**SOLID principles**

SOLID Principles is a coding standard that all developers should have a clear concept for developing software properly to avoid a bad design. It was promoted and first conceptualized by Robert C Martin in his 2000 paper, Design Principles and Design Patterns. and is used across the object-oriented design spectrum. When applied properly it makes your code more extendable, logical, and easier to read.

These concepts were later built upon by Michael Feathers, who introduced us to the SOLID acronym. And in the last 20 years, these 5 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?

When the developer builds software following a bad design, the code can become inflexible and more brittle. Small changes in the software can result in bugs. For these reasons, we should follow SOLID Principles.

It takes some time to understand, but if you write code following these principles, it will improve code quality and help you understand the most well-designed software.

To understand SOLID principles, you have to know the use of the interface clearly.

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 **each class should have** **one responsibility, one single purpose**. This means that a class will do only one job, which leads us to conclude it should have **only one reason to change**.

We don’t want objects that know too much and have unrelated behavior. These classes are harder to maintain.

For example, if we have a class that we change a lot, and for different reasons, then this class should be broken down into more classes, each handling a single concern. Surely, **if an error occurs, it will be easier to find.**

Let’s consider a class that contains code that changes the text in some way. The only job of this class should be **manipulating text**.

public class TextManipulator {

private String text;

public TextManipulator(String text) {

this.text = text;

}

public String getText() {

return text;

}

public void appendText(String newText) {

text = text.concat(newText);

}

public String findWordAndReplace(String word, String replacementWord) {

if (text.contains(word)) {

text = text.replace(word, replacementWord);

}

return text;

}

public String findWordAndDelete(String word) {

if (text.contains(word)) {

text = text.replace(word, "");

}

return text;

}

public void printText() {

System.out.println(textManipulator.getText());

}

}

Although this may seem fine, it is not a good example of the SRP. Here we have **two** **responsibilities**: **manipulating and printing the text**.

Having a method that prints out text in this class violate the Single Responsibility Principle. For this purpose, we should create another class, which will only handle printing text

public class TextPrinter {

TextManipulator textManipulator;

public TextPrinter(TextManipulator textManipulator) {

this.textManipulator = textManipulator;

}

public void printText() {

System.out.println(textManipulator.getText());

}

public void printOutEachWordOfText() {

System.out.println(Arrays.toString(textManipulator.getText().split(" ")));

}

public void printRangeOfCharacters(int startingIndex, int endIndex) {

System.out.println(textManipulator.getText().substring(startingIndex, endIndex));

}

}

Now, in this class, we can create methods for as many variations of printing text as we want, because that's its job.

What this means is that sometimes only we, as designers of our application, can decide if something is in the scope of a class or not.

When writing a class according to the SRP principle, we have to think about the problem domain, business needs, and application architecture.

**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.

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:

|  |  |
| --- | --- |
|  | **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****

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 behavior of our program.**

 It states that “[subclasses](https://web.archive.org/web/20150906155800/http:/www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf) should be substitutable for their base classes“, meaning that code expecting a certain class to be used should work if passed any of this class’ subclasses.

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); |
|  | } |
|  | } |

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! |
|  | } |
|  | } |

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:

|  |  |
| --- | --- |
|  | **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!

}

}

## ****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(); |
|  | } |
|  |  |
|  | } |

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:

|  |  |
| --- | --- |
|  | **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 { } |

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.

we've taken a **deep dive into the SOLID principles of object-oriented design.**

We **started with a quick bit of SOLID history and the reasons these principles exist.**

Letter by letter, we've **broken down the meaning of each principle with a quick code example that violates it. We then saw how to fix our code**and make it adhere to the SOLID principles.