SOLID Principles every Developer Should Know

Chidume Nnamdi 🔥💻🎵🎮 Follow
Oct 9, 2018 · 11 min read

Object-Oriented type of programming brought a new design to software development.

This enables developers to combine data with the same purpose/functionality in one class to deal with the sole purpose there, regardless of the entire application.

But, this Object-oriented programming doesn't prevent confusing or unmaintainable programs.

As such, five guidelines were developed by Robert C. Martin. These five guidelines/principles made it easy for developers to create readable and maintainable programs.

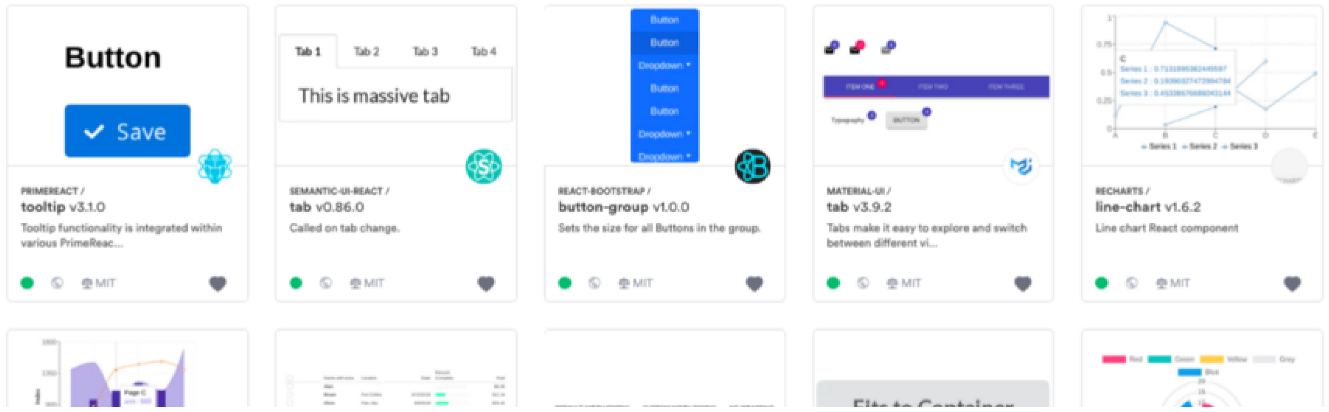
These five principles were called the S.O.L.I.D principles (the acronym was derived by Michael Feathers).

- S: Single Responsibility Principle
- O: Open-Closed Principle
- L: Liskov Substitution Principle
- I: Interface Segregation Principle
- D: Dependency Inversion Principle

We will discuss them in detail below.

Note: Most of the examples in this article, may not suffice as/for the real thing or not applicable in real world applications. It all depends on your own design and use case. The **most important** thing is to understand and know how to apply/follow the principles.

Tip: Use tools like **Bit** **(GitHub)** to easily share and reuse components (and small modules) across projects and applications. It also helps you and your team save time, stay in sync and build faster together. It's free, give it a try.



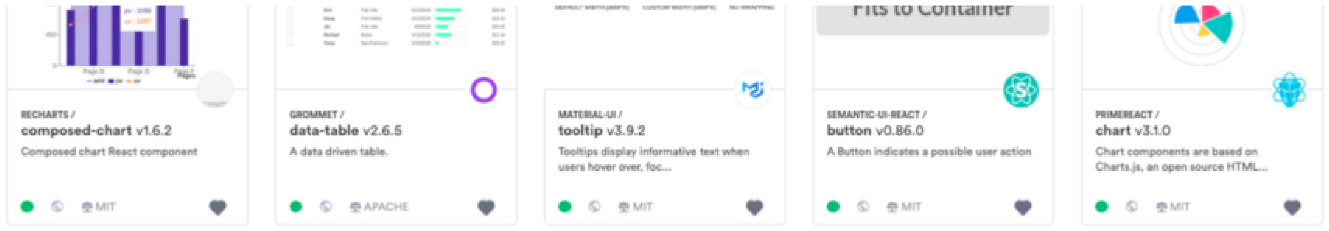
PRIMEREACT / **tooltip** v3.1.0
Tooltip functionality is integrated within various PrimeReac...
MIT

