# TEMA 5

## Contenido

# Programación Orientada a Objetos en PHP.

**Caso práctico**
***Carlos*** *empieza a darse cuenta de que, con lo que lleva aprendido de PHP, ya es capaz de hacer bastantes cosas. Ha acabado de programar la aplicación web para gestionar su colección de comics, y está satisfecho con el resultado obtenido. De vez en cuando ha tenido que echar mano de la documentación del lenguaje, para buscar información sobre cómo hacer algo, o los parámetros que requiere cierta función, pero siempre ha podido solucionarlo por sí mismo.*
*Sin embargo, cuando revisa el código de su aplicación, se da cuenta de que está muy desorganizado. Cada vez que necesita hacer algún cambio, o introducir un añadido, tiene que rebuscar entre las páginas para encontrar el código a reprogramar.*
*Y si eso le pasa con su pequeña aplicación, no se imagina lo que sucederá cuando tenga que programar una aplicación más compleja. —¡Tiene que haber algún modo de obtener un código más limpio y estructurado!*

## 1.- Características de orientación a objetos en PHP

**Caso práctico**
***Carlos*** *comenta a* ***Juan*** *su problema, y éste le dice que seguramente gran parte del problema se arregle si utiliza programación orientada a objetos en sus aplicaciones. Le explica por encima de qué se trata y cuáles son sus ventajas, pero no puede ayudarlo mucho más. La última vez que programó en PHP, intentó utilizar objetos en su aplicación, pero las características de orientación a objetos del lenguaje dejaban mucho que desear, por lo que acabó haciéndola como siempre.*
*Sin embargo, por lo que ha oído, parece ser que en las últimas versiones de PHP eso ha cambiado considerablemente. El lenguaje evoluciona, y él lleva bastante tiempo sin utilizarlo, así que tendrá que actualizarse.*
*Mientras tanto,* ***Carlos*** *ya se ha puesto un nuevo reto: debe aprender a utilizar objetos en PHP. Y cuando tenga cierta soltura, lo primero que hará es modificar el código de su aplicación de comics.*

La programación orientada a objetos (POO, o OOP en lenguaje inglés), es una metodología de programación basada en objetos. Un objeto es una estructura que contiene datos y el código que los maneja.

La estructura de los objetos se define en las clases. En ellas se escribe el código que define el comportamiento de los objetos y se indican los miembros que formarán parte de los objetos de dicha clase. Entre los miembros de una clase puede haber:

- ✓ **Métodos**. Son los miembros de la clase que contienen el código de la misma. Un método es como una función. Puede recibir parámetros y devolver valores.
- ✓ **Atributos o propiedades**. Almacenan información acerca del estado del objeto al que pertenecen (y por tanto, su valor puede ser distinto para cada uno de los objetos de la misma clase).

A la creación de un objeto basado en una clase se le llama **instanciar una clase** y al objeto obtenido también se le conoce como **instancia de esa clase**.

Los pilares fundamentales de la POO son:

- ✓ **Herencia**. Es el proceso de crear una clase a partir de otra, heredando su comportamiento y características y pudiendo redefinirlos.
- ✓ **Abstracción**. Hace referencia a que cada clase oculta en su interior las peculiaridades de su implementación, y presenta al exterior una serie de métodos (interface) cuyo comportamiento está bien definido. Visto desde el exterior, cada objeto es un ente abstracto que realiza un trabajo.

- ✓ **Polimorfismo**. Un mismo método puede tener comportamientos distintos en función del objeto con que se utilice.
- ✓ **Encapsulación**. En la POO se juntan en un mismo lugar los datos y el código que los manipula.

Las ventajas más importantes que aporta la programación orientada a objetos son:

- ✓ **Modularidad**. La POO permite dividir los programas en partes o módulos más pequeños, que son independientes unos de otros pero pueden comunicarse entre ellos.
- ✓ **Extensibilidad**. Si se desean añadir nuevas características a una aplicación, la POO facilita esta tarea de dos formas: añadiendo nuevos métodos al código, o creando nuevos objetos que extiendan el comportamiento de los ya existentes.
- ✓ **Mantenimiento**. Los programas desarrollados utilizando POO son más sencillos de mantener, debido a la modularidad antes comentada. También ayuda seguir ciertas convenciones al escribirlos, por ejemplo, escribir cada clase en un fichero propio. No debe haber dos clases en un mismo fichero, ni otro código aparte del propio de la clase.

En este módulo no se pretende realizar un estudio profundo de las ventajas y características de la POO, sino simplemente recordar conceptos que ya deberías haber asimilado con anterioridad. Si tienes dudas sobre algo de lo que acabamos de repasar, puedes consultar este tutorial de la web desarrolloweb.com.

**http://www.desarrolloweb.com/articulos/499.php**

## 1.1.- Características de orientación a objetos en PHP.

Seguramente todo, o la mayoría de lo que acabas de ver, ya lo conocías, y es incluso probable que sepas utilizar algún lenguaje de programación orientado a objetos, así que vamos a ver directamente las peculiaridades propias de PHP en lo que hace referencia a la POO.

Como ya has visto en las unidades anteriores, especialmente con las extensiones para utilizar bases de datos, con PHP puedes utilizar dos estilos de programación: estructurada y orientada a objetos.

```
// utilizando programación estructurada
$dwes = mysqli_connect('localhost', 'dwes', 'abc123.', 'dwes');
// utilizando POO
$dwes = new mysqli();
$dwes->connect('localhost', 'dwes', 'abc123.', 'dwes');
```

Sin embargo, el lenguaje PHP original no se diseñó con características de orientación a objetos. Sólo a partir de la versión 3, se empezaron a introducir algunos rasgos de POO en el lenguaje. Esto se potenció en la versión 4, aunque todavía de forma muy rudimentaria. Por ejemplo, en PHP4:

- ✓ Los objetos se pasan siempre por valor, no por referencia.
- ✓ No se puede definir el nivel de acceso para los miembros de la clase. Todos son públicos.
- ✓ No existen los interfaces.
- ✓ No existen métodos destructores.

En la versión actual, PHP5, se ha reescrito y potenciado el soporte de orientación a objetos del lenguaje, ampliando sus características y mejorando su rendimiento y su funcionamiento general. Aunque iremos detallando y explicando cada una posteriormente con detenimiento, las características de POO que soporta PHP5 incluyen:

- ✓ Métodos estáticos.
- ✓ Métodos constructores y destructores.
- ✓ Herencia.
- ✓ Interfaces.
- ✓ Clases abstractas.

Entre las características que no incluye PHP5, y que puedes conocer de otros lenguajes de programación, están:

- ✓ Herencia múltiple.
- ✓ Sobrecarga de métodos (*tener varios métodos con el mismo nombre, pero con funcionalidades distintas. Los métodos sobrecargados deben tener distintos parámetros. Cuando se llama al método, se utiliza una u otra versión en función de los parámetros con que se realice la llamada (pudiendo distinguir por su número y por su tipo, según el lenguaje de programación utilizado)*) (incluidos los métodos constructores).
- ✓ Sobrecarga de operadores (*similar a la sobrecarga de métodos. Se pueden definir varias funcionalidades a un mismo operador, y se utilizará una u otra en función del tipo de los operandos que se usen en cada instante*).

**Antes de PHP5, el comportamiento cuando se pasaba una variable a una función era siempre el mismo, independientemente de si la variable fuera un objeto o de cualquier otro tipo: siempre se creaba una nueva variable copiando los valores de la original.**

(•) **Verdadero.** ( ) Falso.

*Solo a partir de PHP5, cuando se pasa un objeto como parámetro a una función, se hace por referencia y no por valor.*

## 1.2.- Creación de clases.

La declaración de una clase en PHP se hace utilizando la palabra `class`. A continuación y entre llaves, deben figurar los miembros de la clase. Conviene hacerlo de forma ordenada, primero las propiedades o atributos, y después los métodos, cada uno con su código respectivo.

```
class Producto {
    private $codigo;
    public $nombre;
    public $PVP;

    public function muestra() {
        print "<p>" . $this->codigo . "</p>";
    }
}
```

Como comentábamos antes, es preferible que cada clase figure en su propio fichero (`producto.php`). Además, muchos programadores prefieren utilizar para las clases nombres que comiencen por letra mayúscula, para, de esta forma, distinguirlos de los objetos y otras variables.

Una vez definida la clase, podemos usar la palabra **new** para instanciar objetos de la siguiente forma:

```
$p = new Producto();
```

Para que la línea anterior se ejecute sin error, previamente debemos haber declarado la clase. Para ello, en ese mismo fichero tendrás que incluir la clase poniendo algo como:

```
require_once('producto.php');
```

Los atributos de una clase son similares a las variables de PHP. Es posible indicar un valor en la declaración de la clase. En este caso, todos los objetos que se instancien a partir de esa clase, partirán con ese valor por defecto en el atributo.

Para acceder desde un objeto a sus atributos o a los métodos de la clase, debes utilizar el **operador flecha** (fíjate que sólo se pone el símbolo `$` delante del nombre del objeto):

```
$p->nombre = 'Samsung Galaxy S';
$p->muestra();
```

Cuando se declara un atributo, se debe indicar su nivel de acceso. Los principales niveles son:

- ✓ `public`. Los atributos declarados como `public` pueden utilizarse directamente por los objetos de la clase. Es el caso del atributo `$nombre` anterior.
- ✓ `private`. Los atributos declarados como `private` sólo pueden ser accedidos y modificados por los métodos definidos en la clase, no directamente por los objetos de la misma. Es el caso del atributo `$codigo`.

En PHP4 no se podía definir nivel de acceso para los atributos de una clase, por lo que todos se precedían de la palabra `var`.

```
var $nombre;
```

Hoy en día, aunque aún es aceptado por PHP5, no se recomienda su uso. Si encuentras algún código que lo utilice, ten en cuenta que tiene el mismo efecto que `public`.

Uno de los motivos para crear atributos privados es que su valor forma parte de la información interna del objeto y no debe formar parte de su interface. Otro motivo es mantener cierto control sobre sus posibles valores.

Por ejemplo, no quieres que se pueda cambiar libremente el valor del código de un producto. O necesitas conocer cuál será el nuevo valor antes de asignarlo. En estos casos, se suelen definir esos atributos como privados y además se crean dentro de la clase métodos para permitirnos obtener y/o modificar los valores de esos atributos. Por ejemplo:

```
private $codigo;
public function setCodigo($nuevo_codigo) {
   if (noExisteCodigo($nuevo_codigo)) {
      $this->codigo = $nuevo_codigo;
      return true;
   }
   return false;
}
public function getCodigo() { return $this->codigo; }
```

Aunque no es obligatorio, el nombre del método que nos permite obtener el valor de un atributo suele empezar por `get`, y el que nos permite modificarlo por `set`.

En PHP5 se introdujeron los llamados métodos mágicos, entre ellos `__set` y `__get`. Si se declaran estos dos métodos en una clase, PHP los invoca automáticamente cuando desde un objeto se intenta usar un atributo no existente o no accesible. Por ejemplo, el código siguiente simula que la clase Producto tiene cualquier atributo que queramos usar.

```
class Producto {
    private $atributos = array();

    public function __get($atributo) {
        return $this->atributos[$atributo];
    }
    public function   set($atributo, $valor) {
        $this->atributos[$atributo] = $valor;
    }
}
```

En la documentación de PHP tienes más información sobre los métodos mágicos.
**http://es.php.net/manual/es/language.oop5.magic.php**

**En lugar de programar un método set para modificar el valor de los atributos privados en que sea necesario, puedo utilizar el método mágico __set.**

(•) **Verdadero.** ( ) Falso.

