

Programación orientada a objetos

PHP



Programación orientada a objetos

La **programación orientada a objetos (POO)** es un **paradigma de programación** basado en la organización del código en objetos. Es utilizada actualmente en la programación moderna debido a su capacidad de adaptarse al medio y modelar de manera efectiva el mundo real, además de promover la reutilización del código.

Para trabajar en PHP es imprescindible entender este paradigma, conocer sus elementos y sus principios fundamentales.

¡Empezamos!

Autor: Jesús Donoso

Introducción

Instancias

Constructores

Encapsulación

Abstracción

Herencia

Interfaces

Polimorfismo

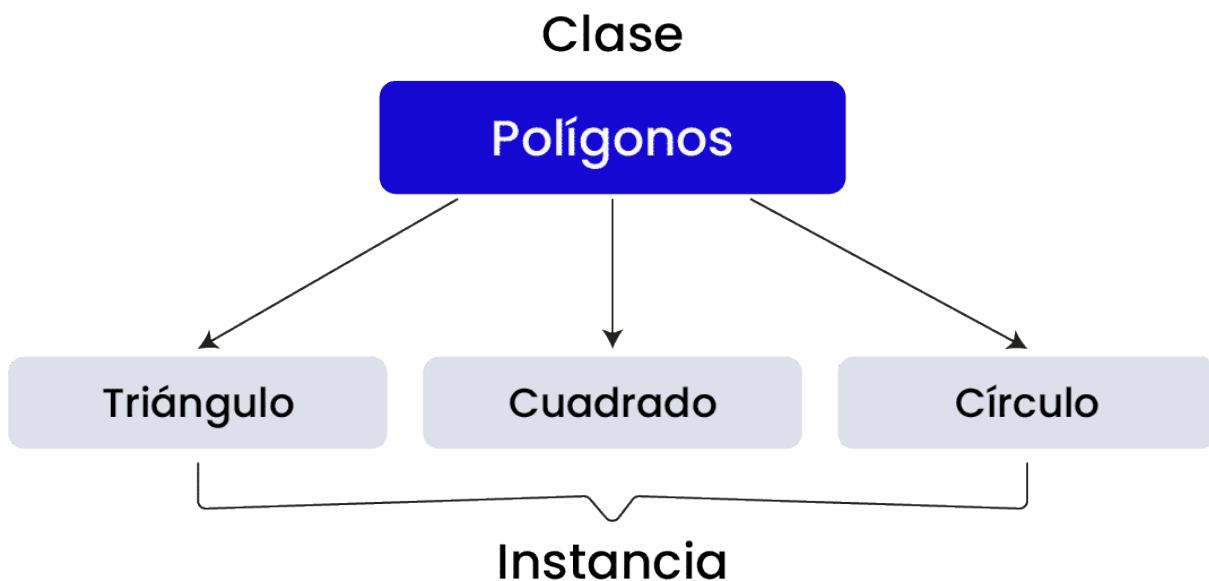
Principios SOLID

Organización de un Proyecto PHP con POO

Introducción



En la POO un objeto es una instancia de una clase, y una clase es una plantilla que define unas propiedades y comportamientos que los objetos pueden tener. En la imagen a continuación se puede observar un ejemplo sencillo de una clase *Polígono* y tres objetos *Triángulo*, *Cuadrado* y *Círculo*.



A continuación, te detallo algunos de los principales conceptos en la programación orientada a objetos, que seguro conoces, pero no está demás refrescarlos en la memoria.

- Objetos:** son instancias de clases que encapsulan datos (atributos) y comportamientos (métodos). En el ejemplo anterior, los objetos o instancias son los tres polígonos.
- Atributos:** se llama así a las propiedades o características que tiene un objeto. En ejemplo de los polígonos, sus propiedades o atributos pueden ser el número de lados o el número de ángulos.
- Métodos:** son las funciones o procedimientos que definen el comportamiento de un objeto. Estos métodos también pueden consultar o modificar los valores de los atributos. Un ejemplo de un método con polígonos sería contar el número de lados.
- Clases:** son plantillas o moldes que definen la estructura y comportamiento de los objetos. La clase controla o especifica qué atributos se necesitan para crear un objeto.
- Encapsulación:** dentro de una clase, los atributos y métodos pueden ser públicos, protegidos o privados. Esto es necesario para mantener la integridad de los datos.
- Herencia:** permite la creación de nuevas clases basadas en una clase padre. Este es un concepto fundamental en la programación orientada a objetos para la reutilización de código.
- Polimorfismo:** permite que objetos de diferentes clases sean tratados de manera uniforme si comparten una interfaz común. Esto simplifica el código y permite la flexibilidad en la implementación de clases.

El lenguaje PHP admite este tipo de programación y, por lo tanto, permite crear clases y objetos para desarrollar aplicaciones más organizadas y reutilizables.

