Mapeador Objeto/Relacional

Persistencia no invasiva y por alcance en PHP

Leonardo Tadei

Especialista en Sistemas; Maestrando Ingeniería del Software UNLP

<leonardot@pegasusnet.com.ar>

Mar del Plata - Argentina

(+54)(+223) 471-2880

Licencia Creative Commons Reconocimiento-NoComercial-SinObraDerivada 3.0 Unported.

Resumen: de los paradigmas de programación, la Programación Orientada a Objetos (POO) se consolidó en la década de 1990 como la respuesta que ofrecía la flexibilidad y velocidad de desarrollo que no podían satisfacer otros paradigmas. Sin embargo a aplicación masiva y la implementación de POO en lenguajes híbridos de este enfoque se encontró con el problema de la persistencia, que fue solucionado con técnicas nativas como las OODB o híbridas como el Mapeo Objeto/Relacional (ORM)

Sin embargo el ORM y la hibridación parecen conspirar contra las buenas prácticas de la POO, sobre todo por implementaciones deficientes a nivel metodológico e los mismos, como por la natural confusión que utilizar un lenguaje híbrido conlleva.

Por todo esto y para poner un ejemplo de ORM que permita respetar la POO, implementamos uno que funciona de forma no invasiva y provee persistencia por alcance.

1. El problema de la Persistencia

Un problema que supuso la adopción de la Programación Orientada a Objetos (POO) fue el de la *persistencia,* entendiendo esta como el mecanismo para que el software pueda almacenar y recuperar los Objetos por los que está compuesto. Cabe destacar que la persistencia no es parte del paradigma de la POO, sino un mecanismo para que cuando dejamos de ejecutar un software, los datos se mantengan de manera tal que al volverlo a ejecutar, no hayamos perdido información.

Lenguajes de Objetos puros como Smalltalk implementan este mecanismo de la forma más simple posible: almacenan en el disco una imagen de la memoria de la aplicación, y al volver a ejecutarla, restauran dicha imagen. Esto proceso es válido, pero presenta problemas a la hora de almacenar grandes cantidades de datos (del orden de los millones), de almacenarlos distribuidamente y de compartirlos con otras instancias del software en ejecución.

2. El problema del desajuste por impedancia

Por otra parte, con un sólido soporte matemática, implementaciones eficientes y capacidad para almacenar millones de datos de forma distribuida y accesibles de manera concurrente, tenemos las Sistemas Manejadores de Bases de Datos Relacionales (RDBMS, *relational data base manegement system*).

Sin embargo, estos sistemas de almacenamiento tienen el problema del desajuste por impedancia con la la POO. Este desajuste se refiere a la imposibilidad de representar fielmente un modelo de software Orientado a Objetos a un almacenamiento en un RDBMS, dado que en la POO tenemos conceptos como herencia, polimorfismo, encapsulamiento, binding dinámico y recolección de basura, que usando como únicos elementos conceptuales "objetos" y "mensajes" basta para construir el software, y por contraparte los RDBMS están construidos en base a conceptos como tablas, filas, campos, índices, procedimientos almacenados y disparadores. Tal vez esto pueda expresarse de mejor manera diciendo que la metodología del diseño orientado a objetos no tiene ningún elemento en común con la normalización de bases de datos.

Son mundos conceptuales distintos orientados a proveer soluciones a problemas diferentes. Por todo esto es necesario para que un software orientado a objetos use como mecanismo de persistencia un RDBMS tener alguna estrategia de mapeo o conversión, entre los objetos del software y la entidades almacenadas.

3. Arquitectura MVC

Esta arquitectura ampliamente utilizada por sus buenos resultados a la hora de estructurar una

aplicación, consiste en la separación del Modelo, las reglas de negocio que implementan la solución, la Vista, presentación de datos que arroja el Modelo, y los Controladores, que son los manejadores de eventos generados en la Vista y que disparan acciones del Modelo.

