UNIVERSIDAD BOLIVIANA DE INFORMATICA

TALLER DE SISTEMAS

ESTUDIANTE: YAPU FLORES CLAUDIA LORENA

**PRINCIPIOS SOLID**

**Introducción**

*SOLID* es un acrónimo de los primeros cinco principios del diseño orientado a objetos (OOD) de Robert C. Martin (también conocido como el [Tío Bob](http://en.wikipedia.org/wiki/Robert_Cecil_Martin) ).

Estos principios fundamentan prácticas que se prestan al desarrollo de software con consideración para su mantenimiento y expansión a medida que el proyecto se amplía. Adoptar estas prácticas también puede ayudar a evitar los aromas de código, refactorizar el código y aprender sobre el desarrollo ágil y adaptativo de software.

Representación SÓLIDA:

1. [**S** : (Única) Principio de responsabilidad única](https://www.digitalocean.com/community/conceptual_articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design-es#single-responsibility-principle)
2. [**O: (Open)** Principio abierto-cerrado](https://www.digitalocean.com/community/conceptual_articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design-es#open-closed-principle)
3. [**L: (Liskov)** Principio de sustitución de Liskov](https://www.digitalocean.com/community/conceptual_articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design-es#liskov-substitution-principle)
4. [**I: (Interfaz)** Principio de segregación de interfaz](https://www.digitalocean.com/community/conceptual_articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design-es#interface-segregation-principle)
5. [**D: (Dependency)** Principio de inversión de dependencia](https://www.digitalocean.com/community/conceptual_articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design-es#dependency-inversion-principle)
6. **Principio de responsabilidad única**

El principio de responsabilidad única (SRP) establece:

Una clase debe tener una y una sola razón para cambiar, lo que significa que una clase debe tener solo un trabajo.

Por ejemplo, considere una aplicación que toma una colección de formas, entre círculos y cuadrados, y calcule la suma del área de todas las formas de la colección.

Primero, cree las clases de forma y haga que los constructores configuren los parámetros requeridos.

Para los cuadrados, debe saber la longitud de un lado:

class Square{

public $length;

public function construct($length)

{$this->length = $length; }

}

Para los círculos, deberás saber el radio:

class Circle{

public $radius;

public function construct($radius)

{$this->radius = $radius; }

}

A continuación, cree la clase AreaCalculatory luego escriba la lógica para sumar las áreas de todas las formas proporcionadas. El area de un cuadrado se calcula por longitud al cuadrado. El área de un círculo se calcula mediante pi por el radio al cuadrado.

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(), '',]);

}}

Para usar la claseAreaCalculator , deberá crear una instancia de la clase y pasar una matriz de formas para mostrar el resultado en la parte inferior de la página.

A continuación, se muestra un ejemplo con una colección de tres formas:

* un circulo con una radio de 2
* un cuadrado con una longitud de 5
* un segundo cuadrado con una longitud de 6

$shapes = [

new Circle(2),

new Square(5),

new Square(6),

];

$areas = new AreaCalculator($shapes);

echo $areas->output();

El problema con el metodo de salida es queAreaCalculator maneja la logica para generar los datos.

Considere un escenario en el que el resultado debe convertirse en otro formato como JSON.

La clase AreaCalculatormanejaría toda la lógica. Esto violaría el principio de responsabilidad única. la claseAreaCalculator solo debe ocuparse de la suma de las áreas de las formas proporcionadas. No debería importar si el usuario desea JSON o HTML.

Para abordar esto, puede crear una clase SumCalculatorOutputterseparada por separado y usarla para manejar la lógica que necesita para mostrar los datos al usuario:

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(),

'',

]);

}}

La clase SumCalculatorOutputterfuncionaría así:

$shapes = [

new Circle(2),

new Square(5),

new Square(6),

];

$areas = new AreaCalculator($shapes);

$output = new SumCalculatorOutputter($areas);

echo $output->JSON();

echo $output->HTML();

Ahora, la clase SumCalculatorOutputtermaneja cualquier lógica que necesite para enviar los datos al usuario.

Eso cumple con el principio de responsabilidad única.

1. **Principio abierto-cerrado**

Principio abierto-cerrado (SRP) establece:

Los objetos o entidades deben estar abiertos por extensión, pero cerrados por modificación.

Esto significa que una clase debe ser ampliable sin modificar la clase en sí.

Volvamos a ver la clase AreaCalculatory enfoquémonos en el método sum:

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);

}

}

Considere un escenario en el que el usuario desea las sumformas adicionales como triángulos, pentágonos, hexágonos, etc. Tendría que editar constantemente este archivo y agregar más bloques if/ else. Eso violaría el principio abierto-cerrado.

Una forma de mejorar el método sumes eliminar la lógica para calcular el área de cada forma fuera del método de la clase AreaCalculatory adjuntarlo a la clase de cada forma.

A continuación, se muestra areadefinida en Square:

class Square{

public $length;

public function \_\_construct($length) {

$this->length = $length;

}

public function area() {

return pow($this->length, 2);

}

}

Y aquí es el método areadefinido en Circle:

class Circle{

public $radius;

public function construct($radius) {

$this->radius = $radius;

}

public function area(){

return pi() \* pow($shape->radius, 2);

}

}

El método sumpara AreaCalculatorreescribirse así:

class AreaCalculator{

// ...

public function sum() {

foreach ($this->shapes as $shape) {

$area[] = $shape->area();

}

return array\_sum($area);

}}

Ahora, puede crear otra clase de forma y pasarla al calcular la suma sin romper el código.

