Eugenia Bahit

POO y MVC en PHP

POO y MVC en PHP

El paradigma de la Programación Orientada a Objetos en PHP y el patrón de arquitectura de Software MVC

por
Eugenia Bahit
http://eugeniabahit.blogspot.com

AVISO LEGAL

Índice de contenidos

ANTES DE COMENZAR CON EL LIBRO

Antes de comenzar con la lectura puntual sobre POO y MVC en PHP, hago meritorio notar ciertas cuestiones que considero deben contemplarse a fin de lograr un mejor entendimiento de lo expuesto.

OBJETIVO DEL LIBRO

El objetivo de este libro, es dar una introducción general a conceptos básicos de la programación como lo son, el **paradigma de la programación orientada a objetos** y el **patrón arquitectónico MVC (modelo, vista, controlador)**, a la vez de intentar introducir estos conceptos, en la programación específica del lenguaje **PHP**.

REQUISITOS PREVIOS

Para poder entender este libro, los requisitos previos que debe tener el lector, no son muchos. Recomiendo que se tengan **conocimientos básicos del lenguaje PHP** así como la **facilidad para entender código HTML básico** (se utilizará HTML Transitional 4.0 en algunos ejemplos, evitando el uso de XHTML, a fin de que programadores sin experiencia en el diseño y maquetado de sitios Web, puedan comprender más fácilmente los ejemplos utilizados).

¿A QUIÉNES ESTÁ DIRIGIDO ESTE LIBRO?

Este libro está dirigido principalmente, a **programadores PHP** que quieran salir de la programación estructurada para insertarse en la programación orientada a objetos y, a **programadores de otros lenguajes** que quieran comenzar a insertarse en la POO en PHP y que cuenten **con mínimos conocimientos de este lenguaje**.

EL ¿POR QUÉ? DE ESTE LIBRO Y MI DEDICATORIA

Mi frase de cabecera es ***"el por qué de las cosas es relativo ya que no tiene forma de ser comprobado"***. Así que se me hace muy difícil explicar un "por qué". Solo puedo decir **"cómo"** fue que se me ocurrió escribirlo.

Perdí la cuenta de la cantidad de programadores con los que he trabajado. Han sido muchísimos en los últimos 15 años de mi vida. Me ha tocado trabajar con **excelentes programadores** y con unos pocos no tan buenos. Pero hay algo, que he notado en una gran parte de progrmadores, con los que tal vez, no me ha tocado trabajar en forma directa, pero sí, tener que continuar desarrollando algo ya empezado por ellos o modificarlo. Y ésto que he notado y me ha llamado mucho la atención, es que:

- Se usan elementos de la programación orientada a objetos, con la falsa creencia de que escribiendo una "clase" ya se está programando orientado a objetos
- Se desconoce por completo o no se logra entender del todo, la diferencia entre "estilo arquitectónico", "patrón arquitectónico" y "patrón de diseño", provocando en consecuencia, el desaprovechamiento o mal uso, de patrones arquitectónicos

como MVC desencadenando así, en una arquitectura de software deficiente

Las alternativas que me quedaban eran dos:

1. No hacer nada
2. Transmitir a otros programadores, todo lo que había aprendido, para que pudiesen implementarlo si ese fuese su deseo

Por lógica y sin dudarlo ni tener que pensarlo demasiado, elegí la segunda opción. Porque estoy segura que **el conocimiento debe ser transmitido** y **no se debe escatimar en su difusión**. Pues **no me interesa hacerme eco de la competitividad** que invade varios ámbitos de la vida. Por el contrario, **prefiero seguir el ejemplo de aquellos, mis colegas, que lo han brindado todo al saber general**. Y es por ello, que **este libro, lo dedico a todos ellos**, a los creadores de **MastrosDelWeb.com**[1] y **ForosDelWeb.com** y **en especial**, a mis **dos grandes colegas**, de quienes he aprendido gran parte de lo que hoy se:

- a la barcelonesa **Helena Heidenreich** (aka: **tunait**[2]), gran colaboradora del foro JavaScript de FDW
- y al madrileño **Miguel Ángel Álvarez**, fundador y director del portal de programación, **DesarrolloWeb.com**[3].

Contenido del libro

A fin de lograr cumplir con los objetivos propuestos, este libro ha sido diseñado con una estructura esquemática y seguiendo normas de estilo a fin de diferenciar su contenido.

Estructura del libro

Este libro se encuentra dividido en **4 capítulos** donde dos de ellos, introducen a conceptos básicos de la POO y el patrón MVC (en general) y los otros dos, colocan a la POO y al patrón MVC en el contexto del lenguaje PHP.

Entender el contenido diferenciado

Este libro no pretende ser un compendio de conceptos meramente teóricos. A fin de alcanzar un abordaje más preciso y fácilmente entendible, se ha divido el libro en **7 tipificaciones de contenido** diferenciadas:

1. **marco teórico** (contenido propio y citas de terceros)
2. **ejemplo práctico** (código fuente de ejemplo)
3. **metáforas** (comparación con ejemplos de la vida real – tomando esta técnica de un principio básico de Extreme Programming)
4. **sugerencias de estilo** (reglas de estilo sugeridas para la programación en PHP)

1 http://www.maestrosdelweb.com/

2 **tunait** nos ofrece una de las mejores colecciones de códigos **JavaScript**, en su sitio Web http://tunait.com/javascript/

3 http://www.desarrolloweb.com Portal de **desarrollo Web y programación**, de la comunidad hispanoparlante

5. **ejercicios de autoevaluación**
6. **programación real** (ejemplos con códigos fuente reales)
7. **preguntas a respuestas frecuentes**

Marco teórico (contenido propio)
Texto general del libro.

Marco teórico (cita de terceros)
Las citas presentan el siguiente formato:

> *"Esto es una cita textual de terceros. La fuente se encuentra al pie de página."*

Ejemplo práctico (código fuente)
El código fuente se encuentra con una tipografía *monoespaciada* y con la sintaxis del lenguaje coloreada[4].

```
class Usuario {
    # Propiedades
    public $nombre;

    # Métodos
    public function set_usuario() {
        $this->nombre = 'Juan';
    }
}
```

Metáforas (comparación con la vida real)
A fin de ejemplificar con facilidad un concepto netamente teórico, se darán ejemplos utilizando hechos de la vida cotideana, que puedan asemejarse al concepto que se esté explicando. Estas metáforas, se presentarán con un diseño como el siguiente:

Esto es una **metáfora**. 

Si no tengo el ícono, soy un **ejemplo** de la vida real, pero no soy una metáfora.

Sugerencias de estilo
A diferencia de otros lenguajes de programación como Python, por ejemplo, PHP no tiene una guía de prácticas recomendadas o sugerencias de estilo. Esto, hace que muchas veces, cada programador le de al código fuente un "toque" característico. Si bien es un rasgo de personalidad tal vez admirable, en la práctica, provoca que sea difícil modificar el código fuente escrito previamente por otro programador.

A fin de proponer una alternativa para solucionar este inconveniente y basándome en el principio de "propiedad colectiva" de **Extreme Programming** así como en la técnica de

4 Este libro ha sido editado utilizando **OpenOffice Writer 3.2** con el plug-in **COOoder** para el coloreado de sintaxis

establecimiento de pautas de estilo que hagan un código homogéneo, a lo largo del libro iré sugiriendo algunas **reglas de estilo** según surjan, cuyo finalidad tiende a:

1. hacer los códigos fuente más intuitivos;
2. escribir códigos fuente fácilmente legibles.

Las **Sugerencias de Estilo** presenta un formato similar al siguiente:

Reglas de estilo sugeridas

Esto es una regla de estilo sugerida en referencia al tema que se está tratando.

Ejercicios de autoevaluación

Al final de los capítulos II y IV, hay una breve serie de ejercicios, cuya finalidad, es la de asimilar los conocimientos adquiridos en los capítulos anteriores. Las soluciones a dichos ejercicios, se encuentran al final de cada uno de ellos.

Programación real

Tras los ejercicios presentados al finalizar los capítulos II y IV, se podrá encontrar código fuente de casos reales. Dichos códigos, pretenden dar al lector, una noción más práctica de los temas que se han tratado, aplicándola a la vida cotideana de un desarrollador. Los encontrarás fácilmente identificando el ícono de PHP como el que se muestra a la izquierda de este párrafo.

Respuestas a preguntas frecuentes

Aquí encontrarás todas las preguntas sobre los códigos fuente que se encuentren en la sección "programación real". Dicha sección, se encuentra identificada con el icono que se muestra a la derecha de este párrafo.

Hecha estas aclaraciones, ahora...

¡A programar!

CAPÍTULO I

Introducción a la Programación Orientada a Objetos (POO)

INTRODUCCIÓN A LA PROGRAMACIÓN ORIENTADA A OBJETOS (POO)

La POO es un **paradigma de programación** (o técnica de programación) que utiliza objetos e interacciones en el diseño de un sistema.

ELEMENTOS DE LA POO

La POO está compuesta por una serie de elementos que se detallan a continuación.

CLASE

Una clase es un **modelo** que se utiliza para crear objetos que comparten un mismo comportamiento, estado e identidad.

Metáfora

Persona es la metáfora de una clase (la abstracción de Juan, Pedro, Ana y María), cuyo comportamiento puede ser caminar, correr, estudiar, leer, etc. Puede estar en estado despierto, dormido, etc. Sus características (propiedades) pueden ser el color de ojos, color de pelo, su estado civil, etc.

```
class Persona {
    # Propiedades
    # Métodos
}
```

OBJETO

Es una **entidad** provista de métodos o mensajes a los cuales responde (comportamiento); atributos con valores concretos (estado); y propiedades (identidad).

```
$persona = new Persona();
/*
   El objeto, ahora, es $persona,
   que se ha creado siguiendo el modelo de la clase Persona
*/
```

MÉTODO

Es el **algoritmo** asociado a un objeto que indica la capacidad de lo que éste puede hacer.

```
function caminar() {
    #...
}
```

EVENTO Y MENSAJE

Un **evento** es un suceso en el sistema mientras que un **mensaje** es la comunicación del suceso dirigida al objeto.

PROPIEDADES Y ATRIBUTOS

Las propiedades y atributos, son **variables** que contienen datos asociados a un objeto.

```
$nombre = 'Juan';
$edad = '25 años';
$altura = '1,75 mts';
```

Características conceptuales de la POO

La POO debe guardar ciertas características que la identifican y diferencian de otros paradigmas de programación. Dichas características se describen a continuación.

Abstracción

Aislación de un elemento de su contexto. Define las características esenciales de un objeto.

Encapsulamiento

Reúne al mismo nivel de abstracción, a todos los elementos que puedan considerarse pertenecientes a una misma entidad.

Modularidad

Característica que permite dividir una aplicación en varias partes más pequeñas (denominadas módulos), independientes unas de otras.

Ocultación (aislamiento)

Los objetos están aislados del exterior, protegiendo a sus propiedades para no ser modificadas por aquellos que no tengan derecho a acceder a las mismas.

Polimorfismo

Es la capacidad que da a diferentes objetos, la posibilidad de contar con métodos, propiedades y atributos de igual nombre, sin que los de un objeto interfieran con el de otro.

Herencia

Es la relación existente entre dos o más clases, donde una es la principal (madre) y otras son secundarias y dependen (heredan) de ellas (clases "hijas"), donde a la vez, los objetos heredan las características de los objetos de los cuales heredan.

Recolección de basura

Es la técnica que consiste en destruir aquellos objetos cuando ya no son necesarios, liberándolos de la memoria.

CAPÍTULO II

Programación Orientada a Objetos en PHP 5

PROGRAMACIÓN ORIENTADA A OBJETOS EN PHP 5

En este capítulo veremos como aplicar los conceptos de la POO en el entorno del lenguaje PHP 5+.

CLASES Y OBJETOS EN PHP 5

DEFINICIÓN DE CLASES

Según el **Manual Oficial de PHP**, una **Clase** es:

> *[…] "una colección de variables y funciones que trabajan con estas variables. Las variables se definen utilizando **var** y las funciones utilizando **function**" [...]*[5]

Para definir una clase, el **Manual Oficial de PHP**, continúa diciendo:

> *[…] "La definición básica de clases comienza con la palabra clave class, seguido por un nombre de clase, continuado por un par de llaves que encierran las definiciones de las propiedades y métodos pertenecientes a la clase. El nombre de clase puede ser cualquier etiqueta válida que no sea una palabra reservada de PHP. Un nombre válido de clase comienza con una letra o un guión bajo, seguido de la cantidad de letras, números o guiones bajos que sea." […]*[6]

Veamos un ejemplo de **definición de clase**:

```
class NombreDeMiClase {
    #...
}
```

Reglas de Estilo sugeridas

Utilizar **CamelCase** para el nombre de las clases.

La **llave de apertura en la misma línea** que el nombre de la clase, permite una mejor legibilidad del código.

DECLARACIÓN DE CLASES ABSTRACTAS

Las clases abstractas son aquellas que **no necesitan ser instanciadas** pero sin embargo, **serán heredadas en algún momento**. Se definen anteponiendo la palabra clave ***abstract*** *a class*:

```
abstract class NombreDeMiClaseAbstracta {
    #...
}
```

Este tipo de clases, será la que contenga **métodos abstractos** (que veremos más adelante) y generalmente, su finalidad, es la de declarar clases "genéricas" que necesitan ser declaradas pero a las cuales, no se puede otorgar una definición precisa (de eso, se encargarán las clases que la hereden).

