgN2M5MTkwM2EtNWFkZC00ZTI3LWFjZTUtNTFhZGZiYmUzODc1&hl=en) | You should be able to extend a classes behavior, without modifying it. |
| LSP | [The Liskov Substitution Principle](http://docs.google.com/a/cleancoder.com/viewer?a=v&pid=explorer&chrome=true&srcid=0BwhCYaYDn8EgNzAzZjA5ZmItNjU3NS00MzQ5LTkwYjMtMDJhNDU5ZTM0MTlh&hl=en) | Derived classes must be substitutable for their base classes. |
| ISP | [The Interface Segregation Principle](http://docs.google.com/a/cleancoder.com/viewer?a=v&pid=explorer&chrome=true&srcid=0BwhCYaYDn8EgOTViYjJhYzMtMzYxMC00MzFjLWJjMzYtOGJiMDc5N2JkYmJi&hl=en) | Make fine grained interfaces that are client specific. |
| DIP | [The Dependency Inversion Principle](http://docs.google.com/a/cleancoder.com/viewer?a=v&pid=explorer&chrome=true&srcid=0BwhCYaYDn8EgMjdlMWIzNGUtZTQ0NC00ZjQ5LTkwYzQtZjRhMDRlNTQ3ZGMz&hl=en) | Depend on abstractions, not on concretions. |