



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

**PHP WebDevelopment 2019**

Милена Томова  
Vratsa Software

<https://vratsasoftware.com/>

# Table of Contents

1. S.O.L.I.D. /SOLID/ principles in OOP design
2. namespaces
3. traits





# SOLID principles



The **First 5 principles** in the OOP design, are as follows

**S** - Single-responsibility principle

**O** - Open-closed principle

**L** - Liskov substitution principle

**I** - Interface segregation principle

**D** - Dependency Inversion Principle



**S** - single  
responsibility  
principle

# Single responsibility principle

## TASK

**Create** a web application using OOP that will sum the areas of two different geometric figures /for example a circle and a triangle/.



# Single responsibility principle

**A class should have ONLY one job.**

## Step 1 Define

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

## Step 2 Define

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



# Single responsibility principle

**A class should have ONLY one job.**

Step 3 Define

```
class AreaCalculator {  
  
    protected $shapes;  
  
    public function __construct($shapes = []) {  
        $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();  
    }  
}
```



# Single responsibility principle

**A class should have ONLY one job.**

**Step 4** Create an array of geometric figures /objects of class Circle, Square etc./

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

**Step 5** Create an instance of class AreaCalculator. Pass the geometric figures array as a parameter.

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

**Step 6** Print the sum of all areas.

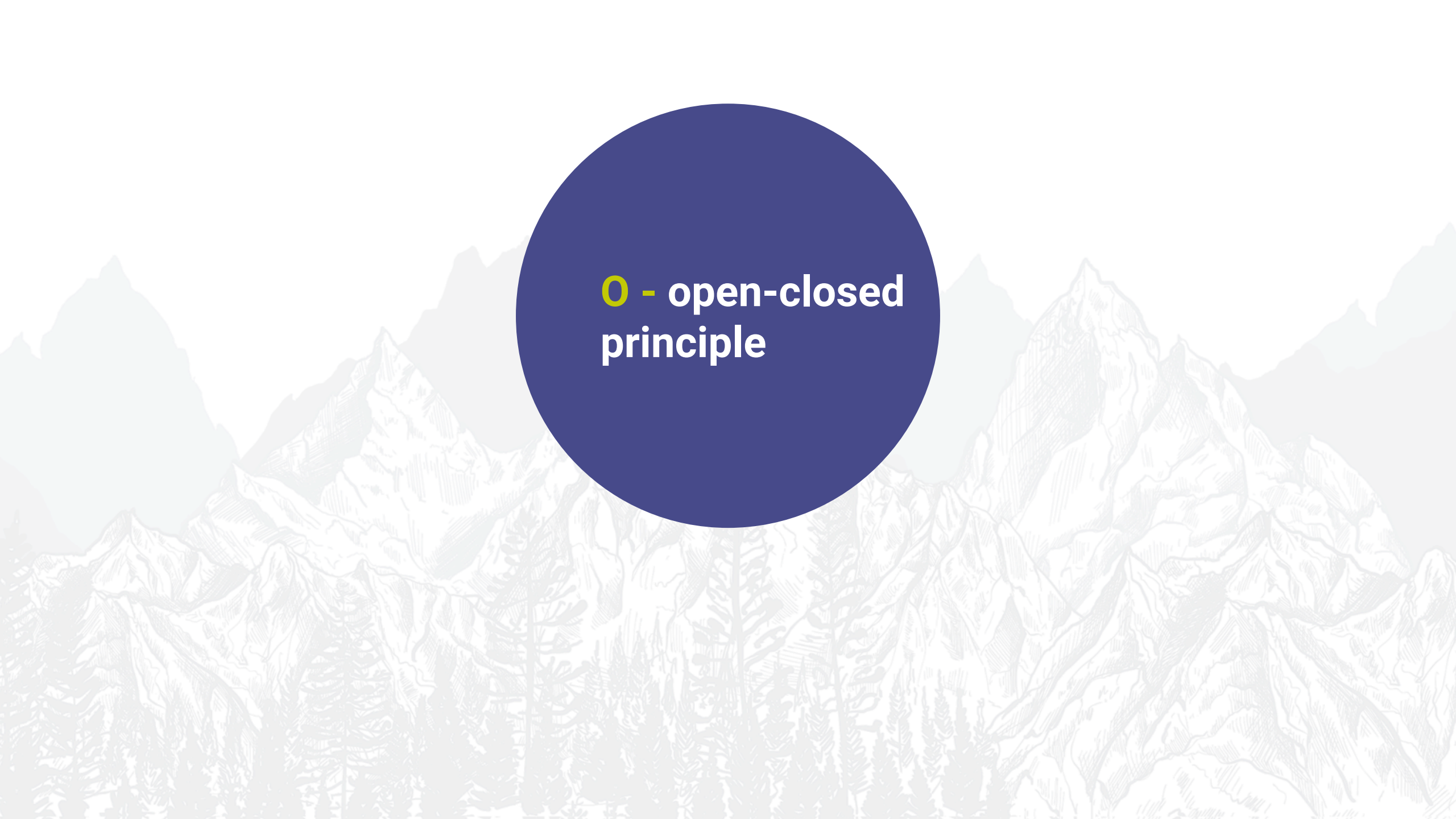
```
echo $areas->output();
```



# Single responsibility principle

## TASK 1

**B. Print** the results of AreaCalculator in different formats. /for example JSON, HTML/



**0 - open-closed  
principle**

# Open-closed principle

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

Lets implement **AreaCalculator**'s **sum** method this way

```
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);  
}
```

We have limited the application to calculate the areas of 2 type of figures.