En la *ilustración 1* puede verse un diagrama de esta arquitectura, en dónde las líneas sólidas representan conocimiento directo, y las líneas punteadas representan conocimiento indirecto; esto significa que, por ejemplo, un cambio en el Modelo requiere que la Vista cambie, pero un cambio en la Vista no afecta de ninguna manera al código del Modelo.

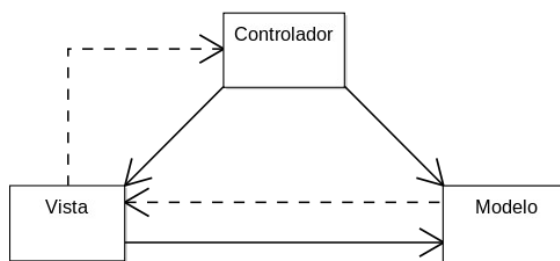

Ilustración 1: Arquitectura MVC

Se debe notar que el espíritu detrás de esta arquitectura consiste en que el Modelo es la cosa a preservar, ya que refleja el problema a resolver y la solución, en cambio la forma de ver e interactuar con el Modelo puede cambiar de diversas maneras sin afectarlo. El MVC es una solución tan ampliamente usada y validada, ya que justamente escribir un Modelo es una tarea muy compleja en comparación con una Vista y con un Controlador: se preserva lo complejo separándolo de los más volátil y simple.

4. El problema del Active Record y los Data Access Object

Los Data Access Object (DAO, *Objeto de Acceso a Datos*) es un mecanismo por el cual se ofrece una interfaz común a un medio de almacenamiento. Cuando un DAO es invocado, este devuelve los 'datos' que es capaz de leer. El Active Record podría pensarse como una especialización del DAO que cuando es invocado, devuelve un registro de una tabla de base de datos, o el registro que dé por resultado una proyección sobre la base de datos, exponiéndolos con una sintaxis acorde a los objetos.

Es una solución ampliamente difundida, en cuanto que permite automatizar tareas de alta, baja y modificación sobre registros de una tabla de base de datos, pero que trae como consecuencia nefasta lo que se describe como Modelo Anémico: se diseñan los Objetos del Modelo para solo tener atributos, que se corresponden con los campos de las tablas y tienen solo setters y getters, pero no tienen ningún comportamiento asociado, con lo que se desdibujan las responsabilidades y las reglas de negocio, deviniendo en una solución cuando menos mediocre del problema, que en vez de reflejar la realidad que se está modelando, refleja en su lugar el resultado de la normalización.

Dado que en POO se diseña creando jerarquías por comportamiento, el Active Record trae además generalmente aparejado una jerarquía por atributos, con lo que se rompe la premisa fundamental del

diseño Orientado a Objetos, que es modelar en base al comportamiento.

5. Solución Orientada a Objetos Pura: ODBMS

Una solución a estos problemas es usar una Base de Datos Orientada a Objetos (ODBMS, *object database management system*). Estos sistemas de almacenamiento consisten en que desde el lenguaje de programación que se usen, se le da directamente los Objetos a persistir, y posteriormente pueden ser recuperados de forma transparente. Todo el mecanismo de persistencia y de recuperación subyacente es manejado por la ODBMS.

Existen varios ODBMS, de los cuales podemos citar Versant, Gemstone y ObjectStore. Son excelentes piezas de software, pero no ampliamente difundidas principalmente por varias cuestiones: una cuestión técnica, es que necesitan tener un binding con el lenguaje de programación, lo cual los restringe a solo a algunos lenguajes, siendo PHP uno de los que no están soportados; otra cuestión, en este caso de infraestructura, es que de existir un RDBMS ya en funcionamiento, nuestro software tiene que acceder a los datos existentes; otra cuestión más es la económica, ya que si la organización licenció un RDBMS debe ahora amortizar la solución y debería además afrontar los costos de migración de un almacenamiento a otro.

6. Mapeo Objeto-Relacional

En este contexto surge la estrategia del Mapeo Objeto-Relacional (ORM, Object-Relational Mapping) que consiste en que un Modelo de Objetos puro, sea persistido en un RDBMS usando reglas para pasar del Modelo de Objetos al Modelo Relacional y viceversa, sin sacrificar las cualidades ni los paradigmas de uso, resolviendo así la impedancia entre modelos.

