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).

The following 5 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

**1. Single Responsibility**

Let's kick things off 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

Take, for example, a class to represent a simple book:

|  |  |
| --- | --- |
|  | **public** **class** Book { |
|  |  |
|  | **private** String name; |
|  | **private** String author; |
|  | **private** String text; |
|  |  |
|  | //constructor, getters and setters |
|  | } |

In this code, we store the name, author, and text associated with an instance of a *Book*.

Let's now add a couple of methods to query the text:

|  |  |
| --- | --- |
|  | **public** **class** Book { |
|  |  |
|  | **private** String name; |
|  | **private** String author; |
|  | **private** String text; |
|  |  |
|  | //constructor, getters and setters |
|  |  |
|  | // methods that directly relate to the book properties |
|  | **public** String replaceWordInText(String word){ |
|  | **return** text.replaceAll(word, text); |
|  | } |
|  |  |
|  | **public** **boolean** isWordInText(String word){ |
|  | **return** text.contains(word); |
|  | } |
|  | } |

Now, our *Book* class works well, and we can store as many books as we like in our application. But, what good is storing the information if we can't output the text to our console and read it?

Let's throw caution to the wind and add a print method:

|  |  |
| --- | --- |
|  | **public** **class** Book { |
|  | //... |
|  |  |
|  | **void** printTextToConsole(){ |
|  | // our code for formatting and printing the text |
|  | } |
|  | } |

This code does, however, violate the single responsibility principle we outlined earlier. To fix our mess, we should implement a separate class that is concerned only with printing our texts:

|  |  |
| --- | --- |
|  | **public** **class** BookPrinter { |
|  |  |
|  | // methods for outputting text |
|  | **void** printTextToConsole(String text){ |
|  | //our code for formatting and printing the text |
|  | } |
|  |  |
|  | **void** printTextToAnotherMedium(String text){ |
|  | // code for writing to any other location.. |
|  | } |
|  | } |

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

Now, time for the ‘O' – more formally 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.

Let's explore the concept further 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. However, after a few months, we decide the *Guitar*is a little bit boring and could do with an awesome flame pattern to make it look a bit 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.

## ****Liskov Substitution****

Next up on our list is [Liskov substitution](https://www.baeldung.com/cs/liskov-substitution-principle), which is arguably the most complex of the 5 principles. Simply put, **if class A is a subtype of class B, then we should be able to replace B with A without disrupting the behavior of our program.**

Let's just jump straight to the code to help wrap our heads around 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.

We are now living in the era of electric cars:

|  |  |
| --- | --- |
|  | **public** **class** ElectricCar **implements** Car { |
|  |  |
|  | **public** **void** turnOnEngine() { |
|  | **throw** **new** AssertionError(**"I don't have an engine!"**); |
|  | } |
|  |  |
|  | **public** **void** accelerate() { |
|  | //this acceleration is crazy! |
|  | } |
|  | } |

## ****5. 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. However, we're all too aware of the dangers of petting them. Unfortunately, our interface is rather large, and we have no choice than to implement the code to pet the bear.

Let's **fix this by splitting our large interface into 3 separate ones:**

|  |  |
| --- | --- |
|  | **public** **interface** BearCleaner { |
|  | **void** washTheBear(); |
|  | } |
|  |  |
|  | **public** **interface** BearFeeder { |
|  | **void** feedTheBear(); |
|  | } |
|  |  |
|  | **public** **interface** BearPetter { |
|  | **void** petTheBear(); |
|  | } |

Now, thanks to interface segregation, we're free to implement only the methods that matter to us:

|  |  |
| --- | --- |
|  | **public** **class** BearCarer **implements** BearCleaner, BearFeeder { |
|  |  |
|  | **public** **void** washTheBear() { |
|  | //I think we missed a spot... |
|  | } |
|  |  |
|  | **public** **void** feedTheBear() { |
|  | //Tuna Tuesdays... |
|  | } |
|  | } |

And finally, we can leave the dangerous stuff to the crazy people:

|  |  |
| --- | --- |
|  | **public** **class** CrazyPerson **implements** BearPetter { |
|  |  |
|  | **public** **void** petTheBear() { |
|  | //Good luck with that! |
|  | } |
|  | } |

## ****6. 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 pre-packed 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(); |
|  | } |
|  |  |
|  | } |

Let's decouple our machine from the *StandardKeyboard* by adding a more general *Keyboard* interface and using this in our class:

|  |  |
| --- | --- |
|  | **public** **interface** Keyboard { } |

|  |  |
| --- | --- |
|  | **public** **class** Windows98Machine{ |
|  |  |
|  | **private** **final** Keyboard keyboard; |
|  | **private** **final** Monitor monitor; |
|  |  |
|  | **public** Windows98Machine(Keyboard keyboard, Monitor monitor) { |
|  | **this**.keyboard = keyboard; |
|  | **this**.monitor = monitor; |
|  | } |
|  | } |

Here, we're using the dependency injection pattern here to facilitate adding the *Keyboard* dependency into the *Windows98Machine* class.

Let's also modify our *StandardKeyboard* class to implement the *Keyboard* interface so that it's suitable for injecting into the *Windows98Machine* class:

|  |  |
| --- | --- |
|  | **public** **class** StandardKeyboard **implements** Keyboard { } |