Estándares

En la programación orientada a objetos, los estándares se refieren a las **convenciones y buenas prácticas** que se siguen para escribir código orientado a objetos de manera coherente y legible. Estos estándares son importantes para mantener el código más organizado, por lo tanto, facilita la colaboración en proyectos de desarrollo de software y mejora la compresión y el mantenimiento del código.

Estos son algunos de los estándares más comunes:

- **Nomenclatura de clases y métodos:** la convención de nomenclatura *CamelCase* es la más utilizada para nombrar clases y métodos. Los nombres de clases comienzan con letra mayúscula *MiClase*; mientras que los nombres de métodos empiezan con minúscula *miMetodo*.
- **Encapsulación de propiedades y métodos:** se utilizan las palabras claves *public*, *protected* y *private* para definir la visibilidad de una propiedad o un método.
- **Documentación:** es importante hacer uso de comentarios de documento. En el caso de PHP se aplica *PHPDoc* para describir el propósito de las clases, métodos o propiedades; así como para definir el tipo de dato esperado y retornado. Esto facilita la comprensión de nuestro código.

Instancias



Qualentum Lab

En la programación orientada a objetos, y por lo tanto en PHP, una instancia es la **creación de un objeto a partir de una clase**. Una clase es una plantilla o un modelo que define una estructura y un comportamiento de un objeto; mientras que una instancia es un objeto real que se crea en tiempo de ejecución a partir de la clase.

A continuación, se puede observar el código de una clase y de la declaración de una clase y la creación de una instancia:

```
<?php
class Coche {
    public $marca;
    public $modelo;
    public function arrancar() {
        echo "El coche está arrancando";
    }
}

// Crear una instancia (objeto) de la clase Coche
$miCoche = new Coche();
```

En el ejemplo anterior se puede observar la clase *Coche* declarada con la palabra clave *class* y el nombre de la clase. Dentro de la misma se pueden ver declarados dos atributos de un coche, marca y modelo, y la función arrancar.

Igualmente se puede ver cómo, haciendo uso de la palabra clave *new* para la declaración de instancias, se crea una instancia de la clase *Coche* llamada *miCoche*.

En el fragmento de código siguiente se observa cómo al objeto creado se le añaden valores, a los atributos, y se realiza una llamada al método declarado en la clase. También se puede ver cómo se accede a los valores asignados a los atributos.

```
// Asignar valores a las propiedades de la instancia
$coche->marca = "Volkswagen";
$coche->modelo = "Golf";

// Llamar a un método de la instancia
$coche->arrancar();

// Acceder a las propiedades de la instancia
echo "Marca del coche: ". $coche->marca; //Echo Marca del coche: Volkswagen
echo "Modelo del coche ". $coche->modelo;//Echo Modelo del coche :Golf
```

Las instancias son un concepto básico dentro de la POO. Uno de los pros de este tipo de programación es la posibilidad de creación de instancias de la misma clase sin que se vean afectadas otras creadas previamente. Esto facilita la gestión de información y mantenimiento del código.

Constructores



Qualentum Lab

Los constructores son métodos especiales declarados dentro de las clases que se llaman automáticamente en el momento de la creación de una instancia.

Un constructor se utiliza para inicializar las propiedades básicas de un objeto y realizar las configuraciones necesarias al crear la instancia. ¿Y cómo se define el constructor de una clase?

Con la palabra clave `__construct()`. Observa un fragmento de código con la declaración de un constructor:

```
<?php
class Coche {
    public $marca;
    public $modelo;
    public function __construct($marca, $modelo)
    {
        $this->marca = $marca;
        $this->modelo = $modelo;
    }
}
// Crear una instancia (objeto) de la clase coche
$miCoche = new Coche("Volkswagen", "Golf");
```

En este ejemplo, el constructor de la clase *Coche* requiere como parámetros los atributos marca y modelo. Se puede ver cómo, para la creación de la instancia, se introducen directamente los valores de marca y modelo. El constructor se ejecuta automáticamente y proporciona los valores introducidos a las propiedades.

Si en el momento de la creación de una instancia no se introducen los atributos declarados como argumentos del constructor la ejecución del script se detiene debido a un error. Por otro lado, si no se define un constructor en una clase, PHP proporciona uno por defecto que no realiza ninguna acción específica.

**Hay que incidir en que los constructores son muy útiles
para asegurar que los atributos de los objetos tengan
los valores que se consideren necesarios en el
momento de su creación.**

Encapsulación



Qualentum Lab

La encapsulación es uno de los principios fundamentales de la POO y restricción del acceso a detalles internos de un objeto y protección de sus datos.

En PHP, se puede lograr la encapsulación utilizando tres tipos de acceso a propiedades y métodos: público, protegido y privado.