Existen varias estrategias de mapeo:

<u>clase concreta a tabla</u>, en la que los atributos de un objeto son mapeados a un registro de una tabla. Esta estrategia funciona en muchos casos pero tiende a que provocar que se modelen objetos con estructuras más simples que las reales (POJO).

<u>jerarquía de clases a tabla</u>, en la que los atributos de superclase y los atributos de todas las subclases son mapeados a una tabla. En este caso, dependiendo del objeto concreto, la tabla tendrá registros de más, que deben ser ignorados por el mapeador cuando corresponde.

cada clase a su propia tabla, en la que los atributos de las clases abstractas se representan en una tabla, y los de las clases concretas en otra, con lo que hidratar un objeto que herede de una clase abstracta implica hacer un JOIN a dos o más tablas.

modelo a tabla universal, en la que todo el modelo se almacena en una gran tabla, que tendrá campos de discriminación para poder asociar un registro y los campos que corresponden a una clase.

Todas estas estrategias de mapeo tienen ventajas y desventajas, debiéndose elegir la mejor en base a la estructura del modelo, y de ser posible, diferentes estrategias para cada conjunto conceptual de clases.

7. Romper el paradigma o viciar el Modelo

Como vimos en la sección 3, la idea subyacente a la arquitectura MVC es preservar el Modelo. Existen implementaciones de ORM que pierden de vista la preservación del modelo proponiendo mecanismos aparentemente inocentes, pero que implican o violar el paradigma de la POO o introducir cambios en el Modelo que no tienen que ver con las reglas del negocio.

Enumeramos aquí algunas de estas malas prácticas, siempre tomando como referencia el paradigma de la POO.

Mala práctica 1: Heredar del ORM

Una practica muy habitual es que, para que los Objetos puedan tener su mecanismo de persistencia, estos deben heredar de una clase del ORM que es quien la provee. El código tiene la siguiente forma:

```
class Cliente extends MyORM {…}
```

Con esta estrategia hacemos que todo nuestro Modelo esté subordinado a una clase que no se corresponde con las reglas de negocio, que deberemos agregar al ORM a los test de unidad, ya que por herencia está presente en todo el Modelo, y que, para usar este ejemplo, ahora todo *Cliente* tendrá la capacidad de guardarse, aunque nuestro software pueda necesitar usarlos temporalmente o no requieran ser persistidos nunca.

De esta manera, el persistir a la instancia quedaría con la siguiente sintaxis:

```
$c = new Cliente();
// código que modifica el valor de los atributos
$c->save();
```

Nótese que ahora los Objetos de clase *Cliente* tienen como comportamiento *save()*, pero el ser persistidos no es en realidad un comportamiento de las reglas de negocio, sino un recurso técnico para almacenar los datos.

Mala práctica 2: violar el encapsulamiento

Algunos ORM requieren que los atributos del objeto sean públicos, para poder acceder a ellos al persistir y luego para hidratarlo:

```
class Cliente {
	public $nombre = '';
	public $limiteCredito = 0.00;
	…
}
```

De esta manera, ahora cualquier objeto del sistema, podría modificar el valor en límite de crédito, saltándose las restricciones del Modelo que deberían estar en el getter, ya sea por tener implementada la regla ahí o por aceptar solo mensajes de cambio de otro Objeto con esta responsabilidad.

Mala práctica 3: usar anotaciones

Cuando las anotaciones no son parte de la definición del lenguaje, como es el caso de PHP, son meros comentarios en el código:

```
class Cliente {
	/**
	* @ORM\Id
	* @ORM\Column(type="integer")
	* @ORM\GeneratedValue(strategy="AUTO")
	*/
	protected $id;
	…
}
```

Si bien de esta manera conseguimos mantener el encapsulamiento del atributo, el correcto funcionamiento del código queda supeditado a un comentario, que no generará nunca un mensaje de error, que no puede ser testeado salvo por el efecto de que el *Cliente* se persistirá, y sobre todo porque cualquier optimizador, ofuscador o minimizador de código lo eliminará.

