SOLID Principles
DRY & YAGNI

SOLID Principles: Introduction

The SOLID principles were first conceptualized by Robert C. Martin in his 2000 paper, *Design Principles and Design Patterns*.

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?

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 .**

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 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

Single Responsibility

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*.

```
public class Book {

    private String name;
    private String author;
    private String text;

    //constructor, getters and setters
}
```

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);
    }
}
```

Single Responsibility

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?

```
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..
    }
}
```

Not only have we developed a class that relieves the *Book* of its printing duties, but we can also leverage our *BookPrinter* class to send our text to other media.

Whether it's email, logging, or anything else, we have a separate class dedicated to this one concern.

Open for Extension, Closed for Modification

Open for Extension, Closed for Modification

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.

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
}
```

Open for Extension, Closed for Modification

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 FlameGutar 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

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.

Which means, Derived class objects must be substitutable for the base class objects. That means objects of the derived class must behave in a manner consistent with the promises made in the base class contract.

In short it means, *Derived class objects should* complement*, not* substitute *base class behavior*

Classic examples are *Square* class extending *Rectangle class* and the other one is *Circle* class extending *Ellipse* class.

Interface Segregation Principle

Clients should not be forced to depend on methods that they do not use.

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.**

Example

```
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.

Interface Segregation Principle

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();
}
```

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 Principle

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.

Good practice is anything that reduces coupling, which improves testability, maintainability, and replace-ability.

Dependency Inversion Principle

A. High level modules should not depend upon low level modules. Both should depend upon abstractions.

B. Abstractions should not depend upon details. Details should depend upon abstractions.

```
//Tight coupling
public class Windows98Machine {

    private final StandardKeyboard keyboard;
    private final Monitor monitor;

    public Windows98Machine() {
        monitor = new Monitor();
        keyboard = new StandardKeyboard();
    }
}
```

By declaring the *StandardKeyboard* and *Monitor* with the *new* keyword, we've tightly coupled these 3 classes together.

Dependency Inversion Principle

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.

```
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;
    }
}
```

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

DRY & WET

Don’t’ **R**epeat **Y**ourself (DRY) formulated by Andy Hunt and Dave Thomas in their book *The Pragmatic Programmer*

Don't repeat yourself (**DRY**, or sometimes **do not repeat yourself**) is a principle of software development aimed at reducing repetition of software patterns replacing it with abstractions or using data normalization to avoid redundancy.

It says every piece of knowledge must have a single, unambiguous, authoritative representation within a system.

They apply it quite broadly to include "database schemas, test plans, the build system, even documentation".

Write Everything Twice (WET) is a cheeky abbreviation to mean the opposite i.e. code that doesn’t adhere to DRY principle.

YAGNI

YAGNI is an acronym for ***You Aren't Gonna Need It*.**

It is a mantra from ExtremeProgramming that's often used generally in agile software teams. It's a statement that some capability we presume our software needs in the future should not be built now because "you aren't gonna need it".

It states a programmer should not add functionality until deemed necessary. i.e. Implement things when you actually need them, never when you just foresee that you need them.

The most common contention is *timing*. We continually write code sooner than we *actually need* them. This is the **over-engineer** in us. We confuse *foreseeing* with *needing*.

https://martinfowler.com/bliki/Yagni.html

Thank You!