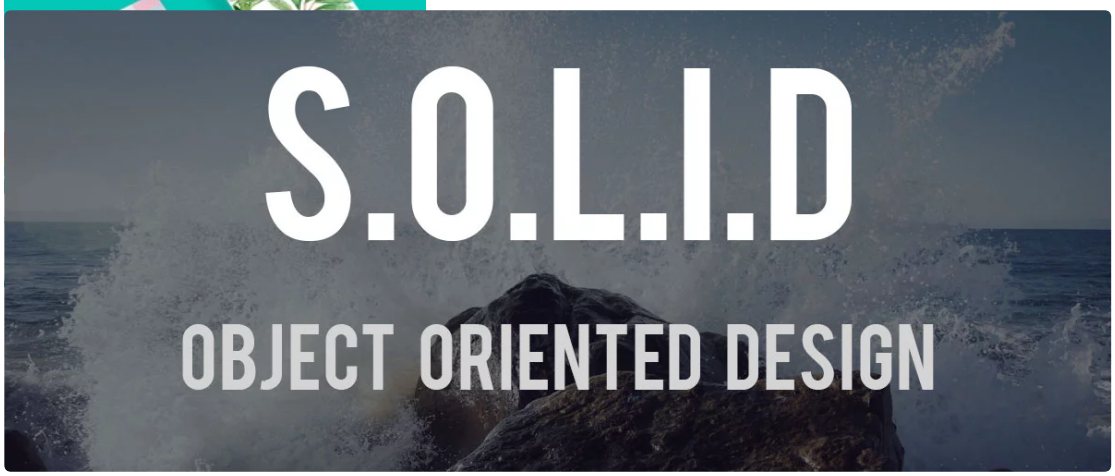
S.O.L.I.D: The First 5 Principles of Object Oriented Design

Samuel Oloruntoba (/@kayandrae) (@kayandrae07) (https://twitter.com/kayandrae07)

March 18, 2015

#ood (/tutorials?hFR%5Bcategory%5D%5B0%5D=Tutorials&dFR%5B_tags%5D%5B0%5D=ood)
#solid (/tutorials?hFR%5Bcategory%5D%5B0%5D=Tutorials&dFR%5B_tags%5D%5B0%5D=solid)
#php (/tutorials?hFR%5Bcategory%5D%5B0%5D=Tutorials&dFR%5B_tags%5D%5B0%5D=php)



(/@kayandrae)
Samuel Oloruntoba (/@kayandrae)
Follow @kayandrae07 890 followers

March 18, 2015

#ood (/tutorials?hFR%5Bcategory%5D%5B0%5
#solid (/tutorials?hFR%5Bcategory%5D%5B0%5
#php (/tutorials?hFR%5Bcategory%5D%5B0%5

S.O.L.I.D is an acronym for the **first five object-oriented design(OOD) principles** by Robert C. Martin, popularly known as Uncle Bob (https://en.wikipedia.org/wiki/Robert_Cecil_Martin).

hen combined together, make it easy for a programmer to develop asy to maintain and extend. They also make it easy for developers to easily refactor code, and are also a part of the agile or adaptive ent.

Note: *this is just a simple "welcome to **S.O.L.I.D**" article, it simply sheds light on what **S.O.L.I.D** is.*

f Contents

esponsibility Principle

osed Principle

ubstitution principle

segregation principle

ency Inversion principle

on

ands for:

e acronyms might seem complicated, but they are pretty simple to grasp.

- **S** - Single-responsiblity principle
- **O** - Open-closed principle
- **L** - Liskov substitution principle
- **I** - Interface segregation principle
- **D** - Dependency Inversion Principle

Let's look at each principle individually to understand why S.O.L.I.D can help make us better developers.

#Single-responsibility Principle

is principle states that:

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

e have some shapes and we wanted to sum all the areas of the pretty simple right?

PHP

```
class Circle {
    public $radius;

    public function __construct($radius) {
        $this->radius = $radius;
    }
}

class Square {
    public $length;

    public function __construct($length) {
        $this->length = $length;
    }
}
```

First, we create our shapes classes and have the constructors setup the required parameters. Next, we move on by creating the **AreaCalculator** class and then write up our logic to sum up the areas of all provided shapes.

