Whale Ambience unlocked!

CONTENTS

RELATED

CodeIgniter: Getting Started With a Simple Example
View

How To Install Express, a Node.js Framework, and Set Up Socket.io on a VPS
View

// Conceptual Article //

SOLID: The First 5 Principles of Object Oriented Design

Published on September 21, 2020 · Updated on November 30, 2021

Development

By Samuel Oloruntoba
Developer and author at DigitalOcean.

English

Whale Ambience unlocked!

Introduction

SOLID is an acronym for the first five object-oriented design (OOD) principles by Robert C. Martin (also known as [Uncle Bob]).

> **Note:** While these principles can apply to various programming languages, the sample code contained in this article will use PHP.

These principles establish practices that lend to developing software with considerations for maintaining and extending as the project grows. Adopting these practices can also contribute to avoiding code smells, refactoring code, and Agile or Adaptive software development.

SOLID stands for:

- **S** - Single-responsiblity Principle
- **O** - Open-closed Principle
- **L** - Liskov Substitution Principle
- **I** - Interface Segregation Principle
- **D** - Dependency Inversion Principle

In this article, you will be introduced to each principle individually to understand how SOLID can help make you a better developer.

Single-Responsibility Principle

Single-responsibility Principle (SRP) states:

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

For example, consider an application that takes a collection of shapes—circles, and squares—and calculates the sum of the area of all the shapes in the collection.

First, create the shape classes and have the constructors set up the required parameters.

For squares, you will need to know the `length` of a side:

Copy

```
class Square
{
    public $length;
```

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

For circles, you will need to know the `radius`:


```
class Circle
{
    public $radius;

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

Next, create the `AreaCalculator` class and then write up the logic to sum up the areas of all provided shapes. The area of a square is calculated by length squared. The area of a circle is calculated by pi times radius squared.


```
class AreaCalculator
{
    protected $shapes;

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

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

        return array_sum($area);
    }

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

To use the `AreaCalculator` class, you will need to instantiate the class and pass in an array of shapes and display the output at the bottom of the page.

Here is an example with a collection of three shapes:

- a circle with a radius of 2
- a square with a length of 5
- a second square with a length of 6

```
$shapes = [
  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.

Consider a scenario where the output should be converted to another format like JSON.

All of the logic would be handled by the `AreaCalculator` class. This would violate the single-responsibility principle. The `AreaCalculator` class should only be concerned with the sum of the areas of provided shapes. It should not care whether the user wants JSON or HTML.

To address this, you can create a separate `SumCalculatorOutputter` class and use that new class to handle the logic you need to output the data to the user:

```
class SumCalculatorOutputter
{
    protected $calculator;

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

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

        return json_encode($data);
    }

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


The `SumCalculatorOutputter` class would work like this:

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

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

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


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

That satisfies the single-responsibility principle.

Open-Closed Principle

Open-closed Principle (OCP) states:

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

This means that a class should be extendable without modifying the class itself.

Let's revisit the `AreaCalculator` class and focus on the `sum` method:

```
class AreaCalculator
{
    protected $shapes;

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

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

        return array_sum($area);
    }
}
```

Consider a scenario where the user would like the `sum` of additional shapes like triangles, pentagons, hexagons, etc. You would have to constantly edit this file and add more `if`/`else` blocks. That would violate the open-closed principle.

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

Here is the `area` method defined in `Square`:

```
class Square
{
    public $length;

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

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

And here is the `area` method defined in `Circle`:

```
class Circle
{
```

```
    public $radius;

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

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

Whale Ambience unlocked!

The `sum` method for `AreaCalculator` can then be rewritten as:

Copy

```
class AreaCalculator
{
    // ...

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

        return array_sum($area);
    }
}
```

Now, you can create another shape class and pass it in when calculating the sum without breaking the code.

However, another problem arises. How do you 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 SOLID.

Create a `ShapeInterface` that supports `area`:

Copy

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

Modify your shape classes to `implement` the `ShapeInterface`.