SEMANTIC-UI-REACT / **tab** v0.86.0
Called on tab change.
MIT

REACT-BOOTSTRAP / **button-group** v1.0.0
Sets the size for all Buttons in the group.
MIT

MATERIAL-UI / **tab** v3.9.2
Tabs make it easy to explore and switch between different vi...
MIT

RECHARTS / **line-chart** v1.6.2
Line chart React component
MIT


RECHARTS /
composed-chart v1.6.2
Composed chart React component
MIT
GROMMET /
data-table v2.6.5
A data driven table.
APACHE
MATERIAL-UI /
tooltip v3.9.2
Tooltips display informative text when users hover over, foc...
MIT
SEMANTIC-UI-REACT /
button v0.86.0
A Button indicates a possible user action
MIT
PRIMEREACT /
chart v3.1.0
Chart components are based on Charts.js, an open source HTML...
MIT

Easily share components across apps and projects

Component Discovery and Collaboration · Bit

Bit is where developers share components and collaborate to build amazing software together. Discover components shared…

bit.dev

Single Responsibility Principle

> "…You had one job" — Loki to Skurge in Thor: Ragnarok
>
> A class should have only one job.

A class should be responsible for only one thing. If a class has more than one responsibility, it becomes coupled. A change to one responsibility results to modification of the other responsibility.

- **Note**: This principle applies not only to classes, but also to software components and microservices.

For example, consider this design:

```
class Animal {
    constructor(name: string){ }
    getAnimalName() { }
    saveAnimal(a: Animal) { }
}
```

The Animal class violates the SRP.

How does it violate SRP?

SRP states that classes should have one responsibility, here, we can draw out two responsibilities: animal database management and animal properties management. The constructor and getAnimalName manage the Animal properties while the saveAnimal manages the Animal storage on a database.

How will this design cause issues in the future?

If the application changes in a way that it affects database management functions. The classes that make use of Animal properties will have to be touched and recompiled to compensate for the new changes.

You see this system smells of rigidity, it’s like a domino effect, touch one card it affects all other cards in line.

To make this conform to SRP, we create another class that will handle the sole responsibility of storing an animal to a database:

```
class Animal {
    constructor(name: string){ }
    getAnimalName() { }
}

class AnimalDB {
    getAnimal(a: Animal) { }
    saveAnimal(a: Animal) { }
}
```

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

With the proper application of these, our application becomes highly cohesive.

Open-Closed Principle

Software entities(Classes, modules, functions) should be open for extension, not modification.

Let’s continue with our Animal class.

```
class Animal {
    constructor(name: string){ }
    getAnimalName() { }
}
```

We want to iterate through a list of animals and make their sounds.

```
//...
const animals: Array<Animal> = [
    new Animal('lion'),
    new Animal('mouse')
];

function AnimalSound(a: Array<Animal>) {
    for(int i = 0; i <= a.length; i++) {
        if(a[i].name == 'lion')
            log('roar');
        if(a[i].name == 'mouse')
            log('squeak');
    }
}
AnimalSound(animals);
```

The function AnimalSound does not conform to the open-closed principle because it cannot be closed against new kinds of animals.

If we add a new animal, Snake:

```
//...
const animals: Array<Animal> = [
    new Animal('lion'),
    new Animal('mouse'),
    new Animal('snake')
]
//...
```

We have to modify the AnimalSound function:

```
//...
function AnimalSound(a: Array<Animal>) {
    for(int i = 0; i <= a.length; i++) {
        if(a[i].name == 'lion')
            log('roar');
        if(a[i].name == 'mouse')
            log('squeak');
        if(a[i].name == 'snake')
            log('hiss');
    }
}

AnimalSound(animals);
```

You see, for every new animal, a new logic is added to the AnimalSound function. This is quite a simple example. When your application grows and becomes complex, you will see that the `if` statement would be repeated over and over again in the `AnimalSound` function each time a new animal is added, all over the application.