Mala práctica 4: acceder a todos las instancias mediante una instancia

Muchos ORM inducen a tratar al Objeto como puente para acceso a los datos. Esto no parece una mala práctica en este ejemplo, en el que se instancia el objeto `$c` de clase `Cliente` invocando un método de clase heredado para hidratar uno en particular:

```
$c = Cliente::findById(34);
```

Pero en el siguiente ejemplo:

```
$c = Cliente::findByName('Perez');
```

en realidad *$c* no es una instancia de cliente, sino alguna estructura tal que pueda contener todos los objetos de clase `Cliente` que tienen por nombre "Perez", con lo que se desdibuja la identidad del

Objeto, que según como se use, es el Objeto en sí o es una colección de Objetos.

8. Persistencia no invasiva

Viendo los inconvenientes de los mecanismos que implican romper el paradigma de programación o viciar el modelo, es que decidimos implementar un ORM que haga persistencia no invasiva, es decir, que al modelar el Objeto, el código no tenga ninguna línea de código que haga referencia al ORM, y que a su vez su diseño no esté condicionado por él. Cabe destacar que las técnica usadas se pueden implementar en cualquier lenguaje de programación que soporte introspección (Reflection) sobre los Objetos.

Dada la siguiente clase:

```
class Auto {
      private $patente; // la patente del automóvil
      private $carga; // peso en Kg
      function __construct($p = '', $c = 0) {
            $this->setPatente($p);
            $this->setCarga($c);
      }
      function setPatente($v) { $this->patente= $v; }
      function getPatente() { return $this->patente; }
      function setCarga($v) { $this->carga = $v; }
      function getCarga() { return $this->carga; }
      function calcularConsumo() {
            return 8 * (1 + $this->getCarga() / 1000;
      }
}
```

Nótese que los atributos son privados, y que en el código no hay ninguna referencia a la persistencia. Podría usarse y persistirse de la siguiente manera:

```
$p = new Persistent(); // instanciamos al ORM
$a1 = new Auto('GNU-007', 280);  // creamos una instancia de Auto
print('El consumo es: '. $a1->calcularConsumo());  // mostramos el consumo
$p->save($a1);  // el mapeador persiste al objeto
```

El ORM funciona como un repositorio virtual de Objetos. Intenta dar la sensación de que, una vez creado un objeto, este existe, y por tanto no debe ser vuelto a crear para ser usado nuevamente. Veamos ahora un ejemplo de modificación, en el que previamente es necesario hidratar el objeto para recuperarlo del repositorio:

```
$p = new Persistent(); // instanciamos al ORM
$a1 = $p->retrieve('Auto','GNU-007');  // hidratamos la instancia
print('El consumo es: '. $a1->calcularConsumo());  // mostramos el consumo
$a->setCarga(500);
print('El consumo cambió: '. $a1->calcularConsumo());  // mostramos el consumo
$p->save($a1);  // el mapeador persiste al objeto con su estado actual
```

Nótese que al hidratarlo, el ORM crea la instancia de *Auto* solicitada, con lo cual el método `calcularConsumo()` sigue existiendo, ya que el comportamiento se mantiene intacto. Para hidratarlo, el ORM crea un objeto de la clase dada, e invoca a los setters para completar la hidratación, con lo que si hubiera alguna restricción en los valores implementada esta se ejecutaría, con lo que una alteración de los datos en la base de datos no puede corromper el Modelo.

Nótese que tanto para almacenar el Objeto en el repositorio virtual como para actualizarlo, se invoca al método *save()* del ORM. Esto es así porque usando el principio de indentidad, el ORM creará un Objeto nuevo en el repositorio virtual si el ObjectId no existe, y lo actualizará en caso de que ya exista. Es por esto que en el mapeo es necesario especificar el atributo que funcionará como ObjectId.

