



web разработка

ооп  
S.O.L.I.D.  
принципи

# S.O.L.I.D. principles

Първите пет принципа в ОО дизайн.

**S** - Single-responsibility principle

**O** - Open-closed principle

**L** - Liskov substitution principle

**I** - Interface segregation principle

**D** - Dependency Inversion Principle

**S - single responsibility principle**

# S - single responsibility principle

**Задача** Създайте приложение с PHP ООП, което сумира площта на две различни геометрични фигури.

# S - single responsibility principle

A class should have one job.

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

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

# S - single responsibility principle

A class should have one job.

```
class AreaCalculator {  
  
    protected $shapes;  
  
    public function __construct($shapes = array()) {  
        $this->shapes = $shapes;  
    }  
  
    public function sum() {  
        // logic to calculate and sum the areas  
    }  
  
    public function output() {  
        return "Sum of the areas of provided shapes: ". $this->sum();  
    }  
}
```

# S - single responsibility principle

A class should have one job.

- Създаваме обект от клас AreaCalculator;
- Подаваме му масив от геометрични фигури;
- Отпечатваме сумата от площите

```
$shapes = array(  
    new Circle(2),  
    new Square(5),  
    new Square(6)  
);  
  
$areas = new AreaCalculator($shapes);  
  
echo $areas->output();
```

# S - single responsibility principle

A class should have one job.

- Ако искаме да доразвием задачата и да отпечатваме резултатът в различни формати. Редно е да създадем нов клас SumCalculatorOutputter, който ще има задачата да реализира извеждането на резултата във формата, който ни трябва.

*Кой вариант бихте избрали?*

Вариант 1 - \$output = new SumCalculatorOutputter(\$areas);

Вариант 2 - \$output = new SumOutputter(\$sum);

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

# O - open-closed principle

# O - open-closed principle

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

*Трябва да е възможно разширяването на класа, без да се налага да го променяме.*

Да разгледаме **AreaCalculator** и неговия **sum** метод в този му вид.

```
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);  
}
```

Ограничени сме с два вида геометрични фигури.

*Ако приложението се развие иискаме да изчисляваме площта на неограничен брой геометрични фигури?*

# O - open-closed principle

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

Трябва да е възможно разширяването на класа, без да се налага да го променяме.

**Ако приложението се развие иискаме да изчисляваме площта на неограничен брой геометрични фигури?**

*Вариант 1* - добавяме още if/else блокове за всяка фигура - противоречи с open-closed принципа

*Вариант 2* - махаме логиката за изчисляване на площта от sum метода и я поверяваме на логиката на класа на всяка фигура

```
class Square {  
    public $length;  
  
    public function __construct($length) {  
        $this->length = $length;  
    }  
  
    public function calc_area() {  
        return pow($this->length, 2);  
    }  
}
```

*Същото ще направим и в класовете за всяка друга фигура.*

# O - open-closed principle

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

Трябва да е възможно разширяването на класа, без да се налага да го променяме.

Изчисляването на сумата от площите на геометричните фигури няма да е свързано с броя или вида им -

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

Сега можем да създадем друг клас за друга геометрична фигура и да подаваме негови обекти при изчисляване на общата сума.

Възниква въпросът - *Как да сме сигурни, че обектът подаван към AreaCalculator е геометрична фигура или дали този обект има метод, наречен area?*

# O - open-closed principle

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

Трябва да е възможно разширяването на класа, без да се налага да го променяме.

Как да сме сигурни, че обектът подаван към *AreaCalculator* е геометрична фигура или дали този обект има метод, наречен *area*?

За да решим този проблем използваме интерфейс, което е неразделна част от S.O.L.I.D

Този интерфейс ще се имплементира от всеки клас, отговарящ за геометрична фигура.

```
interface ShapeInterface {  
    public function calc_area();  
}  
  
class Circle implements ShapeInterface {  
    public $radius;  
  
    public function __construct($radius) {  
        $this->radius = $radius;  
    }  
  
    public function area() {  
        return pi() * pow($this->radius, 2);  
    }  
}
```

# O - open-closed principle

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

*Трябва да е възможно разширяването на класа, без да се налага да го променяме.*

В sum методът на **AreaCalculator** проверяваме дали подаваната геометрична фигура в действителност е истанция на клас, който имплементира **ShapeInterface**.

В случай, че не е - връщаме подходящо съобщение, грешка и т.н

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

# L - Liskov substitution principle

# L - Liskov substitution principle

any class that is the child of a parent class should be usable in place of its parent without any unexpected behaviour

LSP - обектите от клас наследник трябва да заменят обекти на родителския клас, без това да води до грешки в програмата и без да променят поведението на родителския клас.

