Mosh Hamedani

Coding Made Simple

June 10th, 2019
Comments

SOLID – 5 Principles Of Object Oriented Design Every Developer Must Learn

Connect with Me

You may have come across the term SOLID principles in your programming career. SOLID principles are a set of five principles that ensure good Object Oriented Design. SOLID principles were introduced by Robert C.Martin, otherwise called "Uncle Bob".

In this post, we will go over all the five SOLID principles and how they can be used in any OOP language of your choice.

Let's jump right in.

What does SOLID stand for?

1. **S** – Single responsibility principle
2. **O** – Open closed principle
3. **L** – Liskov substitution principle
4. **I** – Interface segregation principle
5. **D** – Dependency Inversion principle

Now, let's look into each of the principle and how they can be achieved in JavaScript.

Principle 1 – Single Responsibility Principle

> A class should have one and only one reason to change, meaning that a class should have only one job.

This is by far the most important principle in my opinion, to write clean, maintainable and extensible code. This principle assures the importance of writing classes/functions that do only one thing. This is a common advice that you will hear from architects in the industry. You may have got feedback from your code reviewers, that a certain piece of function/module needs to be refactored into smaller functions/modules for better maintainability and readability. This is indeed the Single Responsibility Principle.

> When your function is doing more than one thing, it is time to create another function.

This sounds easy, but sometimes harder to keep up with.

Why is it useful?

- **Readability** – As your applications grow in size and complexity, readability becomes one of the top priorities. Code that is not readable, will lead to several failure points. The Single Responsibility Principle, ensures that your code is clean and readable at all times.
- **Testability** – Breaking down your code into small modules, that do only one thing, makes them easy to test.
- **Reusability** – Your code is now tested, and clean which means that they can be reused in several parts of your code

Data Structures
Django
Flutter
General
Interviews
Java
JavaScript
Mobile
Python
React
React Native
Redux
Redux
Svelte

Popular Posts

53 Python Exercises and Questions for Beginners
React file upload: proper and easy way, with NodeJS!

reused in several parts of your code.

- **Maintainability –** Code written with this principle in mind is easy to maintain on a long run.

Let's look into a simple example to make more sense of this principle.

```
class Person {
    constructor(name: string){ }
    getPersonName() { }
    savePerson(p: Person) { }
}
```

The class above is a *Person* class. If you take a brief look at it, you will realise that it does not honor the Single Responsibility Principle.

Why?

The class has both Person properties and Person database management functions. The function *savePerson,* is a database action, that need not be a part of the generic *Person* class. Let's refactor this, by separating the concerns, so that the class does only one thing.

```
class Person {
    constructor(name: string){ }
    getPersonName() { }
}
class PersonDB {
    getPerson(p: Person) { }
    savePerson(p: Person) { }
}
```

Does the above snippet look better? We have separated the *Person* class and the *PersonDB* class, ensuring the Single Responsibility Principle.

> When designing our classes, we should aim to put related features together, so whenever they tend to change they change for the same reason. And we should try to separate features if they will change for different reasons. – Steve Fenton

Keep this in mind, the next time you are coding and make sure that you try your best to seperate the concerns.

Principle 2 – Open Closed Principle

Alright, the next principle that we are going to look into is the Open Closed Principle. This may sound confusing to some, but it really is not that hard to learn. Let's dive in.

> Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.

The idea behind this principle, is to ensure that the functions/classes have scope for extension in the future. New features can be added to it, without introducing new bugs. You'll add new features by extending or writing new code without modifying old code.

Why is it useful?

- **Agile –** By being open to adding new features, the development process is agile. Much time won't be spent on refactoring the code to add new features.
- **Reliable –** This principle ensures reliability, that adding new features will not introduce new bugs, since the code is closed to modification.

If you start coding with this principle in your mind, you will produce high quality

React vs. Angular: The Complete Comparison

React Lifecycle Methods – A Deep Dive

How to use localStorage with React

4 Common Mistakes with the Repository Pattern

5 C# Collections that Every C# Developer Must Know

If you start coding, with this principle in your mind, you will produce high quality bug-free code that will be appreciated over time.