*Sí, pero tendrías que comprobar el nombre del atributo usado y asignar el valor al adecuado*

Cuando desde un objeto se invoca un método de la clase, a éste se le pasa siempre una referencia al objeto que hizo la llamada. Esta referencia se almacena en la variable `$this`. Se utiliza, por ejemplo, en el código anterior para tener acceso a los atributos privados del objeto (que sólo son accesibles desde los métodos de la clase).

```
print "<p>" . $this->codigo . "</p>";
```

Una referencia es una forma de utilizar distintos nombres de variables para acceder al mismo contenido. En los puntos siguientes aprenderás a crearlas y a utilizarlas.

**http://es.php.net/manual/es/language.references.php**

Además de métodos y propiedades, en una clase también se pueden definir **constantes**, utilizando la palabra `const`. Es importante que no confundas los atributos con las constantes. Son conceptos distintos: las constantes no pueden cambiar su valor (obviamente, de ahí su nombre), no usan el carácter `$` y, además, su valor va siempre entre comillas y está asociado a la clase, es decir, no existe una copia del mismo en cada objeto. Por tanto, para acceder a las constantes de una clase, se debe utilizar el nombre de la clase y el operador `::`, llamado **operador de resolución de ámbito** (que se utiliza para acceder a los elementos de una clase).

```
class DB {
    const USUARIO = 'dwes';
    …
}
echo DB::USUARIO;
```

Es importante resaltar que no es necesario que exista ningún objeto de una clase para poder acceder al valor de las constantes que defina. Además, sus nombres suelen escribirse en mayúsculas.

Tampoco se deben confundir las constantes con los miembros estáticos de una clase. En PHP5, una clase puede tener atributos o métodos estáticos, también llamados a veces atributos o métodos de clase. Se definen utilizando la palabra clave `static`.

```
class Producto {
    private static $num productos = 0;
    public static function nuevoProducto() {
        self::$num_productos++;
    }
    …
}
```

Los atributos y métodos estáticos no pueden ser llamados desde un objeto de la clase utilizando el operador `->`. Si el método o atributo es público, deberá accederse utilizando el nombre de la clase y el operador de resolución de ámbito.

```
Producto::nuevoProducto();
```

Si es privado, como el atributo `$num_productos` en el ejemplo anterior, sólo se podrá acceder a él desde los métodos de la propia clase, utilizando la palabra `self`. De la misma forma que `$this` hace referencia al objeto actual, `self` hace referencia a la clase actual.

```
self::$num_productos ++;
```

Los atributos estáticos de una clase se utilizan para guardar información general sobre la misma, como puede ser el número de objetos que se han instanciado. Sólo existe un valor del atributo, que se almacena a nivel de clase.

Los métodos estáticos suelen realizar alguna tarea específica o devolver un objeto concreto. Por ejemplo, las clases matemáticas suelen tener métodos estáticos para realizar logaritmos o raíces cuadradas. No tiene sentido crear un objeto si lo único que queremos es realizar una operación matemática.

Los métodos estáticos se llaman desde la clase. No es posible llamarlos desde un objeto y por tanto, no podemos usar `$this` dentro de un método estático.

Como ya viste, para instanciar objetos de una clase se utiliza `new`:

```
$p = new Producto();
```

En PHP5 puedes definir en las clases métodos constructores, que se ejecutan cuando se crea el objeto. El constructor de una clase debe llamarse `__construct`. Se pueden utilizar, por ejemplo, para asignar valores a atributos.

```
class Producto {
    private static $num productos = 0;
    private $codigo;

    public function __construct() {
        self::$num_productos++;
    }
    …
}
```

El constructor de una clase puede llamar a otros métodos o tener parámetros, en cuyo caso deberán pasarse cuando se crea el objeto. Sin embargo, sólo puede haber un método constructor en cada clase.

```
class Producto {
    private static $num productos = 0;
    private $codigo;

    public function __construct($codigo) {
        $this->$codigo = $codigo;
        self::$num_productos++;
    }
    …
}
$p = new Producto('GALAXYS');
```

Por ejemplo, si como en este ejemplo, definimos un constructor en el que haya que pasar el código, siempre que instancies un nuevo objeto de esa clase tendrás que indicar su código.

Una de las posibilidades de los métodos mágicos de PHP5 es utilizar `__call` para capturar llamadas a métodos que no estén implementados en la clase. Entonces, en función del nombre del método y del número de parámetros que se pasen, se podrían realizar unas acciones u otras.
**http://es.php.net/manual/es/language.oop5.overloading.php#language.oop5.overloading.methods**

También es posible definir un método destructor, que debe llamarse `__destruct` y permite definir acciones que se ejecutarán cuando se elimine el objeto.

```
class Producto {
    private static $num productos = 0;
    private $codigo;

    public function __construct($codigo) {
        $this->$codigo = $codigo;
        self::$num_productos++;
    }

    public function __destruct() {
        self::$num_productos--;
    }
    …
}
$p = new Producto('GALAXYS');
```

Los métodos constructores también existen en PHP4, pero en lugar de llamarse `construct`, se deben llamar del mismo modo que la clase. Los métodos destructores son nuevos en PHP5; no existían en versiones anteriores del lenguaje.

**¿Cuál es la utilidad del operador de resolución de ámbito ::?**

- ○ Nos permite hacer referencia a la clase del objeto actual.
- ◉ **Se utiliza para acceder a los elementos de una clase, como constantes y miembros estáticos.**

*Sí; y por tanto debe usarse precedido por el nombre de una clase, o por una referencia a una clase como self.*

## 1.3.- Utilización de objetos.

Ya sabes cómo instanciar un objeto utilizando `new`, y cómo acceder a sus métodos y atributos públicos con el operador flecha:

```
$p = new Producto();
$p->nombre = 'Samsung Galaxy S';
$p->muestra();
```

Una vez creado un objeto, puedes utilizar el operador `instanceof` para comprobar si es o no una instancia de una clase determinada.

```
if ($p instanceof Producto) {
   …
}
```

Además, en PHP5 se incluyen una serie de funciones útiles para el desarrollo de aplicaciones utilizando POO.

| Funciones de utilidad para objetos y clases en PHP5 | | |
|---|---|---|
| **Función** | **Ejemplo** | **Significado** |
| **get_class** | `echo "La clase es: " . get_class($p);` | Devuelve el nombre de la clase del objeto. |
| **class_exists** | `if (class exists('Producto') { $p = new Producto(); … }` | Devuelve true si la clase está definida o false en caso contrario. |
| **get_declared_classes** | `print_r(get_declared_classes());` | Devuelve un array con los nombres de las clases definidas. |
| **class_alias** | `class_alias('Producto', 'Articulo'); $p = new Articulo();` | Crea un alias para una clase. |
| **get_class_methods** | `print_r(get_class_methods('Producto'));` | Devuelve un array con los nombres de los métodos de una clase que son accesibles desde dónde se hace la llamada. |
| **method_exists** | `if (method exists('Producto', 'vende'){ … }` | Devuelve true si existe el método en el objeto o la clase que se indica, o false en caso contrario, independientemente de si es accesible o no. |
| **get_class_vars** | `print_r(get_class_vars('Producto'));` | Devuelve un array con los nombres de los atributos de una clase que son accesibles desde dónde se hace la llamada. |
| **get_object_vars** | `print_r(get_object_vars($p));` | Devuelve un array con los nombres de los métodos de un objeto que son accesibles desde dónde se hace la llamada. |
| **property_exists** | `if (property_exists('Producto', 'codigo') { … }` | Devuelve true si existe el atributo en el objeto o la clase que se indica, o false en caso contrario, independientemente de si es accesible o no. |

Desde PHP5, puedes indicar en las funciones y métodos de qué clase deben ser los objetos que se pasen como parámetros. Para ello, debes especificar el tipo antes del parámetro.

```
public function vendeProducto(Producto $p) {
     …
}
```

Si cuando se realiza la llamada, el parámetro no es del tipo adecuado, se produce un error que podrías capturar. Además, ten en cuenta que sólo funciona con objetos (y a partir de PHP5.1 también con arrays).

Una característica de la POO que debes tener muy en cuenta es qué sucede con los objetos cuando los pasas a una función, o simplemente cuando ejecutas un código como el siguiente:

```
$p = new Producto();
$p->nombre = 'Samsung Galaxy S';
$a = $p;
```

En PHP4, la última línea del código anterior crea un nuevo objeto con los mismos valores del original, de la misma forma que se copia cualquier otro tipo de variable. Si después de hacer la copia se modifica, por ejemplo, el atributo 'nombre' de uno de los objetos, el otro objeto no se vería modificado.

Sin embargo, en PHP5 este comportamiento varía. El código anterior simplemente crearía un **nuevo identificador del mismo objeto**. Esto es, en cuanto se utilice uno cualquiera de los identificadores para cambiar el valor de algún atributo, este cambio se vería también reflejado al acceder utilizando el otro identificador. Recuerda que, aunque haya dos o más identificadores del mismo objeto, en realidad todos se refieren a la única copia que se almacena del mismo.

Para crear nuevos identificadores en PHP5 a un objeto ya existente, se utiliza el operador `=`. Sin embargo, como ya sabes, este operador aplicado a variables de otros tipos, crea una copia de la misma. En PHP puedes crear referencias a variables (como números enteros o cadenas de texto), utilizando el operador `&`:

```
$a = 'Samsung Galaxy S';
$b = & $a;
```

En el ejemplo anterior, `$b` es una referencia a la variable `$a`. Cuando se cambia el valor de una de ellas, este cambio se refleja en la otra.
Las referencias se pueden utilizar para pasarlas como parámetros a las funciones. Si utilizamos el operador `&` junto al parámetro, en lugar de pasar una copia de la variable, se pasa una referencia a la misma.

```
function suma(&$v) {
$v ++;
}
$a = 3;
suma ($a);
echo $a;  // Muestra 4
```

De esta forma, dentro de la función se puede modificar el contenido de la variable que se pasa, no el de una copia.

**http://es.php.net/manual/es/language.references.php**

Por tanto, a partir de PHP5 no puedes copiar un objeto utilizando el operador `=`. Si necesitas copiar un objeto, debes utilizar `clone`. Al utilizar `clone` sobre un objeto existente, se crea una copia de todos los atributos del mismo en un nuevo objeto.

```
$p = new Producto();
$p->nombre = 'Samsung Galaxy S';
$a = clone($p);
```

Además, existe una forma sencilla de personalizar la copia para cada clase particular. Por ejemplo, puede suceder que quieras copiar todos los atributos menos alguno. En nuestro ejemplo, al menos el código de cada producto debe ser distinto y, por tanto, quizás no tenga sentido copiarlo al crear un

nuevo objeto. Si éste fuera el caso, puedes crear un método de nombre `__clone` en la clase. Este método se llamará automáticamente después de copiar todos los atributos en el nuevo objeto.

```
class Producto {
    …
    public function __clone($atributo) {
        $this->codigo = nuevo_codigo();
    }
    …
}
```

**¿Cuál es el nombre de la función que se utiliza para hacer una copia de un objeto?**

(•) **clone.** ( ) __clone.

*Además, cuando utilizas la función clone, si la clase tiene definido un método de nombre __clone, se llama automáticamente.*

A veces tienes dos objetos y quieres saber su relación exacta. Para eso, en PHP5 puedes utilizar los operadores `==` y `===`.

Si utilizas el operador de comparación `==`, comparas los valores de los atributos de los objetos. Por tanto dos objetos serán iguales si son instancias de la misma clase y, además, sus atributos tienen los mismos valores.

```
$p = new Producto();
$p->nombre = 'Samsung Galaxy S';
$a = clone($p);
// El resultado de comparar $a == $p da verdadero
//  pues $a y $p son dos copias idénticas
```

Sin embargo, si utilizas el operador `===`, el resultado de la comparación será `true` sólo cuando las dos variables sean referencias al mismo objeto.

```
$p = new Producto();
$p->nombre = 'Samsung Galaxy S';
$a = clone($p);
// El resultado de comparar $a === $p da falso
//  pues $a y $p no hacen referencia al mismo objeto
$a = & $p;
// Ahora el resultado de comparar $a === $p da verdadero
//  pues $a y $p son referencias al mismo objeto.
```

Los operadores de comparación `==` y `===` no funcionan de la misma manera que acabas de ver en PHP4. Si utilizas objetos en tus programas, es recomendable que te asegures de la versión del intérprete que los va a ejecutar, utilizando por ejemplo la función `phpversion`.

