

# ¿Qué es un patrón de diseño?

Los **patrones de diseño** son soluciones habituales a problemas que ocurren con frecuencia en el diseño de software. Son como planos prefabricados que se pueden personalizar para resolver un problema de diseño recurrente en tu código.

No se puede elegir un patrón y copiarlo en el programa como si se tratara de funciones o bibliotecas ya preparadas. El patrón no es una porción específica de código, sino un concepto general para resolver un problema particular. Puedes seguir los detalles del patrón e implementar una solución que encaje con las realidades de tu propio programa.

A menudo los patrones se confunden con algoritmos porque ambos conceptos describen soluciones típicas a problemas conocidos. Mientras que un algoritmo siempre define un grupo claro de acciones para lograr un objetivo, un patrón es una descripción de más alto nivel de una solución. El código del mismo patrón aplicado a dos programas distintos puede ser diferente.

Una analogía de un algoritmo sería una receta de cocina: ambos cuentan con pasos claros para alcanzar una meta. Por su parte, un patrón es más similar a un plano, ya que puedes observar cómo son su resultado y sus funciones, pero el orden exacto de la implementación depende de ti.

## ¿En qué consiste el patrón?

La mayoría de los patrones se describe con mucha formalidad para que la gente pueda reproducirlos en muchos contextos. Aquí tienes las secciones que suelen estar presentes en la descripción de un patrón:

- El **propósito** del patrón explica brevemente el problema y la solución.
- La **motivación** explica en más detalle el problema y la solución que brinda el patrón.
- La **estructura** de las clases muestra cada una de las partes del patrón y el modo en que se relacionan.
- El **ejemplo de código** en uno de los lenguajes de programación populares facilita la asimilación de la idea que se esconde tras el patrón.

Algunos catálogos de patrones enumeran otros detalles útiles, como la aplicabilidad del patrón, los pasos de implementación y las relaciones con otros patrones.

## Clasificación de los patrones

Los patrones de diseño varían en su complejidad, nivel de detalle y escala de aplicabilidad al sistema completo que se diseña. Me gusta la analogía de la construcción de carreteras: puedes hacer más segura una intersección instalando semáforos o construyendo un intercambiador completo de varios niveles con pasajes subterráneos para peatones.

Los patrones más básicos y de más bajo nivel suelen llamarse *idioms*. Normalmente se aplican a un único lenguaje de programación.

Los patrones más universales y de más alto nivel son los *patrones de arquitectura*. Los desarrolladores pueden implementar estos patrones prácticamente en cualquier lenguaje. Al contrario que otros patrones, pueden utilizarse para diseñar la arquitectura de una aplicación completa.

Además, todos los patrones pueden clasificarse por su *propósito*. Este libro cubre tres grupos generales de patrones:

- Los **patrones creacionales** proporcionan mecanismos de creación de objetos que incrementan la flexibilidad y la reutilización de código existente.
- Los **patrones estructurales** explican cómo ensamblar objetos y clases en estructuras más grandes a la vez que se mantiene la flexibilidad y eficiencia de la estructura.
- Los **patrones de comportamiento** se encargan de una comunicación efectiva y la asignación de responsabilidades entre objetos.

## Práctico 10 - Patrones de diseño (CHAT GPT)

1) Patrón singleton. La conexión a una base de datos requiere tres parámetros: Nombre de la BBDD, usuario, y password. Su utilización ocurre en diversas partes del sistema. Para crear una conexión se utiliza la clase PDO de la siguiente forma:  
\$conexion = new PDO(\$base, \$usuario, \$clave): El problema es que cada nueva conexión consume recursos que se van agotando hasta bloquear el sistema. Como haría un constructor único que siempre devuelva la misma instancia de conexión ?

Para implementar el patrón Singleton en PHP para gestionar la conexión a la base de datos de manera que siempre obtengas la misma instancia de conexión, puedes seguir este enfoque:

```
class DatabaseConnection {
    private static $instance = null;
    private $pdo;

    // Datos de la base de datos
    private $host = 'localhost';
    private $db = 'nombre_db';
    private $user = 'usuario';
    private $pass = 'contraseña';

    private function __construct() {
        // Evitar que se pueda crear una instancia directamente
        try {
            $this->pdo = new PDO("mysql:host={$this->host};dbname={$this->db}", $this->user, $this->pass);
        } catch (PDOException $e) {
            die("Error de conexión: " . $e->getMessage());
        }
    }

    public static function getInstance() {
        if (self::$instance === null) {
            self::$instance = new self();
        }
        return self::$instance;
    }

    public function getConnection() {
        return $this->pdo;
    }
}

// Uso del Singleton para obtener la conexión a la base de datos
$conexion = DatabaseConnection::getInstance()->getConnection();
```

En este ejemplo:

1. La clase **DatabaseConnection** tiene un constructor privado para evitar que se puedan crear instancias directamente desde fuera de la clase.
2. El método **getInstance()** se encarga de crear una única instancia de la clase **DatabaseConnection** la primera vez que se llama y luego devuelve la misma instancia en sucesivas llamadas.
3. La conexión PDO a la base de datos se establece en el constructor de la clase **DatabaseConnection**.
4. Puedes obtener la conexión PDO utilizando el método **getConnection()**.

