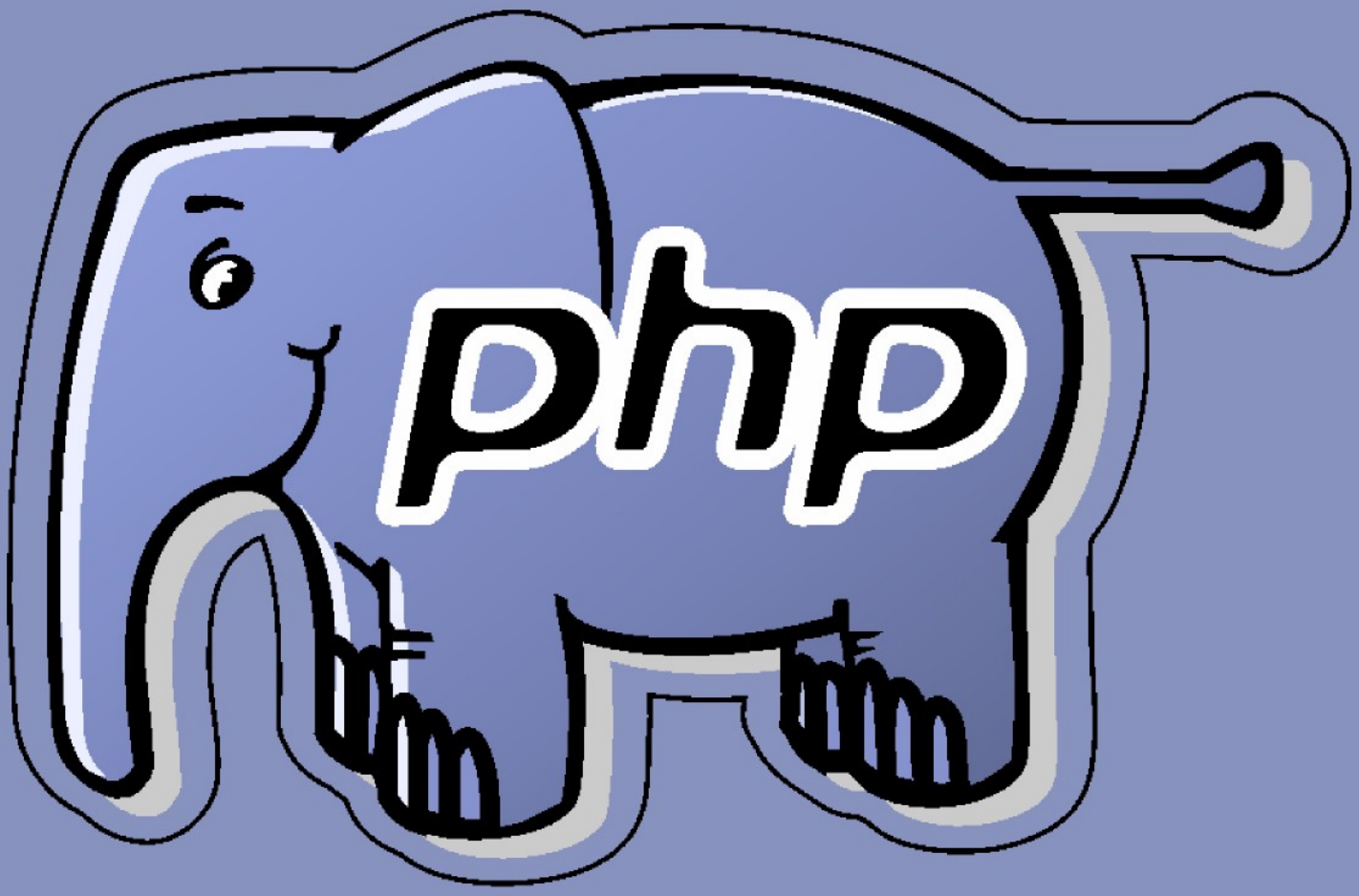
Programación en



Versión 2.0

php

Gugler
Laboratorio de Investigación Gugler

FCyT
Facultad de Ciencia y Tecnología

UADER
Universidad Autónoma de Entre Ríos

CONTENIDO

Patrones de diseño **5**

Introducción 5
Teoría de los patrones de diseño 5
¿ Qué es un patrón ? 5
Objetivos 6
Clasificación de los patrones 6
De creación 7
Estructurales 7
De comportamiento 8
The Singleton Pattern 9
Definición 9
Ejemplo de aplicación 9
Scripts 9
Codificación de los script 9
The Factory Pattern 11
Definición 11
Scripts 11
Codificación de los script 11
The Active Record 13
Definición 13
Ejemplo de aplicación 14
Scripts 14
Codificación de los script 14
The Registry 16
Definición 16
Scripts 17
Codificación de los scripts 17
Value Object 20
Definición 20
Scripts 21
Codificación de los scripts 21

Capítulo 5

Patrones de diseño

Introducción

Los patrones de diseño son generalmente reconocidos como un excelente conjunto de soluciones probadas a los problemas comunes que los desarrolladores enfrentan cada día. En este capítulo, nos centraremos en cómo algunas de PHP 5 de las nuevas instalaciones, como la orientación objeto propio, puede hacer que el desarrollo de aplicaciones basadas en patrones más fácil.

Teoría de los patrones de diseño

¿ Qué es un patrón ?

Como mencionamos en el inicio del capítulo, los patrones de diseño comprenden soluciones a

problemas comunes. No se enfocan en el código que usted debe escribir, sino que proporcionan directrices que puede traducir en su código fuente sin importar el lenguaje de programación que este utilizando.

Podemos decir que un patrón es una estructura de clases que aparece repetidamente en diversos diseños orientados a objetos, la cuál es utilizada para resolver un problema determinado de forma flexible y adaptable dinámicamente. El propósito de un patrón de diseño es capturar el conocimiento de un diseño de software y hacerlo reusable.

