

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



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](#).

These principles, when combined together, make it easy for a programmer to develop software that are easy to maintain and extend. They also make it easy for developers to avoid code smells, easily refactor code, and are also a part of the agile or adaptive software development.

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.*

S.O.L.I.D stands for:

When expanded the acronyms might seem complicated, but they are pretty simple to grasp.

- S – Single-responsibility 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](#)

S.R.P for short – this principle states that:

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

For example, say we have some shapes and we wanted to sum all the areas of the shapes. Well this is pretty simple right?

```
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.

```
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(
            "<h1>",
            "Sum of the areas of provided shapes: ",
            $this->sum(),
            "</h1>"
        ));
    }
}
```

```
}  
}
```

To use the AreaCalculator class, we simply instantiate the class and pass in an array of shapes, and display the output at the bottom of the page.

```
$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 output the data. Therefore, what if the user wanted to output the data as json or something else?

All of that logic would be handled by the AreaCalculator class, this is what SRP frowns against; the AreaCalculator class should only sum the areas of provided shapes, it should not care whether the user wants json or HTML.

So, to fix this you can create an SumCalculatorOutputter class and use this to handle whatever logic you need to handle how the sum areas of all provided shapes are displayed.

The SumCalculatorOutputter class would work like this:

```
$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();
```

Now, whatever logic you need to output the data to the user is now handled by the SumCalculatorOutputter class.

Open-closed Principle

Objects or entities should be open for extension, but closed for modification.

This simply means that a class should be easily extendable without modifying the class itself. Let's take a look at the AreaCalculator class, especially its sum method.

```
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);
}
```

If we wanted the sum method to be able to sum the areas of more shapes, we would have to add more if/else blocks and that goes against the Open-closed principle.

A way we can make this sum method better is to remove the logic to calculate the area of each shape out of the sum method and attach it to the shape's class.

```
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:

```

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

    return array_sum($area);
}

```

Now we can create another shape class and pass it in when calculating the sum without breaking our code. 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 named area?

Coding to an interface is an integral part of S.O.L.I.D, a quick example is we create an interface, that every shape implements:

```

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:

```

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

```

```

        throw new AreaCalculatorInvalidShapeException;
    }

    return array_sum($area);
}

```

[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 .

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

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

```

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

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

```

In the SumCalculatorOutputter class:

```

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(
            '<h1>',
            'Sum of the areas of provided shapes: ',
            $this->calculator->sum(),
            '</h1>'
        ));
    }
}

```

If we tried to run an example like this:

```

$areas = new AreaCalculator($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:

```

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

```

The summed data as a float, double or integer.

[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.

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:

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

Any shape we create must implement the volume method, but we know that squares are flat shapes and that they do not have volumes, so this interface would force the Square class to implement 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 implement this interface:

```
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:

```
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();
    }
}

```

Now in AreaCalculator class, we can easily replace the call to the area method with calculate and also check if the object is an instance of the ManageShapeInterface and not the ShapeInterface.

[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.

This might sound bloated, but it is really easy to understand. This principle allows for decoupling, an example that seems like the best way to explain this principle:

```

class PasswordReminder {
    private $dbConnection;

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

```

First the MySqlConnection 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:

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

The interface has a 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 the type of database your application uses, the PasswordReminder class can easily connect to the database without any problems and OCP is not violated.

```
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.