A continuación, se detallan estos **tres tipos de acceso** y se muestran fragmentos de código PHP de cada uno de ellos.

Acceso público

Las propiedades y métodos públicos son accesibles desde cualquier lugar, ya sea desde dentro de la misma clase, desde clases derivadas o desde código externo que instancia al objeto. Para ello, como se puede ver en el siguiente ejemplo, se hace uso de la palabra clave *public* antes de la declaración de la propiedad o la función.

```
<?php
class Clase {
    // Acceso a la propiedad y la función desde cualquier sitio
    public $propiedadPublic;
    public function metodoPublic(){}
}
?>
```

Acceso protegido

Las propiedades y métodos protegidos son aquellos que solo son accesibles dentro de la misma clase y desde clases derivadas. Se definen con la palabra clave *protected*. El concepto de clase derivada se detalla más adelante en este documento.

```
<?php
class Clase {
    // Acceso a la propiedad y la función desde cualquier clase o
    // clases derivadas
    protected $propiedadProtected;
    protected function metodoProtected($valor) {}
}
?>
```

Acceso privado

Las propiedades o métodos privados son aquellos únicamente accesibles desde la misma clase en la que están definidos. Se definen haciendo uso de la palabra clave *private*.

```
<?php
class Clase {
    // Acceso a la propiedad y la función únicamente desde la misma clase
    private $propiedadPrivate;
    private function metodoPrivate() {}
}
?>
```

Es importante tener en cuenta que el constructor de una clase siempre debe ser una función pública. Un constructor privado imposibilita la creación de objetos, ya que no se puede llamar a dicha función.

En resumen, **la encapsulación permite controlar quién accede y puede modificar las propiedades y métodos de una clase**. Es una forma de protección de las propiedades internas de una clase. Un desarrollador puede definir reglas y validaciones que impidan la modificación de valores de propiedades si no se cumplen.

A continuación, te comarto un ejemplo de código de una clase con una propiedad privada y métodos públicos.

```
<?php
class Usuario {
    private $nombre;
    public function __construct($nombre){
        $this->nombre = $nombre;
    }
    public function getNombre(){
        return $this->nombre;
    }
    public function setNombre($nombre){
        if ($nombre != ''){
            $this->nombre = $nombre;
        }
    }
}
// Creación de un usuario llamado José
$usuario = new Usuario('José');
//Acceder al atributo directamente resulta en error porque es privado
echo "El nombre del usuario es: ". $usuario->nombre;
// Acceder al atributo a través de la función pública getNombre()
echo "El nombre del usuario es: ". $usuario->getNombre();
// Se modifica a través de la función pública setNombre()
$usuario->setNombre('Luis');
// La siguiente instrucción no cambia el valor del nombre
$usuario->setNombre('');
?>
```

En este ejemplo, hemos podido observar cómo la propiedad privada *nombre* no se puede acceder ni modificar desde fuera de la clase, ya que está encapsulada. Para consultar o modificar el valor del atributo privado *nombre* es necesario hacer uso de las funciones públicas *getNombre* y *setNombre*. También se puede ver que en la función *setNombre* se encuentra una sentencia de control que evita que se asignen cadenas de texto vacías.

De esta forma se consigue que los datos o la información de una clase sea únicamente accesible para su consulta o modificación a través de los métodos públicos que se crean para ello. Así se consigue un mayor control sobre los cambios de valores y la obtención de información.

Abstracción



La abstracción es uno de los pilares fundamentales de la programación orientada a objetos. Este concepto se refiere a la capacidad que tiene este tipo de programación de representar un objeto del mundo real de manera más simplificada. Enfocándose en las características más importantes y ocultando detalles innecesarios en el problema del mundo real.

Para entender mejor la definición de abstracción se puede pensar en la clase *Alimentos*. Los alimentos pueden tener diversos atributos comunes a todos los alimentos del mundo. La clase *Alimentos* puede ser una clase abstracta y que una clase *Verduras* se extienda de *Alimentos*. Al mismo tiempo la clase *Verduras* también puede ser una clase abstracta y que una clase *Cebolla* se extienda de ella.

En definitiva, la abstracción implica enfocarse en los aspectos esenciales de un objeto en el momento de programar una clase, ocultando así detalles internos no relevantes para el problema que se está resolviendo en un momento específico.

En definitiva, la abstracción implica enfocarse en los aspectos esenciales de un objeto en el momento de programar una clase, ocultando así detalles internos no relevantes para el problema que se está resolviendo en un momento específico.