Para borrar la instancia del repositorio el código es como sigue:

```
$p = new Persistent(); // instanciamos al ORM
$a1 = $p->retrieve('GNU-007');  // hidratamos la instancia
print('El consumo es: '. $a1->calcularConsumo());  // mostramos el consumo
$p->delete($a1);  // el Objeto es removido del repositorio
var_dump($a1);  // devuelve NULL: el Objeto ya no existe
```

Colecciones de Objetos

El ORM implementa un mecanismo para hidratar una colección de Objetos, de manera tal que sin saber como en el ejemplo anterior la patente del *Auto*, podamos obtener un listado de los existentes. Para esto es necesaria una colección que implemente la interfaz `IteratorAggregate` de PHP; en nuestro ejemplo, la colección estará casteada (forzada a aceptar mensajes de un solo tipo de Clase) para evitar que se agreguen Objetos de clases diferentes a las deseadas:

```
class CollectionAuto implements IteratorAggregate {
      private $items = array();
      // Métodos
      public function getIterator() {
            return new ArrayIterator($this->items);
      }
      // el método add() solo acepta instancias de Auto
      public function add(Auto $obj) {
            $this->items[] = $obj;
      }
}
```

Y ahora estamos en condiciones de hidratar los Autos existentes en el repositorio:

```
$p = new Persistent(); // instanciamos al ORM
$c = new CollectionAuto();  // instanciamos la colección
$p->retrieveCollection('Auto', $c);  // hidrata los Autos y los agrega a $c
```

```
foreach ($c as $auto) { // recorre los autos hidratados
	var_dump($auto);
}
```

El método `retrieveCollection()` devuelve la cantidad de Objetos existentes en el repositorio, e hidrata la colección paginada, devolviendo por default la primer página y los primeros 10 objetos. Para devolver la página 3 en grupos de 20, deberíamos escribir:

```
$total = $p->retrieveCollection('Auto', $c, 3, 20);
```

También es posible ordenar la Colección que se va a devolver por los valores en un atributo y agregar diferentes filtros para obtener un resultad parcial o una búsqueda sobre la misma:

```
$p->setCollectionOrder('patente'); // ordenado por la patente
$p->addCollectionFilter('carga', 200, '>'); // carga mayor a 200
$p->retrieveCollection('Auto', $c);  // hidrata los Autos y los agrega a $c
```

Configuración

El mapeo del ORM se realiza mediante un archivo XML, en dónde se definen sus reglas. Sin embargo, se contemplan cierta configuración con defaults razonables que simplifica mucho el mapeo, además de que el propio ORM tiene un método que, recibiendo como mensaje la instancia de un objeto o el nombre de una clase, genera un XML para usar de base.

Los defaults que se contemplan son que los getters y setters se llamen getNombreAtributo y setNombreAtributo, que el mapeo sea clase a tabla, que la clase se llame igual que la tabla y que el atributo que identifique unívocamente al Objeto (ObjectId) se llame se llame 'id'.

Por lo tanto, si para el ejemplo actual de la clase Auto tuviéramos una tabla que se llama Automoviles con los campos 'patente' y 'carga', el mapeo correspondiente sería:

```
<?xml version="1.0" encoding="UTF-8"?>
<Persistent>
	<Auto>
		<Table>
			<Name>Automoviles</Name>
		</Table>
		<ObjectId>patente</ObjectId>
	</Auto>
</Persistent>
```

Si la estrategia de mapeo no fuera “clase a tabla” es posible mapear a una consulta SQL la proyección deseada para hidratar y persistir a los objetos.