A pesar de que podemos implementar patrones de diseño utilizando un *diseño procedural*, los patrones se ilustran mejor con programación orientada a objetos. Es por eso que es sólo con PHP 5 que se han convertido en una herramienta muy relevante para el mundo de PHP: con una arquitectura apropiada orientada a objetos, la creación de patrones de diseño es fácil y proporciona un método comprobado y verdadero para el desarrollo de código robusto.

Objetivos

Los objetivos que ofrece el uso de patrones de diseño son los siguientes:

1. Reducción de tiempo.
2. Disminuir tiempo y esfuerzo aplicado al mantenimiento.
3. Aumentar la eficiencia.
4. Asegurar la consistencia.
5. Aumentar la fiabilidad.
6. Estandarizar el modo en que se realiza el diseño.
7. Facilitar el aprendizaje de las nuevas generaciones de diseñadores condensando conocimiento ya existente.

No es obligatorio utilizar los patrones, solo es aconsejable en el caso de tener el mismo problema o similar que soluciona el patrón, siempre teniendo en cuenta que en un caso particular puede no ser aplicable. Abusar o forzar el uso de los patrones puede ser un error.

Clasificación de los patrones

Según la publicación de *Design Patterns: Elements of Reusable Object Oriented Sofware* escrita por Gang of Four (GoF, que en español es la pandilla de los cuatro) formada por Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides, existen 23 patrones de diseño aplicados usualmente por expertos diseñadores de software orientado a objetos, los cuales son clasificados en 3 grandes categorías basadas en su propósito: creacionales, estructurales y de comportamiento.

De creación

Estos patrones tratan acerca de la mejor manera de crear instancias de objetos. Esto es importante porque los programas no deberían depender de como los objetos son creados.

Los patrones de creación abstraen el proceso de creación de instancias. Ayudan a hacer a un sistema independiente de cómo se crean, se componen y representan sus objetos. Un patrón de *creación de clases* usa la herencia para cambiar la clase de la instancia a crear, mientras que un patrón de *creación de objetos* delega la creación de la instancia en otro objeto.