5 Fuente de la cita: http://php.net/manual/es/keyword.class.php

6 Fuente de la cita: http://www.php.net/manual/es/language.oop5.basic.php

HERENCIA DE CLASES

Los **objetos pueden heredar propiedades y métodos de otros objetos**. Para ello, PHP permite la "extensión" (herencia) de clases, cuya característica representa la relación existente entre diferentes objetos. Para definir una clase como extención de una clase "madre" se utiliza la palabra clave ***extends***.

```
class NombreDeMiClaseMadre {
    #...
}

class NombreDeMiClaseHija extends NombreDeMiClaseMadre {
    /* esta clase hereda todos los métodos y propiedades de
       la clase madre NombreDeMiClaseMadre
    */
}
```

DECLARACIÓN DE CLASES FINALES EN PHP

PHP 5 incorpora clases finales que **no pueden ser heredadas por otra**. Se definen anteponiendo la palabra clave ***final***.

```
final class NombreDeMiClaseFinal {
    #esta clase no podrá ser heredada
}
```

¿QUÉ TIPO DE CLASE DECLARAR?

Hasta aquí, han quedado en claro, cuatro tipos de clases diferentes: **Instanciables**, **astractas**, **heredadas** y **finales**. ¿Cómo saber qué tipo de clase declarar? Todo dependerá, de lo que necesitemos hacer. Este cuadro, puede servirnos como guía básica:

Necesito...	Instanciable	Abstracta	Heredada	Final
Crear una clase que pueda ser instanciada y/o heredada	X			
Crear una clase cuyo objeto guarda relación con los métodos y propiedades de otra clase			X	
Crear una clase que solo sirva de modelo para otra clase, sin que pueda ser instanciada		X		
Crear una clase que pueda instanciarse pero que no pueda ser heredada por ninguna otra clase				X

OBJETOS EN PHP 5

Una vez que las clases han sido declaradas, será necesario crear los objetos y utilizarlos, aunque hemos visto que algunas clases, como las clases abstractas son solo modelos para otras, y por lo tanto no necesitan instanciar al objeto.

INSTANCIAR UNA CLASE

Para instanciar una clase, solo es necesario utilizar la palabra clave ***new***. El objeto será creado, asignando esta instancia a una variable (la cual, adoptará la forma de objeto). Lógicamente, la clase debe haber sido declarada antes de ser instanciada, como se muestra a continuación:

```
# declaro la clase
class Persona {
    #...
}

# creo el objeto instanciando la clase
$persona = new Persona();
```

Reglas de Estilo sugeridas

Utilizar nombres de variables (objetos) descriptivos, siempre en letra minúscula, separando palabras por guiones bajos. Por ejemplo si el nombre de la clase es **NombreDeMiClase** como variable utilizar **$nombre_de_mi_clase**. Esto permitirá una mayor legibilidad del código.

PROPIEDADES EN PHP 5

Las propiedades representan ciertas características del objeto en sí mismo. Se definen anteponiendo la palabra clave ***var*** al nombre de la variable (propiedad):

```
class Persona {
    var $nombre;
    var $edad;
    var $genero;
}
```

Las propiedades pueden gozar de diferentes características, como por ejemplo, la visibilidad: pueden ser **públicas, privadas** o **protegidas**. Como veremos más adelante, la visiblidad de las propiedades, es aplicable también a la visibilidad de los métodos.

PROPIEDADES PÚBLICAS

Las propiedades públicas se definen anteponiendo la palabra clave ***public*** al nombre de la variable. Éstas, **pueden ser accedidas desde cualquier parte de la aplicación**, sin restricción.

```
class Persona {
    public $nombre;
    public $genero;
}
```

PROPIEDADES PRIVADAS

Las propiedades privadas se definen anteponiendo la palabra clave ***private*** al nombre de la variable. Éstas solo **pueden ser accedidas por la clase que las definió**.

```
class Persona {
    public $nombre;
    public $genero;
    private $edad;
}
```

PROPIEDADES PROTEGIDAS

Las propiedades protegidas **pueden ser accedidas por la propia clase que la definió, así como por las clases que la heredan**, pero no, desde otras partes de la aplicación. Éstas, se definen anteponiendo la palabra clave ***protected*** al nombre de la variable:

```php
class Persona {
    public $nombre;
    public $genero;
    private $edad;
    protected $pasaporte;
}
```

PROPIEDADES ESTÁTICAS

Las propiedades estáticas representan una característica de "variabilidad" de sus datos, de gran importancia en PHP 5. Una propiedad declarada como estática, **puede ser accedida sin necesidad de instanciar un objeto**. y su valor es estático (es decir, no puede variar ni ser modificado). Ésta, se define anteponiendo la palabra clave ***static*** al nombre de la variable:

```php
class PersonaAPositivo extends Persona {
    public static $tipo_sangre = 'A+';
}
```

ACCEDIENDO A LAS PROPIEDAD DE UN OBJETO

Para acceder a las propiedad de un objeto, existen varias maneras de hacerlo. Todas ellas, dependerán del ámbito desde el cual se las invoque así como de su condición y visibilidad.

ACCESO A VARIABLES DESDE EL ÁMBITO DE LA CLASE

Se accede a una propiedad **no estática** dentro de la clase, utilizando la pseudo-variable ***$this*** siendo esta pseudo-variable una referencia al objeto mismo:

```php
return $this->nombre;
```

Cuando la variable **es estática**, se accede a ella mediante el operador de resolución de ámbito, doble dos-puntos **::** anteponiendo la palabra clave ***self*** o ***parent*** según si trata de una variable de la misma clase o de otra de la cual se ha heredado, respectivamente:

```php
print self::$variable_estatica_de_esta_clase;
print parent::$variable_estatica_de_clase_madre;
```

ACCESO A VARIABLES DESDE EL EXTERIOR DE LA CLASE

Se accede a una propiedad **no estática** con la siguiente sintáxis: `$objeto->variable`

Nótese además, que este acceso dependerá de la visibilidad de la variable. Por lo tanto, solo variables públicas pueden ser accedidas desde cualquier ámbito fuera de la clase o clases heredadas.

```php
# creo el objeto instanciando la clase
$persona_a_positivo = new PersonaAPositivo();

# accedo a la variable NO estática
print $persona_a_positivo->nombre;
```

Para acceder a una propiedad pública y **estática** el objeto **no necesita ser instanciado**, permitiendo así, el acceso a dicha variable mediante la siguiente sintáxis:

`Clase::$variable_estática`

```
# accedo a la variable estática
print PersonaAPositivo::$tipo_sangre;
```

CONSTANTES DE CLASE

Otro tipo de "propiedad" de una clase, son las constantes, aquellas que **mantienen su valor de forma permanente y sin cambios**. A diferencia de las propiedades estáticas, **las constantes solo pueden tener una visibilidad pública**.

Puede declararse una constante de clase como cualquier constante normal en PHP 5. El acceso a constantes es exactamente igual que al de otras propiedades.

Reglas de Estilo sugeridas

Utilizar **NOMBRE_DE_CONSTANTE** en letra MAYÚSCULA, ayuda a diferenciar rápidamente constantes de variables, haciendo más legible el código.

```
const MI_CONSTANTE = 'Este es el valor estático de mi constante';
```

MÉTODOS EN PHP 5

Cabe recordar, para quienes vienen de la programación estructurada, que el método de una clase, es un algoritmo igual al de una función. La única diferencia entre método y función, es que **llamamos método a las funciones de una clase** (en la POO), mientras que llamamos funciones, a los algoritmos de la programación estructurada.

Reglas de Estilo sugeridas

Utilizar **nombres_de_funciones_descriptivos**, en letra minúscula, separando palabras por guiones bajos, ayuda a comprender mejor el código fuente haciéndolo más intuitivo y legible.

La forma de declarar un método es anteponiendo la palabra clave ***function*** al nombre del método, seguido por un par paréntesis de apertura y cierre y llaves que encierren el algoritmo:

```
# declaro la clase
class Persona {
    #propiedades

    #métodos
    function donar_sangre() {
        #...
    }
}
```

Al igual que cualquier otra función en PHP, los métodos recibirán los parámetros necesarios indicando aquellos requeridos, dentro de los paréntisis:

```
# declaro la clase
class Persona {
    #propiedades

    #métodos
    function donar_sangre($destinatario) {
        #...
    }
}
```

MÉTODOS PÚBLICOS, PRIVADOS, PROTEGIDOS Y ESTÁTICOS

Los métodos, al igual que las propiedades, pueden ser **públicos, privados, protegidos** o **estáticos**. La forma de declarar su visibilidad tanto como las características de ésta, es exactamente la misma que para las propiedades.

```
static function a() { }
protected function b() { }
private function c() { }
# etc...
```

MÉTODOS ABSTRACTOS

A diferencia de las propiedades, los métodos, pueden ser **abstractos** como sucede con las clases.

El **Manual Oficial de PHP**, se refiere a los métodos abstractos, describiéndolos de la siguiente forma:

> *[…] "Los métodos definidos como abstractos simplemente declaran la estructura del método, pero no pueden definir la implementación. Cuando se hereda de una clase abstracta, todos los métodos definidos como abstract en la definición de la clase parent deben ser redefinidos en la clase child; adicionalmente, estos métodos deben ser definidos con la misma visibilidad (o con una menos restrictiva). Por ejemplo, si el método abstracto está definido como protected, la implementación de la función puede ser redefinida como protected o public, pero nunca como private." […]*[7]

Para entender mejor los métodos abstractos, podríamos decir que a grandes rasgos, los **métodos abstractos son aquellos que se declaran inicialmente en una clase abstracta, sin especificar el algoritmo que implementarán**, es decir, que solo son declarados pero no contienen un "código" que especifique qué harán y cómo lo harán.

Tal vez, te preguntes **¿Cuál es la utilidad de definir métodos abstractos y clases abstractas?** Para responder a esta pregunta, voy enfocarme en un caso de la vida real, en el cual estuve trabajando hace poco tiempo.

Ejemplo

Se trataba de hacer un sistema de gestión informática, para las farmacias de los Hospitales del Gobierno de la Ciudad de Buenos Aires. Un punto fundamental, era pensar en los insumos farmacéuticos como "un todo abstracto". ¿Por qué? Fundamentalmente, porque si bien existen insumos farmacéuticos de todo tipo y especie, cada uno de ellos, comparte características comunes, que por sí solas no pueden definirse con precisión. Por ejemplo, todos los insumos

7 Fuente de la cita: http://www.php.net/manual/es/language.oop5.abstract.php

farmacéuticos requieren de un tipo de conservación especial. Algunos requieren refrigeración a determinada temperatura que solo puede ser satisfecha conservándolos en una heladera; otros requieren conservarse en un ambiente seco; otros, no pueden tomar contacto con el exterior, a fin de conservar su capacidad estéril; etc. ¿Podía definirse con exactitud una clase Insumo? La respuesta a esa pregunta, es justamente su pregunta retórica ¿irías a la farmacia a pedirle al farmacéutico "deme un insumo de 500 mg"? Insumo, representa la entidad "abstracta" y para eso, sirven las clases abstractas. Con ellas declaramos aquellos "objetos" que no pueden ser definidos con presición pero aseguramos allí, todas aquellas características que dichos objetos, guardarán entre sí. Declarar un método conservar_insumo() como abstracto, serviría para luego definir con exactitud, en una clase heredada, el algoritmo exacto que determinado insumo necesitaría para procesar su conservación. Es así entonces, que una clase InsumoRefrigerado heredaría de Insumo, y redefiniría el método conservar_insumo() indicando un algoritmo que solicitara la temperatura a la cual debía conservarse en heladera, etc.

MÉTODOS MÁGICOS EN PHP 5

PHP 5, nos trae una gran cantidad de auto-denominados **"métodos mágicos"**. Estos métodos, otorgan una funcionalidad pre-definida por PHP, que pueden aportar valor a nuestras clases y ahorrarnos grandes cantidades de código. Lo que muchos programadores consideramos, ayuda a convertir a PHP en un lenguaje orientado a objetos, cada vez más robusto.

Entre los métodos mágicos, podemos encontrar los siguientes:

El Método Mágico `__construct()`

El método `__construct()` es aquel que será invocado de manera automática, al instanciar un objeto. Su función es la de ejecutar cualquier inicialización que el objeto necesite antes de ser utilizado.

```
# declaro la clase
class Producto {
    #defino algunas propiedades
    public $nombre;
    public $precio;
    protected $estado;

    #defino el método set_estado_producto()
    protected function set_estado_producto($estado) {
        $this->estado = $estado;
    }

    # constructor de la clase
    function __construct() {
        $this->set_estado_producto('en uso');
    }
}
```

En el ejemplo anterior, el constructor de la clase se encarga de definir el estado del producto como "en uso", antes de que el objeto (Producto) comience a utilizarse. Si se agregaran otros métodos, éstos, podrán hacer referencia al estado del producto, para determinar si ejecutar o no determinada función. Por ejemplo, no podría mostrarse a la venta un producto "en uso por el sistema", ya que a éste, se le podría estar modificando el precio.