Definición de clase abstracta y clase extendida

Una clase abstracta no se puede instanciar directamente y generalmente se utiliza como una plantilla para otras clases. Las clases abstractas se declaran con la palabra clave *abstract*. En una clase abstracta se definen propiedades y métodos que son comunes a los objetos creados en clases derivadas. A continuación, se observa un fragmento de código con la creación de una clase abstracta:

```
<?php
abstract class FiguraGeometrica {
    private $lados;
    public function __construct($lados)
    {
        $this->lados = $lados;
    }
    abstract public function calcularArea();
}
?>
```

Como se puede observar en el ejemplo anterior, dentro de la clase abstracta se encuentra el atributo *lados* común a todas las figuras geométricas. Por otra parte, se define una función también abstracta para calcular el área de las figuras geométricas. Se puede observar que esta función no tiene programada una fórmula de cálculo de área, ya que será algo a definir en las clases extendidas de cada una de las figuras geométricas.

En el fragmento de código que muestro a continuación, se observa la creación de una clase *círculo* que extiende de la clase abstracta de figuras geométricas creada en el ejemplo anterior. Es necesario hacer uso de la palabra clave *extends*.

```
<?php
class Circulo extends FiguraGeometrica {
    private $radio;
    public function __construct($lados, $radio) {
        if ($lados == 0) {
            parent::__construct($lados);
            $this->radio = $radio;
        }
    }
    public function calcularArea() {
        return pi() * pow($this->radio , 2);
    }
}
?>
```

Se deben implementar todos los métodos definidos en la clase abstracta para la creación de objetos de la clase extendida. Por ejemplo, en la clase círculo se puede observar cómo se implementa el método de calcular el área específicamente para un círculo.

Y en el siguiente fragmento de código se crea un objeto círculo tras la implementación de las dos clases ya vistas en los ejemplos anteriores:

```
<?php
$circulo = new Circulo(0, 5);
echo "Área del circulo: " . $circulo->calcularArea() . "unidades
cuadradas\n";
?>
```

En último lugar, se puede observar, implementado en código PHP, un ejemplo distinto al de las figuras geométricas correspondiente al que se enunció al principio del apartado:

```
<?php
//Creación de la clase abstracta alimentos
abstract class Alimentos {
    private $color;
    public function __construct($color)
    {
        $this->color = $color;
    }
    abstract function tiempoCoccion($tiempo);
}
//Creación de la clase abstracta verduras que extiende de alimentos
abstract class Verduras extends Alimentos {
    private $nombre;
    public function __construct($nombre, $color)
    {
        parent::__construct($color);
        $this->nombre = $nombre;
    }
}
//Creación de la clase abstracta cebolla que extiende de verduras
class Cebolla extends Verduras {
    public function __construct($color, $nombre)
    {
        parent::__construct($nombre, $color);
    }
    public function tiempoCoccion($tiempo) {
        return '15 minutos';
    }
}
?>
```

En el ejemplo vemos cómo el método abstracto del tiempo de cocción se implementa en la clase que ya no es abstracta. También observamos cómo en la clase intermedia se añade un atributo más al constructor y en la clase extendida simplemente se usa el mismo.

En resumen, en este apartado hemos explicado cómo la abstracción permite trabajar a los desarrolladores con conceptos a alto nivel sin la necesidad de conocer los detalles de la implementación a niveles más bajos.

Herencia



La herencia es otro de los pilares fundamentales de la programación orientada a objetos. Al igual que la abstracción, este concepto permite crear una nueva clase a partir de una ya existente. Sin embargo, en este caso, la clase extendida hereda las propiedades y métodos ya implementados en la clase padre.

A continuación, se puede observar como ejemplo la implementación en código PHP de una clase *coche*:

```
<?php
Class Coche {
    protected $marca;
    public function __construct($marca)
    {
        $this->marca = $marca;
    }
    public function arrancar() {
        return "Arranca motor";
    }
}
?>
```

La clase *coche* se implementa al igual que cualquier otra clase que se haya visto en este documento. Tiene un atributo protegido, un constructor que necesita la marca para crear el objeto *coche* y una función *arrancar*. A continuación, se observa la implementación de otras dos clases que heredan de esta:

```
<?php
// Clase derivada de Coche
class cocheF1 extends Coche {
    public function arrancar()
    {
        return "Arranca F1";
    }
    public function correr()
    {
        return "Fiiiiuuuumm";
    }
}

// Otra clase derivada de Coche
class cocheUtilitario extends Coche {
    public function frenar()
    {
        return "Frena el coche";
    }
}
?>
```

Como podemos observar, para la implementación de la herencia se hace uso de la palabra clave *extends* seguida del nombre de la clase padre. La clase derivada hereda las propiedades y los métodos de la clase padre (superclase) y al mismo tiempo puede agregar nuevas propiedades o métodos.