De esta manera, cada vez que necesites una conexión a la base de datos, puedes usar **DatabaseConnection::getInstance()->getConnection()**, y siempre obtendrás la misma instancia de conexión, evitando así consumir recursos innecesarios al crear múltiples conexiones.

- 2) Patron Builder. Una clase Auto, se compone de una clase Motor, de varias Instancias de Ruedas y una clase Carroceria. Realice un constructor general de vehículos.

El patrón Builder es útil cuando necesitas construir objetos complejos con múltiples partes de manera flexible. En tu caso, estás construyendo un objeto **Auto** que se compone de un **Motor**, varias instancias de **Ruedas** y una **Carrocería**. Aquí tienes un ejemplo de cómo implementar un constructor general de vehículos utilizando el patrón Builder en PHP:

```
1  // Clase Motor
2  class Motor {
3      private $tipo;
4
5      public function __construct($tipo) {
6          $this->tipo = $tipo;
7      }
8
9      public function getTipo() {
10         return $this->tipo;
11     }
12 }
13
14 // Clase Ruedas
15 class Ruedas {
16     private $cantidad;
17
18     public function __construct($cantidad) {
19         $this->cantidad = $cantidad;
20     }
21
22     public function getCantidad() {
23         return $this->cantidad;
24     }
25 }
26
27 // Clase Carroceria
28 class Carroceria {
29     private $tipo;
30 }
```

```

31     public function __construct($tipo) {
32         $this->tipo = $tipo;
33     }
34
35     public function getTipo() {
36         return $this->tipo;
37     }
38 }
39
40 // Clase AutoBuilder para construir un Auto
41 class AutoBuilder {
42     private $motor;
43     private $ruedas;
44     private $carroceria;
45
46     public function setMotor($tipo) {
47         $this->motor = new Motor($tipo);
48     }
49
50     public function setRuedas($cantidad) {
51         $this->ruedas = new Ruedas($cantidad);
52     }
53
54     public function setCarroceria($tipo) {
55         $this->carroceria = new Carroceria($tipo);
56     }
57
58     public function build() {
59         return new Auto($this->motor, $this->ruedas, $this->carroceria);
60     }
61 }

```

```

62
63 // Clase Auto que representa un vehículo completo
64 class Auto {
65     private $motor;
66     private $ruedas;
67     private $carroceria;
68
69     public function __construct(Motor $motor, Ruedas $ruedas, Carroceria $carroceria) {
70         $this->motor = $motor;
71         $this->ruedas = $ruedas;
72         $this->carroceria = $carroceria;
73     }
74
75     public function getDescription() {
76         return "Auto con motor {$this->motor->getTipo()}, {$this->ruedas->getCantidad()}
77         ruedas, y carrocería {$this->carroceria->getTipo()}";
78     }
79 }
80
81 // Uso del Builder para crear un Auto
82 $autoBuilder = new AutoBuilder();
83 $autoBuilder->setMotor('Gasolina');
84 $autoBuilder->setRuedas(4);
85 $autoBuilder->setCarroceria('Sedán');
86
87 $miAuto = $autoBuilder->build();
88
89 echo $miAuto->getDescription(); // Imprime: "Auto con motor Gasolina, 4 ruedas, y carrocería Sedán"

```

En este ejemplo:

1. Las clases **Motor**, **Ruedas** y **Carroceria** representan las partes individuales de un vehículo, cada una con sus propios atributos y métodos para obtener información sobre ellas.
2. La clase **AutoBuilder** se encarga de construir un objeto **Auto** paso a paso. Puedes configurar el tipo de motor, la cantidad de ruedas y el tipo de carrocería antes de llamar al método **build()** para obtener el objeto **Auto** completo.
3. La clase **Auto** representa un vehículo completo y tiene un método **getDescription()** para obtener información sobre el auto construido.
4. En el ejemplo final, se crea un objeto **Auto** utilizando el **AutoBuilder** y se muestra su descripción.