El método mágico `__destruct()`

El método `__destruct()` es el encargado de liberar de la memoria, al objeto cuando ya no es referenciado. Se puede aprovechar este método, para realizar otras tareas que se

estimen necesarias al momento de destruir un objeto.

```
# declaro la clase
class Producto {
    #defino algunas propiedades
    public $nombre;
    public $precio;
    protected $estado;

    #defino el método set_estado_producto()
    protected function set_estado_producto($estado) {
        $this->estado = $estado;
    }

    # constructor de la clase
    function __construct() {
        $this->set_estado_producto('en uso');
    }

    # destructor de la clase
    function __destruct() {
        $this->set_estado_producto('liberado');
        print 'El objeto ha sido destruido';
    }
}
```

Otros métodos mágicos

PHP nos ofrece otros métodos mágicos tales como `__call`, `__callStatic`, `__get`, `__set`, `__isset`, `__unset`, `__sleep`, `__wakeup`, `__toString`, `__invoke`, `__set_state` y `__clone`.

Puede verse una descripción y ejemplo de su uso, en el sitio Web oficial de PHP:

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

Ejercicios Prácticos

Para practicar lo que hemos visto hasta ahora, les propongo hacer algunos ejercicios, a fin de asimilar los conocimientos de los capítulos I y II.

Ejercicio Nº1: Sobre la programación orientada a objetos

1.1) ¿Qué es la Programación Orientada a Objetos?

a) Un patrón de diseño de software

b) Un paradigma de programación

c) La única forma en la que se puede programar en PHP

d) Ninguna de las anteriores

1.2) ¿Cuál de las siguientes opciones, responde mejor a los elementos que forman parte de la POO?

a) Clases, objetos, métodos, atributos y propiedades

b) Atributos, eventos y funciones

c) Métodos, inmutabilidad, abstracción, funciones y prototipos

e) Variables, constantes y funciones

1.3) ¿Cuál de las siguientes afirmaciones es FALSA con respecto a las características de la POO?

a) La abstracción y el polimorfismo son característica esenciales de la programación orientada a objetos

b) Encapsulamiento es sinónimo de aislamiento

c) En la POO, la modularidad, es la característica que permite dividir una aplicación, en partes más pequeñas, con independencia unas de las otras

Ejercicio Nº2: Sobre la POO en PHP

2.1) Dado el siguiente código:

```
<?php

final class ItemProducto {
    #...
}

class Producto extends ItemProducto {
    #...
}
?>
```

¿Qué crees que fallaría al intentar ejecutarlo?

a) ItemProducto fallará ya que está heredando de otra clase

b) No fallaría nada

c) Producto no puede heredar de ItemProducto ya que esta clase ha sido declarada como clase final

2.2) ¿Cuál crees que será la salida del siguiente código?

```
<?php

class Cliente {
    static public $nombre_completo = 'Cliente desconocido';
    protected $id = 1001;
}

print Cliente::$nombre_completo;
print Cliente::$id;

?>
```

a) Cliente desconocido 1001

b) Cliente desconocido

c) Se imprimiría Cliente desconocido pero fallaría luego el acceso a $id ya que es una propiedad protegida

2.3) ¿Cuál crees que será la salida del siguiente código?

```
<?php

class Cliente {
    static public $nombre_completo = 'Cliente desconocido';
```

```
    protected $id = 1001;

    function __construct() {
        self::$nombre = 'Juan Pérez';
    }
}

print Cliente::$nombre_completo;

?>
```

a) Cliente desconocido

b) Juan Pérez

Explica porqué has elegido esa respuesta:

2.4) Teniendo en cuenta el siguiente código:

```
<?php

class Cliente {
    static public $nombre_completo = 'Cliente desconocido';
    protected $id = 1001;
}
?>
```

¿Cuál de las siguientes opciones, estimas que sería la apropiada para imprimir en pantalla, la propiedad "nombre_completo"

a) `print Cliente::nombre_completo;`

b) `print Cliente::$nombre_completo;`

c)
```
$cliente = new Cliente();
print $cliente->nombre_completo;
```

d)
```
$cliente = new Cliente();
print $cliente->$nombre_completo;
```

Soluciones a los ejercicios 1 y 2

Ejercicio nº1

Pregunta 1.1: respuesta **b**

Pregunta 1.2: respuesta **a**

Pregunta 1.3: respuesta **b**

Ejercicio nº2

Pregunta 2.1: respuesta **c**

Pregunta 2.2: respuesta **c**

Pregunta 2.3: respuesta **a**

porque a las propiedades estáticas no se les puede modificar su valor

Pregunta 2.4: respuesta **b**

PROGRAMACIÓN REAL ORIENTADA A OBJETOS EN PHP

Veremos ahora, un ejemplo basado en la realidad de un programador.

En este caso, se trata de un **ABM de usuarios**.

***Nótese** que los métodos respectivos han sido resumidos no encontrándose en éstos, algoritmos de validación de datos. Se ejemplifica todo aquello que es relevante en la POO.*

ARCHIVOS FUENTE DEL ABM DE USUARIOS

ARCHIVO **db_abstract_model.php**

```
<?php
abstract class DBAbstractModel {

    private static $db_host = 'localhost';
    private static $db_user = 'usuario';
    private static $db_pass = 'contraseña';
    protected $db_name = 'mydb';
    protected $query;
    protected $rows = array();
    private $conn;

    # métodos abstractos para ABM de clases que hereden
    abstract protected function get();
    abstract protected function set();
    abstract protected function edit();
    abstract protected function delete();

     # los siguientes métodos pueden definirse con exactitud
     # y no son abstractos
     # Conectar a la base de datos
     private function open_connection() {
         $this->conn = new mysqli(self::$db_host, self::$db_user,
                                  self::$db_pass, $this->db_name);
     }

     # Desconectar la base de datos
     private function close_connection() {
           $this->conn->close();
     }

     # Ejecutar un query simple del tipo INSERT, DELETE, UPDATE
     protected function execute_single_query() {
         $this->open_connection();
         $this->conn->query($this->query);
         $this->close_connection();
     }

     # Traer resultados de una consulta en un Array
     protected function get_results_from_query() {
         $this->open_connection();
        $result = $this->conn->query($this->query);
        while ($this->rows[] = $result->fetch_assoc());
        $result->close();
        $this->close_connection();
```

```
            array_pop($this->rows);
        }
    }
    ?>
```

ARCHIVO **usuarios_model.php**

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

class Usuario extends DBAbstractModel {

    public $nombre;
    public $apellido;
    public $email;
    private $clave;
    protected $id;

    function __construct() {
        $this->db_name = 'book_example';
    }

    public function get($user_email='') {

        if($user_email != ''):
            $this->query = "
                SELECT      id, nombre, apellido, email, clave
                FROM        usuarios
                WHERE       email = '$user_email'
            ";
            $this->get_results_from_query();
        endif;

        if(count($this->rows) == 1):
            foreach ($this->rows[0] as $propiedad=>$valor):
                $this->$propiedad = $valor;
            endforeach;
        endif;
    }

    public function set($user_data=array()) {
        if(array_key_exists('email', $user_data)):
            $this->get($user_data['email']);
            if($user_data['email'] != $this->email):
                foreach ($user_data as $campo=>$valor):
                    $$campo = $valor;
                endforeach;
                $this->query = "
                    INSERT INTO     usuarios
                    (nombre, apellido, email, clave)
                    VALUES
                    ('$nombre', '$apellido', '$email', '$clave')
                ";
                $this->execute_single_query();
            endif;
        endif;
    }

    public function edit($user_data=array()) {
        foreach ($user_data as $campo=>$valor):
            $$campo = $valor;
```

```
            endforeach;
            $this->query = "
                UPDATE      usuarios
                SET         nombre='$nombre',
                            apellido='$apellido',
                            clave='$clave'
                WHERE       email = '$email'
            ";
            $this->execute_single_query();
        }

        public function delete($user_email='') {
            $this->query = "
                DELETE FROM     usuarios
                WHERE           email = '$user_email'
            ";
            $this->execute_single_query();
        }

        function __destruct() {
            unset($this);
        }

    }
    ?>
```

ARCHIVO **abm_example.php**

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

# Traer los datos de un usuario
$usuario1 = new Usuario();
$usuario1->get('user@email.com');
print $usuario1->nombre . ' ' . $usuario1->apellido . ' existe<br>';

# Crear un nuevo usuario
$new_user_data = array(
    'nombre'=>'Alberto',
    'apellido'=>'Jacinto',
    'email'=>'albert2000@mail.com',
    'clave'=>'jacaranda'
);
$usuario2 = new Usuario();
$usuario2->set($new_user_data);
$usuario2->get($new_user_data['email']);
print $usuario2->nombre . ' ' . $usuario2->apellido . ' ha sido creado<br>';

# Editar los datos de un usuario existente
$edit_user_data = array(
    'nombre'=>'Gabriel',
    'apellido'=>'Lopez',
    'email'=>'marconi@mail.com',
    'clave'=>'69274'
);
$usuario3 = new Usuario();
$usuario3->edit($edit_user_data);
$usuario3->get($edit_user_data['email']);
print $usuario3->nombre . ' ' . $usuario3->apellido . ' ha sido editado<br>';

# Eliminar un usuario
```

```php
$usuario4 = new Usuario();
$usuario4->get('lei@mail.com');
$usuario4->delete('lei@mail.com');
print $usuario4->nombre . ' ' . $usuario4->apellido . ' ha sido eliminado';
?>
```

EXPLICANDO EL CÓDIGO EN EL CONTEXTO DE LA POO: LA MEJOR FORMA DE APRENDER Y COMPRENDER

En principio tenemos **3 archivos**:

- 2 archivos son "clases" de dos modelos de objetos.
- 1 archivo, es el que realiza las instancias creando la cantidad de objetos necesaria.

RESPUESTAS A PREGUNTAS FRECUENTES SOBRE EL CÓDIGO

1. Respuestas a preguntas frecuentes de la clase **DBAbstractModel**

1.1 ¿Por qué la clase está definida como abstracta?

Una base de datos, puede tener varios métodos, como **insertar datos, editar datos, eliminar datos** o simplemente **consultar datos**. El **algoritmo** de cada uno de esos métodos **no puede definirse con exactitud** ¿por qué? Porque cada dato que se inserte, modificque, elimine o consulte, variará en infinidad de aspectos: desde los tipos de datos hasta las tablas y los campos a los que se deba acceder, etc.

Basados en el **principio de modularidad** de la POO, es necesario tener en cuenta, que HOY, necesito un ABM de usuarios, pero mañana, este requisito puede ampliarse y, debo dejar todo preparado para que el sistema pueda ser **escalable** y **modular** (otros módulos pueden ser requeridos en el futuro).

Si solo contemplara los métodos de inserción, edición, consulta y eliminación de usuarios en esta clase, estaría cometiendo dos **errores imperdonable**:

1. No estaría respetando el principio de modularidad de la POO
2. Los métodos mencionados ¿no son a caso métodos de un nuevo objeto llamado Usuario?

Por otro lado, **por cuestiones de seguridad**, se hacía necesario **no permitir instanciar esta clase** desde ningún lugar de la aplicación y controlar con mucha precaución, el acceso a sus métodos y propiedades, ya que una base de datos, es el "alma" de toda aplicación.

1.2 ¿Para qué declarar propiedades estáticas?

En principio, tanto el **host como el usuario y contraseña de la base de datos**, en esta aplicación, son únicos. No varían. De allí, que como característica se les asignó ser **estáticos**, a fin de, por cuestiones de seguridad, no puedan ser modificados dinámicamente (tengamos en cuenta que **una clase abstracta debe ser heredada sí o sí para ser utilizada**).

1.3 ¿Por qué propiedades estáticas y no constantes de clase?

Porque a las constantes, aunque lógicamente guardan en sí mismas la condición de "estáticas" en el sentido de que "no pueden ser modificadas", pueden ser **accedidas** desde cualquier clase que herede de la clase que las declare. Pues entonces, la pregunta es **¿Para qué querría una clase "hija" conocer el host, usuario y contraseña de una base de datos?** Es una cuestión nétamente de seguridad, ya que a una constante no se la puede definir como **privada** y si se la quiere ocultar a las clases "hijas", no quedará otra alternativa que declararlas como propiedades estáticas.

1.4 ¿Por qué propiedades estáticas y a la vez privadas?

(ver respuestas anteriores)

1.5 ¿Con qué sentido declarar las propiedades como protegidas?

Es simple. Algunos módulos pueden necesitar utilizar una base de datos diferente. Todos los módulos, necesitarán *queries* personalizados para sus consultas. Y lógicamente, todos necesitarán acceder y conocer los resultados de los datos arrojados por una consulta. Es decir, que tanto la propiedad **db_name** como **query y rows**, deben tener permisos para ser leídas y modificadas, pero ¡Ojo! Solo por las clases hijas de DBAbstractModel ¿Qué sentido tendría permitir a un programador acceder a estas propiedades si puede ser la clase hija, quien las controle?

1.6 ¿Por qué los métodos open y close_connection son privados?