Los patrones de creación son cinco y están estrechamente relacionados:

- ***Factory Method (método de fafricación):*** es aquel que retorna una instancia de una o varias posibles clases dependiendo de los datos que se le provea. Este patrón define una interfaz para crear un objeto, pero deja que sean las subclases quienes decidan qué clase instanciar. Permite que una clase delegue en sus subclases la creación de objetos.
- ***Abstract Factory (Fábrica Abstracta):*** presenta un nivel de abstracción más alto que el *Factory*. Proporciona una interfaz para crear *familias de objetos* relacionados o que dependen entre sí, sin especificar sus clases concretas.
- ***Singleton (Único):*** Existen muchas situaciones en las que se necesita asegurarse de que solo habrá *una instancia de una clase*. Por ejemplo, si el sistema puede tener solo una ventana de administración o una cola de impresión , o un único punto de acceso a una base de datos.
- **Builder (Constructor):** separa la construcción de un objeto completo de su representación, de forma que el mismo proceso de construcción pueda crear diferentes representaciones.
- **Prototype (Prototipo):** especifica los tipos de objetos a crear por medio de una instancia prototípica, y crea nuevos objetos copiando dicho prototipo.

Estructurales

Estos patrones tratan cómo las clases y los objetos se combinan para dar lugar a estructuras más complejas. Puede hacerse aquí la misma distinción que hacíamos en los patrones de creación y hablar de patrones estructurales asociados a clases (Adapter) y asociados a objetos (Bridge, Composite, Decorator, Facade, Flyweight, Proxy), los primeros utilizarán la herencia, los segundos la composición.

Los patrones Estructurales son cinco:

- ***Adapter (adaptador):*** convierte la interfaz de una clase en otra distinta que es la que esperan los clientes. Permiten que cooperen clases que de otra manera no podrían por tener interfaces incompatibles.
- **Bridge:** desvincula una abstracción de su implementación, de manera que ambas puedan variar de forma independiente.
- **Composite:** combina objetos en estructuras de árbol para representar jerarquías de

parte-todo. Permite que los clientes traten de manera uniforme a los objetos individuales y a los compuestos.
- **Decorator:** Añade dinámicamente nuevas responsabilidades a un objeto, proporcionando una alternativa flexible a la herencia para extender la funcionalidad.
- **Facade:** proporciona una interfaz unificada para un conjunto de interfaces de un subsistema. Define una interfaz de alto nivel que hace que el subsistema se más fácil de usar.

De comportamiento

Identifican patrones comunes de comunicación entre objetos y tratan de incrementar la flexibilidad de esta comunicación. Describen no solo patrones de objetos o clases sino también patrones de comunicación entre ellos. Nuevamente se pueden clasificar en función de que trabajen con clases (Template Method, Interpreter) u objetos (Chain of Responsability, Command, Iterator, Mediator, Memento, Observer, State, Strategy, Visitor).

- **Command:** encapsula una petición en un objeto, permitiendo así parametrizar a los clientes con distintas peticiones, encolar o llevar un registro de las peticiones y poder deshacer la operaciones.
- **Strategy:** define una familia de algoritmos, encapsula uno de ellos y los hace intercambiables. Permite que un algoritmo varíe independientemente de los clientes que lo usan.
- **Observer:** define una dependencia de uno-a-muchos entre objetos, de forma que cuando un objeto cambia de estado se notifica y actualizan automáticamente todos los objetos.
- **Chain of Responsibility:** evita acoplar el emisor de una petición a su receptor, al dar a más de un objeto la posibilidad de responder a la petición. Crea una cadena con los objetos receptores y pasa la petición a través de la cadena hasta que esta sea tratada por algún objeto.

The Singleton Pattern

Definición

El *Singleton* es probablemente, el patrón de diseño más simple. Su objetivo es facilitar el acceso a un recurso y que éste nunca se duplique, sino que se ponga a disposición de cualquier parte de la aplicación que lo requiera sin necesidad de realizar un seguimiento de su existencia.