- 3) Realice una clase **Lista**, que permita agregar elementos, y mostrarlos en distintos ordenes (ascendente, descendente, aleatorio, por orden de llegada) según indique el usuario.

Para crear una clase **Lista** que permita agregar elementos y mostrarlos en diferentes órdenes según lo indique el usuario, puedes seguir este ejemplo en PHP:

```
1  class Lista {
2      private $elementos = [];
3
4      public function agregarElemento($elemento) {
5          $this->elementos[] = $elemento;
6      }
7
8      public function mostrar($orden = 'ascendente') {
9          switch ($orden) {
10             case 'ascendente':
11                 sort($this->elementos);
12                 break;
13             case 'descendente':
14                 rsort($this->elementos);
15                 break;
16             case 'aleatorio':
17                 shuffle($this->elementos);
18                 break;
19             case 'ordenLlegada':
20                 default:
21                     // No hacemos nada, mantenemos el orden de llegada
22                     break;
23             }
24
25             echo "Elementos en orden $orden: ";
26             echo implode(', ', $this->elementos) . "\n";
27         }
28     }
29 }
```

```

30 // Ejemplo de uso
31 $lista = new Lista();
32
33 $lista->agregarElemento(5);
34 $lista->agregarElemento(2);
35 $lista->agregarElemento(8);
36 $lista->agregarElemento(1);
37 $lista->agregarElemento(3);
38
39 echo "Orden de llegada: ";
40 $lista->mostrar('ordenLlegada');
41
42 echo "Ascendente: ";
43 $lista->mostrar('ascendente');
44
45 echo "Descendente: ";
46 $lista->mostrar('descendente');
47
48 echo "Aleatorio: ";
49 $lista->mostrar('aleatorio');

```

En este ejemplo:

1. La clase **Lista** tiene un arreglo privado llamado **\$elementos** donde se almacenan los elementos agregados.
2. El método **agregarElemento(\$elemento)** permite agregar elementos a la lista.
3. El método **mostrar(\$orden)** muestra los elementos en el orden especificado por el usuario. Utiliza un **switch** para determinar el orden y luego aplica las funciones de ordenamiento o aleatorización apropiadas. Finalmente, muestra los elementos con **implode()**.
4. En el ejemplo de uso, se crea una instancia de **Lista**, se agregan elementos y se muestra la lista en diferentes órdenes: orden de llegada, ascendente, descendente y aleatorio.

Puedes personalizar esta clase según tus necesidades y agregar más métodos si es necesario para manipular la lista de elementos de otras formas.

- 4) Una Lista puede componerse de elementos simples, como números o textos, pero también puede componerse de otras listas. Realice un programa que permita este tipo de estructuras, y que pueda listar el contenido, independiente de si este es simple o compuesto.

Para permitir que una lista pueda componerse de elementos simples como números o textos, así como de otras listas (estructura anidada), y listar su contenido independientemente de su complejidad, puedes utilizar la recursión para recorrer la estructura anidada. Aquí te muestro un ejemplo en PHP:

```

1  class Lista {
2      private $elementos = [];
3
4      public function agregarElemento($elemento) {
5          $this->elementos[] = $elemento;
6      }
7
8      public function listar() {
9          $this->listarRecurso($this->elementos);
10     }
11
12     private function listarRecurso($elementos, $nivel = 0) {
13         foreach ($elementos as $elemento) {
14             if (is_array($elemento)) {
15                 // Si el elemento es un arreglo (lista anidada), llamamos recursivamente
16                 echo str_repeat(' ', $nivel) . "Lista anidada:\n";
17                 $this->listarRecurso($elemento, $nivel + 1);
18             } else {
19                 // Si es un elemento simple, lo mostramos
20                 echo str_repeat(' ', $nivel) . "$elemento\n";
21             }
22         }
23     }
24 }
25
26 // Ejemplo de uso
27 $lista = new Lista();
28
29 $lista->agregarElemento(1);
30 $lista->agregarElemento(2);
31 $lista->agregarElemento([3, 4, 5]);
32 $lista->agregarElemento("Texto");
33 $lista->agregarElemento([6, [7, 8], 9]);
34
35 echo "Contenido de la lista:\n";
36 $lista->listar();

```

En este ejemplo:

1. La clase **Lista** puede almacenar elementos simples o listas anidadas. El método **agregarElemento(\$elemento)** agrega elementos a la lista.
2. El método **listar()** inicia el proceso de listado y llama al método privado **listarRecurso(\$elementos, \$nivel)** para recorrer los elementos, donde **\$nivel** se utiliza para indentar el contenido anidado.
3. **listarRecurso()** es un método privado que se encarga de listar los elementos de manera recursiva. Si encuentra un arreglo (lista anidada), llama a sí mismo recursivamente para procesar ese subconjunto.
4. En el ejemplo de uso, se crean elementos simples como números y texto, así como listas anidadas. Luego, se llama al método **listar()** para mostrar el contenido de la lista, independientemente de su complejidad.