Pues porque la clase es la única con permiso para abrir y cerrar conexiones a la base de datos. Es una cuestión de control y seguridad. Si necesitas insertar datos, consultarlos o hacer con ellos otras actividades, tienes métodos protegidos pero no privados, que te permiten hacerlo.

1.7 ¿Por qué los métodos de consulta y ejecución son protegidos y no públicos?

Consultar una base de datos, modificar o eliminar sus datos, así como insertar nuevos, no puede ser una tarea que se deje librada al azar y al libre acceso de cualquiera. Es necesario "proteger" los datos a fin de resguardar su integridad y, la forma de hacerlo, es "proteger" los métodos que se encuentran al servicio de dichos datos.

1.8 ¿Por qué hay métodos declarados como abstractos y además protegidos?

Esta pregunta, en gran parte, la responde la respuesta de la pregunta 1.1. El hecho de que se definan como abstractos, es porque necesariamente DEBEN ser métodos comunes a toda clase que los hereden. Pero solo la clase hija, podrá definirlos (re-definirlos técnicamente hablando) ya que tal cual se explicó en la pregunta 1.1., solo ellas conocen los requerimientos específicos.

Se declaran además como protegidos, por una cuestión de seguridad ("por las dudas" para que solo puedan ser vistos por clases heredadas). Esto, da lugar a las clases hijas, a que modifiquen su visibilidad de acuerdo al criterio de cada una de ellas.

2. Respuestas a preguntas sobre la clase **Usuario**

2.1 ¿Por qué la clase Usuario es una extensión de DBAbstractModel?

La clase usuario TIENE necesariamente que heredar de DBAbstractModel ya que DEBE utilizar sus métodos (propios y definidos para acceder a la base de datos) y redefinir obligadamente aquellos declarados como abstractos, a fin de continuar un modelo de administración de los ABM (altas, bajas y modificaciones).

2.2 ¿Con qué fin nombre, apellido e e-mail son propiedades públicas mientras que clave es privada y id, protegida?

Pensemos esto como en la vida real: tu nombre, apellido e e-mail, puede ser necesario para cientos de "trámites" y acciones de tu vida diaria y no andas por la vida negándolos a todo quien te los pida. Podríamos decir, que **no tiene nada de malo que facilites públicamente estos datos, a quien decidas**.

Ahora bien **¿le das tu contraseña a cualquiera?** Si lo haces, deja de hacerlo. **La contraseña es un dato que debe ser privado.** Nadie puede tener acceso a él. Es una cuestión de seguirdad.

El número de **ID, sería como tu número de documento o pasaporte**. Es un dato que debes mantener **protegido** pero que en algunos casos, te será requerido y necesitarás darlo obligadamente. Por ejemplo, no podrías negarte a mostrar tu pasaporte en migraciones. Sin embargo, DEBES negarte a decirle el PIN de tu tarjeta de crédito a cualquier persona.

2.3 ¿Por qué se utiliza el método __construct() para modificar el nombre de la base de datos? ¿Por qué no se encarga la clase madre de modificar ese valor?

El nombre de la base de datos, es algo que variará de acuerdo al **módulo** que lo esté trabajando. Seguramente, de existir en el futuro una clase llamada Producto, ésta, sobreescribirá el valor de `db_name`. Modificarlo automáticamente utilizando un cosntructor, nos asegura instanciar el objeto con un valor de propiedad adecuado.

2.4 ¿Por qué se utiliza require_once y no include_once?

La diferencia entre include y require es que require, si no puede incluir el archivo al que se está importando, frena la ejecución del script, siendo que `include`, solo arroja un error, pero continúa la ejecución. Al tratarse de una clase que "requiere" sí o sí, de su clase madre, no tendría sentido permitir que se continúe ejecutando, si no ha podido cargarse el archivo.

3. Respuestas a preguntas sobre el **archivo de instancias**

3.1 ¿Por qué el archivo tiene cuatro instancias al mismo objeto?

Es necesario hacer notar que **si bien los objetos comparten propiedades y métodos en común, no todos son idénticos**. Es decir, todos los seres humanos tenemos ciertas características en común y realizamos acciones similares. Pero cada uno de nosotros, tiene una **única identidad**. Por eso, no puede decirse que se instancia cuadro veces el mismo objeto, porque en realidad, son cuatro objetos diferentes donde cada uno de ellos, tiene una identidad propia.

CAPÍTULO III

Introducción al patrón arquitectónico MVC

INTRODUCCIÓN AL PATRÓN ARQUITECTÓNICO MVC

MVC, son las siglas de **modelo-vista-controlador** (o en inglés, model-view-controller), que es uno de los tantos **patrones de arquitectura de software**.

Antes de abordar de lleno este patrón, vamos a intentar hacer una introducción a la arquitectura de software, desmembrándola de lo general hacia lo particular, para al fin llegar al detalle, procurando entender exactamente de que se trata, en el contexto adecuado.

Probablemente, **este capítulo sea el más complejo** (y mucho más extenso en relación al Capítulo I). Pero si quieres poder aplicar de forma decuada el patrón MVC, debes hacer el esfuerzo de seguirlo con especial interés y actitud "entuciasta".

INTRODUCCIÓN A LA ARQUITECTURA DE SOFTWARE

¿QUÉ ES LA ARQUITECTURA DE SOFTWARE?

Es necesario aclarar, que no existe una definición única, exacta, abarcativa e inequívoca de "arquitectura de software". La bibliografía sobre el tema es tan extensa como la cantidad de definiciones que en ella se puede encontrar. Por lo tanto trataré, no de definir la arquitectura de software, sino más bien, de introducir a un concepto simple y sencillo que permita comprender el punto de vista desde el cual, este libro abarca a la arquitectura de software pero, sin ánimo de que ello represente "una definición más".

*A grandes rasgos, puede decirse que "**la Arquitectura de Software es la forma en la que se organizan los componentes de un sistema, interactúan y se relacionan entre sí y con el contexto, aplicando normas y principios de diseño y calidad, que fortalezcan y fomenten la usabilidad a la vez que dejan preparado el sistema, para su propia evolución".***

Tal vez estés pensando *"...al fin y al cabo, me haz dado una definición más..."*. La respuesta es sí. Probablemente no pude contenerme de hacerlo – no lo niego -. Pero, **no debes tomar lo anterior como una definición**, sino como **la forma en la que en este libro, se abarca la arquitectura de software**.

TENDENCIAS DE LA ARQUITECTURA DE SOFTWARE

Si bien no puede decirse o mejor dicho, no es muy académico decirlo, voy a tomarme el atrevimiento de mencionar **solo dos tendencias arquitectónicas**, claramente diferenciadas entre sí:

- La **Arquitectura de Software Orientada a Objetos** (como "ingeniría" de sistemas)
- La **Arquitectura Estructurada** (como "desarrollo" de una aplicación)

Como podemos ver, en este libro, es a la primera a la cual nos enfocamos.

A fin de satisfacer la inquietud de curiosos (y de paso, no ocultar que existen), voy a mencionar la existencia de otras dos corrientes: la **arquitectura basada en patrones** y la **arquitectura basada en procesos y metodologías**.

Personalmente, **no las considero como cuatro corrientes** ¿Por qué? Porque independientemente de cual te agrade más, si la AS orientada o objetos o la AS estructural, implican necesariamente dos corrientes que deben aplicarse de forma optativa (o utilizas una o la otra pero no las dos simultáneamente), mientras que la AS basada en patrones y la AS basada en procesos, pueden utilizarse como complemento de las dos primeras. Pero esto, **es mera filosofía propia**, con la cual, no he encontrado demasiado consenso. Pues no los aburriré con ello e iremos a lo práctico.

Características de la Arquitectura de Software: Atributos de Calidad

La **Calidad del Software** puede definirse como **los atributos implícitamente requeridos en un sistema que deben ser satisfechos**. Cuando estos atributos son satisfechos, puede decirse (aunque en forma objetable), que la calidad del software es satisfactoria.

Estos atributos, se gestan desde la arquitectura de software que se emplea, ya sea cumpliendo con aquellos requeridos durante la ejecución del software, como con aquellos que forman parte del proceso de desarrollo de éste.

Atributos de calidad que pueden observarse **durante la ejecución** del software

1. **Disponibilidad** de uso
2. **Confidencialidad**, puesto que se debe evitar el acceso no autorizado al sistema
3. Cumplimiento de la **Funcionalidad** requerida
4. **Desempeño** del sistema con respecto a factores tales como la capacidad de respuesta
5. **Confiabilidad** dada por la constancia operativa y permanente del sistema
6. **Seguridad externa** evitando la pérdida de información debido a errores del sistema
7. **Seguridad interna** siendo capaz de impedir ataques, usos no autorizados, etc.

Atributos de calidad inherentes al **proceso de desarrollo** del software

8. Capacidad de **Configurabilidad** que el sistema otorga al usuario a fin de realizar ciertos cambios
9. **Integrabilidad** de los módulos independientes del sistema
10. **Integridad** de la información asociada
11. Capacidad de **Interoperar** con otros sistemas (interoperabilidad)
12. Capacidad de permitir ser **Modificable** a futuro (modificabilidad)
13. Ser fácilmente **Mantenible** (mantenibilidad)
14. Capacidad de **Portabilidad**, es decir que pueda ser ejecutado en diversos ambientes tanto de software como de hardware
15. Tener una estructura que facilite la **Reusabilidad** de la misma en futuros sistemas
16. Mantener un diseño arquitectónico **Escalable** que permita su ampliación (escalabilidad)
17. Facilidad de ser **Sometido a Pruebas** que aseguren que el sistema falla cuando es lo que se espera (testeabilidad)

DE LO GENERAL A LO PARTICULAR: DEL ESTILO ARQUITECTÓNICO AL PATRÓN DE DISEÑO

Existe una diferencia entre **Estilo Arquitectónico, Patrón Arquitectónico y Patrón de Diseño**, que debe marcarse a fin de evitar las grandes confusiones que inevitablemente, concluyen en el mal entendimiento y en los resultados poco satisfactorios. Éstos, son los que en definitiva, aportarán "calidad" al sistema resultante. En lo sucesivo, trataremos de establecer la diferencia entre estos tres conceptos, viendo como los mismos, se relacionan entre sí, formando parte de un todo: la arquitectura de software.

RELACIÓN Y DIFERENCIA

Estilo Arquitectónico, Patrón Arquitectónico y Patrón de Diseño, representan, de lo general a lo particular, los niveles de abstracción que componen la Arquitectura de Software. En este sentido, puede decirse que:

- **El Estilo Arquitectónico** es el encargado de:
  - Describir la *estructura general* de un sistema, *independientemente de otros estilos*
  - Definir los componentes del sistema, su relación e interactividad
  - Ejemplos: *flujo de datos, llamada y retorno, etc.*
- **El Patrón Arquitectónico** es el nivel en el cual la arquitectura de software:
  - Define la *estructura básica* de un sistema, pudiendo estar *relacionado con otros patrones*
  - Representa una *plantilla de construcción* que provee un conjunto de *subsistemas* aportando las *normas para su organización*
  - *Ejemplos: Capas, MVC, Tuberías y Filtros, Pizarra, etc.*
- **El Patrón de Diseño** es el tercer nivel de abstracción de la arquitectura de software, cuya finalidad es la de *precisar en detalle los subsistemas y componentes* de la aplicación
  - Ejemplos: *Proxy, Command, Factory, etc..*

EL PATRÓN ARQUITECTÓNICO MODELO-VISTA-CONTROLADOR (MVC)

Habiendo dejado en claro de qué hablamos exactamente cuando nos referimos a "patrón arquitectónico", estamos en condiciones de ahondar en los detalles del patrón MVC.

ACLARCIONES PREVIAS

Antes de caer inmersos en el conocimiento sobre el patrón MVC, quiero dejar en claro que en este libro, no hará referencia a ningún framework.

El **objetivo de *POO y MVC en PHP* no es el de formar a programadores en el uso de frameworks**, sino que el mismo, parte de una clara pregunta ¿a caso los frameworks no han sido desarrollados por programadores? Tenemos la alternativa de utilizar frameworks para ahorrar tiempo de programación o, **si realmente nos apasiona programar, adquirimos los conocimientos necesarios, tenemos la capacidad y nos animamos, podemos crear nuestros propios frameworks.** De hecho, **no se trata de reinventar la rueda, sino de crear una, adaptada a nuestras necesidades**. Y si lo que

necesitamos es una rueda para acoplar a un mueble que acabamos de crear ¿cuál sería el sentido de intentar modificar una rueda de camión si tenemos todas las herramientas necesarias para ser "creadores" y no "modificadores"?

Claro que esto, depende de la pasión, gusto, capacidad y por qué no, de la ambición de cada uno. El tiempo, es discutible. Pues puede demandarte más tiempo modificar algo que ya existe, que crearlo. Si quieres reparar un mueble te llevará más tiempo repararlo que ir a comprar uno nuevo. Y si eres ebanista, seguramente te llevará menos tiempo hacer un nuevo mueble que ir a comprarlo. Pues sabes exactamente lo que necesitas y como hacerlo y eso, hará que ahorres el tiempo de recorrer todas las mueblerías y termines comprando "el mueble que más se asemeje al que quieres". El mueble, será similar, pero no exactamente igual al que tu imaginaste. De todas formas, la alternativa de modificar, siempre la tienes... al igual que también tienes la de crear. Todo dependerá del criterio de elección que apliques y sea cual sea, tu elección será la correcta, justamente porque habrás "elegido" y eso, es lo más importante.