Ejemplo de aplicación

El ejemplo más típico de este patrón es una conexión de base de datos, que normalmente sólo hay que crear una vez al comienzo de un script y luego se usa a través de toda la aplicación.

Scripts

Para la realización del ejemplo desarrollaremos los siguientes script:

- index.php
- DB.php

Codificación de los script

DB.php

```
class DB
{
	private static $_singleton;
	private static $_host = '127.0.0.1';
	private static $_user = 'root';
	private static $_pass = '';
	private $_connection;

	private function __construct()
	{
		$this->_connection = new PDO('mysql:host='.self::$_host, self::$_user, self::$_pass);
	}

	public static function getInstance()
	{
		if (is_null (self::$_singleton)) {
			self::$_singleton = new DB();
		}
		return self::$_singleton;
	}
}
```

Como podemos ver, la clase DB define una propiedad estática (*static*) de nombre *$_singleton*. Esta propiedad va ha hacer referencia a una instancia del propio objeto DB que estamos

declarando.

Como podemos observar, el método *__construct()* es privado, de manera que no podremos instanciar directamente un objeto de esta clase mediante un new(). Para poder realizar esto, deberemos llamar al método *getInstance()* el cuál sí es publico. Si analizamos este método veremos que evalúa si el atributo *$_singleton* de la propia clase ya ha sido definido (haciendo uso de self). Si ya ha sido definido lo devuelve, sino llama al método *__construct()* el cuál crea una conexión con la BD y se la asigna al atributo *$_singleton*.

index.php

```
$oDataBase = DataBase::getInstancia();
var_dump($oDataBase);

$oDataBase1 = DataBase::getInstancia();
var_dump($oDataBase);
```

En esta implementación de la clase DB aprovechamos algunos conceptos de la programación orientada a objetos que están disponibles en PHP 5: hemos declarado el constructor privado, lo cual garantiza efectivamente que únicamente la propia *clase* pueda crear instancias de si misma.

De esta forma, el método *getInstance()* comprueba si la propiedad *$_singleton* ha sido declarada y de no ser así, se establece en una nueva instancia de DB. Esto nos dará la certeza de que la conexión se realizará al invocarse *getInstance()* por primera vez. En las próximas invocaciones se devolverá la instancia creada.

The Factory Pattern

Definición

El patrón *Factory* se utiliza en escenarios donde se tiene una clase genérica (Factory) que proporciona las facilidades para la creación de instancias de una o más clases que manejan la misma tarea de diferentes formas. Una buena situación en la que el patrón Factory ofrece una excelente solución es la gestión de múltiples mecanismos de almacenamiento para una tarea determinada.

Por ejemplo, considere la posibilidad de almacenamiento de alguna configuración, que pueden ser guardados por almacenes de datos como archivos INI, bases de datos o archivos XML de manera intercambiable. La API que cada una de estas clases proporciona es la misma, pero la implementación de fondo cambia. El patrón *Factory* nos proporciona una forma fácil de devolver un almacén de datos de diferentes clases en función de cualquiera de las preferencias

del usuario.

Scripts

Para la realización del ejemplo desarrollaremos los siguientes script:

- index.php
- ActiveRecordFactory.php
- MysqlActiveRecordFactory.php
- PgsqlActiveRecordFactory.php
- OracleActiveRecordFactory.php

Codificación de los script

ActiveRecordFactory.php

```
<?php
require_once 'MysqlActiveRecordFactory.php';
require_once 'PgsqlActiveRecordFactory.php';
require_once 'OracleActiveRecordFactory.php';
class ActiveRecordFactory
{
	const MYSQL = 1;
	const PGSQL = 2;
	const ORACLE = 3;
	public static function getActiveRecordFactory($motor = selft::MYSQL)
	{
		switch ($motor) {
			case selft::MYSQL:
				return new MysqlActiveRecordFactory();
			case selft::PGSQL:
				return new PgsqlActiveRecordFactory();
			case self:ORACLE:
				return new OracleActiveRecordFactory();
		}
	}
}
```

Como podemos ver, esta clase contiene un único método publico llamado getActiveRecordFactory(), el cual espera como parámetro el motor de base de datos al cual queramos conectarnos. Una vez recibido el parámetro, es evaluado por la instrucción switch y se instancia el objeto de conexión.