Este enfoque te permite manejar listas anidadas de manera recursiva y listar su contenido de forma estructurada.

5) En una empresa, un departamento tiene un jefe y empleados directos. Además un departamento puede componerse de otros departamentos. En términos concretos, la organización puede definir otros nombres, como áreas o jefaturas, pero es lo mismo. Realice un programa que permita definir una jerarquía de organización, asignando y manipulando los distintos departamentos por un nombre único, dentro del departamento que componen. Debe poder crearse nuevos departamentos y agregarlos dentro de otro departamento. El único departamento que no se encuentra dentro de otro, es la Gerencia General.

a. Que patrones o patron aplicaría ?

b. Que controles especiales debe realizar para evitar la duplicidad de departamentos dentro de uno preexistente?

Para modelar la jerarquía de organización de la empresa donde un departamento puede tener un jefe, empleados directos y puede componerse de otros departamentos, puedes utilizar varios patrones de diseño y controles especiales para evitar la duplicidad de departamentos dentro de uno preexistente.

**a. Patrones a aplicar:**

1. **Composite Pattern (Patrón Composite):** Este patrón se utiliza para tratar objetos individuales y composiciones de objetos de manera uniforme. En este caso, puedes considerar cada departamento como un componente, ya sea un departamento simple o un departamento compuesto (que contiene otros departamentos). Esto permite crear una jerarquía flexible de departamentos.
2. **Singleton Pattern (Patrón Singleton):** Puedes utilizar el patrón Singleton para asegurarte de que solo existe una instancia de la organización en toda la aplicación, de modo que todos los componentes (departamentos) compartan la misma instancia de organización.

**b. Controles para evitar la duplicidad de departamentos:**

Para evitar la duplicidad de departamentos dentro de uno preexistente, puedes implementar controles adicionales en tu programa. Aquí hay algunas sugerencias:

1. **Utilizar un diccionario o mapa:** Mantén un diccionario o mapa donde la clave sea el nombre único de cada departamento y el valor sea el objeto de departamento correspondiente. Antes de agregar un nuevo departamento, verifica si ya existe en el diccionario. Si existe, muestra un mensaje de error o lanza una excepción.
2. **Validación al agregar departamentos:** Cuando se agrega un departamento a otro departamento, verifica si ya existe en la lista de subdepartamentos del departamento padre. Si ya existe, evita agregarlo nuevamente y muestra un mensaje de error.
3. **Nombres únicos:** Asegúrate de que cada departamento tenga un nombre único dentro de su nivel de jerarquía. Esto significa que los nombres de los departamentos pueden ser iguales en niveles diferentes de la jerarquía, pero no pueden repetirse dentro del mismo nivel.
4. **Validación recursiva:** Si estás agregando un departamento compuesto que contiene otros departamentos, asegúrate de que los departamentos en niveles más bajos de la jerarquía también cumplan con la regla de nombres únicos.

Aquí tienes un ejemplo básico de cómo podrías implementar esto en PHP:



```

1  class Organization {
2      private $departments = [];
3
4      public function addDepartment($name, $department) {
5          if (!isset($this->departments[$name])) {
6              $this->departments[$name] = $department;
7          } else {
8              // Mostrar un mensaje de error o lanzar una excepción
9              echo "El departamento '$name' ya existe en la organización.\n";
10         }
11     }
12 }
13
14 class Department {
15     private $name;
16     private $subDepartments = [];
17
18     public function __construct($name) {
19         $this->name = $name;
20     }
21
22     public function addSubDepartment($name, $subDepartment) {
23         if (!isset($this->subDepartments[$name])) {
24             $this->subDepartments[$name] = $subDepartment;
25         } else {
26             // Mostrar un mensaje de error o lanzar una excepción
27             echo "El subdepartamento '$name' ya existe en el departamento '{$this->name}'.\n";
28         }
29     }
30 }

```

```

// Uso
$org = new Organization();

$gerenciaGeneral = new Department("Gerencia General");
$departamentoVentas = new Department("Ventas");

$org->addDepartment("Gerencia General", $gerenciaGeneral);
$gerenciaGeneral->addSubDepartment("Ventas", $departamentoVentas);

```

Este es un ejemplo simplificado, pero puedes adaptarlo a tus necesidades específicas y ampliarlo para incluir más funcionalidad según la estructura de tu organización.