¿QUÉ ES EL PATRÓN MVC?

El patrón MVC es un **patrón de arquitectura de software encargado de separar la lógica de negocio de la interfaz del usuario** y es el más utilizado en aplicaciones Web, ya que facilita la funcionalidad, mantenibilidad y escalabilidad del sistema, de forma simple y sencilla, a la vez que **permite** *"no mezclar lenguajes de programación en el mismo código"*.

MVC divide las aplicaciones en tres niveles de abstracción:

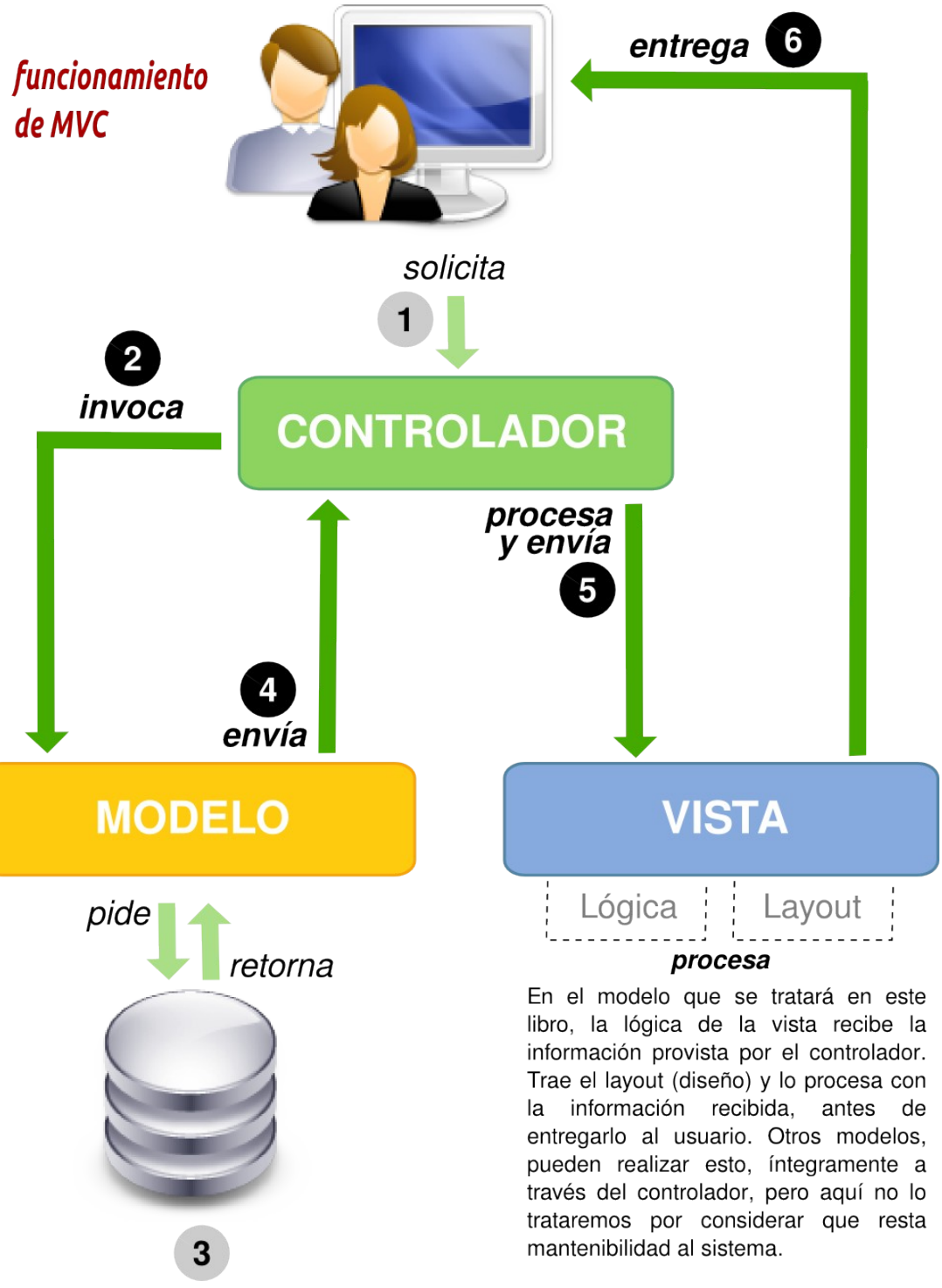
- **Modelo:** representa la lógica de negocios. Es el encargado de accesar de forma directa a los datos actuando como "intermediario" con la base de datos. Lo que en nuestro ejemplo de programación orientada a objetos, serían las clases DBAbstractModel y Usuario.
- **Vista:** es la encargada de mostrar la información al usuario de forma gráfica y "humanamente legible".
- **Controlador:** es el intermediario entre la vista y el modelo. Es quien controla las interacciones del usuario solicitando los datos al modelo y entregándolos a la vista para que ésta, lo presente al usuario, de forma "humanamente legible".

¿CÓMO FUNCIONA EL PATRÓN MVC?

El **funcionamiento básico del patrón MVC**, puede resumirse en:

- El **usuario realiza una petición**
- El **controlador captura el evento** (puede hacerlo mediante un manejador de eventos – *handler* -, por ejemplo)
- Hace la **llamada al modelo**/modelos correspondientes (por ejemplo, mediante una llamada de retorno – *callback* -) efectuando las modificaciones pertinentes sobre el modelo
- El **modelo será el encargado de interactuar con la base de datos**, ya sea en forma directa, con una capa de abstracción para ello, un Web Service, etc. y retornará esta información al controlador

- El **controlador recibe la información y la envía a la vista**
- La vista, procesa esta información pudiendo hacerlo desde el enfoque que veremos en este libro, creando una capa de abstracción para la lógica (quien se encargará de procesar los datos) y otra para el diseño de la interfaz gráfica o GUI. **La lógica de la vista, una vez procesados los datos, los "acomodará" en base al diseño de la GUI - *layout* – y los entregará al usuario** de forma "humanamente legible".



Funcionamiento del patrón modelo-vista-controlador

Hasta quí, hemos hablado de estilos arquitectónicos, nos hemos introducido en uno de los patrones arquitectónicos (MVC) pero, en todo esto **¿en qué momento intervienen los patrones de diseño?**

Es aquí donde debemos notar que éstos, intervienen en la forma en la que cada capa (modelo, vista y controlador), "diseña" su estructura. El controlador decidirá (aunque en realidad, nosotros lo haremos) si utilizará un *handler* para manejar los eventos del usuario. En ese caso, estaría optando por un **patrón de diseño**. Si para llamar al modelo, utiliza un *callback*, estaría utilizando otro, y así en lo sucesivo.

Personalmente, sugiero que, a no ser que se trate de sistemas realmente robustos y complejos, no se compliquen demasiado **– por ahora -** en "aplicar" todos y cada uno de los patrones de diseño que encuentren en el camino. La forma de hacerlo **bien**, es:

- Sencillez y simplicidad
- Fácil mantenibilidad
- Practicidad para evolucionar

Si se tienen en cuenta estas tres premisas, el resultado de un código limpio, legible y fácil de interpretar para cualquier programador, estará 90% asegurado.

CAPÍTULO IV

El patrón MVC en PHP

EL PATRÓN MVC EN PHP

Para que el concepto de modelo-vista-controlador nos queda más claro, vamos a ir analizándolo con ejemplos concretos, en el lenguaje que impulsa este libro: PHP.

Al final de este capítulo, vamos a encontrar ejercicios prácticos de autoevaluación y luego, continuaremos transformando y ampliando nuestro primer ejemplo del Capítulo II, siguiendo el patrón arquitectónico MVC, con sus correspondientes comentarios y aclaraciones.

EL MODELO

Con respecto al modelo, no tenemos demasiada novedad. El modelo, no es más que el conjunto de clases con las que hemos trabajado hasta ahora, incluyendo entre ellas, la capa de abstracción de la base de datos.

INTERFACES EN PHP: UN NUEVO CONCEPTO PARA CREAR MODELOS

En esta etapa, podemos ampliar las referencias de PHP, incorporando un nuevo concepto: **las interfaces**.

El **Manual Oficial de PHP** describe las interfaces de la siguiente forma:

> *"Las interfaces de objetos permiten crear código con el cual especificar qué métodos deben ser implementados por una clase, sin tener que definir cómo estos métodos son manipulados. Las interfaces son definidas utilizando la palabra clave interface, de la misma forma que con clases estándar, pero sin métodos que tengan su contenido definido. Todos los métodos declarados en una interfaz deben ser public, ya que ésta es la naturaleza de una interfaz."*[8] *[...]*

Una interface se define utilizando la palabra clave ***interface*** y se la implementa utilizando la palabra clave ***implements***.

En varias ocasiones, las interfaces nos servirán para ampliar los modelos.

Generamos la interface:

```
interface Postre {
    public function set_ingredientes();
}
```

Implementamos la interface:

```
class Bizcochuelo implements Postre {

    var $ingredientes = array();

    public function set_ingredientes() {
        $this->ingredientes = array('harina'=>'2 tazas', 'leche'=>'1 taza',
```

8 Fuente de la cita: http://php.net/manual/es/language.oop5.interfaces.php

```
                                    'azucar'=>'1 taza', 'huevo'=>1
                                );
    }
}
```

Extendemos una clase (que por defecto, implementará la interface):

```
class BizcochueloVainilla extends Bizcochuelo {

    public function set_ingredientes() {
        $this->ingredientes['escencia de vainilla'] = 'a gusto';
    }

    function __construct() {
        parent::set_ingredientes();
        $this->set_ingredientes();
    }
}
```

También podemos crear otra clase que implemente la interface:

```
class Alfajor implements Postre {

    public function set_ingredientes() {
        $this->ingredientes = array('Tapas de maizena' => 2,
                                    'dulce de leche'=>'1 cda. sopera',
                                    'coco rallado'=>'1 cdta. de te');
    }

    function __construct() {
        $this->set_ingredientes();
    }
}
```

Diferencia entre Interfaces y Clases abstractas

Probablemente te preguntes **¿cuál es la diferencia entre una interface y una clase abstracta?** Créeme: todos nos lo hemos preguntado alguna vez :)

En principio, existe una **diferencia conceptual** que a niveles "prácticos" tal vez no es del todo clarificadora. Esta diferencia, radica en que ***las clases abstractas, no dejan de ser "clases", las cuales representan la "abstracción de un objeto" siguiendo un orden de "relación jerarquíca" entre ellas:*** Clase B hereda de Clase A y Clase C hereda de Clase B, pero no puede tener herencias múltiples, es decir no puede heredar de más de una clase, ya que ellas, guardan una relación de orden jerárquico entre sí.

```
class A { }
class B extends class A { } #correcto
class C extends class B { } #correcto
class D  { } #correcto
class E extends class A, class D { } # incorrecto!!! no se puede hacer!!!
# lo correcto sería:
class D extends class A { }
class E extends class D { }
```

A diferencia de las clases abstractas, conceptualmente, ***las interfaces son solo un conjunto de métodos característicos de diversas clases***, independientemente de la relación que dichas clases mantengan entre sí. De hecho, una misma clase, puede implementar múltiples interfaces:

```
interface A { }
interface B { }
interface C { }
class MiClase implements A, B, C { }
```

No obstante la diferencia conceptual, **podemos establecer claras diferencias prácticas**, que nos ayudarán a asimilar mejor estos conceptos:

- **Las interfaces no pueden definir propiedades** (de hecho, no poseen propiedades porque a diferencia de las clases abstractas no tienen relación con un objeto, y solo los objetos tienen propiedades)
- **Las interfaces no pueden tener métodos definidos con su algoritmo correspondiente** (solo pueden declarar los métodos pero no pueden indicar el "qué" ni el "cómo" hacerlo, mientras que las clases abstractas, sí pueden)
- **Las interfaces no pueden instanciarse porque no tienen referencias asociadas a objetos**, mientras que las clases abstractas no pueden instanciarse, porque sus objetos son "entes abstractos" que necesitan de una clase no abstracta para definirse con exactitud y poder ser instanciados.
- **Todos los métodos declarados en una interface deben ser públicos**, puesto que la finalidad de una interface, es brindar un "ámbito público de un modelo". Las clases abstractas, pueden tener métodos abstractos tanto públicos como protegidos (aquellos que deberán ser obligatoriamente redefenidos por las clases que la hereden) y métodos privados que solo ella utilice.

Lo que NO deben hacer las insterfaces

Las interfaces no pueden:

- **Tener el mismo nombre que una clase** (PHP las interpreta como una clase más. Crear una interface con el mismo nombre que una clase, sería interpretado por PHP como una "re-declaración de clase")
- Diferentes interfaces no pueden **tener nombres de métodos idénticos** si serán implementadas por la misma clase

LA VISTA

Como hemos visto anteriormente, la vista representa la interfaz gráfica del usuario (GUI), es decir, es la encargada de mostrar la información al usuario de manera "humanamente legible".

A tal fin, divideremos la vista en dos sub-capas: la GUI propiamente dicha y su lógica.