**http://es.php.net/manual/es/function.phpversion.php**

## 1.4.- Mecanismos de mantenimiento del estado.

En la unidad anterior aprendiste a usar la sesión del usuario para almacenar el estado de las variables, y poder recuperarlo cuando sea necesario. El proceso es muy sencillo; se utiliza el array superglobal `$_SESSION`, añadiendo nuevos elementos para ir guardando la información en la sesión.
El procedimiento para almacenar objetos es similar, pero hay una diferencia importante. Todas las variables almacenan su información en memoria de una forma u otra según su tipo. Los objetos, sin embargo, no tienen un único tipo. Cada objeto tendrá unos atributos u otros en función de su clase. Por tanto, para almacenar los objetos en la sesión del usuario, hace falta convertirlos a un formato estándar. Este proceso se llama serialización.

En PHP, para serializar un objeto se utiliza la función `serialize`. El resultado obtenido es un `string` que contiene un flujo de bytes, en el que se encuentran definidos todos los valores del objeto.

```
$p = new Producto();
$a = serialize($p);
```

Esta cadena se puede almacenar en cualquier parte, como puede ser la sesión del usuario, o una base de datos. A partir de ella, es posible reconstruir el objeto original utilizando la función **unserialize**.

```
$p = unserialize($a);
```

Las funciones `serialize` y `unserialize` se utilizan mucho con objetos, pero sirven para convertir en una cadena cualquier tipo de dato, excepto el tipo `resource`. Cuando se aplican a un objeto, convierten y recuperan toda la información del mismo, incluyendo sus atributos privados. La única información que no se puede mantener utilizando estas funciones es la que contienen los atributos estáticos de las clases.

Si simplemente queremos almacenar un objeto en la sesión del usuario, deberíamos hacer por tanto:

```
session_start();
$_SESSION['producto'] = serialize($p);
```

Pero en PHP esto aún es más fácil. Los objetos que se añadan a la sesión del usuario son serializados automáticamente. Por tanto, no es necesario usar **serialize** ni **unserialize**.

```
session_start();
$_SESSION['producto'] = $p;
```

Para poder deserializar un objeto, debe estar definida su clase. Al igual que antes, si lo recuperamos de la información almacenada en la sesión del usuario, no será necesario utilizar la función **unserialize**.

```
session_start();
$p = $_SESSION['producto'];
```

Como ya viste en el tema anterior, el mantenimiento de los datos en la sesión del usuario no es perfecta; tiene sus limitaciones. Si fuera necesario, es posible almacenar esta información en una base de datos. Para ello tendrás que usar las funciones `serialize` y `unserialize`, pues en este caso PHP ya no realiza la serialización automática.

En PHP además tienes la opción de personalizar el proceso de serialización y deserialización de un objeto, utilizando los métodos mágicos **__sleep** y **__wakeup**. Si en la clase está definido un método con nombre **__sleep**, se ejecuta antes de serializar un objeto. Igualmente, si existe un método **__wakeup**, se ejecuta con cualquier llamada a la función `unserialize`.

**http://www.php.net/manual/es/language.oop5.magic.php#language.oop5.magic.sleep**

**Si serializas un objeto utilizando serialize, ¿puedes almacenarlo en una base de datos MySQL?**

(•) **Verdadero.** ( ) Falso.

*No solo puedes, sino que además es la forma correcta de hacerlo.* `serialize` *convierte el objeto en una cadena, que se puede almacenar donde sea necesario.*

## 1.5.- Herencia.

La herencia es un mecanismo de la POO que nos permite definir nuevas clases en base a otra ya existente. Las nuevas clases que heredan también se conocen con el nombre de **subclases**. La clase de la que heredan se llama **clase base** o **superclase**.

Por ejemplo, en nuestra tienda web vamos a tener productos de distintos tipos. En principio hemos creado para manejarlos una clase llamada **Producto**, con algunos atributos y un método que genera una salida personalizada en formato HTML del código.

```
class Producto {
   public $codigo;
   public $nombre;
```

```
   public $nombre_corto;
   public $PVP;

   public function muestra() {
      print "<p>" . $this->codigo . "</p>";
   }
}
```

Esta clase es muy útil si la única información que tenemos de los distintos productos es la que se muestra arriba. Pero, si quieres personalizar la información que vas a tratar de cada tipo de producto (y almacenar, por ejemplo para los televisores, las pulgadas que tienen o su tecnología de fabricación), puedes crear nuevas clases que hereden de **`Producto`**. Por ejemplo, **`TV`**, **`Ordenador`**, **`Movil`**.

```
class TV extends Producto {
   public $pulgadas;
   public $tecnologia;
}
```

Como puedes ver, para definir una clase que herede de otra, simplemente tienes que utilizar la palabra **`extends`** indicando la superclase. Los nuevos objetos que se instancien a partir de la subclase son también objetos de la clase base; se puede comprobar utilizando el operador **`instanceof`**.

```
$t = new TV();
if ($t instanceof Producto) {
   // Este código se ejecuta pues la condición es cierta
…
}
```

Antes viste algunas funciones útiles para programar utilizando objetos y clases. Las de la siguiente tabla están además relacionadas con la herencia.

**Funciones de utilidad en la herencia en PHP5**

| Función | Ejemplo | Significado |
|---|---|---|
| **get_parent_class** | `echo "La clase padre es: " . get_parent_class($t);` | Devuelve el nombre de la clase padre del objeto o la clase que se indica. |
| **is_subclass_of** | `if (is_subclass_of($t, 'Producto') { … }` | Devuelve true si el objeto o la clase del primer parámetro, tiene como clase base a la que se indica en el segundo parámetro, o false en caso contrario. |

La nueva clase hereda todos los atributos y métodos públicos de la clase base, pero no los privados. Si quieres crear en la clase base un método no visible al exterior (como los privados) que se herede a las subclases, debes utilizar la palabra **`protected`** en lugar de **`private`**. Además, puedes redefinir el comportamiento de los métodos existentes en la clase base, simplemente creando en la subclase un nuevo método con el mismo nombre.

```
class TV extends Producto {
   public $pulgadas;
   public $tecnologia;

   public function muestra() {
      print "<p>" . $this->pulgadas . " pulgadas</p>";
   }
}
```

Existe una forma de evitar que las clases heredadas puedan redefinir el comportamiento de los métodos existentes en la superclase: utilizar la palabra **`final`**. Si en nuestro ejemplo hubiéramos hecho:

```
class Producto {
   public $codigo;
   public $nombre;
   public $nombre_corto;
   public $PVP;

   public final function muestra() {
      print "<p>" . $this->codigo . "</p>";
   }
}
```

En este caso el método `muestra` no podría redefinirse en la clase `TV`.

Incluso se puede declarar una clase utilizando `final`. En este caso no se podrían crear clases heredadas utilizándola como base.

```
final class Producto {
   …
}
```

Opuestamente al modificador `final`, existe también `abstract`. Se utiliza de la misma forma, tanto con métodos como con clases completas, pero en lugar de prohibir la herencia, obliga a que se herede. Es decir, una clase con el modificador `abstract` no puede tener objetos que la instancien, pero sí podrá utilizarse de clase base y sus subclases sí podrán utilizarse para instanciar objetos.

```
abstract class Producto {
   …
}
```

Y un método en el que se indique `abstract`, debe ser redefinido obligatoriamente por las subclases, y no podrá contener código.

```
class Producto {
   …

   abstract public function muestra();
}
```

Obviamente, no se puede declarar una clase como `abstract` y `final` simultáneamente. `abstract` obliga a que se herede para que se pueda utilizar, mientras que `final` indica que no se podrá heredar.

**La función is_subclass_of recibe como primer parámetro:**

( ) Un objeto. (•) **Un objeto o una clase.**

*Efectivamente, puedes pasarle un objeto o el nombre de una clase como primer parámetro, y te dirá si es o no una subclase del nombre de clase que le pases como segundo parámetro.*

Vamos a hacer una pequeña modificación en nuestra clase `Producto`. Para facilitar la creación de nuevos objetos, crearemos un constructor al que se le pasará un array con los valores de los atributos del nuevo producto.

```
class Producto {
   public $codigo;
   public $nombre;
   public $nombre corto;
   public $PVP;

   public function muestra() {
      print "<p>" . $this->codigo . "</p>";
   }

   public function   construct($row) {
      $this->codigo = $row['cod'];
      $this->nombre = $row['nombre'];
      $this->nombre_corto = $row['nombre_corto'];
      $this->PVP = $row['PVP'];
   }
}
```

¿Qué pasa ahora con la clase `TV`, qué hereda de `Producto`? Cuando crees un nuevo objeto de esa clase, ¿se llamará al constructor de `Producto`? ¿Puedes crear un nuevo constructor específico para `TV` que redefina el comportamiento de la clase base?

Empezando por esta última pregunta, obviamente puedes definir un nuevo constructor para las clases heredadas que redefinan el comportamiento del que existe en la clase base, tal y como harías

con cualquier otro método. Y dependiendo de si programas o no el constructor en la clase heredada, se llamará o no automáticamente al constructor de la clase base.

En PHP5, si la clase heredada no tiene constructor propio, se llamará automáticamente al constructor de la clase base (si existe). Sin embargo, si la clase heredada define su propio constructor, deberás ser tú el que realice la llamada al constructor de la clase base si lo consideras necesario, utilizando para ello la palabra `parent` y el operador de resolución de ámbito.

```
class TV extends Producto {
   public $pulgadas;
   public $tecnologia;

   public function muestra() {
      print "<p>" . $this->pulgadas . " pulgadas</p>";
   }

   public function __construct($row) {
      parent::  construct($row);
      $this->pulgadas = $row['pulgadas'];
      $this->tecnologia = $row['tecnologia'];
   }
}
```

Ya viste con anterioridad cómo se utilizaba la palabra clave `self` para tener acceso a la clase actual. La palabra `parent` es similar. Al utilizar `parent` haces referencia a la clase base de la actual, tal y como aparece tras `extends`.

**Si una subclase no tiene método constructor, y su clase base sí lo tiene, cuando se instancie un nuevo objeto de la subclase:**

- (●) **Se llamará automáticamente al constructor de la clase base.**
- ( ) No se llamará automáticamente al constructor de la clase base.

*Fíjate que esta llamada automática solo ocurre cuando la clase heredada no tiene constructor; en otro caso tendrás que hacer tú la llamada manualmente.*

## 1.6.- Interfaces.

Un interface es como una clase vacía que solamente contiene declaraciones de métodos. Se definen utilizando la palabra `interface`.

Por ejemplo, antes viste que podías crear nuevas clases heredadas de `Producto`, como `TV` o `Ordenador`. También viste que en las subclases podías redefinir el comportamiento del método `muestra` para que generara una salida en HTML diferente para cada tipo de producto.

Si quieres asegurarte de que todos los tipos de productos tengan un método `muestra`, puedes crear un interface como el siguiente.

```
interface iMuestra {
   public function muestra();
}
```

Y cuando crees las subclases deberás indicar con la palabra `implements` que tienen que implementar los métodos declarados en este interface.

```
class TV extends Producto implements iMuestra {
   …
   public function muestra() {
      print "<p>" . $this->pulgadas . " pulgadas</p>";
   }
   …
}
```

Todos los métodos que se declaren en un interface deben ser públicos. Además de métodos, los interfaces podrán contener constantes pero no atributos.