En el ejemplo anterior también vemos cómo en la clase de coche de fórmula 1 se sobrescribe el método *arrancar*. Por lo tanto, los objetos de coche de fórmula 1 utilizan el método de su propia clase, mientras que los coches utilitarios hacen uso del método de la clase padre. A continuación, te muestro un ejemplo de código con la creación de los objetos:

```
<?php
$cocheF1 = new cocheF1('Aston Martin');
$cocheUtilitario = new cocheUtilitario('Seat');

echo $cocheF1->arrancar(); //Echo Arranca F1
echo $cocheUtilitario->arrancar(); //Echo Arranca motor

echo $cocheF1->correr(); //Echo Fiiiuuuuumm
echo $cocheUtilitario->frenar(); //Echo Frena el coche
?>
```

Ambos objetos se inicializan con el atributo de la marca, ya que es el único que necesita la clase padre y en las clases derivadas no se ha indicado otro. Sin embargo, como hemos comentado previamente, la llamada a la función *arrancar* desde cada uno de los objetos no tendría el mismo resultado.

Al igual que en la herencia, una clase derivada al mismo tiempo puede ser la clase padre de otra. Esto fomenta la reutilización del código y la organización jerárquica de conceptos. Facilitando así la extensión y modificación de funcionalidades y mejorando la eficiencia y la mantenibilidad del código.

Al igual que en la herencia, una clase derivada al mismo tiempo puede ser la clase padre de otra. Esto fomenta la reutilización del código y la organización jerárquica de conceptos. Facilitando así la extensión y modificación de funcionalidades y mejorando la eficiencia y la mantenibilidad del código.

Interfaces



Qualentum Lab

Las interfaces en la programación orientada a objetos son una herramienta poderosa que permite definir un conjunto de métodos o propiedades que deben ser implementados por cualquier clase que haga uso de esa interfaz.

Para la creación de una interfaz en PHP se hace uso de la palabra clave *interface*. Dentro de la interfaz se declaran los métodos que deben ser implementados en las diferentes clases que use la interfaz. Estos métodos no se implementan, sino que únicamente se definen. Es decir, se indican los parámetros necesarios y el tipo de retorno si es necesario especificarlo.

En el siguiente ejemplo se puede observar la definición de una interfaz llamada *Animal*. Cualquier clase que haga uso de esta interfaz debe implementar los métodos *hablar()* y *patas()*.

```
<?php
interface Animal {
    public function hablar();
    public function patas($numero);
}
?>
```

Para implementar una interfaz se debe hacer uso de la palabra clave *implements* en la declaración de clase. A continuación, muestro un ejemplo de una declaración de una clase que implementa la interfaz creada antes:

```
<?php
interface Animal {
    public function hablar();
}
class Gato implements Animal {
    public function hablar() {
        return "Miau";
    }
    public function patas($numero) {
        return `El gato tiene {$numero} patas`;
    }
}
?>
```

Si te fijas, se puede ver cómo la clase *Gato* implementa la interfaz *Animal* y, por lo tanto, a su vez se implementan los métodos *hablar()* y *patas(\$numero)*. Pues bien, ahora vamos a ver la creación de un objeto *gato* que hace uso de los métodos implementados:

```
<?php
$gato = new Gato();
echo $gato->hablar(); //Echo Miau
echo $gato->patas(4); //Echo: El gato tiene 4 patas
?>
```

Varias clases pueden hacer uso de una misma interfaz. Esto facilita la extensión y el mantenimiento del código, ya que se pueden agregar nuevas clases que cumplan con la misma interfaz sin afectar a las clases ya existentes.

Por otro lado, es importante tener en cuenta, que si se modifica una interfaz también se deben modificar todas las clases que la implementen. Por lo tanto, es recomendable realizar una buena definición de la interfaz antes del desarrollo de las clases.

Recuerda: las interfaces son especialmente útiles para garantizar que las clases cumplan con ciertos estándares o comportamientos. Logrando así un alto nivel de abstracción en el código.

Polimorfismo



Qualentum Lab

El polimorfismo es la capacidad que pueden tener los diferentes objetos de las clases para responder a la llamada de un método, de manera que tenga sentido en el contexto de cada objeto.

Esta propiedad ayuda a escribir código de manera más genérica y reutilizable, ya que se puede trabajar con objetos de diferentes clases. En PHP podemos implementar el polimorfismo a través de la **herencia** y las **interfaces**.

Polimorfismo y herencia

La forma más común de lograr el polimorfismo en PHP es mediante la herencia. En la clase padre se implementa un método que en las clases derivadas se sobrescribe. Para ello, en la clase derivada se crea un método con el mismo nombre que en la clase padre. Sin embargo, se implementa un comportamiento diferente en la clase derivada.

