Patrones de diseño en PHP

Los patrones de diseño, design patterns, proporcionan soluciones esquemáticas para problemas comunes generados durante el desarrollo

Los **patrones de diseño**, design patterns, son métodos reusables para solucionar problemas de diseño comunes dentro de un contexto. Es una descripción o plantilla de **cómo solucionar un problema** que puede usarse en múltiples situaciones y lenguajes. Los patrones están formalizados como "*mejores prácticas*" que el desarrollador puede usar para resolver problemas comunes cuando diseña una aplicación. Esto no significa que tengan que aplicarse siempre, depende del criterio del desarrollador cuándo y cómo aplicarlos en una situación y proyecto concretos.

Los patrones de diseño pueden **mejorar el proceso y rendimiento del desarrollo**. Un diseño de proyecto efectivo requiere considerar problemas que pueden no verse hasta después en la implementación. Reusar patrones de diseño ayuda a **prevenir problemas que pueden causar mayores problemas** y mejora la legibilidad del código para desarrolladores familiarizados con los patrones.

A menudo la gente sólo entiende cómo aplicar determinados patrones para problemas concretos, esto es debido a que estas técnicas son difíciles de aplicar para un rango amplio de problemas. Los patrones de diseño proveen soluciones generales, documentadas en un formato que no requiere reglas específicas para problemas específicos. Los patrones más comunes **van mejorando con el tiempo**, haciéndolos más robutos y efectivos.

Los **patrones de diseño** se dividen generalmente en:

Creational patterns

Tratan la **instancia de las clases**. Estos patrones pueden dividirse después en **class-creation patterns** y **object-creational patterns**. Mientras que los class-creation patterns utilizan la **herencia** en el proceso de instanciación, object-creation patterns utilizan la **delegación** para terminar la tarea.

- **Abstract factory**. Crea una instancia de diferentes familias de clases.
- **Builder**. Separa la construcción de objetos de su representación.
- **Factory method**. Crea una instancia de varias clases derivadas.
- **Object pool**. Evita adquisiciones costosas y libera recursos reciclando objetos que no se usan.
- **Prototype**. Una instancia iniciada lista para ser copiada o clonada.
- **Singleton**. Una clase de la que sólo puede existir una instancia.

Structural patterns

Tratan la **composición de clases y objetos**. Los patrones **structural class-creation** utilizan la herencia para componer interfaces. Los patrones **structural object-patterns** definen formas de componer objetos para obtener nueva funcionalidad.

- **Adapter**. Ajusta las interfaces de distintas clases para que coincidan.
- **Bridge**. Separa la interface de un objeto de su implementación.
- **Composite**. Una estructura en árbol de objetos simples y compuestos.
- **Decorator**. Añade responsabilidades a objetos de forma dinámica.
- **Facade**. Una clase simple que representa un subsistema entero.
- **Flyweight**. Ligera instancia usada para que sea eficiente de compartir.
- **Proxy**. Un objeto que representa a otro objeto.

Behavioral patterns

Tratan la **comunicación de las clases de los objetos**, y por tanto tratan la comunicación entre objetos.

- **Chain of responsibility**. Forma de pasar un request entre una cadena de objetos.
- **Command**. Encapsula un command request como un objeto.
- **Interpreter**. Forma de incluir elementos del lenguaje en un programa.
- **Iterator**. Acceder a elementos de una colección de forma secuencial.
- **Mediator**. Define una comunicación simplificada entre clases.
- **Memento**. Captura y restaura el estado interno de un objeto.
- **Null Object**. Diseñado para actuar como valor por defecto de un objeto.
- **State**. Altera el comportamiento de un objeto.
- **Strategy**. Encapsula un algoritmo dentro de una clase.
- **Template method**. Aplaza los pasos exactos de un algoritmo a una subclase.
- **Visitor**. Define una nueva operación a una clase sin cambios.

Architectural patterns

Tienen una aplicación más amplia que las anteriores pero tienen un rango más amplio de posibilidades. Son muy utilizados en los **frameworks** actuales de **PHP**.

- **Front controller**. Proporciona una forma centralizada de manejar los requests.
- **MVC**. Model-View-Controller. Divide una aplicación en tres partes interconectadas, separando las partes internas de la representación.
- **ADR**. Action-Domain-Responder. Se plantea como un patrón MVC más refinado y orientado al desarrollo web.
- **Service locator**. Emplea un registro central que devuelve información necesaria para tareas cuando se hacen peticiones.
- **Active record**. Acceso de datos de una base de datos mediante un objeto.
- **Publish-subscribe**. Forma de notificar los cambios en un número de clases.