MysqlActiveRecordFactory.php

```
<?php
class MysqlActiveRecordFactory
{
	public function __construct()
	{
		return mysql_connect('localhost', 'root',' ');
	}
}
```

Al ser instanciado un objeto MysqlActiveRecordFactory desde la clase ActiveRecordFactory, el constructor realizará la conexión con el motor y lo devolverá al método desde donde fue invocado (en este caso getActiveRecordFactory());.

PgsqlActiveRecordFactory.php

```
<?php
class PgsqlActiveRecordFactory
{
	// ...
}
```

OracleActiveRecordFactory.php

```
<?php
class OracleActiveRecordFactory
{
	// ...
}
```

index.php

```
<?php
require_once 'ActiveRecordFactory.php';

class Index
{
	public function ejecutar()
	{

		$oMysqlActiveRecord =
ActiveRecordFactory::getActiveRecordFactory(ActiveRecordFactory::MYSQL);

	}
}

$oIndex = new Index();
$oIndex->ejecutar();
```

Para hacer uso del patrón deberemos de invocar al método público getActiveRecordFactory() y enviarle como parámetro el motor de base de datos con el que queramos conectarnos.

The Active Record

Definición

El siguiente patrón que vamos a examinar es el Active Record. Este patrón busca dar solución al problema de acceder a los datos de una base de datos. La idea detrás de Active Record es sencilla, una fila en la tabla de la base de datos es representada por un objeto, de manera que se asocian filas únicas de la base de datos con objetos.

La forma de implementarlo constaría en definir una clase que por cada tabla en nuestra base de datos. De esta manera, nuestra clase contaría con tantos atributos (privados) como campos tengan cada una de las tablas. Además deberíamos de programar los métodos que nos permitan insertar, recuperar, actualizar y borrar los datos que tenemos instanciados en nuestro objeto en la tabla de la base de datos.

Ejemplo de aplicación

Se cuenta con una base de datos *sgp* la cual posee una tabla *Personas* con la siguiente definición de campos:

Personas:
idPersona: int autoincrement not null;
apellido: varchar (50);
nombres: varchar (50);
domicilio: varchar (50);
telefono: varchar (50);

Realizaremos haciendo uso de Active Record, una clase que nos permita mapear objetos con filas en nuestra tabla.

Scripts

Para la realización del ejemplo desarrollaremos los siguientes script:

- index.php
- MysqlPersonaActiveRecord.php

Codificación de los script

MysqlPersonaActiveRecord.php

```
<?php
class MysqlPersonaActiveRecord
{
	private $_idPersona;
	private $_docu, $_apellido, $_nombres, $_domicilio,$_telefono;

	public function setDocu ($documento)
	{
		$this->_docu = $documento;
	}

	public function setApellido($apellido)
	{
		$this->_apellido = $apellido;
	}

	public function setNombres($nombres)
	{
		$this->_nombres= $nombres;
	}

	public function setDomicilio ($domicilio)
	{
		$this->_domicilio= $domicilio;
	}

	public function insert()                                    // Insertar una fila
	{
		$sql="INSERT INTO personas  VALUES ";
		$sql.="('',$this->_docu, '$this->_aplellido', '$this->_nombres', '$this->_domicilio' )";
		if ($dbh->exec($sql) >= 1) {
			return true;
		} else {
			return false
		}
	}

	public function update($id)                                 // Modificar una fila
	{
		$sql="UPDATE INTO personas  SET documento=$this->_docu";
		$sql.="apellido=$this->_apellido, nombres=$this->_nombres,";
		$sql.= "domicilio=$this->_domicilio WHERE id_persona=$id";
		if ($dbh->exec($sql) >= 1) {
```

```
                return true;
            } else {
                return false
            }
        }
        public function delete($id)                    // Borrar una fila
        {
            $sql="DELETE FROM personas WHERE id_persona=$id";
            if ($dbh->exec($sql) >= 1) {
                return true;
            } else {
                return false
            }
        }

        public function fetch ($id)                    // Seleccionar una fila
        {
            $sql="SELECT * FROM personas WHERE id_persona=$id";
            $stmt = $dbh->query($sql);
            return $stmt->fetch();
        }
}
```