Vamos a analizar este ejemplo en código.

```
<?php
class Animal {
    public function patas() {
        return "Patas del animal";
    }
}

class Gallo extends Animal {
    public function patas() {
        return "Tiene 2 patas";
    }
}

class Elefante extends Animal {
    public function patas() {
        return "Tiene 4 patas";
    }
}

$gallo = new Gallo();
$elefante = new Elefante();

echo $gallo->patas(); //Echo Tiene 2 patas
echo $elefante->patas(); //Echo Tiene 4 patas
?>
```

Si te fijas, la creación de la clase *Animal* con el método *patas()* devuelve “*Patas del animal*”. Las clases *Gallo* y *Elefante* heredan de la clase *Animal* y ambas implementan el método *patas()* con el número exacto de patas del animal. Las clases derivadas sobrescriben el método de la clase padre. Esto permite que diferentes objetos respondan de manera polimórfica el método *patas()*.

Polimorfismo e interfaces

Otra forma de implementar en PHP el polimorfismo es a través de las interfaces. Las interfaces, como ya se ha visto en el apartado correspondiente, definen el conjunto de propiedades y métodos que deben de ser implementados en cada clase que las interprete.

Esto permite que diferentes clases interpreten la misma interfaz, por lo tanto, respondan de manera polimórfica a los métodos definidos. A continuación, comparto un ejemplo en código:

```
<?php
interface Animal {
    public function patas();
}

class Gallo extends Animal {
    public function patas() {
        return "Tiene 2 patas";
    }
}

class Elefante extends Animal {
    public function patas() {
        return "Tiene 4 patas";
    }
}

$gallo = new Gallo();
$elefante = new Elefante();

echo $gallo->patas(); //Echo Tiene 2 patas
echo $elefante->patas(); //Echo Tiene 4 patas
?>
```

En este ejemplo las Clases *Gallo* y *Elefante* implementan la interfaz *Animal*. Esto garantiza que deben implementar el método *patas()*. De esta manera permite que los objetos de estas clases compartan el método *patas()*.

**El polimorfismo mejora el trabajo con diferentes
objetos de manera uniforme, lo que facilita la extensión
y reutilización de código.**

Principios SOLID



Qualentum Lab

SOLID es el acrónimo de cinco principios de diseño para la programación, los que promueven buenas prácticas en el desarrollo de software y facilitan la creación de código limpio, modular y mantenible. Estos principios fueron establecidos por **Robert C. Martin**, autor también de los conocidos principios del desarrollo *Agile*.

Los **principios SOLID** son los siguientes:

S	<i>Single responsibility principle.</i>
O	<i>Open/closed principle.</i>
L	<i>Liskov substitution principle.</i>
I	<i>Interface segregation principle.</i>
D	<i>Dependency inversion principle.</i>

A continuación, vamos a dedicar un apartado para cada uno de los principios con el objetivo de que comprendas a fondo cada uno de ellos.

1

Principio de responsabilidad única (*Single responsibility principle*)

Este principio establece que una clase debe tener una única responsabilidad y, por lo tanto, solo podría haber una única razón válida para modificarla.

```
<?php
class Coche {
    public $marca;
    public $modelo;
    public function __construct($marca, $modelo)
    {
        $this->marca = $marca;
        $this->modelo = $modelo;
    }
    public function obtenerInfo() {
        return "Marca: {$this->marca}, Modelo: {$this->modelo}";
    }
}
class Motor {
    public function encender() {}
    public function apagar() {}
}
?>
```

En el ejemplo anterior se puede observar la clase *Coche*. Esta clase tiene la responsabilidad de representar un coche y proporcionar los métodos necesarios para acelerar, frenar y obtener información acerca del coche.

La clase *Coche* no se encarga de proporcionar los métodos con las funcionalidades del motor. Para ello, existe una clase *Motor* explícita que representa los motores como objetos.

Al seguir este principio, se está asegurando que la clase *Coche* se concentre en una única responsabilidad: **gestionar la información y comportamiento** relacionado con un coche. Esto hace que el código sea más organizado y fácil de mantener.

2

Principio de abierto/cerrado (*Open/closed principle*)

Las clases, módulos, funciones y otros elementos del software deben estar abiertos para la extensión, pero cerrados para la modificación. Esto significa que deben poder agregarse nuevas funcionalidades sin cambiar el código existente.

```
<?php
interface FiguraGeometrica {
    public function calcularArea();
}

class Circulo implements FiguraGeometrica {
    private $radio;
    public function __construct($radio)
    {
        $this->radio = $radio;
    }
    public function calcularArea()
    {
        return pi() * pow($this->radio, 2);
    }
}
class Cuadrado implements FiguraGeometrica {
    private $lado;
    public function __construct($lado)
    {
        $this->lado = $lado;
    }
    public function calcularArea()
    {
        return pow($this->lado, 2);
    }
}
?>
```

En este caso, se ha creado una interfaz llamada *FiguraGeometrica* que define el método *calcularArea()*, pero no proporciona una implementación. Las clases *Circulo* y *Cuadrado* implementan esta interfaz y, por lo tanto, deben desarrollar la lógica necesaria para el cálculo del área de cada una de las figuras respectivamente.