How to resolve the problem of such limitation - to open the application to calculate and sum the areas of unlimited number of figures?



# Open-closed principle

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

We have limited the application to calculate the areas of 2 type of figures - Circles and Square.

How to resolve the problem of such inefficiency -  
to **open the application** to calculate and sum the areas of  
**unlimited number** of figures?

# Open-closed principle

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

## Variant A

- add more if/else blocks for every geometric figure ...in advance ... or later ...

**Which is against the 2nd SOLID principle.**

## Variant B

**Move** the logic of the area calculation **from AreaCalculator** class and **define it in** every single geometric figure class - **class Circle, class Square** etc.



# Open-closed principle

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

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

*Define **calc\_area** method  
in every geometric figure class.*



# Open-closed principle

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

Replace the **sum method** implementation in **AreaCalculator** with -

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

The method logic **does not depend** on the **type** or the **number** of the figures anymore.

# Open-closed principle

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

We can define a class for other geometric type of figure, create instances and include them in the **\$shapes** array to calculate the sum of areas.

*There are some important questions.*

- How to be sure that every **object** in **\$shapes** is a geometric figure?
- How to be sure that the class the object is instance of **has a definition of calc\_area()**?



# Open-closed principle

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

**Step 1** Define a iShape interface

```
interface iShape {  
    public function calc_area();  
}
```



# Open-closed principle

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

**Step 2** Every class responsible for a geometric figure will implement that interface

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


# Open-closed principle

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

**Step 3** The **sum** method of **AreaCalculator** checks if every object is an instance of a class that implements **iShape**.

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





**L** - liskov  
substitution  
principle



# 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

*If a class **S** **extends** the class **T**, an instance of **T** can **be replaced** with an instance of **S** without affecting the program expected results or causing any errors.*



# 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

## Step 1

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

## Step 2

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





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



# 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

## Step 3

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

# 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

## Step 4

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

A client class - **Driver**

should be able **to use** either of them -

**Vehicle,**  
**Car,**  
**ElectricBus**

if it can use **Vehicle**.



# SOLID principles

to be continued ...





**Trait**

Traits are a mechanism for code reuse in single inheritance languages such as PHP.

## Step 1

```
trait Validator {  
  
    public function is_positive_number( $num ) {  
        if( $num < 0 || $num == 0 ){  
            return "You need to provide dimension  
> 0!";  
        }  
    }  
}
```

- Declared with the keyword **trait** in front of the name;
- Define one or more **methods** for reuse in different, not connected by inheritance classes;
- If a class uses a trait - trait's methods **are available** for classes that inherit the class also;



## Step 2

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

- *Declare the class is using the trait with **use Validator**;*
- *Call the trait's methods from the class with **\$this** keyword;*
- *You are **not obliged** to call all trait's method in the class;*





# Namespaces

# Namespace

```
<?php
namespace Foo\Bar\subnamespace;

class Foo{

    static function staticmethod() {}

}
```

- *Declare the current namespace with the keyword **namespace**;*
- *The namespace must be declared **immediately after** the opening PHP tag - no blank lines allowed!*
- *The path that follows is the path that corresponds to the current file's **location in files hierarchy** of the project*
- *You are **not obliged** to call all trait's method in the class;*



# Namespace

```
<?php
namespace Foo\Bar\subnamespace2;

use namespace Foo\Bar\subnamespace\Foo;
use ...

class Demo {

    public function some_method(){
        //script
        Foo::staticmethod();
    }
}
```

- *Declare the current namespace with the keyword **namespace**;*
- ***After** the namespace declaration call the needed files with the **use** keyword;*
- *Call a class method by referring to **class name only**;*



# Questions?



Гнездото  
Coworking

Цялостен  
курс по  
програми  
ране

Дизайн  
курс

Курс по  
дигит.  
маркетинг

MindHub



# Partners



**Telerik  
Academy**



**MindHub**

**ПРОМЯНАТА**



# Trainings @ Vratsa Software



- Vratsa Software – High-Quality Education, Profession and Jobs
  - [www.vratsasoftware.com](http://www.vratsasoftware.com)
- The Nest Coworking
  - [www.nest.bg](http://www.nest.bg)
- Vratsa Software @ Facebook
  - [www.fb.com/VratsaSoftware](http://www.fb.com/VratsaSoftware)
- Slack Channel
  - [www.vso.slack.com](http://www.vso.slack.com)