Un interface es como un contrato que la clase debe cumplir. Al implementar todos los métodos declarados en el interface se asegura la interoperabilidad entre clases. Si sabes que una clase implementa un interface determinado, sabes qué nombre tienen sus métodos, qué parámetros les debes pasar y, probablemente, podrás averiguar fácilmente con qué objetivo han sido escritos.
Por ejemplo, en la librería de PHP está definido el interface **`Countable`**.

```
Countable {
   abstract public int count ( void )
}
```

Si creas una clase para la cesta de la compra en la tienda web, podrías implementar este interface para contar los productos que figuran en la misma.

Antes aprendiste que en PHP5 una clase sólo puede heredar de otra. En PHP5 no existe la herencia múltiple. Sin embargo, sí es posible crear clases que implementen varios interfaces, simplemente separando la lista de interfaces por comas después de la palabra **`implements`**.

```
class TV extends Producto implements iMuestra, Countable {
   …
}
```

La única restricción es que los nombres de los métodos que se deban implementar en los distintos interfaces no coincidan. Es decir, en nuestro ejemplo, el interface **`iMuestra`** no podría contener un método **`count`**, pues éste ya está declarado en **`Countable`**.

En PHP5 también se pueden crear nuevos interfaces heredando de otros ya existentes. Se hace de la misma forma que con las clases, utilizando la palabra `extends`.

Una de las dudas más comunes en POO, es qué solución adoptar en algunas situaciones: interfaces o clases abstractas. Ambas permiten definir reglas para las clases que los implementen o hereden respectivamente. Y ninguna permite instanciar objetos. Las diferencias principales entre ambas opciones son:

- ✓ En las clases abstractas, los métodos pueden contener código. Si van a existir varias subclases con un comportamiento común, se podría programar en los métodos de la clase abstracta. Si se opta por un interface, habría que repetir el código en todas las clases que lo implemente.
- ✓ Las clases abstractas pueden contener atributos, y los interfaces no.
- ✓ No se puede crear una clase que herede de dos clases abstractas, pero sí se puede crear una clase que implemente varios interfaces.

Por ejemplo, en la tienda online va a haber dos tipos de usuarios: clientes y empleados. Si necesitas crear en tu aplicación objetos de tipo **`Usuario`** (por ejemplo, para manejar de forma conjunta a los clientes y a los empleados), tendrías que crear una clase no abstracta con ese nombre, de la que heredarían **`Cliente`** y **`Empleado`**.

```
class Usuario {
   …
}
class Cliente extends Usuario {
   …
}
class Empleado extends Usuario {
   …
}
```

Pero si no fuera así, tendrías que decidir si crearías o no `Usuario`, y si lo harías como una clase abstracta o como un interface.
Si por ejemplo, quisieras definir en un único sitio los atributos comunes a `Cliente` y a **`Empleado`**, deberías crear una clase abstracta `Usuario` de la que hereden.

```
abstract class Usuario {
   public $dni;
```

```
    protected $nombre;
    …
}
```

Pero esto no podrías hacerlo si ya tienes planificada alguna relación de herencia para una de estas dos clases.

Para finalizar con los interfaces, a la lista de funciones de PHP relacionadas con la POO puedes añadir las siguientes.

| Funciones de utilidad para interfaces en PHP5 | | |
|---|---|---|
| **Función** | **Ejemplo** | **Significado** |
| get_declared_interfaces | `print_r (get_declared_interfaces());` | Devuelve un array con los nombres de los interfaces declarados. |
| interface_exists | `if (interface_exists('iMuestra') { … }` | Devuelve true si existe el interface que se indica, o false en caso contrario. |

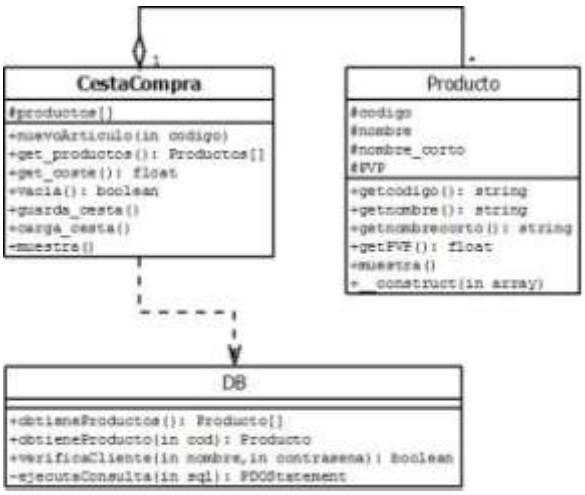
**Si en tu código utilizas un interface, y quieres crear uno nuevo basándote en él:**

- (•) **Puedes utilizar la herencia para crear el nuevo constructor extendiendo al primero.**
- ( ) No puedes hacerlo, pues no se puede utilizar herencia con los interfaces; solo con las clases.

*Sí; se hace utilizando extends, de la misma forma que con las clases*

## 1.7.- Ejemplo de POO en PHP.

Es hora de llevar a la práctica lo que has aprendido. Vamos a aplicar los principios de la POO a la aplicación de tienda web con la que trabajamos en la unidad anterior. Concretamente, vamos a crear en un subdirectorio **`include`** las siguientes clases:




- ✓ **`DB`**. Va a ser la clase encargada de interactuar con la base de datos.
- ✓ **`Producto`**. Las instancias de esta clase representan los productos que se venden en la tienda.
- ✓ **`CestaCompra`**. Con esta clase vas a gestionar los productos que escoge el cliente de la tienda para comprar.

Obviamente en una tienda web real, la lista de clases que deberíamos crear sería mucho más amplia y debería haberse estudiado el diagrama resultante. Veamos estas tres clases una a una.

Las instancias de la clase **`Producto`** van a almacenar la siguiente información de cada producto: **`código`**, **`nombre`**, **`nombre_corto`** y **`PVP`**. Cada valor se almacenará en un atributo de tipo `protected`, para limitar el acceso a su contenido y, a la vez, permitir su herencia (en caso de que en el futuro creemos clases heredadas). Además, en nuestra aplicación no será necesario cambiar sus valores, pero sí acceder a ellos, por lo que se creará un método de tipo **`get`** para cada uno.

También necesitamos un constructor. En nuestro caso, para facilitar la creación de objetos de la clase **`Producto`** a partir del contenido de la base de datos, le pasaremos como parámetro un array obtenido de una fila de la base de datos.
Por último, vamos a crear también un método para mostrar el código del producto. La clase **`Producto`** quedará por tanto como sigue:

```
class Producto {
   protected $codigo;
   protected $nombre;
```

```
    protected $nombre_corto;
    protected $PVP;

    public function getcodigo() {return $this->codigo; }
    public function getnombre() {return $this->nombre; }
    public function getnombrecorto() {return $this->nombre_corto; }
    public function getPVP() {return $this->PVP; }

    public function muestra() {print "<p>" . $this->codigo . "</p>";}

    public function   construct($row) {
        $this->codigo = $row['cod'];
        $this->nombre = $row['nombre'];
        $this->nombre_corto = $row['nombre_corto'];
        $this->PVP = $row['PVP'];
    }
}
```

La clase `DB` no necesita almacenar ninguna información; simplemente deberá contener métodos para realizar acciones sobre la base de datos. Por tanto, vamos a definir en la misma únicamente métodos estáticos. No hará falta instanciar ningún objeto de esta clase.

Los métodos son los siguientes:

- ✓ `obtieneProductos`. Devuelve un array con todos los productos de la base de datos.
- ✓ `obtieneProducto($codigo)`. Devuelve el producto que coincide con el código que se indica.
- ✓ `verificaCliente($nombre, $contrasena)`. Devuelve `true` o `false`, según sean correctas o no las credenciales que se proporcionen.

```
<?php
require_once('Producto.php');

class DB {
    protected static function ejecutaConsulta($sql) {
        $opc = array(PDO::MYSQL_ATTR_INIT_COMMAND => "SET NAMES utf8");
        $dsn = "mysql:host=localhost;dbname=dwes";
        $usuario = 'dwes';
        $contrasena = 'abc123.';

        $dwes = new PDO($dsn, $usuario, $contrasena, $opc);
        $resultado = null;
        if (isset($dwes)) $resultado = $dwes->query($sql);
            return $resultado;
    }

    public static function obtieneProductos() {
        $sql = "SELECT cod, nombre_corto, nombre, PVP FROM producto;";
        $resultado = self::ejecutaConsulta ($sql);
        $productos = array();

        if($resultado) {
            // Añadimos un elemento por cada producto obtenido
            $row = $resultado->fetch();
            while ($row != null) {
                $productos[] = new Producto($row);
                $row = $resultado->fetch();
            }
        }

        return $productos;
    }

    public static function obtieneProducto($codigo) {
        $sql = "SELECT cod, nombre_corto, nombre, PVP FROM producto";
        $sql .= " WHERE cod='" . $codigo . "'";
        $resultado = self::ejecutaConsulta ($sql);
        $producto = null;

        if(isset($resultado)) {
            $row = $resultado->fetch();
            $producto = new Producto($row);
        }

        return $producto;
```

```
    }

    public static function verificaCliente($nombre, $contrasena) {
        $sql = "SELECT usuario FROM usuarios ";
        $sql .= "WHERE usuario='$nombre' ";
        $sql .= "AND contrasena='" . md5($contrasena) . "';";
        $resultado = self::ejecutaConsulta ($sql);
        $verificado = false;

        if(isset($resultado)) {
            $fila = $resultado->fetch();
            if($fila !== false) $verificado=true;
        }
        return $verificado;
    }

}

?>
```

Utilizaremos de apoyo un método **`protected`**, **`ejecutaConsulta`**, que será el que realmente ejecute las consultas sobre la base de datos.

Por último, la clase **`CestaCompra`** debe almacenar un array con los productos que figuran en la cesta. Ese array lo crearemos como **`protected`** para controlar el acceso a su contenido, pero necesitamos un método **`get_productos`** para devolverlo.

Además, hemos implementado los siguientes métodos en la misma.

- ✓ `nuevo articulo($codigo)`. Introduce en la cesta el artículo indicado por su código.
- ✓ `get productos`. Devuelve un array con todos los productos de la base de datos.
- ✓ `get_coste`. Devuelve el coste de los productos que figuran en la cesta.
- ✓ `vacia`. Devuelve `true` o `false`, según la cesta esté o no vacía.
- ✓ `obtieneProductos`. Devuelve un array con todos los productos de la base de datos.

También necesitamos dos funciones para guardar la cesta en la sesión del usuario, y para recuperarla. Y programaremos otra más para mostrar el contenido de la cesta en formato HTML.