How do we make it (the AnimalSound) conform to OCP?

```
class Animal {
        makeSound();
        //...
}

class Lion extends Animal {
    makeSound() {
        return 'roar';
    }
}

class Squirrel extends Animal {
    makeSound() {
        return 'squeak';
    }
}
class Snake extends Animal {
    makeSound() {
        return 'hiss';
    }
}

//...
function AnimalSound(a: Array<Animal>) {
    for(int i = 0; i <= a.length; i++) {
        log(a[i].makeSound());
    }
}
AnimalSound(animals);
```

Animal now has a virtual method `makeSound`. We have each animal extend the Animal class and implement the virtual makeSound method.

Every animal adds its own implementation on how it makes a sound in the makeSound. The AnimalSound iterates through the array of animal and just calls its makeSound method.

Now, if we add a new animal, AnimalSound doesn’t need to change. All we need to do is add the new animal to the animal array.

AnimalSound now conforms to the OCP principle.

Another example:

Let’s imagine you have a store, and you give a discount of 20% to your favorite customers using this class:

```
class Discount {
    giveDiscount() {
        return this.price * 0.2
    }
}
```

When you decide to offer double the 20% discount to VIP customers. You may modify the class like this:

```
class Discount {
    giveDiscount() {
        if(this.customer == 'fav') {
            return this.price * 0.2;
        }
        if(this.customer == 'vip') {
            return this.price * 0.4;
        }
    }
}
```

No, this fails the OCP principle. OCP forbids it. If we want to give a new percent discount maybe, to a diff. type of customers, you will see that a new logic will be added.

To make it follow the OCP principle, we will add a new class that will extend the Discount. In this new class, we would implement its new behavior:

```
class VIPDiscount: Discount {
    getDiscount() {
        return super.getDiscount() * 2;
    }
}
```

If you decide 80% discount to super VIP customers, it should be like this:

```
class SuperVIPDiscount: VIPDiscount {
    getDiscount() {
        return super.getDiscount() * 2;
    }
}
```

You see, extension without modification.

Liskov Substitution Principle

A sub-class must be substitutable for its super-class

The aim of this principle is to ascertain that a sub-class can assume the place of its super-class without errors. If the code finds itself checking the type of class then, it must have violated this principle.

Let’s use our Animal example.

```
//...
function AnimalLegCount(a: Array<Animal>) {
    for(int i = 0; i <= a.length; i++) {
        if(typeof a[i] == Lion)
            log(LionLegCount(a[i]));
        if(typeof a[i] == Mouse)
            log(MouseLegCount(a[i]));
        if(typeof a[i] == Snake)
            log(SnakeLegCount(a[i]));
    }
}
AnimalLegCount(animals);
```

This violates the LSP principle, (and also the OCP principle). It must know of every Animal type and call the associated `leg-counting` function.

With every new creation of an animal, the function must modify to accept the new animal.

```
//...
class Pigeon extends Animal {

}

const animals[]: Array<Animal> = [
    //...,
    new Pigeon();
]

function AnimalLegCount(a: Array<Animal>) {
    for(int i = 0; i <= a.length; i++) {
        if(typeof a[i] == Lion)
            log(LionLegCount(a[i]));
        if(typeof a[i] == Mouse)
            log(MouseLegCount(a[i]));
         if(typeof a[i] == Snake)
            log(SnakeLegCount(a[i]));
        if(typeof a[i] == Pigeon)
            log(PigeonLegCount(a[i]));
    }
}
AnimalLegCount(animals);
```

To make this function follow the LSP principle, we will follow this LSP requirements postulated by Steve Fenton:

- If the super-class (Animal) has a method that accepts a super-class type (Animal) parameter. Its sub-class(Pigeon) should accept as argument a super-class type (Animal type) or sub-class type(Pigeon type).
- If the super-class returns a super-class type (Animal). Its sub-class should return a super-class type (Animal type) or sub-class type(Pigeon).

Now, we can re-implement AnimalLegCount function:

```
function AnimalLegCount(a: Array<Animal>) {
    for(let i = 0; i <= a.length; i++) {
        a[i].LegCount();
    }
}
AnimalLegCount(animals);
```

The AnimalLegCount function cares less the type of Animal passed, it just calls the LegCount method. All it knows is that the parameter must be of an Animal type, either the Animal class or its sub-class.