A su vez, las colecciones son un Objeto, con lo que, si necesitamos tener un totalizador, por ejemplo de la carga promedio de todos los Autos en el repositorio, simplemente definimos un mapeo para la Colección en la que se mapean atributos a resultados de consultas del tipo *SELECT AVG(carga) FROM Automoviles* y a partir de ese momento, al crear una colección, se ejecuta la consulta y se hidratan los atributos de la misma, manteniendo así la potencia de los RDBMS para acceder a los datos y realizar cálculos.

```
<?xml version="1.0" encoding="UTF-8"?>
<Persistent>
	<Auto>
		<Table>
			<Name>Automoviles</Name>
		</Table>
		<ObjectId>patente</ObjectId>
	</Auto>
	<Collection>
		<cargaPromedio>
		<query>SELECT AGV(carga) AS cargaPromedio FROM Automoviles</query>
		</cargaPromedio>
	</Collection>
</Persistent>
```

Y el código de la colección simplemente cambiaría a:

```
class CollectionAuto implements IteratorAggregate {
	private $items = array();
	private $cargaPromedio = 0; // atributo hidratado por el ORM
	// Métodos
	public function getIterator() {
		return new ArrayIterator($this->items);
	}
	// el método add() solo acepta instancias de Auto
	public function add(Auto $obj) {
		$this->items[] = $obj;
	}
	// Setters y Getters para hidratar atributos de la colección mapeados
	function setCargaPromedio($v) { $this->cargaPromedio = $v; }
	function getCargaPromedio() { return $this->cargaPromedio; }
}
```

9. Persistencia por Alcance

La persistencia por alcance consiste en sobrepasar la limitación de los "Objetos Planos" (POJO, *Plain Old Java Object*), para que si el Modelo lo requiere se puedan implementar Objetos que tengan como atributos a otros Objetos, y sea el ORM el encargado de hidratar y persistir a estas estructuras más complejas.

En POO existen dos formas de asociar a un Objeto con sus componentes, y dependen del ciclo de

vida de dichos Objetos. Son la Agregación y la Composición; en la primera, el Objeto parte sigue existiendo si su contenedor deja de existir, en cambio en la Composición, al destruir el Objeto contenedor, sus partes también se destruyen.

Nuestro ORM permite manejar ambos tipos de asociaciones y también su cardinalidad 1-1 o 1-N. Por ejemplo para un Objeto Localidad que contenga a un Objeto Provincia:

```
class Provincia {
	private $nom;
	public function __construct ($nom='') {
		$this->setNom($nom);
	}
	public function setNom ($v) { $this->nom = $v; }
	public function getNom() { return $this->nom; }
}
class Localidad {
	private $nom;
	private $provincia; // Instancia de Provincia
	public function __construct ($nom='', Provincia $provincia = NULL) {
		$this->setNom($nom);
		$this->setProvincia($provincia);
	}
	public function setNom ($v) { $this->nom = $v; }
	public function getNom() { return $this->nom; }
	public function setProvincia ($v) { $this->provincia = $v; }
	public function getProvincia() { return $this->provincia; }
}
```

Y el mapeo correspondiente para una Agregación:

```
<Persistent>
	<Provincia>
		<Table><Name>Provincias</Name></Table>
		<ObjectId>nom</ObjectId>
	</Provincia>
	<Localidad>
		<Table><Name>Localidades</Name></Table>
		<ObjectId>nom</ObjectId>
		<Attributes>
			<provincia>
				<field>id_pro</field>
				<relation type="agregation">1-1</relation>
				<reference>Provincia</reference>
			</provincia>
		</Attributes>
	</Localidad>
```

En dónde se ve que el atributo “provincia” de la clase *Localidad* está referenciado a la clase *Provincia*, con una relación de Agregación 1-1 a través del campo llamado *id_pro*.

Si en cambio la asociación entre los objetos fuera una Composición el atributo se mapearía indicando:

```
<Attributes>
      <provincia>
            <field>id_pro</field>
            <relation type="composition">1-1</relation>
            <reference>Provincia</reference>
      </provincia>
</Attributes>
```

Y por último si la cardinalidad fuera múltiple se indicaría como *1-N* en vez de *1-1*.

Nótese que la Composición definida en el mapeo debe ir acompañada de los cambios en el código del Modelo para que al destruir el Objeto continente, el contenido también sea destruido, ya que por sus características de no invasivo, el ORM no modifica nunca el comportamiento del Modelo.