Como podemos ver, la clase posee definido un atributo por cada campo en la tabla Personas. Además desarrollamos un método publico por cada una de las operaciones básicas que podríamos realizar con la tabla. De esta manera, podremos incluir nuestra clase MysqlPersonaActiveRecord desde cualquier script que requiera hacer uso de la tabla.

A continuación veremos como utilizar esta clase dentro del script *index.php*.

index.php

```
<php
include('MysqlPersonaActiveRecord.php');

    $oPersona= new  MysqlPersonaActiveRecord();
        $oPersona-> setDocu(22555666);
        $oPersona-> setApellido('Sosa');
        $oPersona-> setNombres('Antonio');
        $oPersona-> setDomicilio('Río Negro');
     $oPersona->insert();
```

Como podemos ver, el código resultante se hace mucho mas sencillo y de mejor lectura. De esta manera lo único que tendremos que hacer es setear los valores de nuestros atributos y realizar la llamada al método que queramos para realizar la operación en nuestra tabla.
Una ventaja que encontramos en el uso de este patrón es que, al tener que realizar un cambio en el esquema de nuestra base de datos, únicamente deberemos modificar los métodos

definidos en nuestra clase sin preocuparnos por las llamadas que realizamos desde nuestros scripts.

The Registry

Definición

El ante último patrón que vamos a examinar es el Registry. Este patrón busca dar solución al problema de compartir datos dentro de la aplicación Web dejando disponible el acceso de los mismos por medio una clase que nos provee de métodos para poder registrar, obtener, borrar los distintos elementos que registremos.

Claramente vemos que tendremos los datos más importantes de una aplicación dentro de la definición de clase y con sus comportamientos para accederlos en cualquier momento de forma ordenada. En la mayoría de los casos vamos a implementar el patrón utilizando la variable global $_SESSION para guardar datos referentes a la navegación de nuestros usuarios en la aplicación Web.

La base del patrón tiene que ver con la propiedad de tipo array que guardará los distintos elementos (objetos) del sistema. Luego tendremos métodos para acceder a esta propiedad y poder agregar, obtener y borrar elementos con sus respectivos identificadores.

Scripts

Para poder entender el uso del patrón vamos a ver un ejemplo que registra un alumno con sus datos personales y los cursos que realiza en una aplicación Web, a continuación los nombres de los scripts que codificaremos:

- index.php
- Registry.php
- Persona.php
- Alumno.php
- Curso.php
- MostrarCurso.php

Codificación de los scripts

index.php

```
<?php
session_start();
```

```
require_once 'Alumno.php';
require_once 'Curso.php';
require_once 'Registry.php';

abstract class Index
{
	public function ejecutar()
	{
		$_SESSION['oRegistry'] = new Registry();
		$oAlumno = new Alumno(25546113, 'DNI', 1, 1);
		$oCursoPhp = new Curso(1, 'Programación en PHP', 'Curso de Desarrollo WEB');
		$oCursoLinux = new Curso(2, 'Administración GNU/Linux', 'Curso de Administración
GNU/Linux');
		$oAlumno->agregarCurso($oCursoPhp);
		$oAlumno->agregarCurso($oCursoLinux);
		$_SESSION['oRegistry']->add('Alumno', $oAlumno);
		header("Location: MostrarCurso.php");
	}
}
Index::ejecutar();
```