- ✓ `guarda_cesta`. Guarda la cesta en la sesión del usuario.
- ✓ `carga cesta`. Recupera el contenido de la cesta de la sesión del usuario.
- ✓ `muestra`. Genera una salida en formato HTML con el contenido de la cesta.

```
<?php
require_once('DB.php');

class CestaCompra {
    protected $productos = array();

    // Introduce un nuevo artículo en la cesta de la compra
    public function nuevo_articulo($codigo) {
        $producto = DB::obtieneProducto($codigo);
        $this->productos[] = $producto;
    }

    // Obtiene los artículos en la cesta
    public function get_productos() { return $this->productos; }

    // Obtiene el coste total de los artículos en la cesta
    public function get_coste() {
        $coste = 0;
        foreach($this->productos as $p) $coste += $p->getPVP();
        return $coste;
    }

    // Devuelve true si la cesta está vacía
    public function vacia() {
        if(count($this->productos) == 0) return true;
        return false;
    }

    // Guarda la cesta de la compra en la sesión del usuario
```

```
   public function guarda_cesta() { $_SESSION['cesta'] = $this; }

   // Recupera la cesta de la compra almacenada en la sesión del usuario
   public static function carga_cesta() {
      if (!isset($ SESSION['cesta'])) return new CestaCompra();
      else return ($_SESSION['cesta']);
   }

   // Muestra el HTML de la cesta de la compra, con todos los productos
   public function muestra() {
      // Si la cesta está vacía, mostramos un mensaje
      if (count($this->productos)==0)  print "<p>Cesta vacía</p>";
      //  y si no está vacía, mostramos su contenido
      else foreach ($this->productos as $producto) $producto->muestra();
   }
}

?>
```

El resto de ficheros que componen la tienda web tendremos que reescribirlos para que utilicen las clases que acabas de definir. En general, el resultado obtenido es mucho más claro y conciso. Veamos algunos ejemplos:

En **`login.php`**, para autentificar al usuario basta con hacer:

```
if (DB::verificaCliente($_POST['usuario'], $_POST['password'])) {
   …
}
```

En **`productos.php`**, el código para vaciar o añadir un nuevo producto a la cesta de la compra del usuario será:

```
// Recuperamos la cesta de la compra
$cesta = CestaCompra::carga_cesta();
// Comprobamos si se ha enviado el formulario de vaciar la cesta
if (isset($ POST['vaciar'])) {
   unset($ SESSION['cesta']);
   $cesta = new CestaCompra();
}
// Comprobamos si se quiere añadir un producto a la cesta
if (isset($_POST['enviar'])) {
   $cesta->nuevo_articulo($_POST['cod']);
   $cesta->guarda cesta();
}
```

Para tener el código más organizado, hemos creado una función para mostrar el listado de todos los productos:

```
function creaFormularioProductos(){
   $productos = DB::obtieneProductos();
   foreach ($productos as $p) {
      // Creamos el formulario en HTML para cada producto
      …
   }
}
```


Prueba a instalar y ejecutar la aplicación de tienda online resultante. Revisa el código y comprueba que entiendes su funcionamiento.
Aplicación de tienda online resultante.

**La clase DB tiene todos sus métodos estáticos. No tiene sentido por tanto crear ningún objeto de esa clase, y podría haberse implementado igualmente como un interface.**

○ Verdadero. ◉ **Falso.**

*Efectivamente, no se podría, pues no se pueden programar métodos en los interfaces.*

## 2.- Programación en capas.

**Caso práctico**
*Después de unas semanas, **Carlos** ha conseguido reprogramar su aplicación de catalogación utilizando orientación a objetos. Le ha costado bastante esfuerzo, pero reconoce que el resultado merece la pena. El código resultante es mucho más limpio, y cada clase de las que ha creado tiene un cometido concreto y bien definido.*
*Ahora es mucho más fácil hacer cambios en el código. Definitivamente, tendrán que utilizar orientación a objetos cuando empiecen el nuevo proyecto. Mientras tanto, se propone volver sobre una asignatura pendiente: mejorar el aspecto de sus páginas. Y aunque conoce el lenguaje HTML, y sabe utilizar las hojas de estilo, decide pedirle consejo a un amigo suyo que se dedica al diseño de webs.*
*Su amigo trabaja en otra empresa y su función es diseñar el aspecto de los sitios web que crean. Cuando ve la aplicación de **Carlos**, queda muy impresionado por lo que ha avanzado en poco tiempo. Tras examinarla, le indica que tal y como está es muy difícil cambiar su aspecto. Tiene el código HTML distribuido en diversos ficheros, y entremezclado con el código PHP.*
*Le comenta que en su empresa utilizan mecanismos de separación del código y le anima a que los pruebe. Si lo hace, él se ofrece a darle un diseño más profesional a su aplicación. —¡Manos a la obra!*

En el ejemplo anterior, hemos programado una aplicación web sencilla utilizando programación orientada a objetos. Sin embargo, si observaste el resultado obtenido, habrás visto como en muchas ocasiones se mezcla el código propio de la lógica de la aplicación, con el código necesario para crear el interface web que se presenta a los usuarios.

Por ejemplo, tanto la clase `CestaCompra` como la clase `Producto`, cuyo objetivo debería ser implementar la lógica de la aplicación, tienen un método llamado `muestra` destinado a generar etiquetas HTML. E inversamente, en algunas páginas que deberían simplemente generar HTML, puedes encontrar código que forma parte de la lógica de la aplicación. Por ejemplo, en la página `productos.php`, el código para incluir un nuevo producto en la cesta de la compra se encuentra mezclado con las etiquetas HTML.

Existen varios métodos que permiten separar la lógica de presentación (en nuestro caso, la que genera las etiquetas HTML) de la lógica de negocio, donde se implementa la lógica propia de cada aplicación. El más extendido es el patrón de diseño Modelo – Vista – Controlador (MVC). Este patrón pretende dividir el código en tres partes, dedicando cada una a una función definida y diferenciada de las otras.

- ✓ **Modelo**. Es el encargado de manejar los datos propios de la aplicación. Debe proveer mecanismos para obtener y modificar la información del mismo. Si la aplicación utiliza algún tipo de almacenamiento para su información (como un SGBD), tendrá que encargarse de almacenarla y recuperarla.
- ✓ **Vista**. Es la parte del modelo que se encarga de la interacción con el usuario. En esta parte se encuentra el código necesario para generar el interface de usuario (en nuestro caso en HTML), según la información obtenida del modelo.
- ✓ **Controlador**. En este módulo se decide qué se ha de hacer, en función de las acciones del usuario con su interface. Con esta información, interactúa con el modelo para indicarle las acciones a realizar y, según el resultado obtenido, envía a la vista las instrucciones necesarias para generar el nuevo interface.

La gran ventaja de este patrón de programación es que genera código muy estructurado, fácil de comprender y de mantener. En la web puedes encontrar algunos ejemplos de implementación del modelo MVC en PHP. Échale un vistazo al siguiente artículo sobre MVC en PHP.

**Anexo I - Patrón MVC Modelo Vista Controlador en PHP**

Aunque puedes programar utilizando MVC por tu cuenta, es más habitual utilizar el patrón MVC en conjunción con un framework o marco de desarrollo. Existen numerosos frameworks disponibles en PHP, muchos de los cuales incluyen soporte para MVC. En esta unidad no profundizaremos en la utilización de un framework específico, pero existen numerosos recursos con información en Internet, incluyendo la página www.phpwebframeworks.com , dedicada exclusivamente a ellos.

| PHP Framework | PHP4 | PHP5 | MVC | Multiple DB's | ORM | DB Objects | Templates | Caching | Validation | Ajax | Auth Module | Modules | EDP |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Akelos | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | |
| Ash.MVC | | ✓ | ✓ | | | ✓ | ✓ | | ✓ | | ✓ | ✓ | |
| CakePHP | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | | ✓ | ✓ | ✓ | ✓ | ✓ | |
| CodeIgniter | ✓ | ✓ | ✓ | ✓ | | ✓ | ✓ | ✓ | ✓ | | | | |
| DIY | | ✓ | ✓ | | ✓ | ✓ | ✓ | ✓ | | ✓ | | | |
| eZ Components | | ✓ | | ✓ | | ✓ | ✓ | ✓ | ✓ | | | | |
| Fusebox | ✓ | ✓ | ✓ | ✓ | | | | ✓ | | ✓ | | ✓ | |
| PHP on TRAX | | ✓ | ✓ | ✓ | ✓ | ✓ | | | ✓ | ✓ | | ✓ | |
| PHPDev Shell | | ✓ | ✓ | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | |
| PhpOpenbiz | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | | ✓ | ✓ | ✓ | | |
| Prado | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| QPHP | ✓ | ✓ | ✓ | ✓ | | ✓ | ✓ | | ✓ | ✓ | ✓ | ✓ | ✓ |
| Seagull | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | |
| Symfony | | ✓ | ✓ | ✓ | ✓ | ✓ | | ✓ | ✓ | ✓ | ✓ | ✓ | |
| WACT | ✓ | ✓ | ✓ | ✓ | | ✓ | ✓ | | ✓ | | | ✓ | |
| WASP | | ✓ | ✓ | | | ✓ | ✓ | | ✓ | ✓ | ✓ | ✓ | |
| Yii | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Zend | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | |
| ZooP | ✓ | ✓ | ✓ | ✓ | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | | |

## 2.1.- Separación de la lógica de negocio.

Otros mecanismos disponibles en PHP, menos complejos que la utilización del patrón MVC, y que también permiten la separación de la lógica de presentación y la lógica de negocio, son los llamados motores de plantillas (template engines).

Un **motor de plantillas web** es una aplicación que genera una página web a partir de un fichero con la información de presentación (denominado plantilla o template, que viene a ser similar a la vista en el patrón MVC) y otro con la lógica interna de la aplicación (similar al modelo de MVC). De esta forma, es sencillo dividir el trabajo de programación de una aplicación web en dos perfiles: un programador, que debe conocer el lenguaje de programación en el que se implementará la lógica de la aplicación (en nuestro caso PHP), y un diseñador, que se encargará de elaborar las plantillas, (en el caso de la web básicamente en HTML, aunque como veremos la lógica de presentación que se incorpore utilizará un lenguaje propio).

En PHP existen varios motores de plantillas con diferentes características. Quizás el más conocido es Smarty, de código abierto y disponible bajo licencia LGPL.

**http://www.smarty.net/**

Entre las características de Smarty cabe destacar:

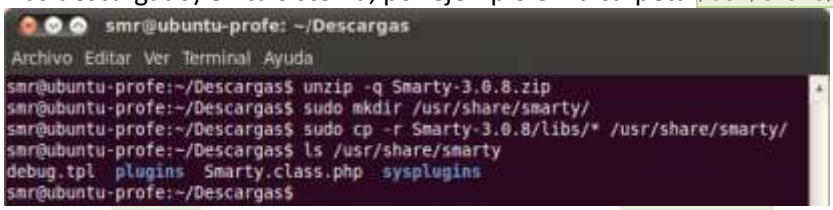
- ✓ Permite la inclusión en las plantillas de una lógica de presentación compleja.
- ✓ Acelera la generación de la página web resultante. Uno de los problemas de los motores de plantillas es que su utilización influye negativamente en el rendimiento. Smarty convierte

internamente las plantillas a guiones PHP equivalentes, y posibilita el almacenamiento del resultado obtenido en memoria temporal.

- ✓ Al ser usado por una amplia comunidad de desarrolladores, existen multitud de ejemplos y foros para la resolución de los problemas que te vayas encontrando al utilizarlo.

Para instalar Smarty, debes descargar la última versión desde su sitio web, y seguir los siguientes pasos:

1. Copiar los archivos de la librería de Smarty (el directorio `libs` que obtienes tras descomprimir el fichero que has descargado) en tu sistema; por ejemplo en la carpeta `/usr/share/smarty`.




2. Modificar el fichero `php.ini` para que se incluya en la variable `include_path` de PHP la ruta en la que acabas de instalar Smarty, y reiniciar Apache para aplicar los cambios.

```
;;;;;;;;;;;;;;;;;;;;;;;;;
; Paths and Directories ;
;;;;;;;;;;;;;;;;;;;;;;;;;

; UNIX: "/path1:/path2"
;include_path = ".:/usr/share/php"
;
; Windows: "\path1;\path2"
;include_path = ".;c:\php\includes"
;
; PHP's default setting for include_path is ".:/path/to/php/pear"
; http://php.net/include-path
include_path = ".:/usr/share/php:/usr/share/pear:usr/share/smarty"
```

3. Crear la estructura de directorios necesaria para Smarty. Concretamente debes crear cuatro directorios, con nombres `templates`, `templates_c`, `configs` y `cache`. Es conveniente que estén ubicados en un lugar no accesible por el servidor web, y en una ubicación distinta para cada aplicación web que programes. Por ejemplo, puedes crear en la raíz de tu servidor una carpeta llamada smarty, y bajo ella otra con el nombre de cada aplicación (por ejemplo `stienda`), que será la que contendrá las carpetas.