De esta manera, si se quisiera agregar un triángulo, simplemente se crearía la clase que represente dicho objeto y que implemente la interfaz *FiguraGeometrica*. Igualmente se proporcionaría la lógica necesaria para el cálculo de su área sin necesidad de modificar el código existente.

3

Principio de sustitución de Liskov (*Liskov substitution principle*)

Los objetos de una clase derivada pueden reemplazar objetos de la clase base sin afectar la corrección del programa.

```
<?php
class Ave {
    public function volar() {
        return "Volando alto en el cielo";
    }
}

class Pinguino extends Ave {
    public function volar() {
        return "No puedo volar, soy un pingüino";
    }
    public function nadar() {
        return "Nado en el agua";
    }
}
?>
```

En este caso, la clase derivada *Pingüino* hereda de la clase *Ave*, pero sobrescribe el método *volar()* con un comportamiento específico para el pingüino. Esto cumple con el principio LSP, el que estamos explicando, ya que los objetos de tipo pingüino pueden sustituir a los objetos de tipo ave sin afectar a la corrección del programa.

Igualmente se puede observar cómo en la clase *Pingüino* se implementa una función *nadar()*. De esta manera, el resultado de la ejecución de las funciones que pueden llevar a cabo los pingüinos es coherente con la naturaleza de estos.

4

Principio de segregación de interfaces (*Interface segregation principle*)

El principio ISP establece que una clase no debe verse obligada a depender de métodos que no necesita. Es decir, es mejor desarrollar interfaces más específicas en vez de una más general que obligue en algunos casos a la implementación de métodos que no son necesarios.

```
<?php
interface Trabajador {
    public function trabajar();
    public function tomarDescanso();
    public function aprobarGastos();
    public function emitirNominas();
}

class Empleado implements Trabajador {
    public function trabajar(){//Implementación}
    public function tomarDescanso(){//Implementación}
    public function aprobarGastos(){//Implementación}
    public function emitirNominas(){//Implementación}
}
?>
```

En el ejemplo, la clase *Empleado* implementa la interfaz *Trabajador*. Se puede observar que en la interfaz se declaran los métodos *trabajar()* y *tomarDescanso()*, que son funcionalidades comunes a todos los empleados. Sin embargo, se obliga a que la clase *Empleado* implemente los métodos *aprobarGastos()* y *emitirNomina()*. Estas acciones representan tareas que no realizan todos los empleados de una empresa.

En el ejemplo a continuación se modifica el código anterior para cumplir con el principio ISP. Por un lado, existe una interfaz *Trabajador()* que contiene los métodos comunes a todos los trabajadores de una empresa independientemente de su puesto de trabajo. Por otro lado, se declara otra interfaz *Administración* que contiene las tareas que realiza el personal administrativo.

La clase *Empleado* ahora solo debe implementar los métodos *trabajar()* y *tomarDescanso()*. Aparece un nuevo elemento, la clase *EmpleadoAdministrativo* que extiende de la clase *Empleado* e implementa la interfaz *Administración*. Ahora, los objetos que representan a los empleados administrativos pueden ejecutar las funciones comunes a cualquier empleado y las específicas de su puesto de trabajo.

```
<?php
interface Trabajador {
    public function trabajar();
    public function tomarDescanso();
}
interface Administracion {
    public function aprobarGastos();
    public function emitirNomina();
}

class Empleado implements Trabajador {
    public function trabajar(){//Implementación}
    public function tomarDescanso(){//Implementación}
}
class EmpleadoAdministrativo extends Empleado implements Administracion {
    public function aprobarGastos(){//Implementación}
    public function emitirNomina(){//Implementación}
}
?>
```

En conclusión, el principio ISP promueve la cohesión y la flexibilidad en el diseño de las interfaces, lo que facilita la creación de interfaces más específicas y evita que las clases deban implementar funcionalidades que no necesitan. Esto mejora la **modularidad** y la **mantenibilidad** del código.

5

Principio de inversión de dependencias (*Dependency inversion principle*)

Dicho principio se centra en **reducir el acoplamiento** entre módulos y clases de un sistema software. Por lo tanto, se apoya en estas dos premisas:



Que los módulos de alto nivel no deben depender de módulos de bajo nivel. Esto significa que las clases que contienen la lógica principal de una aplicación (módulos de alto nivel) no deben depender directamente de las clases o los módulos que implementan detalles de bajo nivel. En cambio, ambos deben depender de abstracciones.



Que las abstracciones no deben contener detalles. Las abstracciones (interfaces o clases) deben definir contratos sin especificar cómo se implementan. Los detalles (las implementaciones específicas) dependen de las abstracciones, no al revés.

A continuación, te muestro un ejemplo en código para comprender este principio. El ejemplo se basa en un sistema de notificaciones que envía mensajes a diferentes destinatarios.