PHP

```
class AreaCalculator {

    protected $shapes;

    public function __construct($shapes = array()) {
        $this->shapes = $shapes;
    }

    public function sum() {
        // logic to sum the areas
    }

    public function output() {
        return implode('', array(
            "",
                "Sum of the areas of provided shapes: ",
                $this->sum(),
            ""
        ));
    }
}
```


culator class, we simply instantiate the class and pass in an array of
the output at the bottom of the page.

etting Started with JavaScript for Web Development
Dcs)

PHP

```
$shapes = array(
    new Circle(2),
    new Square(5),
    new Square(6)
);

$areas = new AreaCalculator($shapes);

echo $areas->output();
```


The problem with the output method is that the **AreaCalculator** handles the logic to erefore, what if the user wanted to output the data as json or

uld be handled by the **AreaCalculator** class, this is what SRP frowns **alculator** class should only sum the areas of provided shapes, it ether the user wants json or HTML.

an create an **SumCalculatorOutputter** class and use this to handle need to handle how the sum areas of all provided shapes are

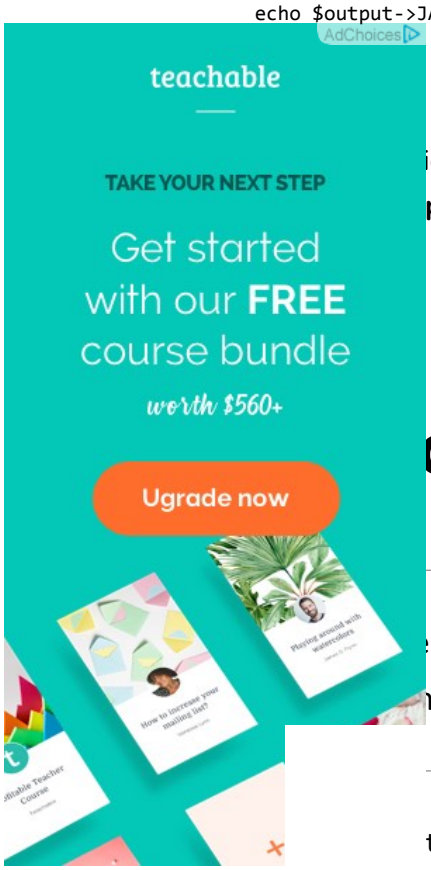
rOutputter class would work like this:

PHP

```
$shapes = array(
    new Circle(2),
    new Square(5),
    new Square(6)
);

$areas = new AreaCalculator($shapes);
$output = new SumCalculatorOutputter($areas);

echo $output->JSON();
echo $output->HAML();
echo $output->HTML();
echo $output->JADE();
```

ic you need to output the data to the user is now handled by the
putter class.

closed Principle

> ntities should be open for extension, but closed for
> n.

that a class should be easily extendable without modifying the class itself. Let's take a look at the **AreaCalculator** class, especially it's **sum** method.

PHP

```
public function sum() {
    foreach($this->shapes as $shape) {
        if(is_a($shape, 'Square')) {
            $area[] = pow($shape->length, 2);
        } else if(is_a($shape, 'Circle')) {
            $area[] = pi() * pow($shape->radius, 2);
        }
    }

    return array_sum($area);
}
```


um method to be able to sum the areas of more shapes, we would f/else blocks and that goes against the Open-closed principle.

e this **sum** method better is to remove the logic to calculate the area of the sum method and attach it to the shape's class.

PHP

```
class Square {
    public $length;

    public function __construct($length) {
        $this->length = $length;
    }

    public function area() {
        return pow($this->length, 2);
    }
}
```

The same thing should be done for the **Circle** class, an **area** method should be added. Now, to calculate the sum of any shape provided should be as simple as:

PHP

```
    public function sum() {
        foreach($this->shapes as $shape) {
            $area[] = $shape->area();
        }

        return array_sum($area);
    }
```


e another shape class and pass it in when calculating the sum without However, now another problem arises, how do we know that the object passed into the **AreaCalculator** is actually a shape or if the shape has a method

ace is an integral part of **S.O.L.I.D**, a quick example is we create an y shape implements:

PHP

```
interface ShapeInterface {
    public function area();
}

class Circle implements ShapeInterface {
    public $radius;

    public function __construct($radius) {
        $this->radius = $radius;
    }

    public function area() {
        return pi() * pow($this->radius, 2);
    }
}
```

In our **AreaCalculator** sum method we can check if the shapes provided are actually instances of the **ShapeInterface**, otherwise we throw an exception:

PHP


```
    public function sum() {
        foreach($this->shapes as $shape) {
            if(is_a($shape, 'ShapeInterface')) {
                $area[] = $shape->area();
                continue;
            }
            throw new AreaCalculatorInvalidShapeException;
        }

        return array_sum($area);
    }
```


substitution principle

> a property provable about objects of **x** of type **T**.
> hould be provable for objects **y** of type **S** where **S** is a
> T.