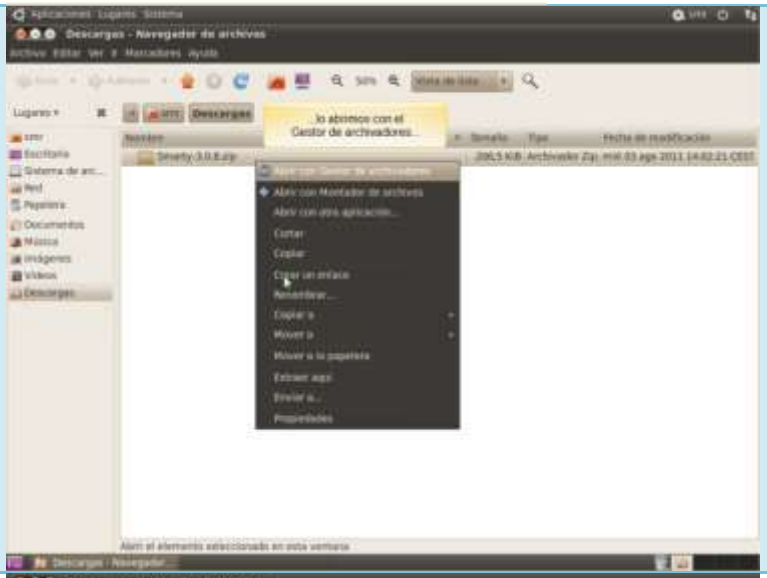
**Descargamos Smarty desde su** [página oficial]

**Existen múltiples versiones, pero descargaremos la última versión estable**

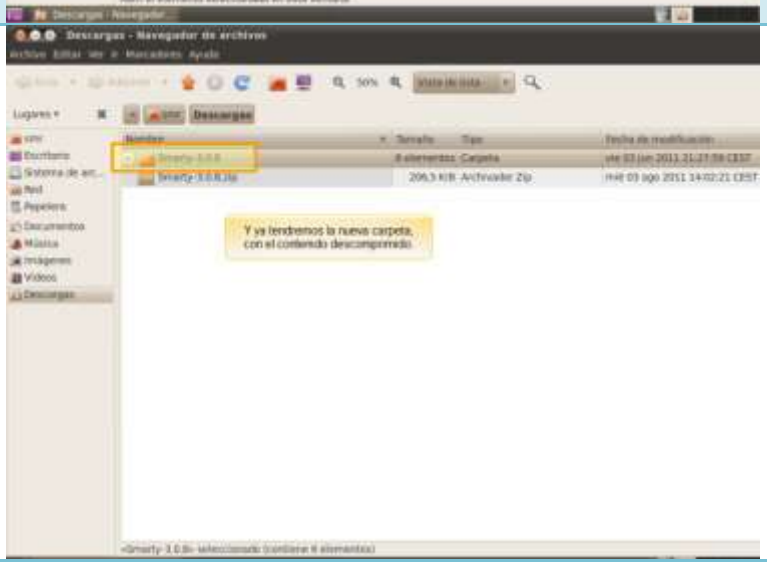

**Abrimos ahora un nuevo terminal y tecleamos:**

```
gksudo nautilus
```

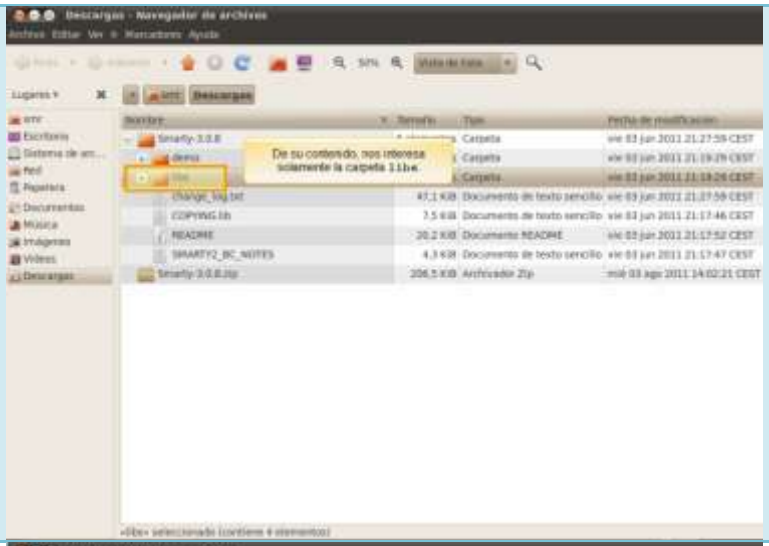
**para abrir un navegador de archivos con permisos de superusuario.**
**Nos desplazamos hasta donde se encuentra el archivo descargado, y lo abrimos con el *Gestor de archivadores***



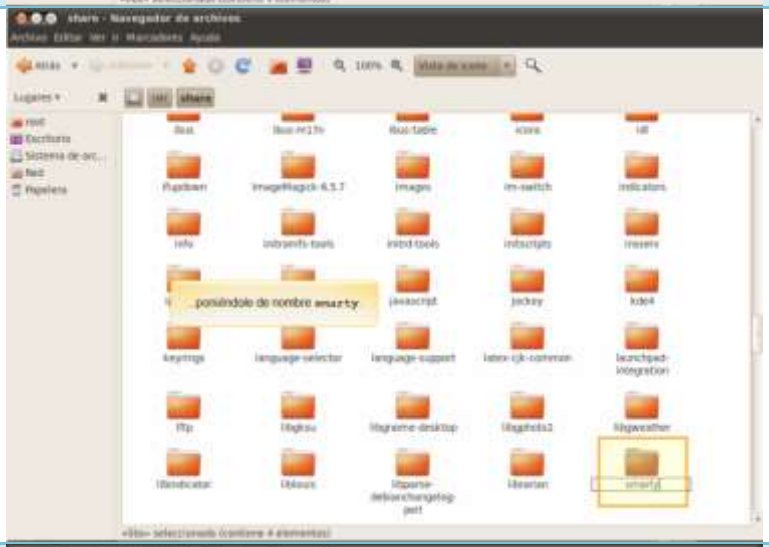

**Extraemos todos los archivos que contiene el fichero comprimido, y tendremos en el mismo lugar de la descarga la nueva carpeta con los archivos descomprimidos.**



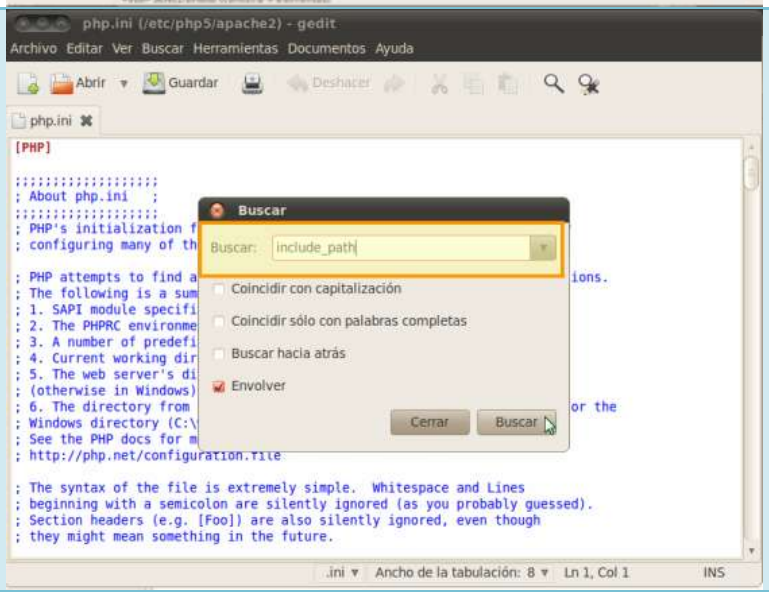
**De su contenido nos interesa la carpeta `libs`, la cual copiaremos a `/usr/share`**



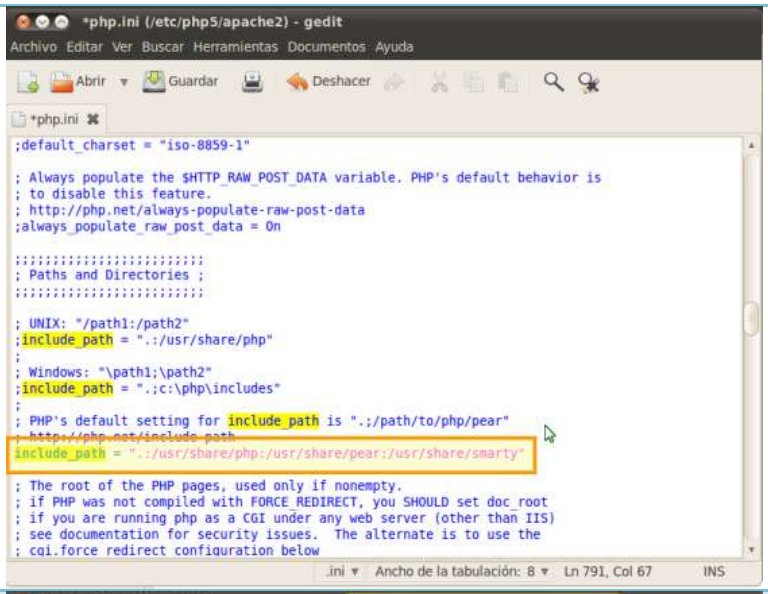

**Una vez copiada, la renombraremos como smarty**




**Ahora, para hacerlo visible a php tendremos que editar `php.ini` que se encuentra en `/etc/php5/apache2` y buscamos la cadena `include_path`**

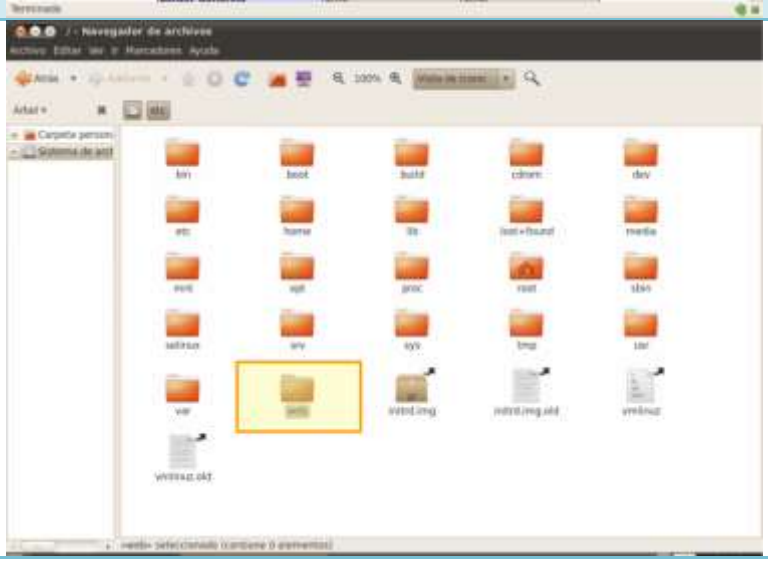


**Y le añadimos la ruta de Smarty**

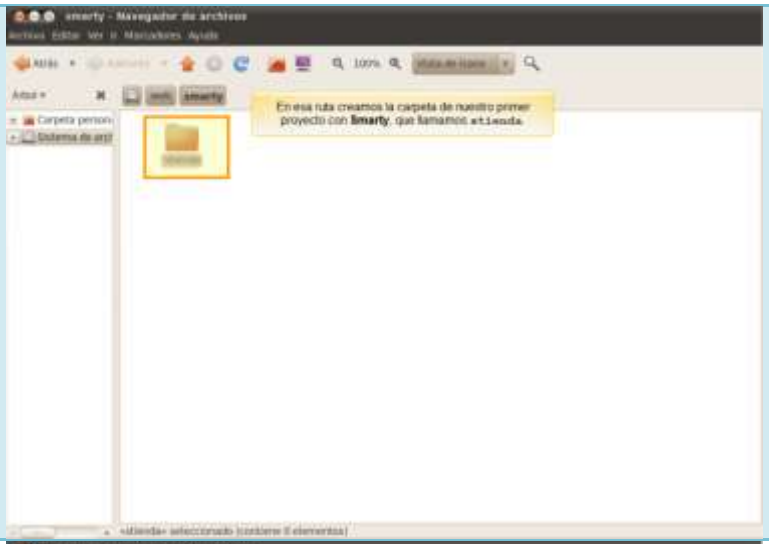



**Tras guardar los cambios, abrimos de nuevo una terminal y recargamos el servidor apache:**
`sudo /etc/init.d/apache2 reload`
**Ahora abrimos un navegador para comprobar que se ha cargado correctamente, para lo que usaremos la función** `phpinfo()`