10. Conclusiones

Como hemos visto, la persistencia por alcance no invasiva es posible en PHP, y permite mantenernos dentro del paradigma de la POO a la vez que no introducimos en el Modelo características que no se corresponden con el problema que se está resolviendo.

Permite además realizar Testeo de Unidad que ejercite al Modelo sin tener que tener en cuenta la persistencia, ya que efectivamente, no hay en el código del Modelo nada que haga referencia a que un Objeto pueda persistir o no.

Se respeta la diferencia de identidad que debe tener un Objeto de una Colección de Objetos, permitiendo además de esta Colección se implemente libremente, respetando la interfaz *IteratorAgregate* que es parte del lenguaje PHP, con lo que no se introduce ninguna restricción significativa y en cambio de promueve una buena práctica de programación.

Se conserva la potencia de los RDBMS sobre todo a la hora de hacer cálculos y de usar funciones de agregación como *SUM()*, *COUNT()*, *MIN()*, *MAX()* y *AVG()*, soportando además vía PDO, que es el mecanismo que usa nuestro ORM para acceder a la Base de Datos, una amplia gama de motores disponibles, contribuyendo a la abstracción que el Modelo debe hacer del almacenamiento.

Fue probado en software que está actualmente en producción, pero lo que se pudo ver en la práctica al ORM sometido a una carga considerable funcionar sin fallos, con eficacia y eficiencia.

Tiene este tipo de tecnología en general una pequeña penalización de la performance para acceder a los datos. Sin embargo, creemos que lo que se gana en metodología, buenas prácticas y conservación del Modelo supera con creces este inconveniente.

Nuestro Mapeador Objeto-Relacional que implementa peristencia no invasiva y por alcance, es además Software Libre bajo licencia GPL v3 o superior, por lo que contribuimos de esta manera a las buenas prácticas al construir software, aumentando de esta así la calidad del diseño y la programación de aplicaciones PHP.

11. Trabajo Futuro

Hay varias mejoras en la versión actual de nuestro ORM en las que iremos trabajando, siempre teniendo como objetivos la aplicación pura del paradigma de la POO. Algunas de las tareas a realizar son:

Implementar niveles de depuración: actualmente se emite un montón de información de debuggueo o ninguna.

Distribución de responsabilidades: separar del ORM en clases más específicas para separar por ejemplo el procesado del XML de configuración, la generación de SQL, el descubrimiento de la estructura de los objetos a persistir con la API de Reflection, el manejo de filtros y las colecciones.

Soporte de atributos no persistibles: para poder definir en el mapeo por ejemplo que un atributo del Objeto no debe ser persistido, ya que actualmente se intentarán persistir todos los atributos encontrados.

Generar querys parametrizadas: para aprovechar la característica de PDO que simplifica el manejo de comillas y hace más segura la ejecución del SQL.

Ampliar la definición del mapeo: soportando así que una clase tenga sus *INSERT*, *UPDATE* y *DELETE* personalizados, así como actualmente se puede personalizar el *SELECT*.

Implementarlo como un Proxie: usando el Principio de Sustitución de Liskov hacer una implementación con el patrón de diseño Proxie para sacar la necesidad de tener un ObjetctId.

Ponerle un nombre: nuestro proyecto no tiene todavía nombre propio; generalmente durante el tiempo que se trabaja en el código, tarde o temprano uno se empieza a referirse al mismo de alguna manera, pero esta vez no pasó, con lo que contamos con la imaginación de la comunidad del Sofware Libre para nombrar a este proyecto y ponerle un logotipo distintivo.

[Fin]

Referencias:

- Design Patterns: Elements of Reusable Object-Oriented Software – Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
- Design Object-Oriented Software – Rebecca Wirf-Brock
- Modelo Anémico - Martin Fowler http://martinfowler.com/bliki/AnemicDomainModel.html
- POJO – Martin Fowler - http://martinfowler.com/bliki/POJO.html
- Simple Smalltalk Testing: With Patterns – Kent Beck
- PHPUnit: Framework para Testeo de Unidad - http://www.phpunit.de/manual/current/en/
- Data Abstraction and Hierarchy – Bárbara Liskov