The Animal class now have to implement/define a LegCount method:

```
class Animal {
    //...
    LegCount();
}
```

And its sub-classes have to implement the LegCount method:

```
//...
class Lion extends Animal{
    //...
    LegCount() {
        //...
    }
}
//...
```

When it's passed to the AnimalLegCount function, it returns the number of legs a lion has.

You see, the AnimalLegCount doesn't need to know the type of Animal to return its leg count, it just calls the LegCount method of the Animal type because by contract a sub-class of Animal class must implement the LegCount function.

Interface Segregation Principle

> Make fine grained interfaces that are client specific
>
> Clients should not be forced to depend upon interfaces that they do not use.

This principle deals with the disadvantages of implementing big interfaces.

Let's look at the below IShape interface:

```
interface IShape {
    drawCircle();
    drawSquare();
    drawRectangle();
}
```

This interface draws squares, circles, rectangles. class Circle, Square or Rectangle implementing the IShape interface must define the methods drawCircle(), drawSquare(),drawRectangle().

```
class Circle implements IShape {
    drawCircle(){
        //...
    }

    drawSquare(){
        //...
    }

    drawRectangle(){
        //...
    }
}

class Square implements IShape {
    drawCircle(){
        //...
    }

    drawSquare(){
        //...
    }

    drawRectangle(){
        //...
    }
}

class Rectangle implements IShape {
    drawCircle(){
        //...
    }

    drawSquare(){
        //...
    }

    drawRectangle(){
        //...
    }
}
```

It’s quite funny looking at the code above. class Rectangle implements methods (drawCircle and drawSquare) it has no use of, likewise Square implementing drawCircle, and drawRectangle, and class Circle (drawSquare, drawSquare).

If we add another method to the IShape interface, like drawTriangle(),

```
interface IShape {
    drawCircle();
    drawSquare();
    drawRectangle();
    drawTriangle();
}
```

the classes must implement the new method or error will be thrown.

We see that it is impossible to implement a shape that can draw a circle but not a rectangle or a square or a triangle. We can just implement the methods to throw an error that shows the operation cannot be performed.

ISP frowns against the design of this IShape interface. clients (here Rectangle, Circle, and Square) should not be forced to depend on methods that they do not need or use. Also, ISP states that interfaces should perform only one job (just like the SRP principle) any extra grouping of behavior should be abstracted away to another interface.

Here, our IShape interface performs actions that should be handled independently by other interfaces.

To make our IShape interface conform to the ISP principle, we segregate the actions to different interfaces:

```
interface IShape {
    draw();
}

interface ICircle {
    drawCircle();
}
```

```
interface ISquare {
    drawSquare();
}

interface IRectangle {
    drawRectangle();
}

interface ITriangle {
    drawTriangle();
}

class Circle implements ICircle {
    drawCircle() {
        //...
    }
}

class Square implements ISquare {
    drawSquare() {
        //...
    }
}

class Rectangle implements IRectangle {
    drawRectangle() {
        //...
    }
}

class Triangle implements ITriangle {
    drawTriangle() {
        //...
    }
}
class CustomShape implements IShape {
   draw(){
      //...
   }
}
```

The ICircle interface handles only the drawing of circles, IShape handles drawing of any shape :), ISquare handles the drawing of only squares and IRectangle handles drawing of rectangles.

OR

Classes (Circle, Rectangle, Square, Triangle, etc) can just inherit from the IShape interface and implement their own `draw` behavior.

```
class Circle implements IShape {
    draw(){
        //...
    }
}

class Triangle implements IShape {
    draw(){
        //...
    }
}

class Square implements IShape {
    draw(){
        //...
    }
}

class Rectangle implements IShape {
    draw(){
        //...
    }
}
```

We can then use the `I` -interfaces to create Shape specifics like Semi Circle, Right-Angled Triangle, Equilateral Triangle, Blunt-Edged Rectangle, etc.

Dependency Inversion Principle

> Dependency should be on abstractions not concretions
>
> A. High-level modules should not depend upon low-level modules. Both should depend upon abstractions.