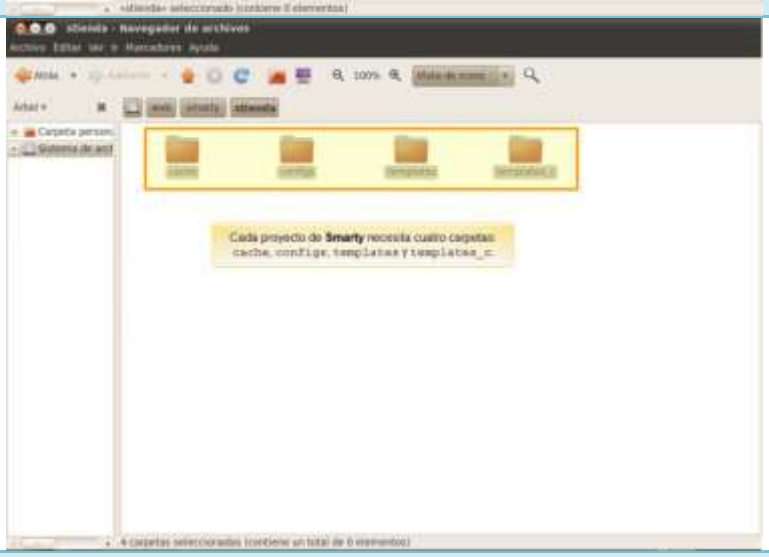
**Por último, hemos de crear las carpetas necesarias para Smarty.**
**Vamos a** `/etc` **y creamos una carpeta denominada** `web` **(por ejemplo)**



**Dentro de la carpeta recién creada, añadiremos otra a la que llamaremos `smarty` y que será la que recoja todos nuestros proyectos con Smarty, por ejemplo, uno al que denominaremos `stienda`**




**Cada proyecto Smarty tendrá al menos 4 carpetas: `cache`, `configs`, `templates` y `templates_c`**




**Y ya tendríamos Smarty instalado. Sólo falta empezar a utilizarlo en nuestras aplicaciones web**

**Al utilizar Smarty, las plantillas que obtienes no precisan ningún tipo de programación; sólo el contenido normal de una página web.**

○ Verdadero ◉ **Falso.**

*Al utilizar Smarty logras separar la lógica de negocio de la información relativa a la presentación; pero en muchas ocasiones, ésta última sigue necesitando algún código, aunque no necesariamente en lenguaje PHP*

Para utilizar Smarty, simplemente tienes que añadir a tus páginas PHP el fichero `Smarty.class.php`, que es donde está declarada la clase `Smarty`. Después debes instanciar un nuevo objeto de esa clase, y configurar la ruta a cada uno de los directorios que acabas de crear.

```
require_once('Smarty.class.php');
$smarty = new Smarty;
$smarty->template_dir = '/web/smarty/stienda/templates/';
$smarty->compile_dir = '/web/smarty/stienda/templates_c/';
$smarty->config_dir = '/web/smarty/stienda/configs/';
$smarty->cache_dir = '/web/smarty/stienda/cache/';
```

Cuando quieras que algún valor o variable obtenido en tus páginas, esté disponible para mostrarlo en las páginas web a través de una plantilla, tienes que usar el método `assign`, indicando el nombre del identificador. Puedes utilizar `assign` con variables de cualquier tipo, incluyendo arrays asociativos y objetos.

```
$smarty->assign('usuario', $_SESSION['usuario']);
$smarty->assign('productoscesta', $cesta->get_productos());
$smarty->assign('coste', $cesta->get_coste());
```

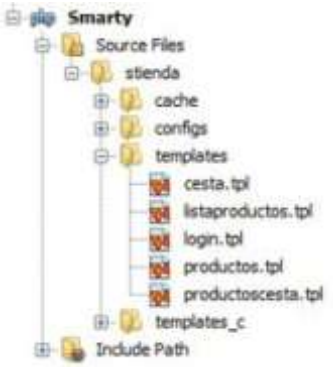
Y una vez que hayas preparado los identificadores que se usarán en la plantilla, deberás mostrarla utilizando el método `display`.

```
$smarty->display('cesta.tpl');
```

Así, por ejemplo, al utilizar Smarty en la página `productos.php` de la tienda online, puedes obtener algo como:

```
require_once('include/DB.php');
require_once('include/CestaCompra.php');
require_once('Smarty.class.php');
// Recuperamos la información de la sesión
session_start();
// Y comprobamos que el usuario se haya autentificado
if (!isset($_SESSION['usuario']))
   die("Error - debe <a href='login.php'>identificarse</a>.<br />");
// Recuperamos la cesta de la compra
$cesta = CestaCompra::carga_cesta();
// Cargamos la librería de Smarty
$smarty = new Smarty;
$smarty->template_dir = '/web/smarty/stienda/templates/';
$smarty->compile_dir = '/web/smarty/stienda/templates_c/';
$smarty->config_dir = '/web/smarty/stienda/configs/';
$smarty->cache_dir = '/web/smarty/stienda/cache/';
// Comprobamos si se ha enviado el formulario de vaciar la cesta
if (isset($_POST['vaciar'])) {
   unset($_SESSION['cesta']);
   $cesta = new CestaCompra();
}
// Comprobamos si se quiere añadir un producto a la cesta
if (isset($_POST['enviar'])) {
   $cesta->nuevo_articulo($_POST['cod']);
   $cesta->guarda_cesta();
}
// Ponemos a disposición de la plantilla los datos necesarios
$smarty->assign('usuario', $_SESSION['usuario']);
$smarty->assign('productos', DB::obtieneProductos());
$smarty->assign('productoscesta', $cesta->get_productos());
// Mostramos la plantilla
$smarty->display('productos.tpl');
```

## 2.2.- Generación del interface de usuario.




Las plantillas de Smarty son ficheros con extensión `tpl`, en los que puedes incluir prácticamente cualquier contenido propio de una página web. Además, en el medio intercalarás delimitadores para indicar la inclusión de datos y de lógica propia de la presentación.

Los delimitadores de Smarty son llaves. De los distintos elementos que puedes incluir entre las llaves están:

- ✓ **Comentarios**. Van encerrados entre asteriscos.

```
{* Este es un comentario de plantilla en Smarty *}
```

- ✓ **Variables**. Se incluye simplemente su nombre, precedido por el símbolo `$`. También se pueden especificar modificadores, separándolos de la variable por una barra vertical. Existen varios modificadores para, por ejemplo, dar formato a una fecha (`date_format`) o mostrar un contenido predeterminado si la variable está vacía (`default`).

```
{$producto->codigo}
```

- ✓ **Estructura de procesamiento condicional**: `if`, `elseif`, `else`. Permite usar condiciones, de forma similar a PHP, para decidir si se procesa o no cierto contenido.

```
{if empty($productoscesta)}
  <p>Cesta vacía</p>
{else}
  …
{/if}
```

- ✓ **Bucles**: `foreach`. Son muy útiles para mostrar varios elementos, por ejemplo en una tabla. Deberás indicar al menos con `from` el array en el que están los elementos, y con `item` la variable a la que se le irán asignado los elementos en cada iteración.

```
{foreach from=$productoscesta item=producto}
  <p>{$producto->codigo}</p>
{/foreach}
```

- ✓ **Inclusión de otras plantillas**. Smarty permite descomponer una plantilla compleja en trozos más pequeños y almacenarlos en otras plantillas, que se incluirán en la actual utilizando la sentencia `include`.

```
<div id="cesta">
  {include file="productoscesta.tpl"}
</div>
<div id="productos">
  {include file="listaproductos.tpl"}
</div>
```

Las que acabas de ver son una pequeña parte de las funcionalidades que ofrece Smarty. En su página web, puedes acceder a la documentación completa. En el momento de escribir estas líneas, la última versión disponible en español de la documentación era la correspondiente a la versión 2.

**http://www.smarty.net/docsv2/es/**

A partir del código obtenido utilizando POO para la tienda web, aplica el motor de plantillas Smarty para dividir la lógica de presentación de la lógica de negocio. Utiliza como apoyo la documentación disponible en Internet. Cuando acabes, puedes comparar lo que has obtenido con la solución propuesta.

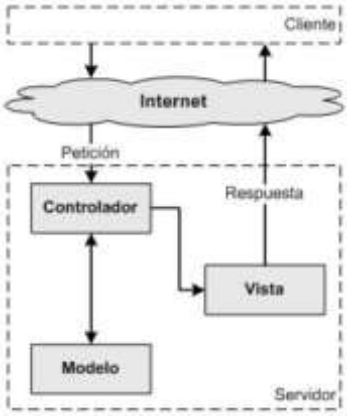
**Solución propuesta.**

**Las plantillas que crees en Smarty es preferible alojarlas:**

- (•) **En un lugar no accesible por el servidor web.**
- ( ) En un lugar accesible por el servidor web.

*En realidad es PHP el que debe acceder a ellas, no el servidor web.*

# Anexo I - Patrón MVC Modelo Vista Controlador en PHP




El patrón clásico del diseño web conocido como arquitectura MVC, está formado por tres niveles:

1. El **modelo** representa la información con la que trabaja la aplicación, es decir, su ***lógica de negocio***.
2. La **vista** transforma el modelo en una página web que permite al usuario interactuar con ella.
3. El **controlador** se encarga de procesar las interacciones del usuario y realiza los cambios apropiados en el modelo o en la vista.

**La arquitectura MVC** separa la ***lógica de negocio*** (**el modelo**) y ***la presentación*** (**la vista**) por lo que se consigue un mantenimiento más sencillo de las aplicaciones. Si por ejemplo una misma aplicación debe ejecutarse tanto en un navegador estándar como un navegador de un dispositivo móvil, solamente es necesario crear una **vista** nueva para cada dispositivo; manteniendo el **controlador** y el **modelo** original. El **controlador se encarga de aislar al modelo y a la vista de los detalles del protocolo** utilizado para las peticiones (HTTP, consola de comandos, email, etc.). **El modelo se encarga de la abstracción de la lógica relacionada con los datos**, haciendo que la vista y las acciones sean independientes de, por ejemplo, el tipo de gestor de bases de datos utilizado por la aplicación.

## Las capas de la arquitectura MVC

Para poder entender las **ventajas de utilizar el patrón MVC**, se va a transformar una aplicación simple realizada con PHP en una aplicación que sigue la **arquitectura MVC**. Un buen ejemplo para ilustrar esta explicación es el de mostrar una lista con las últimas entradas o artículos de un blog como este mismo http://arleytriana.blogspot.com.

### *Programación simple y llana*

Utilizando solamente PHP normal y corriente, el script necesario para mostrar los artículos almacenados en una base de datos se muestra a continuación:

#### *Un script simple*

```
<?php
   // Conectar con la base de datos y seleccionarla
   $conexion = mysql_connect('localhost', 'miusuario', 'micontrasena');
   mysql_select_db('blog_db', $conexion);
   // Ejecutar la consulta SQL
   $resultado = mysql_query('SELECT fecha, titulo FROM articulo',$conexion);
?>
<html>
   <head>
      <title>Listado de Artículos</title>
   </head>
   <body>
      <h1>Listado de Artículos</h1>
      <table>
         <tr><th>Fecha</th><th>Titulo</th></tr>
         <?php
            // Mostrar los resultados con HTML
            while ($fila = mysql fetch array($resultado, MYSQL ASSOC)){
               echo "\t<tr>\n";
               printf("\t\t<td> %s </td>\n", $fila['fecha']);
               printf("\t\t<td> %s </td>\n", $fila['titulo']);
               echo "\t</tr>\n";
            }
         ?>
      </table>
   </body>
</html>
<?php
   // Cerrar la conexion
```

```
   mysql_close($conexion);
?>
```

**El script anterior es fácil de escribir y rápido de ejecutar, pero muy difícil de mantener y actualizar. Los principales problemas del código anterior son:**

- ✓ No existe protección frente a errores (¿qué ocurre si falla la conexión con la base de datos?).
- ✓ El código HTML y el código PHP están mezclados en el mismo archivo e incluso en algunas partes están entrelazados.
- ✓ El código solo funciona si la base de datos es MySQL.

