web разработка

ООП
S.O.L.I.D.
принципи

S.O.L.I.D. principles

Първите пет принципа в ОО дизайн.

S - Single-responsiblity 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

Един клас трябва да отговаря за едно единствено нещо ...

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;
        }
public function calc_area() {
    //логика за изчисляване на площта
    }
}
```

```
class Square {
        public $length;

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

        public function calc_area() {
    //логика за изчисляване на площта
    }
}
```

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 sum the areas
        }

        public function output() {
                return "Sum of the areas of provided shapes: ". $this->sum();
        }
}
```

Домашно - попълнете липсващия код

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();
```

- **Домашно - попълнете липсващия код в класовете**

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.

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

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

- **Домашно - попълнете липсващия код в класовете**

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 и искаме 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 и искаме да изчисляваме площта на неограничен брой геометрични фигури?

Вариант 1 - добавяме още if/elese 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 area() {
        return pow($this->length, 2);
    }
}
```

Същото ще направим и с 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->area();
    }

    return array_sum($area);
}
```

Сега можем да създадем друг клас за друга геометрична фигура и да подаваме негови обекти към 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 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();
            continue;
        } else {
                    .....
            }

      }

    return array_sum($area);
}
```

L - Liskov substitution principle

L - Liskov substitution principle

Обектите от класа наследник, трябва да могат да заменят обектите от родителския клас.

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

Обектите от класа наследник, трябва да могат да заменят обектите от родителския клас.

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);
    }
}
```

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 area();
    public function 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 …

Ние 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

Принципът засяга разделянето на програмния продукт на независими модули.

Следвайки този принцип модулите от по-високо ниво не следва да зависят от начина, по който са реализирани модулите от по-ниско ниво.

Принципът гласи -

A. Модулите от по-високо ниво не трябва да зависят от модули от по-ниско ниво. Всички те трябва да са зависими от абстракции/концепции/.

B. Абстрокциите не трябва да зависят от детайли. Детайлите трябва да зависят от абстракции.

D - Dependency Inversion Principle

Като резултат DIP намалява зависимостта между отделните части на кода.

Идеята зад DIP е следната - има много начини за имплементиране, например, на една логин-система. Начинът, по който ще я използвате трябва да бъде относително стабилен във времето. Ако може да обособите интерфейс, който да представи концепцията за логване, този интерфейс ще бъде много по-стабилен във времето в сравнение с неговата имплементация и съответно местата, които го извикват ще бъдат много по-слабо засегнати подобренията и нововъведенията в механизма на логване.

D - Dependency Inversion Principle

```
class PasswordReminder {

    private $dbConnection;

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

D - Dependency Inversion Principle

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.

След време решавате да смените вида база данни и ще трябва да промените класа **PasswordReminder**, в противоречие с **Open-close principle**.

В нашия пример - модулът от по-ниско ниво е /**MySQLConnection/** $dbConnection.

D - Dependency Inversion Principle

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.

Класът **PasswordReminder** не трябва да се интересува от това каква база данни използвате.

За да оправим това обособяваме интерфейс /абстракция/ - модулите могат да зависят от абстракция.

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

D - Dependency Inversion Principle

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.

```
class MySQLConnection implements DBConnectionInterface {
    public function connect() {
        return //конкретната реализация на връзката към БД
    }
}
```

D - Dependency Inversion Principle

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.

```
class PasswordReminder {
    private $dbConnection;

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

*И двата модула зависят от абстракция - интерфейсът **DBConnectionInterface***

D - Dependency Inversion Principle

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.

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

Забележка - освен име на клас, в скобите като зависимост можем да посочваме и интерфейс, който параметърът е <u>длъжен да имплементира</u>. При извикването на функцията, ако подадения параметър не имплементира посочения в зависимостта интерфейс ще възникне грешка.