All this is stating is that every subclass/derived class should be substitutable for their base/parent class.

Still making use of out **AreaCalculator** class, say we have a **VolumeCalculator** class that extends the **AreaCalculator** class:

PHP

```
class VolumeCalculator extends AreaCalulator {
    public function __construct($shapes = array()) {
        parent::__construct($shapes);
    }

    public function sum() {
        // logic to calculate the volumes and then return and array of output
        return array($summedData);
    }
}
```


orOutputter class:

PHP

```
class SumCalculatorOutputter {
    protected $calculator;

    public function __constructor(AreaCalculator $calculator) {
        $this->calculator = $calculator;
    }

    public function JSON() {
        $data = array(
            'sum' => $this->calculator->sum();
        );

        return json_encode($data);
    }

    public function HTML() {
        return implode('', array(
            '',
            'Sum of the areas of provided shapes: ',
            $this->calculator->sum(),
            ''
        ));
    }
}
```


n example like this:

PHP

```
$areas = new AreaCalculator($shapes);
$volumes = new AreaCalculator($solidShapes);

$output = new SumCalculatorOutputter($areas);
$output2 = new SumCalculatorOutputter($volumes);
```

The program does not squawk, but when we call the **HTML** method on the **$output2** object we get an **E_NOTICE** error informing us of an array to string conversion.

To fix this, instead of returning an array from the **VolumeCalculator** class sum method, you should simply:

PHP

```
public function sum() {
    // logic to calculate the volumes and then return and array of output
    return $summedData;
}
```


as a float, double or integer.

ce segregation principle

> uld 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.

Still using our shapes example, we know that we also have solid shapes, so since we would also want to calculate the volume of the shape, we can add another contract to the **ShapeInterface**:

PHP

```
interface ShapeInterface {
    public function area();
    public function volume();
}
```


te must implement the **volume** method, but we know that squares that they do not have volumes, so this interface would force the plement a method that it has no use of.

ISP says no to this, instead you could create another interface called **SolidShapeInterface** that has the **volume** contract and solid shapes like cubes e.t.c can rface:

PHP

```
interface ShapeInterface {
    public function area();
}

interface SolidShapeInterface {
    public function volume();
}

class Cuboid implements ShapeInterface, SolidShapeInterface {
    public function area() {
        // calculate the surface area of the cuboid
    }

    public function volume() {
        // calculate the volume of the cuboid
    }
}
```


This is a much better approach, but a pitfall to watch out for is when type-hinting these interfaces, instead of using a **ShapeInterface** or a **SolidShapeInterface**.

You can create another interface, maybe **ManageShapeInterface**, and implement it on both the flat and solid shapes, this way you can easily see that it has a single API for managing the shapes. For example:

PHP


```
interface ManageShapeInterface {
    public function calculate();
}

class Square implements ShapeInterface, ManageShapeInterface {
    public function area() { /*Do stuff here*/ }

    public function calculate() {
        return $this->area();
    }
}

class Cuboid implements ShapeInterface, SolidShapeInterface, ManageShapeInterface {
    public function area() { /*Do stuff here*/ }
    public function volume() { /*Do stuff here*/ }

    public function calculate() {
        return $this->area() + $this->volume();
    }
}
```

ator class, we can easily replace the call to the **area** method with check if the object is an instance of the **ManageShapeInterface** and **face**.

#Dependency Inversion principle

The last, but definitely not the least states that:

> 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.

loated, but it is really easy to understand. This principle allows for mple that seems like the best way to explain this principle:

PHP

```
class PasswordReminder {

    private $dbConnection;

    public function __construct(MySQLConnection $dbConnection) {

        $this->dbConnection = $dbConnection;
    }

}
```

nnection is the low level module while the **PasswordReminder** is high level, but according to the definition of **D** in S.O.L.I.D. which states that *Depend on Abstraction not on concretions*, this snippet above violates this principle as the **PasswordReminder** class is being forced to depend on the **MySQLConnection** class.