VISTA: LA INTERFAZ GRÁFICA

Generalmente, en la práctica, no somos los programadores quienes nos hemos de encargar de la GUI. Es tarea que corresponde a diseñadores Web o gráficos, según aplique.

Como "arquitectos", debemos tener preparada la estructura de nuestra aplicación, para que diseñadores puedan trabajar libremente en la GUI, sin tener que acceder al código PHP. En este libro, **veremos** además, **como evitar por completo que los diseñadores tengan que implementar código PHP en sus GUI** y que programadores, tengamos que implementar código HTML en nuestras lógicas.

En principio, **la parte gráfica** de las vistas (archivos HTML, imágenes, CSS, etc.), **deberá tener un directorio reservado** solo para ella. De esta forma, obtenemos varias ventajas de trabajar con el patrón MVC:

- **Los archivos estáticos** (HTML, CSS, imágenes e incluso JavaScript) **pueden almacenarse en un servidor independiente** con todos los beneficios que esto conlleva (al igual que sucede con las bases de datos, los archivos estáticos también pueden independizarse)
- **Nos aseguramos de que el código estático** (aquel desarrollado por diseñadores), **no "rompa" el núcleo de la aplicación**
- **Permitimos a los diseñadores trabajar cómodamente** en la libertad de aplicar todos los conocimientos y herramientas inherentes a su área de conocimiento, sin la presión de tener que implementar un lenguaje de programación que desconocen, les es poco familiar o simplemente, la herramienta que utilizan para diseñar, suele dañarles el código no-estático

En segundo lugar, **cada plantilla** – template - o archivo HTML, **deberá hacer referencia "estática" a los datos que se quieran reemplazar dinámicamente**. Para esto, no será necesario implementar código que no sea HTML.

Veremos aquí como hacer referencia a estos datos, y más adelante, veremos como la lógica de la vista se encargará de renderizarlos.

Supongamos que tenemos una plantilla HTML, donde lo que necesitamos reemplazar de manera dinámica, es: **título de la página, keywords y descripción**. La mejor manera de referenciar a estos datos, facilitando la tarea a un diseñador, es haciendo una referencia textual a éstos. Para que a la vez, los programadores logremos diferenciar que allí se incluyen datos dinámicos, una buena práctica, es encerrar esa referencia textual entre llaves.

Veamos un ejemplo:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
  "http://www.w3.org/TR/html4/loose.dtd">
<html lang="es">

<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>{Título de la Página}</title>
```

```
<meta name="title" content="{Título de la Página}">
<meta name="keywords" content="{keywords}">
<meta name="description" content="{descripción}">
<link type="text/css" rel="stylesheet" href="css/template.css">
</head>

<body>
<div id="page">
    <div id="cab">
        <img alt="home page" src="img/logo.gif"></a>
     </div>
     <div id="contenido">
        <h1>{Título de la Página}</h1>
     </div>
</div>
</body>

</html>
```

Como podemos ver, en la plantilla HTML solo hay código HTML. Para un diseñador, será una tarea sencilla.

Veremos en el siguiente paso, como la lógica de la vista, se encargará de reemplazar estos datos.

LA LÓGICA DE LA VISTA

Tenemos la plantilla que creó nuestro diseñador. Es hora de reemplazar dinámicamente los datos referenciados. Haremos esto, paso a paso, como si de una receta de cocina se tratara.

Primer paso: crear un diccionario de datos

El diccionario de datos es el que contendrá, literalmente hablando, un diccionario, indicando cuales referencias deben ser reemplazadas por cuáles datos, en forma dinámica.

```
$diccionario = array(
    'Título de la Página'=>'POO y MVC en PHP',
    'keywords'=>'poo, mvc, php, arquitectura de software',
    'description'=>'El paradigma de la programación orientada a objetos con el
patrón arquitectónico MVC en PHP'
);
```

Segundo paso: obtener la plantilla HTML

Para traer la plantilla HTML, vamos a utilizar la función **`file_get_contents()`** de PHP, y almacenar el contenido de ella, en una variable.

```
$template = file_get_contents('/carpeta/template.html');
```

Tercer paso: reemplazar el contenido

En el anteúltimo paso, reemplazaremos los datos, utilizando el bucle **`foreach`** de PHP para recorrer el diccionario, y la función **`str_replace()`** de PHP, para reemplazar los

mismos.

```
foreach ($diccionario as $clave=>$valor) {
    $template = str_replace('{'.$clave.'}', $valor, $template);
}
```

Cuarto paso: mostrar el contenido final al usuario

Por último, imprimiremos el resultado obtenido en pantalla.

```
print $template;
```

Y con esto, habremos concluido la lógica de la vista, obteniendo en el navegador del usuario, la siguiente salida (vista de código fuente):

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
  "http://www.w3.org/TR/html4/loose.dtd">
<html lang="es">

<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>POO y MVC en PHP</title>
<meta name="title" content="POO y MVC en PHP">
<meta name="keywords" content="poo, mvc, php, arquitectura de software">
<meta name="description" content="El paradigma de la programación orientada a
objetos con el
                    patrón arquitectónico MVC en PHP">
<link type="text/css" rel="stylesheet" href="css/template.css">
</head>

<body>
<div id="page">
    <div id="cab">
        <img alt="home page" src="img/logo.gif"></a>
     </div>
     <div id="contenido">
        <h1>POO y MVC en PHP</h1>
     </div>
</div>
</body>

</html>
```

EL CONTROLADOR

El controlador, aunque muchos de ustedes no lo crean, es quien tal vez, lleva “la peor parte”. En muchas ocasiones, es complicado programar el controlador de manera previsible para que pueda evolucionar, y generalmente, es sometido a refactorizaciones constantes, incluso mayores que las que puede sufrir el modelo.

Recordemos que el controlador, es quien debe interactuar con el modelo y con la vista. Para hacerlo, deberá previamente reconocer y manejar los distintos eventos del usuario, para saber:

1. A qué modelo / modelos invocar
2. Qué y cuáles propiedades o atributos del modelo/modelos modificar o parámetros

deberá entregar a sus métodos
3. A qué vista entregar la información

¡Pobre hombre! Le toca una tarea extenuante. Veamos paso a paso, como ayudar a este pobre "controller-man" en su tarea.

Supongamos entonces, que tenemos dos modelos:

*Archivo **models.php***

```
class ModeloUno {
    var $propiedad;

    function a($parametro) {
        $this->propiedad = $parametro;
    }
}

class ModeloDos {
    var $propiedad_1;
    var $propiedad_2;

    function b($param1, $param2) {
        $this->propiedad_1 = $param1;
        $this->propiedad_2 = $param2;
    }
}
```

Y a la vez, tenemos dos vistas:

*Template **vista_1.html***

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
  "http://www.w3.org/TR/html4/loose.dtd">
<html lang="es">

<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>Vista Modelo 1</title>
</head>

<body>
<div id="page">
    <p>El valor de la <b>propiedad</b> es <b>{propiedad}</b></p>
</div>
</body>

</html>
```

*Template **vista_2.html***

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
  "http://www.w3.org/TR/html4/loose.dtd">
```

```
<html lang="es">

<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>Vista Modelo 2</title>
</head>

<body>
<div id="page">
    <p>El valor de <b>propiedad_1</b> es <b>{propiedad_1}</b> mientras que el
       de <b>propiedad_2</b> es <b>{propiedad_2}</b></p>
</div>
</body>

</html>
```

Y la lógica de éstas, es la siguiente:

*Archivo **view.php***

```
function set_identificadores($vista) {
    $identificadores = array();
    if($vista) {
        switch ($vista) {
            case 'vista_1':
                $identificadores = array('propiedad');
                break;

            case 'vista_2':
                $identificadores = array('propiedad_1', 'propiedad_2');
                break;
        }
        return $identificadores;
    }
}

function armar_diccionario($vista, $data) {
    $diccionario = array();
    $identificadores = set_identificadores($vista);
    if($identificadores) {
        foreach ($identificadores as $identificador) {
            if(array_key_exists($identificador, $data)) {
                $diccionario[$identificador] = $data[$identificador];
            }
        }
    }
    return $diccionario;
}

function render_data($vista, $data) {
    $html = '';
    if(($vista)&&($data)) {
        $diccionario = armar_diccionario($vista, $data);
        if($diccionario) {
            $html = file_get_contents('html/'.$vista.'.html');
            foreach ($diccionario as $clave=>$valor) {
                $html = str_replace('{'.$clave.'}', $valor, $html);
            }
        }
```

```
    }
    print $html;
}
```

Entonces, nuestro controlador procederá de la siguiente forma:

Primer paso: identificar el modelo

Para esto, el controlador previamente, tendrá que reconocer el evento que ejecuta el usuario y saber como manejarlo. Para ello, la forma de pensarlo "informáticamente" sería:

si usuario **[evento]** entonces **[realizar acción]**

En nuestro caso, los eventos admitidos, serán llamdas por URL mediante el método $_GET.

Primero, identificará el evento respondiendo a las siguientes preguntas:

- ¿existe el evento de llamada mediante "$_GET"?
- Si existe ¿a qué evento, el usuario, quiere que responda?

Para esto, utilizará la siguiente función:

```
function capturar_evento() {
    $vista = '';
    if($_GET) {
        if(array_key_exists('vista', $_GET)) {
            $vista = $_GET['vista'];
        }
    }
    return $vista;
}
```

Devolviendo así, el evento capturado a través de la variable "$vista".

Con el evento "en la mano", se ocupará ahora de identificar el modelo:

```
function identificar_modelo($vista) {
    if($vista) {
        switch ($vista) {
            case 'vista_1':
                $modelo = 'ModeloUno';
                break;
            case 'vista_2':
                $modelo = 'ModeloDos';
                break;
            default:
                exit();
        }
    }
    return $modelo;
}
```

Segundo paso: invocar al modelo efectuando los cambios adecuados

Nuestro controlador, ya sabe a que modelo recurrir. Ahora, solo resta invocarlo y modificarlo si es necesario:

```
function invocar_modelo($modelo) {
    if($modelo) {
        require_once('models.php');
        $data = new $modelo();
        settype($data, 'array');
        return $data;
    }
    #las modificaciones al modelo se harían aquí
}
```

Tercer paso: enviar la información a la vista

Finalmente, nuestro controlador, enviará la información obtenida del modelo, a la vista. Para hacerlo, utilizará una función donde preparará esta información para al fin enviarla:

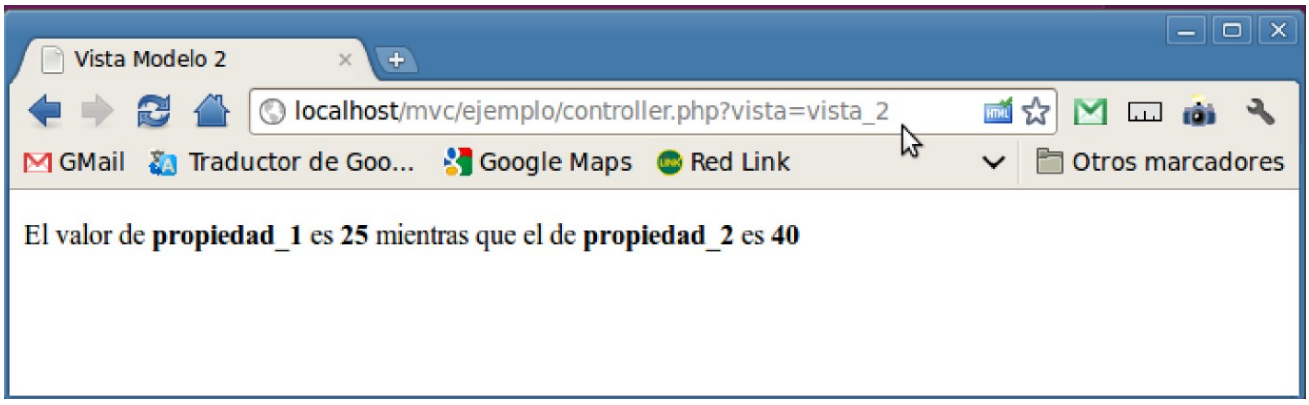
```
function enviar_data() {
    $vista = capturar_evento();
    if($vista) {
        $modelo = identificar_modelo($vista);
        if($modelo) {
            $data = invocar_modelo($modelo);
            if($data) {
                require_once('view.php');
                render_data($vista, $data);
            }
        }
    }
}
```

El controlador, solo deberá llamar a su propia función enviar_data() y la vista automáticamente, estará en condiciones de mostrar los resultados en pantalla al usuario:

```
enviar_data();
```

Si el usuario ingresara en la URL http://servidor/controller.php no verá nada en pantalla.

Si ingresara en http://servidor/controller.php?vista=vista_2 verá lo siguiente:



Ahora, es momento de auto-evaluar lo que hemos aprendido y luego, completar nuestro primer ejemplo para finalizar nuestro aprendizaje.

Ejercicios Prácticos

Veamos cuanto aprendimos sobre los capítulos III y IV.