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

There comes a point in software development where our app will be largely composed of modules. When this happens, we have to clear things up by using *dependency injection*. High-level components depending on low-level components to function.

```
class XMLHttpService extends XMLHttpRequestService {}

class Http {
    constructor(private xmlhttpService: XMLHttpService) { }
    get(url: string , options: any) {
        this.xmlhttpService.request(url,'GET');
    }

    post() {
        this.xmlhttpService.request(url,'POST');
    }
    //...
}
```

Here, Http is the high-level component whereas HttpService is the low-level component. This design violates DIP A: High-level modules should not depend on low-level level modules. It should depend upon its abstraction.

Ths Http class is forced to depend upon the XMLHttpService class. If we were to change to change the Http connection service, maybe we want to connect to the internet through Nodejs or even Mock the http service. We will painstakingly have to move through all the instances of Http to edit the code and this violates the OCP principle.

The Http class should care less the type of Http service you are using. We make a Connection interface:

```
interface Connection {
    request(url: string, opts:any);
}
```

The `Connection` interface has a request method. With this, we pass in an argument of type `Connection` to our `Http` class:

```
class Http {
    constructor(private httpConnection: Connection) { }

    get(url: string , options: any) {
        this.httpConnection.request(url,'GET');
    }

    post() {
        this.httpConnection.request(url,'POST');
    }
    //...
}
```

So now, no matter the type of Http connection service passed to Http it can easily connect to a network without bothering to know the type of network connection.

We can now re-implement our XMLHttpService class to implement the Connection interface:

```
class XMLHttpService implements Connection {
    const xhr = new XMLHttpRequest();
    //...
    request(url: string, opts:any) {
        xhr.open();
        xhr.send();
    }
}
```

We can create many Http `Connection` types and pass it to our Http class without any fuss about errors.

```
class NodeHttpService implements Connection {
    request(url: string, opts:any) {
        //...
    }
}
```

```
class MockHttpService implements Connection {
    request(url: string, opts:any) {
        //...
    }
}
```

Now, we can see that both high-level modules and low-level modules depend on abstractions. `Http` class(high level module) depends on the `Connection` interface(abstraction) and the Http service types(low level modules) in turn, depends on the `Connection` interface(abstraction).

Also, this DIP will force us not to violate the Liskov Substitution Principle: The `Connection` types `Node`-`XML`-`MockHttpService` are substitutable for their parent type `Connection`.

Conclusion

We covered the five principles every software developer must adhere to here. It might be daunting at first to conform to all these principles, but with steady practice and adherence, it will become a part of us and will greatly have a huge impact on the maintenance of our applications.

If you have any question regarding this or anything I should add, correct or remove, feel free to comment below and I'd love to talk!

Thanks for stopping by my little corner of the web. I think you'll love my email newsletter about programming advice, tutoring, tech, programming and software development. Just sign up below:

Sign up for my newsletter!

Email

Sign up

☐ If you're ok with us sending you email updates, please tick the box. Unsubscribe whenever you want!

☐ I agree to leave Blog.bitsrc.io and submit this information, which will be collected and used according to Upscribe's privacy policy.

Check out this new API tool:

POSTly

Tool for managing your APIs right from your browser!

post-ly.github.io

Social Media

You can follow/contact me through these following links:

- Email
- Twitter
- Facebook
- Medium
- LinkedIn

You can also feel free to jump in and take a look at Bit on GitHub here:

teambit/bit

Easily share code between projects with your team. - teambit/bit

github.com

Learn more

5 Tools For Faster Development In React

5 tools to speed the development of your React application, focusing on components.

blog.bitsrc.io

Understanding Execution Context and Execution Stack in Javascript

Understanding execution context and stack to become a better Javascript developer.

blog.bitsrc.io

What is a PWA and why should you care?

A useful intro to progressive web applications in 2018.

blog.bitsrc.io

References/Credits

- Pro TypeScript Application-Scale JavaScript Development by **Steve Fenton**
- S.O.L.I.D: The First 5 Principles of Object Oriented Design by Samuel Oloruntoba

Software Development JavaScript Nodejs Programming Web Development

About Help Legal