**SOLID PRINCIPLES**

The SOLID principles were introduced 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. And in the last 20 years, these five principles have revolutionized the world of object-oriented programming, changing the way that we write software.

So, what is SOLID and how does it help us write better code? Simply put, Martin and Feathers' **design principles encourage us to create more maintainable, understandable, and flexible software.** Consequently, **as our applications grow in size, we can reduce their complexity** and save ourselves a lot of headaches further down the road!

The following five concepts make up our SOLID principles:

1. **S**ingle Responsibility
2. **O**pen/Closed
3. **L**iskov Substitution
4. **I**nterface Segregation
5. **D**ependency Inversion

While these concepts may seem daunting, they can be easily understood with some simple code examples. In the following sections, we'll take a deep dive into these principles, with a quick Java example to illustrate each one

## **Single Responsibility**

Let's begin with the single responsibility principle. As we might expect, this principle states that **a class should only have one responsibility. Furthermore, it should only have one reason to change.**

**How does this principle help us to build better software?** Let's see a few of its benefits:

1. **Testing** – A class with one responsibility will have far fewer test cases.
2. **Lower coupling** – Less functionality in a single class will have fewer dependencies.
3. **Organization** – Smaller, well-organized classes are easier to search than monolithic ones.

For example, let's look at a class to represent a simple book:

public class **Book** {

private String name;

private String author;

private String text;

//constructor, getters and setters

}

## **Open for Extension, Closed for Modification**

It's now time for the O in SOLID, known as the **open-closed principle.** Simply put, **classes should be open for extension but closed for modification.** **In doing so, we** **stop ourselves from modifying existing code and causing potential new bugs** in an otherwise happy application.

Of course, the **one exception to the rule is when fixing bugs in existing code.**

Let's explore the concept with a quick code example. As part of a new project, imagine we've implemented a *Guitar* class.

It's fully fledged and even has a volume knob:

public class **Guitar** {

private String make;

private String model;

private **int** volume;

//Constructors, getters & setters

}

We launch the application, and everyone loves it. But after a few months, we decide the *Guitar* is a little boring and could use a cool flame pattern to make it look more rock and roll.

At this point, it might be tempting to just open up the *Guitar* class and add a flame pattern — but who knows what errors that might throw up in our application.

Instead, let's **stick to the open-closed principle and simply extend our *Guitar* class**:

public class **SuperCoolGuitarWithFlames** extends **Guitar** {

private String flameColor;

//constructor, getters + setters

}

By extending the *Guitar* class, we can be sure that our existing application won't be affected.

## **5. Liskov Substitution**

Next on our list is [Liskov substitution](https://www.baeldung.com/cs/liskov-substitution-principle), which is arguably the most complex of the five principles. Simply put, **if class *A* is a subtype of class *B*, we should be able to replace *B* with *A* without disrupting the behavior of our program.**

Let's jump straight to the code to help us understand this concept:

public interface **Car** {

void **turnOnEngine**();

void **accelerate**();

}

Above, we define a simple *Car* interface with a couple of methods that all cars should be able to fulfill: turning on the engine and accelerating forward.

Let's implement our interface and provide some code for the methods:

public class **MotorCar** implements **Car** {

private Engine engine;

//Constructors, getters + setters

public void **turnOnEngine**() {

//turn on the engine!

engine.on();

}

public void **accelerate**() {

//move forward!

engine.powerOn(1000);

}

}

As our code describes, we have an engine that we can turn on, and we can increase the power.

## **Interface Segregation**

The I in SOLID stands for interface segregation, and it simply means that **larger interfaces should be split into smaller ones. By doing so, we can ensure that implementing classes only need to be concerned about the methods that are of interest to them.**

For this example, we're going to try our hands as zookeepers. And more specifically, we'll be working in the bear enclosure.

Let's start with an interface that outlines our roles as a bear keeper:

public interface **BearKeeper** {

void **washTheBear**();

void **feedTheBear**();

void **petTheBear**();

}

As avid zookeepers, we're more than happy to wash and feed our beloved bears. But we're all too aware of the dangers of petting them. Unfortunately, our interface is rather large, and we have no choice but to implement the code to pet the bear.

Let's **fix this by splitting our large interface into three separate ones**:

## **Dependency Inversion**

**The principle of dependency inversion refers to the decoupling of software modules. This way, instead of high-level modules depending on low-level modules, both will depend on abstractions.**

To demonstrate this, let's go old-school and bring to life a Windows 98 computer with code:

public class **Windows98Machine** {}

But what good is a computer without a monitor and keyboard? Let's add one of each to our constructor so that every *Windows98Computer* we instantiate comes prepacked with a *Monitor* and a *StandardKeyboard*:

public class **Windows98Machine** {

private final StandardKeyboard keyboard;

private final Monitor monitor;

public **Windows98Machine**() {

monitor = new **Monitor**();

keyboard = new **StandardKeyboard**();

}

}

This code will work, and we'll be able to use the *StandardKeyboard* and *Monitor* freely within our *Windows98Computer* class.