Let's look at an example of how this principle can be achieved. In the class *TicketPrice* below, we are adding a function to give a discounted ticket price to some customers. The discount varies based on the customer. Very soon, this piece of code can get complex, with a whole bunch of if-else loops. This does not follow the Open-Closed Principle.

```
class TicketPrice {
    giveDiscountedPrice() {
        if(customer == 'vip') {
            return price * 0.25;
        }
        if(customer == 'family') {
            return price * 0.5;
        }
    }
}
```

A better way to write this same code would be to extend from the *TicketPrice* class.

```
class VIPDiscount: TicketPrice {
    getDiscount() {
        return super.giveDiscountedPrice() * 0.25;
    }
}

class FamilyDiscount: TicketPrice {
    getDiscount() {
        return super.giveDiscountedPrice() * 0.5;
    }
}
```

In the above example, we have extended *VIPDiscount* and *FamilyDiscount* from the base *TicketPrice* class. Now, this ensures that we can add more types of discounts, by simple creating new classes that extend from the base *TicketPrice* class. We have also achieved the closed principle by ensuring that there are no modifications to the base functionality/class.

Principle 3 – Liskov Substitution Principle

> Let q(x) be a property provable about objects of x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T.

Alright, this definition does not really help us. Trust me this principle is not as hard to understand as it's definition.

A classic example that I found explaining this principle on StackOverflow is elaborated below:

```
class Bird{
     void fly() {
     //
    }
}
class Eagle : Bird {
 //
}
class Ostrich : Bird {
 //
}
```

In the example above, we have a base class *Bird*, it has the method *fly()*. Now, specific birds can inherit from this base class. We have an *Eagle* sub-class which inherits from the base class *Bird*. This will work out fine, since the eagle is a bird that can fly. The

the base class *Bird*. This will work out fine, since the eagle is a bird that can fly. The next sub-class we have is *Ostrich.* Here is the catch, although Ostrich is a bird it cannot fly. So it cannot use the inherited *fly()* method.

This is where the Liskov Substitution Principle, comes into play.

Let's refactor this code snippet according to the principle.

```
class Bird{
}
class FlyingBirds : Bird{
    void fly(){
     //
    }
}
class Eagle : FlyingBirds {
 //
}
class Ostrich : Bird{
 //
}
```

We now have a new class *FlyingBirds* that inherits from the base *Bird* class which has a *fly()* method. Now we have taken care of the situation that not all birds can fly. Flying birds can inherit from the *FlyingBirds* class.

Principle 4 – Interface Segregation Principle

> A client should never be forced to implement an interface that it doesn't use or clients shouldn't be forced to depend on methods they do not use.

The next SOLID principle aims to solve the problems of dealing with large interfaces that are hard to use by clients. Here the client is something that is asking for some information from another class. This is one of the easier principles to understand and implement in real-life code scenarios.

Have you seen legacy code, where a class implements an interface, but has several methods/functions that it does not need? You would typically have to define those methods with an empty return statement. This is what the Interface Segregation Principle is trying to get rid of.

This principle is hard to illustrate with code snippets, because it makes more sense with a large code base with real examples. Let me try to showcase it with a simple example anyway.

Take a look at the interface below.

```
interface IColors {
    paintRed();
    paintYellow();
    paintBlue();
}
```

This interface has defined three methods, to paint different colors.

I am going to have classes that implement the IColors interface.

```
class Red implements IColors{
    paintRed() {
      // implementation details to paint red
    }
    paintYellow() {
     // this class doesn't need this method, but still needs an empty implementation.
    }
    paintBlue() {
```

```
    // this class doesn't need this method, but still needs an empty implementation.
    }
}
```

You can see above that our class *Red* represents the red color and implements the *IColors* interface. But there are many methods that are part of the interface that this class does not need. The same thing will happen to the other classes that are specific to a certain color, when they implement the *IColors* interface.

This is what the Interface Segregation Principle is trying to solve. It aims towards keeping your interfaces as simple and specific as possible. Let's refactor our example to follow the Interface Segregation Principle.

```
interface IColors {
    paint();
}
interface IRed {
    paintRed();
}
interface IYellow {
    paintYellow();
}
interface IBlue {
    paintBlue();
}

class Red implements IRed {
    paintRed() {
        //...
    }
}
class Yellow implements IYellow {
    paintYellow() {
        //...
    }
}
class Blue implements IBlue {
    paintBlue() {
        //...
    }
}

class CustomColor implements IColors {
   paint(){
      //...
   }
}
```

Doesn't this look much better? We don't have to have implementation details for methods that the class doesn't need. Keeping interfaces segregated leads to better design and can make a huge difference in larger code bases.