- **Inversion of control**. Diseño que invierte el funcionamiento tradicional en el que el código customizado llama a librerías reusables.

Otros patterns

Estos patterns no están dentro de ninguna de las categorías anteriores, pero también son muy utilizados en los **patrones de diseño** y en los **frameworks**:

- **Dependency Injection**. Implementa Inversion of control para resolver dependencias, que son objetos que pueden usarse (*services*).
- **Lazy loading**. Difiere el inicio de la iniciación de un objeto hasta el punto en el que se necesite, lo que contribuye al rendimiento.
- **Mock object**. Objeto simulado que imita el comportamiento de objetos de forma controlada (creados especialmente para hacer *testing*).
- **Table Data Gateway**. Un objeto actúa como gateway de una base de datos.

A continuación se explican más concretamente algunos de ellos, con ejemplos de aplicación:

1. Factory pattern
2. Singleton pattern
3. Adapter pattern
4. Decorator pattern
5. Strategy pattern
6. MVC pattern
7. ADR pattern
8. Active record pattern
9. Publish-subscribe pattern
10. Mock object pattern

1. Factory pattern

El **Factory pattern** define una interfaz para crear un objeto, pero deja a las subclases decidir que clase instanciar. Este patrón permite a una clase ceder la instanciación a las subclases.

Ejemplo. Tenemos una clase **Vehicle** con las subclases **Car** y **Motorcycle**. En función de los argumentos que le pasemos a la variable estática *Vehicle::create*, se creará una instancia de Car o de Motorcyle.

```
<?php
// Vehicle.php
class Vehicle
{
    public $wheels;

    public static function create($type, $wheels)
    {
        switch($type) {
            case 'car':
                return new Car($wheels);
```

```
            case 'motorcycle':
                return new Motorcycle($wheels);
            default:
                return new Exception("No vehicle found");
        }
    }

    public function getType()
    {
        return get_class($this);
    }
}

class Car extends Vehicle
{
    public function __construct($wheels){
        $this->wheels = $wheels;
    }
}

class Motorcycle extends Vehicle
{
    public function __construct($wheels){
        $this->wheels = $wheels;
    }
}
```

Ahora podemos crear subclases diferentes:

```
// index.php

include_once 'vehicle.php';

$car = Vehicle::create('car', 4);
echo $car->getType() . ": tiene " . $car->wheels . " ruedas" . "<br>";

$motorcycle = Vehicle::create('motorcycle', 2);
echo $motorcycle->getType() . ": tiene " . $motorcycle->wheels . " ruedas" .
"<br>";
```

2. Singleton pattern

Este patrón simplemente restringe la instanciación de una clase a un objeto. Es útil cuando se necesita exactamente un objeto para coordinar las acciones en la aplicación. A menudo es utilizado en situaciones en las que no es beneficioso, ya que añade **restricciones innecesarias**. Muchos recomiendan no utilizar nunca **singletons** (también es engorroso a la hora de hacer testing) y tratar de emplear **dependency injection** en su lugar.

Ejemplo. Ejemplo sencillo de singleton con un constructor private.

```
<?php
// index.php
include 'Connection.php';

$object = Connection::getInstance();
$object2 = Connection::getInstance();
$object3 = Connection::getInstance();
```

El index crea 3 objetos de la misma clase con su método **getInstance**, y el constructor privado se encarga de ejecutar la acción:

```
<?php
// Connection.php
class Connection
{
    private function __construct()
    {
        echo 'Hemos creado un nuevo objeto <br />';
    }

    public static function getInstance()
    {
        static $instance = null;
        if (null === $instance) {
            $instance = new static();
        } else {
            echo 'El objeto ya existe, no puedes volver a crearlo <br />';
        }

        return $instance;
    }
}
```

3. Adapter pattern

El **Adapter pattern** sirve para hacer que las clases existentes funcionen con otras sin modificar sus contenidos. Son útiles cuando quieres utilizar una clase que no tiene los métodos exactos que necesitas y no puedes cambiar la clase original o quieres adaptar varias clases para una funcionalidad común.