## Separando la presentación

Las llamadas a `echo` y `printf` del listado anterior dificultan la lectura del código. De hecho, modificar el código HTML del script anterior para mejorar la presentación es un muy complicado debido a cómo está programado. Así que el código va a ser dividido en dos partes. En primer lugar, el código PHP puro con toda la ***lógica de negocio*** se incluye en el **script del controlador**, como se muestra a continuación.

### La parte del controlador, en `index.php`

```
<?php
   // Conectar con la base de datos y seleccionarla
   $conexion = mysql_connect('localhost', 'miusuario', 'micontrasena');
   mysql_select_db('blog_db', $conexion);
   // Ejecutar la consulta SQL
   $resultado = mysql_query('SELECT fecha, titulo FROM articulo',$conexion);
   // Crear el array de elementos para la capa de la vista
   $articulos = array();
   while ($fila = mysql_fetch_array($resultado, MYSQL_ASSOC)){
      $articulos[] = $fila;
   }
   // Cerrar la conexión
   mysql_close($conexion);
   // Incluir la lógica de la vista
   require('vista.php');
?>
```

El código HTML, que contiene cierto código PHP a modo de plantilla, se almacena en el script de la vista, como se muestra a continuación.

### La parte de la vista, en `vista.php`

```
<html>
   <head>
      <title>Listado de Artículos</title>
   </head>
   <body>
      <h1>Listado de Artículos</h1>
      <table>
         <tr><th>Fecha</th><th>Título</th></tr>
         <?php foreach ($articulos as $articulo): ?>
         <tr>
            <td><?php echo $articulo['fecha'] ?></td>
            <td><?php echo $articulo['titulo'] ?></td>
         </tr>
         <?php endforeach; ?>
      </table>
   </body>
</html>
```

Una buena regla general para determinar si la parte de la vista está suficientemente limpia de código es que debería contener una cantidad mínima de código PHP, la suficiente como para que un diseñador HTML sin conocimientos de PHP pueda entenderla. Las instrucciones más comunes en la parte de la vista suelen ser `echo`, `if/else`, `foreach/endforeach` y poco más. Además, no se deben incluir instrucciones PHP que generen etiquetas HTML.

Toda la lógica se ha centralizado en el script del controlador**,** que solamente contiene código PHP y ningún tipo de HTML. De hecho, y como puedes imaginar, el mismo controlador se puede reutilizar para otros tipos de presentaciones completamente diferentes, como por ejemplo un archivo PDF o una estructura de tipo XML.

## Separando la manipulación de los datos

La mayor parte del script del controlador se encarga de la manipulación de los datos. Pero, ¿qué ocurre si se necesita la lista de entradas del blog para otro controlador, por ejemplo uno que se dedica a generar el canal RSS de las entradas del blog? ¿Y si se quieren centralizar todas las consultas a la base de datos en un único sitio para evitar duplicidades?

¿Qué ocurre si cambia el modelo de datos y la tabla articulo pasa a llamarse articulo_blog? ¿Y si se quiere cambiar a PostgreSQL en vez de MySQL? Para poder hacer todo esto, es imprescindible eliminar del controlador todo el código que se encarga de la manipulación de los datos y ponerlo en otro script, llamado el modelo, tal y como se muestra a continuación.

### La parte del modelo, en `modelo.php`

```
<?php
   function getTodosLosArticulos(){
      // Conectar con la base de datos y seleccionarla
      $conexion = mysql_connect('localhost', 'miusuario', 'micontrasena');
      mysql_select_db('blog_db', $conexion);
      // Ejecutar la consulta SQL
      $resultado = mysql_query('SELECT fecha, titulo FROM articulo',$conexion);
      // Crear el array de elementos para la capa de la vista
      $articulos = array();
      while ($fila = mysql_fetch_array($resultado, MYSQL_ASSOC)){
         $articulos[] = $fila;
      }
      // Cerrar la conexión
      mysql close($conexion);
      return $articulos;
   }
?>
```

El controlador modificado se puede ver aquí.

### La parte del controlador, modificada, en `index.php`

```
<?php
   // Incluir la lógica del modelo
   require once('modelo.php');
   // Obtener la lista de artículos
   $articulos = getTodosLosArticulos();
   // Incluir la lógica de la vista
   require('vista.php');
?>
```

Ahora el controlador es mucho más fácil de leer. Su única tarea es la de **obtener los datos del modelo y pasárselos a la vista**. En las aplicaciones más complejas, **el controlador** se encarga además de **procesar las peticiones, las sesiones de los usuarios, la autenticación,** etc. El uso de nombres apropiados para las funciones del modelo hace que sea innecesario añadir comentarios al código del controlador.

El script del **modelo solamente se encarga del acceso a los datos** y puede ser reorganizado a tal efecto. Todos los parámetros que no dependen de la **capa de datos**(como por ejemplo los parámetros de la petición del usuario) se deben obtener a través del controlador y por tanto, no se puede acceder a ellos directamente desde el modelo. Las funciones del modelo se pueden reutilizar fácilmente en otros controladores.

## Separación en capas más allá del MVC

**El principio más importante de la arquitectura MVC es la separación del código del programa en tres capas,** dependiendo de su naturaleza. La lógica relacionada con los datos se incluye en el modelo, el código de ***la presentación*** en la vista y la lógica de la aplicación en el controlador.

La programación se puede simplificar si se utilizan otros patrones de diseño. De esta forma, las ***capas del modelo***, **la vista** y **el controlador** se pueden subdividir en más *capas*.

### Abstracción de la base de datos

***La capa del modelo*** se puede dividir en la ***capa de acceso a los datos*** y en la ***capa de abstracción de la base de datos***. De esta forma, las funciones que acceden a los datos no utilizan sentencias ni consultas que dependen de una base de datos, sino que utilizan otras funciones para realizar las consultas. Así, si se cambia de sistema gestor de bases de datos, solamente es necesario actualizar la *capa de abstracción de la base de datos*.

**La parte del modelo correspondiente a la abstracción de la base de datos: muestra una capa de acceso a datos específica para MySQL.**

```
<?php
   function crear_conexion($servidor, $usuario, $contrasena){
      return mysql_connect($servidor, $usuario, $contrasena);
   }
   function cerrar_conexion($conexion){
      mysql_close($conexion);
   }
   function consulta_base_de_datos($consulta, $base_datos, $conexion){
      mysql_select_db($base_datos, $conexion);
      return mysql_query($consulta, $conexion);
   }
   function obtener_resultados($resultado){
      return mysql_fetch_array($resultado, MYSQL_ASSOC);
   }
?>
```

**La parte del modelo correspondiente al acceso a los datos: muestra una capa sencilla de abstracción de la base de datos.**

```
<?php
   function getTodosLosArticulos(){
      // Conectar con la base de datos
      $conexion = crear_conexion('localhost', 'miusuario', 'micontrasena');
      // Ejecutar la consulta SQL
      $resultado = consulta_base_de_datos('SELECT fecha, titulo FROM articulo', 'blog_db',
$conexion);
      // Crear el array de elementos para la capa de la vista
      $articulos = array();
      while ($fila = obtener_resultados($resultado)){
         $articulos[] = $fila;
      }
      // Cerrar la conexión
      cerrar_conexion($conexion);
      return $articulos;
   }
?>
```

Como se puede comprobar, ***la capa de acceso a datos*** no contiene funciones dependientes de ningún sistema gestor de bases de datos, por lo que es independiente de la base de datos utilizada. Además, las funciones creadas en la capa de abstracción de la base de datos se pueden reutilizar en otras funciones del modelo que necesiten acceder a la base de datos.

**NOTA**

Estos últimos dos ejemplos no son completos, y todavía hace falta añadir algo de código para tener una completa abstracción de la base de datos (**abstraer el código SQL** mediante un **constructor de**

**consultas** independiente de la base de datos, añadir todas las funciones a una clase, etc.) El propósito de este artículo no es mostrar cómo se puede escribir todo ese código.

### *Los elementos de la vista*

La ***capa de la vista*** también puede aprovechar la **separación de código**. Las páginas web suelen contener elementos que se muestran de forma idéntica a lo largo de toda la aplicación: cabeceras de la página, el layout genérico, el pie de página y la navegación global. Normalmente sólo cambia el interior de la página. Por este motivo, la vista se separa en un ***layout*** y en una plantilla. Normalmente, el layout es global en toda la aplicación o al menos en un grupo de páginas. La plantilla sólo se encarga de visualizar las variables definidas en el controlador. Para que estos componentes interaccionen entre sí correctamente, es necesario añadir cierto código. Siguiendo estos principios, la parte de la vista del scrip inicial se puede separar en tres partes.

### La parte de la plantilla de la vista, en `miplantilla.php`

```
<h1>Listado de Artículos</h1>
<table>
   <tr><th>Fecha</th><th>Título</th></tr>
   <?php foreach ($articulos as $articulo): ?>
   <tr>
      <td><?php echo $articulo['fecha'] ?></td>
      <td><?php echo $articulo['titulo'] ?></td>
   </tr>
   <?php endforeach; ?>
</table>
```

### La parte de la lógica de la vista

```
<?php
   $titulo = 'Listado de Artículos';
   $contenido = include('miplantilla.php');
?>
```

### La parte del layout de la vista

```
<html>
   <head>
      <title><?php echo $titulo ?></title>
   </head>
   <body>
      <?php echo $contenido ?>
   </body>
</html>
```

### *Acciones y controlador frontal*

En el ejemplo anterior, **el controlador** no se encargaba de realizar muchas tareas, pero en las aplicaciones web reales el controlador suele tener mucho trabajo. Una parte importante de su trabajo es común a todos los controladores de la aplicación. Entre las tareas comunes se encuentran el manejo de las peticiones del usuario, el manejo de la seguridad, cargar la configuración de la aplicación y otras tareas similares. Por este motivo, el controlador normalmente se divide en un **controlador frontal**, que es único para cada aplicación, y las acciones, que incluyen el código específico del controlador de cada página.

Una de las principales ventajas de utilizar un ***controlador frontal*** es que **ofrece un punto de entrada único para toda la aplicación**. Así, en caso de que sea necesario impedir el acceso a la aplicación, solamente es necesario editar el script correspondiente al *controlador frontal*. Si la aplicación no dispone de controlador frontal, se debería modificar cada uno de los controladores.

### *Orientación a objetos*

Los ejemplos anteriores utilizan la ***programación procedimental***. Las posibilidades que ofrecen los lenguajes de programación modernos para trabajar con objetos permiten simplificar la programación, ya que los objetos pueden *encapsular la lógica*, *heredar métodos y atributos* entre

diferentes objetos y proporcionan una serie de convenciones claras sobre la forma de nombrar a los objetos.

La implementación de una arquitectura MVC en un lenguaje de programación que no está orientado a objetos puede encontrarse con problemas de *namespaces* y *código duplicado*, dificultando la lectura del código de la aplicación.

La orientación a objetos permite a los *desarrolladores* trabajar con *objetos de la vista*, ***objetos del controlador*** y ***clases del modelo***, transformando las funciones de los ejemplos anteriores en métodos. Se trata de un requisito obligatorio para las arquitecturas de tipo MVC.