Principle 5 – Dependency Inversion Principle

Alright, last but not the least we have come to the final SOLID principle. Let's look at it's definition.

> Entities must depend on abstractions not on concretions. It states that the high level module must not depend on the low level module, but they should depend on abstractions.

Let's take a look at the example given by Uncle Bob in his SOLID Principles explanation. Here the *Button* which is a higher level object is depending on a lower level object the *Lamp.* The *Lamp* has a specific *TurnOn()* and *TurnOff()* methods. This is against the Dependency Inversion Principle. We would need to introduce abstractions to better solve this dependency.

abstractions to better solve this dependency.

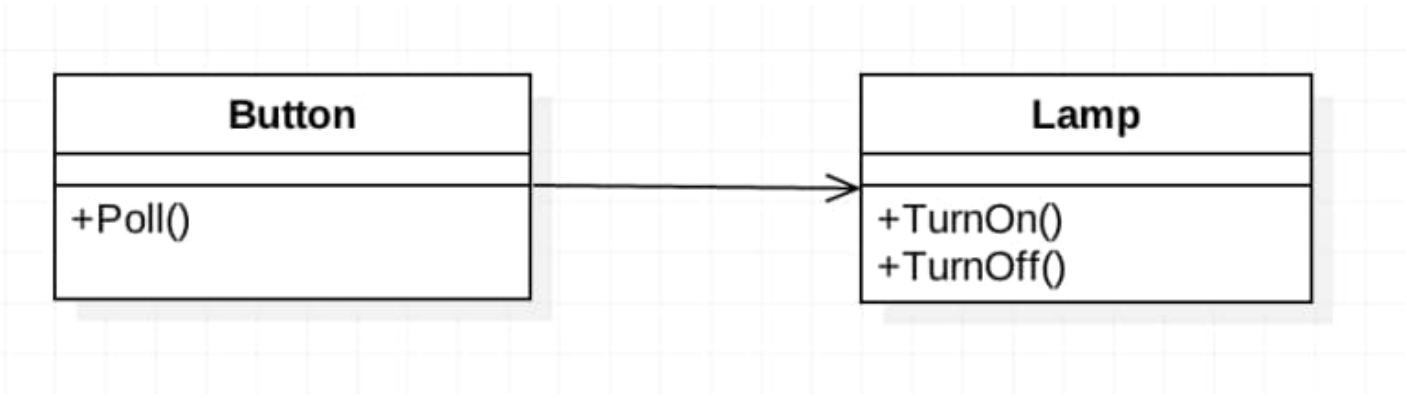

Image source: stackoverflow.com

Let's now see how he solves this with the introduction of an interface. The interface *SwitchableDevice* has the *TurnOn()* and *TurnOff()* methods. Now . the *Button* class does not need to know or interact with the exact device that it is switching on/off. All it does is interact with the *SwitchableDevice* interface. Now different devices can implement the *SwithcableDevice* interface. You can include fans, remotes, lamps, etc.. The *Button* would not need to know about any of these, and hence is completely decoupled from the lower level modules. We want the low-level classes like the *Lamp* to depend (inversely) on our abstraction *(SwitchableDevice)*. And now we are honoring the Dependency inversion principle.