Ejercicio Nº3: Conceptos de Arquitectura, MVC y PHP

3.1) ¿Qué es MVC?

a) Un patrón de diseño

b) Un patrón arquitectónico

c) Un paradigma de programación

d) Un estilo arquitectónico

3.2) ¿Cuál de las siguientes opciones, corresponde a atributos de calidad implícitos en la arquitectura de software?

a) Polimorfismo, herencia, encapsulamiento y disponibilidad

b) Herencia, abstracción, seguridad y aislamiento

c) Disponibilidad, funcionalidad, seguridad, interoperabilidad y portabilidad

d) Todas las anteriores

3.3) Indica V (verdadero) o F (falso) según corresponda

a) [ ] Un patrón arquitectónico es el encargado de describir la arquitectura general de un sistema

b) [ ] Un patrón arquitectónico representa una plantilla de construcción que provee un conjunto de subsistemas y su forma de organización

c) [ ] Estilo arquitectónico y patrón arquitectónico son dos niveles de abstracción diferentes de la arquitectura de software

d) [ ] En MVC, el modelo es un intermediario entre la vista y el controlador

e) [ ] En MVC el controlador es el encargado de capturar los eventos del usuario

f) [ ] En MVC es recomendable mezclar código PHP y HTML

g) [ ] Una interface en PHP es lo mismo que una clase abstracta

h) [ ] Las clases abstractas en PHP cuplen la misma función que las interfaces

i) [ ] Una interface solo debe declarar los métodos pero éstos, no deben estar codificados

j) [ ] Las propiedades declaradas en las interfaces deben tener visibilidad pública

3.4) ¿Cuál de las siguientes afirmaciones es verdadera?

a) Las interfaces no pueden definir propiedades

b) Todos los métodos declarados en una interface deben ser públicos

c) Las interfaces no pueden instanciarse porque no tienen referencias asociadas a objetos

d) Las interfaces no pueden tener métodos definidos con su algoritmo correspondiente

e) Todas las opciones son correctas

3.5) En MVC, el usuario realiza una petición y el _____ captura el evento, identificando el _____ al que debe invocar y al cual realizar las modificaciones. Luego, envía esta información a _____.

Ejercicio Nº4: Revisando código

4.1) ¿Cuál de las siguientes declaraciones es incorrectas?

a) `class Gaseosa extends Bebida implements Producto, BebidaSinAlcohol`

b) `class Gaseosa extends BebidaSinAlcohol, BebidaGasificada implements Bebida, Producto`

d) `class Gaseosa implements Bebida`

4.2) ¿Cuál es el error del siguiente código?

```
interface Producto { }
class Producto implements Producto { }
```

a) La clase no puede tener el mismo nombre que la interface

b) Ni la clase ni la interface tienen código

c) No hay errores

4.3) ¿Cuál es el error en el siguiente código?

```
interface Producto { }
class Bebida extends Producto { }
```

a) Ni la clase ni la interface tienen código

b) Donde utiliza "extends" debería utilizar "implements"

c) No hay errores

Soluciones a los ejercicios 3 y 4

Ejercicio nº3

Pregunta 3.1: respuesta **b**

Pregunta 3.2: respuesta **c**

Pregunta 3.3:

	A	B	C	D	E	F	G	H	I	J
Verdadera		X	X		X				X	
False	X			X		X	X	X		X

Pregunta 3.4: respuesta **e**

Pregunta 3.5:

En MVC, el usuario realiza una petición y el ***controlador*** captura el evento, identificando el ***modelo*** al que debe invocar y al cual realizar las modificaciones. Luego, envía esta información a ***la vista***.

Ejercicio nº4

Pregunta 4.1: respuesta **b**

Pregunta 4.2: respuesta **a**

Pregunta 4.3: respuesta **c**

PROGRAMACIÓN REAL CON EL PATRÓN MVC

Vamos a retormar ahora, nuestro ejemplo del **ABM de usuarios** que comenzamos a programar en el capítulo II. Procederemos a ampliarlo y adaptarlo, seguiendo la estructura del patrón MVC.

***Nótese** que los métodos respectivos han sido resumidos no encontrándose en éstos, extensos algoritmos de validación de datos. Se ejemplifica todo aquello que es relevante en la POO bajo el el patrón arquitectónico MVC.*

ESTRUCTURA DE DIRECTORIOS

Ya estamos en una etapa más avanzada, donde no basta con crear archivos. Nuestro primer ejemplo de "programación real" del capítulo II, será ampliado; agregaremos más archivos y se hace necesario organizarlos de forma tal, que pueda respetarse la sencillez y los beneficios que el patrón MVC nos brinda.

Es necesario que tengamos en cuenta, que NO SIEMPRE la estructura y organización de directorios y archivos puede predecirse. Por el contrario, a medida que una aplicación avanza, SIEMPRE es recomendable refactorizar el código al mismo ritmo que el sistema evoluciona.

Por eso, no se debe tener miedo de comenzar programando con grandes estructuras de código y hasta incluso, por qué no, "desprolijas".

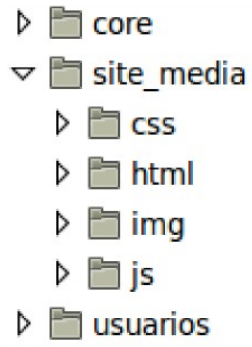
Piensa en cuando vas al supermercado. Realizas las compras de todo lo que necesitas y las llevas a tu casa en bolsas y generalmente, con la mercadería desorganizada. Pero tienes todo lo que necesitas. Una vez que lo tienes, comienzas a organizarlo. Guardarás los alimentos que requieran refrigeración, en la heladera; los congelados en el freezer; los artículos de tocador en tu WC; etc.

No se debe tener miedo de **agilizar la arquitectura de software** y pensarla **adaptativamente**. Y aquí, es donde debemos considerar, la metodología que utilizaremos para gestionar nuestro proyecto.

Una de las prácticas técnicas propuesta por **Extreme Programming**, nos induce a realizar una refactorización constante del código, algo que, desde mi punto de vista, nos dará mejores resultados a la hora de pensar una arquitectura adecuada.

La siguiente estructura de directorios que propondré para adaptar nuestro ejemplo, puede servir de punto de partida para la arquitectura de una aplicación compleja de gran robustez.

Comenzaremos entonces, por crear la siguiente **estructura de directorios**:



core:
el propósito de este directorio, será almacenar todas aquellas clases, *helpers*, *utils*, *decorators*, etc, que puedan ser reutilizados en otros módulos de la aplicación (aquí solo crearemos el de usuarios), es decir, todo aquello que pueda ser considerado "núcleo" del sistema.

site_media:
será el directorio exclusivo para nuestros diseñadores. En él, organizadmente, nuestros diseñadores podrán trabajar libremente, creando los archivos estáticos que la aplicación requiera.

usuarios:
éste, será el directorio de nuestro módulo de usuarios. En él, almacenaremos los archivos correspondientes al modelo, la lógica de la vista y el controlador, exclusivos del módulo de usuarios.

Bajo el supuesto de que los modelos DBAbstractModel y Usuario se conservan y, simplemente a modo de ejercicio **¿en qué directorio crees que se debería almacenar cada uno de estos dos modelos?**

ARCHIVOS DEL MODELO

Los archivos de nuestro "modelo", serán ahora nuestros viejos ***db_abstract_model.php*** y ***usuarios_model.php*** a los cuales, haremos algunas modificaciones.

ARCHIVO `./core/db_abstract_model.php`

Modificaciones incorporadas

1. Agregaremos la propiedad `$mensaje`, a fin de que el objeto, pueda comunicarse a través de mensajes con el resto de la aplicación.
2. El método `execute_single_query()` incorporará una validación que solo permitirá proceder con sus funciones, si recibe sus parámetros a través variables `$_POST`.

Código fuente del archivo

```
<?php

abstract class DBAbstractModel {

    private static $db_host = 'localhost';
    private static $db_user = 'usuario';
    private static $db_pass = 'contraseña';
    protected $db_name = 'mydb';
    protected $query;
    protected $rows = array();
    private $conn;
    public $mensaje = 'Hecho';

    # métodos abstractos para ABM de clases que hereden
    abstract protected function get();
    abstract protected function set();
    abstract protected function edit();
    abstract protected function delete();

    # los siguientes métodos pueden definirse con exactitud y
    # no son abstractos
```

```
        # Conectar a la base de datos
        private function open_connection() {
            $this->conn = new mysqli(self::$db_host, self::$db_user,
                                     self::$db_pass, $this->db_name);
        }

        # Desconectar la base de datos
        private function close_connection() {
              $this->conn->close();
        }

        # Ejecutar un query simple del tipo INSERT, DELETE, UPDATE
        protected function execute_single_query() {
            if($_POST) {
                $this->open_connection();
                $this->conn->query($this->query);
                $this->close_connection();
            } else {
                $this->mensaje = 'Metodo no permitido';
            }
        }

        # Traer resultados de una consulta en un Array
        protected function get_results_from_query() {
            $this->open_connection();
            $result = $this->conn->query($this->query);
            while ($this->rows[] = $result->fetch_assoc());
            $result->close();
            $this->close_connection();
            array_pop($this->rows);
        }
    }
    ?>
```

ARCHIVO ./usuarios/model.php

Modificaciones incorporadas

1. El nombre del archivo cambia a model.php.
2. Se agrega una emisión de mensaje por cada acción concluyente de cada uno de los métodos
3. Sin influencia práctica, el método constructor aparece debajo de los métodos propios de la clase (una simple cuestión organizativa)

Código fuente del archivo

```
<?php
# Importar modelo de abstracción de base de datos
require_once('../core/db_abstract_model.php');


class Usuario extends DBAbstractModel {

################################ PROPIEDADES ################################
    public $nombre;
    public $apellido;
    public $email;
    private $clave;
    protected $id;
```

```
################################## MÉTODOS ###################################
    # Traer datos de un usuario
    public function get($user_email='') {
        if($user_email != '') {
            $this->query = "
                SELECT      id, nombre, apellido, email, clave
                FROM        usuarios
                WHERE       email = '$user_email'
            ";
            $this->get_results_from_query();
        }

        if(count($this->rows) == 1) {
            foreach ($this->rows[0] as $propiedad=>$valor) {
                $this->$propiedad = $valor;
            }
            $this->mensaje = 'Usuario encontrado';
        } else {
            $this->mensaje = 'Usuario no encontrado';
        }
    }

    # Crear un nuevo usuario
    public function set($user_data=array()) {
        if(array_key_exists('email', $user_data)) {
            $this->get($user_data['email']);
            if($user_data['email'] != $this->email) {
                foreach ($user_data as $campo=>$valor) {
                    $$campo = $valor;
                }
                $this->query = "
                        INSERT INTO     usuarios
                        (nombre, apellido, email, clave)
                        VALUES
                        ('$nombre', '$apellido', '$email', '$clave')
                ";
                $this->execute_single_query();
                $this->mensaje = 'Usuario agregado exitosamente';
            } else {
                $this->mensaje = 'El usuario ya existe';
            }
        } else {
            $this->mensaje = 'No se ha agregado al usuario';
        }
    }

    # Modificar un usuario
    public function edit($user_data=array()) {
        foreach ($user_data as $campo=>$valor) {
            $$campo = $valor;
        }
        $this->query = "
                UPDATE      usuarios
                SET         nombre='$nombre',
                            apellido='$apellido'
                WHERE       email = '$email'
        ";
        $this->execute_single_query();
        $this->mensaje = 'Usuario modificado';
    }
```

```
        # Eliminar un usuario
        public function delete($user_email='') {
            $this->query = "
                    DELETE FROM     usuarios
                    WHERE           email = '$user_email'
            ";
            $this->execute_single_query();
            $this->mensaje = 'Usuario eliminado';
        }

        # Método constructor
        function __construct() {
            $this->db_name = 'book_example';
        }

        # Método destructor del objeto
        function __destruct() {
            unset($this);
        }
    }
    ?>
```

ARCHIVOS DE LA VISTA

En esta etapa, crearemos los archivos correspondientes a la GUI y su lógica.

Los primeros, serán los archivos que crearán nuestros diseñadores (5 archivos HTML y 1 archivo CSS, que se expondrán a modo práctico, sin ánimo de servir de ejemplo como interfaces gráficas).

La lógica de la vista, estará en nuestras manos.

Comencemos.

ARCHIVOS DE LA GUI

Recordemos que se trataba de un ABM de Usuarios, donde teníamos la posibilidad de:

- Mostrar los datos de un usuario
- Agregar un nuevo usuario
- Modificar un usuario existente
- Eliminar un usuario de la base de datos

Todos estos archivos, se almacenarán en nuestro directorio estático `site_media`.

Crearemos una plantilla HTML general, cuatro plantillas de formularios y un archivo CSS. Los 5 primeros se almacenarán en `./site_media/html/` con el prefijo `usuario_` que identificará el módulo (también podrían almacenarse en un subdirectorio llamado `usuario`. Eso lo dejo a vuestra libre elección). El archivo CSS, lógicamente se almacenará en `./site_media/css/`.

