SOLID Principle

The Reason for SOLID Principles

https://www.baeldung.com/solid-principles

So, what is SOLID and how does it help us write better code? Simply put, Martin's 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 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

Single Responsibility

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 build better software?

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

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

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:

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:

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 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:

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**:**
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, 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 behaviour of our program.**

Let's just jump straight to the code to help wrap our heads around this concept:

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:

As our code describes, we have an engine that we can turn on, and we can increase the power. But wait, its 2019, and Elon Musk has been a busy man.

We are now living in the era of electric cars:

By throwing a car without an engine into the mix, we are inherently changing the behavior of our program. This is **a blatant violation of Liskov substitution and is a bit harder to fix than our previous 2 principles**.

One possible solution would be to rework our model into interfaces that take into account the engine-less state of our *Car*.

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:

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:**

Now, thanks to interface segregation, we're free to implement only the methods that matter to us:
And finally, we can leave the dangerous stuff to the crazy people:

Going further, we could even split our *BookPrinter* class from our example earlier to use interface segregation in the same way. By implementing a *Printer* interface with a single *print* method, we could instantiate separate *ConsoleBookPrinter* and *OtherMediaBookPrinter* classes.

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:

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*

This code will work, and we'll be able to use the *StandardKeyboard* and *Monitor* freely within our *Windows98Computer* class. Problem solved? Not quite. **By declaring the *StandardKeyboard* and *Monitor* with the *new* keyword, we've tightly coupled these 3 classes together.**

Not only does this make our *Windows98Computer* hard to test, but we've also lost the ability to switch out our *StandardKeyboard* class with a different one should the need arise. And we're stuck with our *Monitor* class, too.

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

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:

Now our classes are decoupled and communicate through the *Keyboard* abstraction. If we want, we can easily switch out the type of keyboard in our machine with a different implementation of the interface. We can follow the same principle for the *Monitor* class.

Excellent! We've decoupled the dependencies and are free to test our *Windows98Machine* with whichever testing framework we choose.