Registy.php

```
<?php
class Registry
{
	private $_datos = array();

	public function add($key, $data)
	{
		$this->_datos[$key] =  $data;
	}

	public function delete($key)
	{
		unset($this->_datos[$key]);
	}
	public function get($key)
	{
		return $this->_datos[$key];
	}

	public function exists($key)
	{
		if (isset($this->_datos[$key])) {
			return true;
		} else {
			return false;
		}
```

```
	}
}
```

Persona.php

```
<?php

class Persona
{
	private $_ndocu;
	private $_tdocu;
	private $_apellidos;
	private $_nombres;

	public function __contruct($ndocu, $tdocu)
	{
		$this->_ndocu = $ndocu;
		$this->_tdocu = $ndocu;
	}
}
```

Alumno.php

```
<?php
require_once 'Persona.php';
require_once 'Curso.php';

class Alumno extends Persona
{
	private $_idAlumno;
	private $_tipoAlumno;
	private $_colCurso = array();

	public function __construct($ndocu, $tdocu, $idAlumno, $tipoAlumno)
	{
		parent::__contruct($ndocu, $tdocu);
		$this->_idAlumno = $idAlumno;
		$this->_tipoAlumno = $tipoAlumno;
	}

	public function agregarCurso(Curso $curso)
	{
		$this->_colCurso[] = $curso;
	}
}
```

Curso.php

```
<?php
```

```
class Curso
{
	private $_idCurso;
	private $_nombre;
	private $_descripcion;

	public function __construct($idCurso, $nombre, $descripcion)
	{
		$this->_idCurso = $idCurso;
		$this->_nombre = $nombre;
		$this->_descripcion = $descripcion;
	}
}
```

MostrarCurso.php

```
<?php

require_once 'Alumno.php';
require_once 'Curso.php';
require_once 'Registry.php';

session_start();

abstract class MostrarCurso
{
	public function mostrar()
	{
		echo "<h2>Obtengo y muestro el objeto Alumno: </h2><br />";

		$oAlumno = $_SESSION['oRegistry']->get('Alumno');

		var_dump($oAlumno);
		echo "<br />";
		echo "<h2>Borro y verifico la existencia del objeto Alumno: </h2><br />";

		$_SESSION['oRegistry']->delete('Alumno');
		echo "<br />";

		echo "Objeto Alumno borrado!!!<br />";
		echo "<br />";

		if ($_SESSION['oRegistry']->exists('Alumno')) {
			echo "El objeto Alumno existe como elemento registrado!";
		} else {
			echo "El objeto Alumno NO existe como elemento registrado!";
		}
	}
}
```

```
MostrarCurso::mostrar();
```

En la implementación y desarrollo del ejemplo podemos ver que se define una variable objeto de la clase registry que se encargará de registrar los distintos objetos en la aplicación que permanecerán disponibles mientras dure la sesión.

Value Object

Definición

El último patrón que vamos a examinar es el Value Object. Este patrón se confunde con frecuencia con los conceptos de Java Bean, DTO y POJO. Value Object es muy parecido a Java Bean con la diferencia que permite mantener inmutables los valores del objeto creado.

Para mantener los datos de un objeto inmutables se debe trabajar en el método constructor y en los setter de la clase. Generalmente, en el constructor se establecen los valores de las propiedades del objeto. Y los métodos setter dejan de existir; o bien, crean un nuevo objeto con el nuevo valor establecido en la propiedad.

Scripts

Para poder entender el uso del patrón vamos a ver un ejemplo que mantiene inmutables los valores de un objeto del tipo/clase persona, a continuación los nombres de los scripts que codificaremos:

- index.php
- Persona.php

Codificación de los scripts

index.php

```
<?php

require_once 'PersonaVO.php';

class Index
{
	public static function ejecutar()
	{
		$juanGarcia = new PersonaVO(1, 'Garcia', 'Juan');

		$juanGonzalez = $unaPersona->setApellidos('Gonzalez');
```

```
                var_dump($juanGarcia);

                echo "<br/>";

                var_dump($juanGonzalez);
        }
}

Index::ejecutar();
```