Ejemplo. Suponemos que tenemos una clase Support donde los usuarios pueden enviar sus problemas. Pueden hacerlo en el tablón público o enviarlo de forma privada al staff de soporte. Para

publicar en el tablón se emplearía una clase **PublishBoard**, y para enviar el mensaje al staff **PublishBackend**.

```
<?php
// Support.php
class Support
{
    private $username;

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

    public function sendMessage($title, $message, $phoneNumber = '')
    {
        if('' == $phoneNumber) {
            return $this->sendPublic($title, $message);
        } else {
            return $this->sendPrivate($title, $message, $phoneNumber);
        }
    }

    protected function sendPublic($title, $message)
    {
        echo "El mensaje se publicaría en el tablón de soporte (público)<br/>";
        echo "Emplearía una clase especial PublishBoard";
    }

    protected function sendPrivate($title, $message, $phoneNumber)
    {
        echo "El mensaje se enviaría al backend del soporte (privado)</br>";
        echo "Emplearía una clase especial PublishBackend";
    }
}
```

Se podrían utilizar las diferentes clases que publican contenidos por separado, pero en este caso hemos adaptado **PublishBoard** y **PublishBackend** en una clase **Support** que se encargará de enviar los mensajes. Si se recibe el número de teléfono, el método *sendMessage()* enviará el mensaje en privado mediante PublishBackend.

```
// index.php
include_once('support.php');

$support = new Support('username');
$support->sendMessage('Problema', 'No funciona algo', '6755642231');
```

4. Decorator pattern

Este patrón añade comportamientos específicos en la instancia de una clase en lugar de adjuntarlos al objeto. Proporciona una alternativa flexible a la herencia para extender funcionalidades.

Ejemplo. Tenemos un objeto **stdClass** que guarda un libro con las propiedades *title*, _author_first*name* y _author_last*name*. Queremos mostrar el nombre del autor de dos formas distintas, por lo que empleamos una clase que recibe la clase stdClass y la modifica.

```
<?php
// PrettyPrint.php
class PrettyPrint
{
    protected $book = null;

    public function __construct(stdClass $book_object)
    {
        $this->book = $book_object;
    }

    public function getAuthor()
    {
        return $this->book->author_first_name . " " . $this->book->author_last_name;
    }

    public function getAuthorSortable()
    {
        return $this->book->author_last_name . ", " . $this->book->author_first_name;
    }
}
```

Ahora creamos un libro, lo pasamos a **PrettyPrint** y podremos mostrar el nombre del autor de las dos formas:

```
<?php
// index.php
include_once('PrettyPrint.php');

$book = new stdClass();

$book->title = "Modern PHP";
$book->author_first_name = "Josh";
$book->author_last_name = "Lockhart";

$bookFormatter = new PrettyPrint($book);

echo $book->title . " es un libro escrito por: " . $bookFormatter->getAuthor() . "<br>";
echo "Pero encontrarás el libro de esta forma: " . $bookFormatter->getAuthorSortable() . "<br>";
```

5. Strategy pattern

Strategy pattern permite que el comportamiento de un algoritmo se seleccione en tiempo de ejecución. El patrón define una **familia de algoritmos**, encapsula cada algoritmo y hace que los algoritmos sean intercambiables en esa familia. De esta forma el algoritmo varía independientemente de los clientes que lo usen.

Permite definir y elegir una implementación dinámicamente y variar esa implementación independientemente del objeto. Surge cuando se necesitan añadir diferentes comportamientos de forma dinámica entre diferentes tipos de objetos.

Este patrón potencia así el **principio de la programación orientada a objetos Open/Closed**, que dice que las clases y los métodos han de estar abiertos para su extensión y cerrados para su modificación. De esta forma si queremos añadir comportamientos lo que se ha de hacer es extender las clases, que se puede hacer con herencia, pero mediante el **patrón strategy** puede ser más efectivo.

Ejemplo. En este ejemplo tenemos una clase (**Output**) desde la cual podemos instanciar clases diferentes (**Square**, **Circle**, **Cube**) en función de la superficie o volumen que queramos calcular.

```
<?php
// index.php
include_once 'Output.php';
include_once 'Square.php';
include_once 'Circle.php';
include_once 'Cube.php';

$input = 10;

$output = new Output(new Square());
echo 'Square of 10: ' . $output->display($input) . '<br/>';

$output->setStrategy(new Circle());
echo 'Circle of 10: ' . $output->display($input) . '<br/>';

$output->setStrategy(new Cube());
echo 'Cube of 10: ' . $output→display($input);
```