Later if you were to change the database engine, you would also have to edit the **PasswordReminder** class and thus violates **Open-close principle**.

The **PasswordReminder** class should not care what database your application uses, to fix this again we "code to an interface", since high level and low level modules should depend on abstraction, we can create an interface:

PHP

```
interface DBConnectionInterface {
    public function connect();
}
```


connect method and the **MySQLConnection** class implements this interface, also instead of directly type-hinting **MySQLConnection** class in the constructor of the **PasswordReminder**, we instead type-hint the interface and no matter se your application uses, the **PasswordReminder** class can easily base without any problems and **OCP** is not violated.

PHP

```
class MySQLConnection implements DBConnectionInterface {
    public function connect() {
        return "Database connection";
    }
}

class PasswordReminder {
    private $dbConnection;

    public function __construct(DBConnectionInterface $dbConnection) {
        $this->dbConnection = $dbConnection;
    }
}
```

According to the little snippet above, you can now see that both the high level and low level modules depend on abstraction.

#Conclusion

Honestly, **S.O.L.I.D** might seem to be a handful at first, but with continuous usage and adherence to its guidelines, it becomes a part of you and your code which can easily be extended, modified, tested, and refactored without any problems.

Learn jQuery for Total Noobs Volume III: Demos & Practice Course (/courses/learn-jquery-for-total-noobs-volume-iii-demos-practice-course)

Learn jQuery for Total Noobs Volume II: jQuery is So Easy Course

(/courses/jquery-for-total-noobs-volume-ii-jquery-is-so-easy-course)

Learn jQuery for Total Noobs Volume I: JavaScript Crash Course (/courses/jquery-for-total-noobs-volume-i-javascript-crash-course)

Build Your First Angular Website

More Courses (/courses)

(https://synd.co/2kx2ZAF)

Stitch (/tutorials/how-to-integrate-mongodb-atlas-and-segment-using-mongodb-stitch)

3 Useful TypeScript Tips for Angular (/tutorials/3-useful-typescript-tips-for-angular)

/tutorials?hFR%5Bcategory%5D%5B0%5D=Tutorials)

(/@kayandrae)

developer and absolutely slays at FIFA. You can reach out to him for business (mailto:sam@rafdel.co) or (https://synd.co/2kx2ZAE)

(/@kayandrae)

Samuel Oloruntoba (/@kayandrae)

39 posts (/@kayandrae)

Samuel is a multi-disciplinary web developer. Nowadays, he spends most of his time in the browser trying to understand it and squeeze as much performance as he can. He's also a proficient Vue/Laravel developer and absolutely slays at FIFA. You

can reach out to him for business (mailto:sam@rafdel.co) or Scotch related matters (mailto:samuel@scotch.io).

(/@kayandrae) (https://twitter.com/kayandrae07) (https://github.com/kayandrae07)

Popular in the Community

GETTING STARTED WITH LIVE CODING IN VISUAL...
GreenDog
30 May
Visual studio code is one of the best...

JAVASCRIPT FUNCTIONAL PROGRAMMING...
PurpleFlower
5d
gdfg

BUILD A PROGRESSIVE WEB APPLICATION WIT...
Сергей Хваль
1d
How are app updates handled with nuxt...

NODE.JS CRON JOBS BY EXAMPLES
dreamzoner
31 May
Why did you use express? I thought th...

BUILD MULTIPLE STACKING STICKY...
BlueCocktail
4d
Great article

BUILD NATIVE MODALS USING THE DIALOG...
Craig Geil
25 May
Will you be able to drag the dialog...

GETTING YOGA AN
Good st informa

A side project brought to you from Las Vegas and DC by... (/about)

(https://scotch.io/@chris) Chris Sevilleja (https://scotch.io/@chris)
Follow @chrisoncode 8,181 followers

(https://scotch.io/@nick) Nick Cerminara (https://scotch.io/@nick)
Follow @whatnicktweets 2,397 followers

(https://bit.ly/2tDcLEK) Easiest Local Dev Environment

(https://scotch.io/courses/getting-started-with-vue) Get Started with Vue.js

scotch

Top shelf learning. Informative tutorials explaining the code **and the choices behind it all.**

(https://twitter.com/scotch_io) (https://facebook.com/scotchdevelopment) (https://github.com/scotch-)