Или ако S е наследник на T, обект от T може да бъде заменен с обект от S без това да се отрази на програмата или да доведе до грешки в системата.

Нека дефинираме клас *Rectangle/правоъгълник/* и друг клас *Square/квадрат/*.

```
Class Rectangle {  
    public $width;  
    public $height;  
  
    public function __construct($w, $h){  
        .....  
    }  
  
    public function area_calc(){  
        return $this->width*$this->height;  
    }  
}
```

# L - Liskov substitution principle

any class that is the child of a parent class should be usable in place of its parent without any unexpected behaviour

В геометрията Квадратът е Правоъгълник или с други думи Square наследява Rectangle.

Дали това е коректно от гледна точка на програмирането и на LSP - трябва да заменяме обектите на Rectangle с обектите на Square без това да води до нежелани промени или грешки в системата ....

```
class Square extends Rectangle {  
    public $side;  
  
    public function __construct($s){  
        ....  
    }  
  
    public function area_calc(){  
        return pow(2, $this->side);  
    }  
}
```

# L - Liskov substitution principle

*В сила е обратното - Правоъгълникът наследява Квадрата ...*

```
class Square {  
    public $a_side;  
  
    public function __construct($s){  
        ....  
    }  
  
    public function area_calc(){  
        return pow(2, $this->side);  
    }  
}
```

```
class Rectangle extends Square {  
    public $b_side;  
  
    public function __construct($a, $b){  
        ....  
    }  
  
    public function area_calc(){  
        return $this->a_side*$this->b_side;  
    }  
}
```

# L - Liskov substitution principle

any class that is the child of a parent class should be usable in place of its parent without any unexpected behaviour

```
class Vehicle {  
  
    function startEngine() {  
        // Default engine start functionality  
    }  
  
    function accelerate() {  
        // Default acceleration functionality  
    }  
}
```

```
class Car extends Vehicle {  
  
    function startEngine() {  
        $this->engagelgnition();  
        parent::startEngine();  
    }  
  
    private function engagelgnition() {  
        // Ignition procedure  
    }  
}
```

```
class ElectricBus extends Vehicle {  
  
    function accelerate() {  
        $this->increaseVoltage();  
        $this->connectIndividualEngines();  
    }  
  
    private function increaseVoltage() {  
        // Electric logic  
    }  
  
    private function connectIndividualEngines() {  
        // Connection logic  
    }  
}
```

# L - Liskov substitution principle

A client class should be able to use either of them, if it can use Vehicle.

```
class Driver {  
    function go(Vehicle $v) {  
        $v->startEngine();  
        $v->accelerate();  
    }  
}
```

# I - Interface segregation principle

# I - 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.

*Да продължим с примера за геометричните фигури.*

Тъй като искаме да изчисляваме и обема на нашите геометрични фигури, може да добавим и друга декларация в ShapeInterface:

```
interface ShapeInterface {  
    public function calc_area();  
    public function calc_volume();  
}
```

# I - 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.

Всяка форма, която създаваме е задължена да имплементира и методът `volume`.

*Но не всяка форма има обем - квадрат, кръг ...*

Ако оставим геометричните класове, които дефенерахме да имплементират `ShapeInterface` в този му вид, задължаваме формите/техните класове/ да имплементират методи, от които нямат полза и/или са неадекватни за тях.

Това е в противоречие с Interface Segregation Principle.

Създаваме `SolidShapeInterface`, който има декларация за `volume`. Триизмерните фигури, като куб, цилиндър и т.н. ще имплементират този интерфейс.

# I - 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.

```
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  
    }  
}
```

# D - Dependency Inversion Principle

# D - Dependency Inversion Principle

A. High-level modules should not depend on low-level modules. Both should depend on abstractions.

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

```
class Mailer
```

```
{
```

```
// Methods for a Mailer class
```

```
}
```

```
class SendWelcomeMessage
{
    private $mailer;
    public function __construct(Mailer $mailer)
    {
        $this->mailer = $mailer;
    }
}
```

# D - Dependency Inversion Principle

A. High-level modules should not depend on low-level modules. Both should depend on abstractions.

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

```
class SendWelcomeMessage
{
    private $mailer;
    public function __construct(MailerInterface $mailer)
    {
        $this->mailer = $mailer;
    }
}
```

## Dependency Injection

# D - Dependency Inversion Principle

A. High-level modules should not depend on low-level modules. Both should depend on abstractions.

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

```
interface MailerInterface
{
    public function send();
}
```

```
class SmtpMailer implements MailerInterface
{
    public function send()
    {
        // Send an email via SMTP
    }
}
```

```
class SendSlackMailer implements MailerInterface
{
    public function send()
    {
        // Send a message via Slack
    }
}
```