Se crea una clase **Output** que recibe en su constructor un objeto **Square**. Después, si queremos emplear otro objeto en su lugar, llamamos al método *setStrategy* y podremos operar con **Circle** de la misma forma:

```
<?php
// Output.php
class Output
{
    protected $formatter = null;

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

    public function setStrategy($formatter)
    {
        $this->formatter = $formatter;
```

```
    }

    public function display($input)
    {
        return $this->formatter->output($input);
    }
}
```

A continuación se muestra cada una de las clases: **Square**, **Circle** y **Cube**, que simplemente tienen un método *output* para mostrar sus superficies/volúmenes:
Clase Square:

```
<?php
// Square.php
class Square
{
    public function output($input)
    {
        return $input * $input;
    }
}
```

Clase Circle:

```
<?php
// Circle.php
class Circle
{
    public function output($input)
    {
        return pi() * $input * $input;
    }
}
```

Clase Cube:

```
<?php
// Cube.php
class Cube
{
    public function output($input)
    {
        return $input * $input * $input;
    }
}
```

6. Model-View-Controller pattern
Este patrón divide la aplicación en tres partes interconectadas (model, view, controller), de forma que la representación de la información queda separada del funcionamiento interno de la aplicación,

lo que la hace mucho más segura. Es uno de los patrones en los que se fundamentan los **frameworks** actuales y la mayoría de **aplicaciones web**:

- Model. Recopila los datos.
- Controller. Maneja y distribuye los datos. Es el puente entre el Model y el View.
- View. Representa los datos.

Se suele procurar que los **controllers** sean reducidos, de forma que se cargue el mayor peso posible en los **models**.

Ejemplo. Este ejemplo es extenso y complejo, pero es lo más que se puede reducir este patrón. Primero representamos el esquema de archivos:

```
mvc/
├─ models/
    └─ Greeting.php
├─ controllers/
    └─ DefaultController.php
    └─ GreetingController.php
├─ views/
    └─ default.php
    └─ message.php
index.php
```

Comenzamos con el archivo *index.php*:

```
<?php
// index.php
include_once 'controllers/DefaultController.php';
include_once 'controllers/GreetingController.php';
include_once 'models/Greeting.php';

$action = isset($_GET['a']) ? $_GET['a'] : 'index';
$section = isset($_GET['s']) ? $_GET['s'] : '';

switch($section){
    case 'greeting':
        $controller = new GreetingController;
        break;
    default:
        $controller = new DefaultController;
}

$controller->run($action);
```

Si no hay ninguna *section* ejecutará el método *run* de **DefaultController**:

```
<?php
// controllers/DefaultController.php
class DefaultController
{
    public function run ($action = 'index', $id = 0)
```

```
    {
        if(!method_exists($this, $action)){
            $action = 'index';
        }

        return $this->$action($id);
    }

    public function index()
    {
        include 'views/default.php';
    }
}
```

Este empleará el método *index*, el cual incluye el template de la página principal:

```
<! views/default.php-->
<ul>
    <li>
        <a href="index.php?s=greeting&a=hello">Decir hola</a>
    </li>
    <li>
        <a href="index.php?s=greeting&a=goodbye">Decir adios</a>
    </li>
</ul>
```

Desde aquí podemos escoger Decir hola y Decir adios, que serían dos tipos de *action*, y el *index.php* nos llevará entonces al **GreetingController**:

```
<?php
// controllers/GreetingController.php
class GreetingController extends DefaultController
{
    protected $section = null;

    public function __construct()
    {
        $this->section = new Greeting;
    }

    public function hello()
    {
        $message = $this->section->hello();

        include_once 'views/message.php';
    }

    public function goodbye()
    {
        $message = $this->section->goodbye();
```

```
        include_once 'views/message.php';
    }

}
```

GreetingController instancia por defecto al model, que es la clase **Greeting**:

```
<?php
// models/Greeting.php
class Greeting
{
    public function hello()
    {
        return "Hola!, ¿que tal?";
    }
    public function goodbye()
    {
        return "Adiós! Hasta Pronto!";
    }
}
```

Finalmente, dependiendo de si hemos hecho click en Decir hola o Decir adios, empleará los métodos *hello()* o *goodbye()*, que incluyen un mensage específico que se mostrará en el view especificado:

```
<!--views/views/message.php-->
<h1><?php echo $message ?></h1>
<ul>
    <li>
        <a href="index.php?s=greeting&a=hello">Saludar</a>
    </li>
    <li>
        <a href="index.php?s=greeting&a=goodbye">Despedirse</a>
    </li>
    <li>
        <a href="index.php?s=greeting&a=unrecognized">Cualquier otra acción</a>
    </li>
</ul>
```