Sin embargo, ahora surge otro problema: ¿Cómo sabe que el objeto pasado a AreaCalculatores realmente una forma o si la forma tiene un método llamado area?

La codificación de una [interfaz](https://www.php.net/manual/en/language.oop5.interfaces.php) es una parte integral de SOLID.

Cree un ShapeInterfaceque sea compatible con area:

interface ShapeInterface{

public function area();

}

Modifique sus clases de forma para implementelShapeInterface .

A continuación, se muestra la actualización a Square:

class Square implements ShapeInterface{

// ...

}

Y aquí está la actualización a Circle:

class Circle implements ShapeInterface{

// ...

}

En el método sumpara AreaCalculator, puede verificar si las formas proporcionadas son realmente instancias de ShapeInterface; de lo contrario, lanzamos una excepción:

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);

}}

Eso cumple con el principio abierto-cerrado.

1. **Principio de sustitución de Liskov**

El principio de sustitución de Liskov establece:

Digamos que q(x) sea una propiedad demostrable sobre objetos de x, de tipo T. Entonces, q(y) debe ser demostrable para los objetos y, de tipo S, donde S es un subtipo de T.

Esto significa que cada subclase o clase derivada debe ser sustituible por su clase base o clase principal.

A partir de la clase AreaCalculatormostrada como ejemplo, considere una nueva clase VolumeCalculatorque extienda la clase AreaCalculator:

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];

}}

Recuerde que la clase SumCalculatorOutputterse asemeja a esto:

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(),

''

)); }}

Si intenta ejecutar un ejemplo como este:

$areas = new AreaCalculator($shapes);

$volumes = new VolumeCalculator($solidShapes);

$output = new SumCalculatorOutputter($areas);

$output2 = new SumCalculatorOutputter($volumes);

Cuando invoca el método HTMLen el objeto $output2, obtiene un errorE\_NOTICE que le informará de conversión de matriz a cadena.

Para solucionar esto, en vez de devolver una matriz desde el método sum de la clase VolumeCalculator, devuelva $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;

}}

$summedData puede ser float, double o integer.

Eso cumple con el principio de sustitución de Liskov.

1. **Principio de segregación de interfaz**

El principio de segregación de interfaz establece:

Un cliente nunca debe ser forzado a implementar una interfaz que no usen ni los clientes no deben ser forzados a depender de métodos que no usen.

Siguiendo con el ejemplo anterior de ShapeInterface, tendrá que admitir las nuevas formas tridimensionales de Cuboidy Spheroid, y estas formas también tendrán que calcular el volumen.

Consideraremos lo que sucedería si modificara ShapeInterface para añadir otro contrato:

interface ShapeInterface{

public function area();

public function volume();

}

Ahora, cualquier forma que crea debe implementar el método volume, pero sabemos que los cuadrados son formas planas y que no tienen volumen, por lo que esta interfaz forzaría a la claseSquare a implementar un método que no usa.

Esto violaría el principio de segregación de interfaz. En su lugar, podría crear otra interfaz llamada ThreeDimensionalShapeInterfaceque tiene el contrato de volumen y las formas tridimensionales pueden implementar esta interfaz:

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

}}

Este es un enfoque mucho mejor, pero hay que tener cuidado cuando se trata de escribir estas interfaces En vez de usar una ShapeInterfaceo una ThreeDimensionalShapeInterface, puede crear otra interfaz, quizás ManageShapeInterfacee implementarla en las formas planas y en las tridimensionales.

De esta manera, puede tener una sola API para administrar las formas:

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();

}}

Ahora en la clase AreaCalculator, puede sustituir la invocación al método área con calculate y también verificar si el objeto es una instancia de ManageShapeInterface y no de ShapeInterface .

Eso cumple con el principio de segregación de interfaz.

1. **Principio de inversión de dependencia**

El principio de inversión de dependencia establece:

Las entidades deben depender de abstracciones, no de concreciones. Indica que el módulo de alto nivel no debe depender del módulo de bajo nivel, sino que debe depender de las abstracciones.

Este principio permite el desacoplamiento.

A continuación, se muestra un ejemplo de un PasswordReminder que se conecta a una base de datos de MySQL:

class MySQLConnection{

public function connect(){

// handle the database connection

return 'Database connection';

}}

class PasswordReminder{

private $dbConnection;

public function \_\_construct(MySQLConnection $dbConnection)

{

$this->dbConnection = $dbConnection;

}}

Primero, MySQLConnectiones el módulo de bajo nivel mientras que PasswordReminderes de alto nivel, pero según la definición de **D** en SOLID, que establece que *depende de la abstracción y no de las concreciones* . Este fragmento de código anterior viola este principio, ya que se está forzando a la clasev PasswordRemindera depender de la claseMySQLConnection .

Si más adelante cambiara el motor de la base de datos, también tendría que editar la clase PasswordReminder, y esto violaría el *principio abierto-cerrado* .

A la clase PasswordReminderno le debe importar qué base de datos usa su aplicación. Para solucionar estos problemas, se puede codificar una interfaz, ya que los módulos de alto nivel y bajo nivel deben depender de la abstracción:

interface DBConnectionInterface{

public function connect();

}

La interfaz tiene un método connect y la clase MySQLConnectionimplementa esta interfaz. Además, en lugar de escribir directamente la clase MySQLConnectionen el constructor del PasswordReminder, se indica la clase DBConnectionInterface y, sin importar el tipo de base de datos que utilice su aplicación, la clase PasswordReminder puede conectarse sin ningún problema a la base de datos y no se viola el principio abierto-cerrado.

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;

}}

Este código establece que los módulos de alto nivel y los de bajo nivel dependen de la abstracción.