Here is the update to `Square`:

Copy

```
class Square implements ShapeInterface
{
    // ...
}
```

And here is the update to `Circle`:

Copy

```
class Circle implements ShapeInterface
{
    // ...
}
```

In the `sum` method for `AreaCalculator`, you can check if the shapes provided are actually instances of the `ShapeInterface`; otherwise, throw an exception:

```php
class AreaCalculator
{
    // ...

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

            throw new AreaCalculatorInvalidShapeException();
        }

        return array_sum($area);
    }
}
```


That satisfies the open-closed principle.

Liskov Substitution Principle

Liskov Substitution Principle states:

> 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.

This means that every subclass or derived class should be substitutable for their base or parent class.

Building off the example `AreaCalculator` class, consider a new `VolumeCalculator` class that extends the `AreaCalculator` class:

```php
class VolumeCalculator extends AreaCalculator
{
    public function construct($shapes = [])
    {
        parent::construct($shapes);
    }

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

Recall that the `SumCalculatorOutputter` class resembles this:

```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(),
            ''
        ));
    }
}
```


If you tried to run an example like this:


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

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

When you call the `HTML` method on the `$output2` object, you will get an `E_NOTICE` error informing you of an array to string conversion.

To fix this, instead of returning an array from the `VolumeCalculator` class sum method, return `$summedData`:


```
class VolumeCalculator extends AreaCalculator
{
    public function construct($shapes = [])
    {
        parent::construct($shapes);
    }

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

The `$summedData` can be a float, double or integer.

That satisfies the Liskov substitution principle.

Interface Segregation Principle

Interface segregation principle states:

> 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 building from the previous `ShapeInterface` example, you will need to support the new three-dimensional shapes of `Cuboid` and `Spheroid`, and these shapes will need to also calculate `volume`.

Let's consider what would happen if you were to modify the `ShapeInterface` to add another contract:


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

    public function volume();
}
```

Now, any shape you create must implement the `volume` method, but you 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.

This would violate the interface segregation principle. Instead, you could create another interface called `ThreeDimensionalShapeInterface` that has the `volume` contract and three-dimensional shapes can implement this interface:


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

interface ThreeDimensionalShapeInterface
{
    public function volume();
}

class Cuboid implements ShapeInterface, ThreeDimensionalShapeInterface
{
    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 `ThreeDimensionalShapeInterface`, you can create another interface, maybe `ManageShapeInterface`, and implement it on both the flat and three-dimensional shapes.

This way, you can have a single API for managing the shapes:


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

class Square implements ShapeInterface, ManageShapeInterface
{
    public function area()
    {
        // calculate the area of the square
    }

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

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

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

    public function calculate()
```

```
    {
        return $this->area();
    }
}
```

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

That satisfies the interface segregation principle.

Dependency Inversion Principle

Dependency inversion principle states:

> 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 principle allows for decoupling.

Here is an example of a `PasswordReminder` that connects to a MySQL database:


```
class MySQLConnection
{
    public function connect()
    {
        // handle the database connection
        return 'Database connection';
    }
}

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 SOLID, which states to *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 this would violate the *open-close principle*.

The `PasswordReminder` class should not care what database your application uses. To address these issues, you can code to an interface since high-level and low-level modules should depend on abstraction:


```
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`, you instead type-hint the `DBConnectionInterface` and no matter the type of database your application uses, the `PasswordReminder` class can connect to the database without any problems and open-close principle is not violated.

```
class MySQLConnection implements DBConnectionInterface
{
    public function connect()
    {
        // handle the database connection
        return 'Database connection';
    }
}

class PasswordReminder
{
    private $dbConnection;

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


This code establishes that both the high-level and low-level modules depend on abstraction.

Conclusion

In this article, you were presented with the five principles of SOLID Code. Projects that adhere to SOLID principles can be shared with collaborators, extended, modified, tested, and refactored with fewer complications.

Continue your learning by reading about other practices for [Agile](#) and [Adaptive software development](#).

About the authors

[Samuel Oloruntoba](#) Author

Developer and author at DigitalOcean.

[Bradley Kouchi](#) Editor

Developer and author at DigitalOcean.