Archivo `./site_media/html/usuario_template.html`
(plantilla HTML general)

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
  "http://www.w3.org/TR/html4/loose.dtd">
<html lang="es">

<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<link rel="stylesheet" type="text/css"
href="/mvc/site_media/css/base_template.css">
<title>ABM de Usuarios: {subtitulo}</title>
</head>

<body>
<div id="cabecera">
    <h1>Administrador de usuarios</h1>
    <h2>{subtitulo}</h2>
</div>
<div id="menu">
    <a href="/mvc/{VIEW_SET_USER}" title="Nuevo usuario">Agregar usuario</a>
    <a href="/mvc/{VIEW_GET_USER}" title="Buscar usuario">Buscar/editar
usuario</a>
    <a href="/mvc/{VIEW_DELETE_USER}" title="Borrar usuario">Borrar
usuario</a>
</div>
<div id="mensaje">
    {mensaje}
</div>
<div id="formulario">
    {formulario}
</div>
</body>

</html>
```

Por favor, obsérvese que en este archivo, se deberán reemplazar dinámicamente: el subtítulo del módulo (método), las URL de los enlaces de las vistas, el mensaje emitido por el sistema y el formulario correspondiente al método/vista solicitado por el usuario.

Archivo `./site_media/html/usuario_agregar.html`
(formulario para agregar un nuevo usuario)

```
    <form id="alta_usuario" action="{SET}" method="POST">
        <div class="item_requerid">E-mail</div>
        <div class="form_requerid"><input type="text" name="email"
         id="email"></div>
        <div class="item_requerid">Clave</div>
        <div class="form_requerid"><input type="password" name="clave"
         id="clave"></div>
        <div class="item_requerid">Nombre</div>
        <div class="form_requerid"><input type="text" name="nombre"
         id="nombre"></div>
        <div class="item_requerid">Apellido</div>
        <div class="form_requerid"><input type="text" name="apellido"
         id="apellido"></div>
        <div class="form_button"><input type="submit" name="enviar"
         id="enviar" value="Agregar"></div>
    </form>
```

Nótese que este archivo no posee datos dinámicos que deban ser reemplazados.

Archivo ./site_media/html/usuario_borrar.html
(formulario para eliminar un usuario identificado por su e-mail)

```
<form id="alta_usuario" action="{DELETE}" method="POST">
    <div class="item_requerid">E-mail</div>
    <div class="form_requerid"><input type="text" name="email"
     id="email"></div>
    <div class="form_button"><input type="submit" name="enviar"
     id="enviar" value="Eliminar"></div>
</form>
```

Nótese que al igual que el archivo anterior, éste, tampoco contiene datos dinámicos que deban ser reemplazados.

Archivo ./site_media/html/usuario_buscar.html
(archivo que permite buscar un usuario por su e-mail para luego mostrarlo en el formulario de edición, ya sea para editarlo o solo para ver sus datos)

```
<form id="alta_usuario" action="{GET}" method="GET">
    <div class="item_requerid">E-mail</div>
    <div class="form_requerid"><input type="text" name="email"
     id="email"></div>
    <div class="form_button"><input type="submit"
     id="enviar" value="Buscar"></div>
</form>
```

Nótese que al igual que los dos archivos anteriores, éste, tampoco contiene datos dinámicos que deban ser reemplazados.

Archivo ./site_media/html/usuario_modificar.html
(archivo para visualizar los datos de un usuario y modificarlos)

```
<form id="alta_usuario" action="{EDIT}" method="POST">
    <div class="item_requerid">E-mail</div>
    <div class="form_requerid"><input type="text" name="email"
     id="email" value="{email}" readonly></div>
    <div class="item_requerid">Nombre</div>
    <div class="form_requerid"><input type="text" name="nombre"
     id="nombre" value="{nombre}"></div>
    <div class="item_requerid">Apellido</div>
    <div class="form_requerid"><input type="text" name="apellido"
     id="apellido" value="{apellido}"></div>
    <div class="form_button"><input type="submit" name="enviar"
     id="enviar" value="Guardar"></div>
</form>
```

Por favor, nótese que en este archivo deberán reemplazarse dinámicamente los datos del usuario en los campos de formulario correspondientes. Obsérvese que el campo "email" es de solo lectura.

Archivo ./site_media/css/base_template.css
(hoja de estilos en cascada)

```
body {
    margin: 0px 0px 0px 0px;
    background-color: #ffffff;
    color: #666666;
    font-family: sans-serif;
    font-size: 12px;
```

```
}

#cabecera {
    padding: 6px 6px 8px 6px;
    background-color: #6699FF;
    color: #ffffff;
}

#cabecera h1, h2 {
    margin: 0px 0px 0px 0px;
}

#menu {
    background-color: #000000;
    color: #ffffff;
    padding: 4px 0px 4px 0px;
}

#menu a {
    width: 100px;
    background-color: #000000;
    padding: 4px 8px 4px 8px;
    color: #f4f4f4;
    text-decoration: none;
    font-size: 13px;
    font-weight: bold;
}

#menu a:hover {
    background-color: #6699FF;
    color: #ffffff;
}

#mensaje {
    margin: 20px;
    border: 1px solid #990033;
    background-color: #f4f4f4;
    padding: 8px;
    color: #990033;
    font-size: 15px;
    font-weight: bold;
    text-align: justify;
}

#formulario {
    margin: 0px 20px 10px 20px;
    border: 1px solid #c0c0c0;
    background-color: #f9f9f9;
    padding: 8px;
    text-align: left;
}

.item_requerid {
    width: 150px;
    height: 22px;
    padding-right: 4px;
    font-weight: bold;
    float: left;
    clear: left;
    text-align: right;
}
```

```
.form_requerid {
    height: 22px;
    float: left;
    clear: right;
}

input {
   border: 1px solid #c0c0c0;
}

.form_button {
    padding-top: 15px;
    margin-left: 154px;
    clear: both;
}

#enviar {
    padding: 5px 10px 5px 10px;
    border: 2px solid #ffffff;
    background-color: #000000;
    color: #ffffff;
    font-family: sans-serif;
    font-size: 14px;
    font-weight: bold;
    cursor: pointer;
}

#enviar:hover {
    background-color: #6699FF;
}
```

ARCHIVO ./usuarios/view.php

Lógica de la vista

```
<?php
$diccionario = array(
    'subtitle'=>array(VIEW_SET_USER=>'Crear un nuevo usuario',
                      VIEW_GET_USER=>'Buscar usuario',
                      VIEW_DELETE_USER=>'Eliminar un usuario',
                      VIEW_EDIT_USER=>'Modificar usuario'
                     ),
    'links_menu'=>array(
        'VIEW_SET_USER'=>MODULO.VIEW_SET_USER.'/',
        'VIEW_GET_USER'=>MODULO.VIEW_GET_USER.'/',
        'VIEW_EDIT_USER'=>MODULO.VIEW_EDIT_USER.'/',
        'VIEW_DELETE_USER'=>MODULO.VIEW_DELETE_USER.'/'
    ),
    'form_actions'=>array(
        'SET'=>'/mvc/'.MODULO.SET_USER.'/',
        'GET'=>'/mvc/'.MODULO.GET_USER.'/',
        'DELETE'=>'/mvc/'.MODULO.DELETE_USER.'/',
        'EDIT'=>'/mvc/'.MODULO.EDIT_USER.'/'
    )
);

function get_template($form='get') {
    $file = '../site_media/html/usuario_'.$form.'.html';
    $template = file_get_contents($file);
    return $template;
}
```

```
function render_dinamic_data($html, $data) {
    foreach ($data as $clave=>$valor) {
        $html = str_replace('{'.$clave.'}', $valor, $html);
    }
    return $html;
}

function retornar_vista($vista, $data=array()) {
    global $diccionario;
    $html = get_template('template');
    $html = str_replace('{subtitulo}', $diccionario['subtitle'][$vista],
$html);
    $html = str_replace('{formulario}', get_template($vista), $html);
    $html = render_dinamic_data($html, $diccionario['form_actions']);
    $html = render_dinamic_data($html, $diccionario['links_menu']);
    $html = render_dinamic_data($html, $data);

    // render {mensaje}
    if(array_key_exists('nombre', $data)&&
       array_key_exists('apellido', $data)&&
       $vista==VIEW_EDIT_USER) {
        $mensaje = 'Editar usuario '.$data['nombre'].' '.$data['apellido'];
    } else {
        if(array_key_exists('mensaje', $data)) {
            $mensaje = $data['mensaje'];
        } else {
            $mensaje = 'Datos del usuario:';
        }
    }
    $html = str_replace('{mensaje}', $mensaje, $html);

    print $html;
}
?>
```

Por favor, nótese que este archivo, utiliza constantes. Las mismas, y a fin de otorgar mayor legibilidad y organización al código, serán definidas en un archivo independiente almacenado en `./usuarios/`, llamado **`constants.php`** que será expuesto más adelante.

EL CONTROLADOR

ARCHIVO ./usuarios/controller.php

```
<?php
require_once('constants.php');
require_once('model.php');
require_once('view.php');

function handler() {
    $event = VIEW_GET_USER;
    $uri = $_SERVER['REQUEST_URI'];
    $peticiones = array(SET_USER, GET_USER, DELETE_USER, EDIT_USER,
                        VIEW_SET_USER, VIEW_GET_USER, VIEW_DELETE_USER,
                        VIEW_EDIT_USER);
    foreach ($peticiones as $peticion) {
        $uri_peticion = MODULO.$peticion.'/';
        if( strpos($uri, $uri_peticion) == true ) {
```

```
            $event = $peticion;
        }
    }

    $user_data = helper_user_data();
    $usuario = set_obj();

    switch ($event) {
        case SET_USER:
            $usuario->set($user_data);
            $data = array('mensaje'=>$usuario->mensaje);
            retornar_vista(VIEW_SET_USER, $data);
            break;
        case GET_USER:
            $usuario->get($user_data);
            $data = array(
                'nombre'=>$usuario->nombre,
                'apellido'=>$usuario->apellido,
                'email'=>$usuario->email
            );
            retornar_vista(VIEW_EDIT_USER, $data);
            break;
        case DELETE_USER:
            $usuario->delete($user_data['email']);
            $data = array('mensaje'=>$usuario->mensaje);
            retornar_vista(VIEW_DELETE_USER, $data);
            break;
        case EDIT_USER:
            $usuario->edit($user_data);
            $data = array('mensaje'=>$usuario->mensaje);
            retornar_vista(VIEW_GET_USER, $data);
            break;
        default:
            retornar_vista($event);
    }
}


function set_obj() {
    $obj = new Usuario();
    return $obj;
}

function helper_user_data() {
    $user_data = array();
    if($_POST) {
        if(array_key_exists('nombre', $_POST)) {
            $user_data['nombre'] = $_POST['nombre'];
        }
        if(array_key_exists('apellido', $_POST)) {
            $user_data['apellido'] = $_POST['apellido'];
        }
        if(array_key_exists('email', $_POST)) {
            $user_data['email'] = $_POST['email'];
        }
        if(array_key_exists('clave', $_POST)) {
            $user_data['clave'] = $_POST['clave'];
        }
    } else if($_GET) {
        if(array_key_exists('email', $_GET)) {
            $user_data = $_GET['email'];
        }
```

```
        }
        return $user_data;
    }


    handler();
    ?>
```

Por favor, obsérvese que el controlador importa el archivo **`constants.php`** y utiliza constantes que serán definidas en dicho archivo (veremos éste más adelante).

Por otro lado, el *handler* del controlador, maneja las peticiones del usuario sobre la base de la URI detectada. Este ejemplo, maneja "URL amigables" (*friendly url*), que serán tratadas más adelante, configurando las mismas en el archivo **`.htaccess`** de la aplicación.

ARCHIVOS DE CONFIGURACIÓN COMPLEMENTARIOS

ARCHIVO `./usuarios/constants.php`

Este archivo contiene todas las constantes del módulo y no se le debería agregar ningún dato que no fuese una constante.

```
<?php
const MODULO = 'usuarios/';

# controladores
const SET_USER = 'set';
const GET_USER = 'get';
const DELETE_USER = 'delete';
const EDIT_USER = 'edit';

# vistas
const VIEW_SET_USER = 'agregar';
const VIEW_GET_USER = 'buscar';
const VIEW_DELETE_USER = 'borrar';
const VIEW_EDIT_USER = 'modificar';
?>
```

ARCHIVO `.htaccess`

Agregar al archivo `.htaccess` de la aplicación, las siguientes líneas:

```
RewriteEngine on
RewriteRule ^usuarios/ usuarios/controller.php
```

Nótese que dependiendo del directorio raíz de la aplicación, probablemente se deba modificar la línea `RewriteRule ^usuarios/ usuarios/controller.php`

NOTA FINAL

Hemos llegado al final de este libro. Espero que todo lo aquí expuesto, les haya sido de utilidad y los invito a hacerme llegar sus dudas, sugerencias y comentarios através de mi e-mail o visitando la **página del libro**:

http://eugeniabahit.blogspot.com/2011/07/poo-y-mvc-en-php.html

¡Gracias por el tiempo que han dedicado!

¡Éxitos en tu carrera!

El paradigma de la Programación Orientada a Objetos en PHP con el patrón arquitectónico MVC. A lo largo del libro, aprenderás a construir una aplicación orientada a objetos en PHP, utilizando el patrón arquitectónico MVC, desde cero, sin necesidad de utilizar ningún framework, incluso aunque tus conocimientos de programación sean básicos. Aprenderás a hacerlo tú mismo, logrando una abstracción completa y absoluta, tanto del modelo, la vista y el controlador, como de la independencia de lenguajes de programación y diseño.