En primer lugar

Se define una interfaz *Notificador* con una función *enviarMensaje()* que es la funcionalidad básica del sistema.

```
<?php
interface Notificador {
    public function enviarMensaje($mensaje);
}
?>
```

En segundo lugar

Se crean las clases que implementan esta interfaz y que representan las vías por las que se hace envío del mensaje.

```
<?php
class Mail implements Notificador {
    public function enviarMensaje($mensaje){//Implementación}
}
class Sms implements Notificador {
    public function enviarMensaje($mensaje){//Implementación}
}
?>
```

Finalmente

Se desarrolla una clase de alto nivel *SistemaDeNotificacion* para la gestión del envío de las notificaciones.

```
<?php
//Clase principal
class SistemaDeNotificacion {
    private $notificador;
    public function __construct(Notificador $notificador) {
        $this->notificador = $notificador;
    }
    public function notificarUsuario($mensaje) {
        $this->notificador->enviarMensaje(($mensaje));
    }
}

$sms = new Sms();
$notificacion = new SistemaDeNotificacion($sms);
$notificacion->notificarUsuario(); //Se envía el sms
?>
```

La clase *SistemaDeNotificacion* solicita en su constructor un objeto de tipo notificador. Es decir, los objetos de esta clase necesitan objetos del tipo SMS, mail u otro que identifique la vía por la que se envía un mensaje.

De este modo el sistema se hace más flexible y extensible, permitiendo añadir nuevos métodos de notificación sin tener que modificar la lógica de alto nivel de la clase *SistemaDeNotificacion*.

Organización de un Proyecto PHP con POO



En este último apartado, quiero resumir una serie de pautas muy recomendables para la creación de proyectos. Estas pautas ayudan a desarrollar un código óptimo, limpio y bien estructurado para un fácil mantenimiento. A continuación, te las detallo, ¡toma buena nota de ellas!

1

La división del código en clases y objetos: es importante basarse en la creación de clases y objetos. Cada clase debe de tener una responsabilidad única y bien definida. Se deben organizar las clases según sus funcionalidades y responsabilidades.

2

El uso de namespaces: los espacios de nombres son útiles para agrupar clases relacionadas y evitar conflictos de nombres. En términos más simples, es un contenedor que agrupa un conjunto de nombres y estos son únicos dentro de ese contenedor.

3

La organización de archivos de clases: es fundamental mantener los archivos en clases en un directorio dedicado solamente a su funcionalidad.

4

El uso de estándares de nomenclatura: *PascalCase* para nombrar las clases y *CamelCase* para nombrar atributos o métodos.

5

La encapsulación: aplica los principios de encapsulación de propiedades y métodos en el acceso de información dentro de una clase.

6

La división de archivos: separa las clases en archivos individuales. Cada archivo debe de tener una clase sola y dicho fichero debe coincidir con el nombre de la clase.

7

La documentación: utiliza comentarios para describir tus clases, métodos y propiedades.

8

La división de responsabilidades: aplica el principio de responsabilidad única para que cada clase tenga una única responsabilidad. Comprueba que un método no se vuelve demasiado extenso o tosco, si es así considera su división.

9

Evitar duplicidad de código: reutiliza código a través de la herencia, composición o la creación de clases utilitarias.

10

La gestión de dependencias: aplica la inyección de dependencias para proporcionar información.

11

Hacer pruebas unitarias: utiliza pruebas unitarias de los métodos para que la funcionalidad funcione correctamente.

12

Utilizar patrones de diseño: usa patrones de diseño software (como Singleton, Factory, Observer, etc.) y aplícalos cuando sean necesarios.

¡Recuerda! El uso de estas y otras buenas prácticas de gestión y organización de código en PHP en la programación orientada a objetos es importante para la creación de aplicaciones más sostenibles y escalables.

Conclusiones

Durante este documento se ha podido observar y comprender como la programación orientada a objetos es clave para la creación de un código modular y reutilizable. La aplicación de conceptos como **la abstracción, el encapsulamiento, la herencia o el polimorfismo** permite desarrollar aplicaciones eficientes y fáciles de mantener. Por su lado, la **estructura jerárquica en clases** facilita la escalabilidad de los proyectos y la inclusión de nuevas funcionalidades sin comprometer la integridad del código ya existente.

En conclusión, la programación orientada a objetos mejora la organización y legibilidad del código y proporciona una base sólida para el desarrollo de aplicaciones robustas y flexibles.

La capacidad de **encapsular** la lógica de una aplicación en objetos que modelan el mundo real facilita la reutilización y comprensión del código agilizando así el proceso de desarrollo. En definitiva, la programación orientada a objetos es una metodología de programación eficaz que **promueve la mantenibilidad y escalabilidad del código** a largo plazo.

¡Enhорabuena! Fastbook superado



Qualentum.com