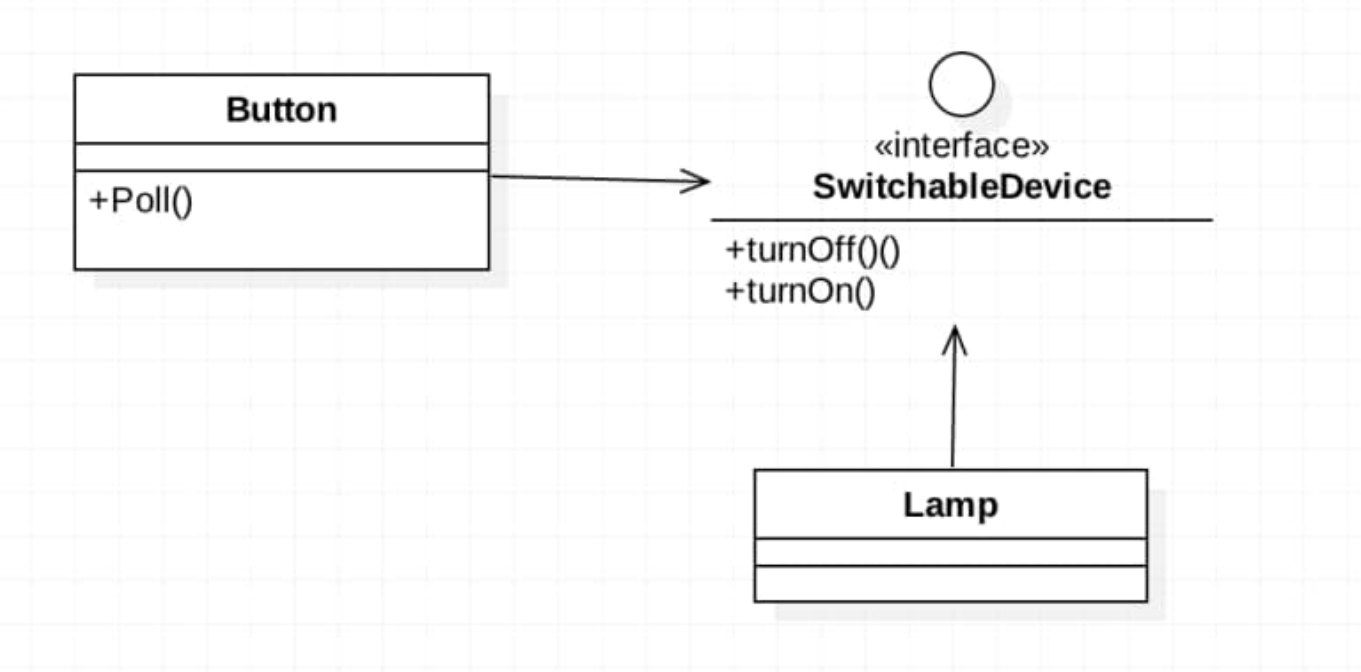

Image source: stackoverflow.com

Conclusion

The SOLID Principles, are some of the most important set of principles to learn and implement, if you are following Object Oriented Design. It may seem a bit overwhelming initially, but with practice and time you will be coding based on these principles, without even realizing it.

If you are looking for a good course on Object Oriented Programming in JavaScript

If you are looking for a good course on Object Oriented Programming in JavaScript, checkout Mosh's course on this topic from the link below.

[Mosh's Object Oriented Programming in JavaScript Course.]

If you liked this post, please share it with your friends and colleagues.

Adhithi Ravichandran

Adhithi Ravichandran is a Software Consultant based in Kansas City. She is currently working on building apps with React, React Native and GraphQL. She is a Conference Speaker, Pluralsight Author, Blogger and Software Consultant. She is passionate about teaching and thrives to contribute to the tech community with her courses, blog posts and speaking engagements.

Share this:

Related

Top Design Patterns to master in Node/TypeScript
July 12, 2022
In "Backend"

Top 3 Reads for C# Developers
December 15, 2015
In "C# / .NET"

React vs. Angular: The Complete Comparison
November 5, 2018
In "Angular"

Tags: javascript, SOLID

2 responses to "SOLID – 5 Principles Of Object Oriented Design Every Developer Must Learn"

1. *Bill* says:
   June 12, 2019 at 12:46 am

   I would like to see this done for functional programming. OOP is kind of dying in the ES world.

   Reply

2. *DBJ* says:

June 16, 2019 at 4:14 pm

Can't resist to chime in 🙂 Above is a fortunate example how younger generations do know what is it not to be forgoten from the OO heritage.

Although, in practice the key problem is inheritance gone wrong. Or should we say gone wild. And it is very simple to control it:

Never (ever) use any form of inheritance but subtyping.

Again. Nice to see OO is understood. But OO is not panacea. Use in moderation.

Reply

Leave a Reply

Enter your comment here...