ÍNDICE

CAPÍTULO 5. PATRONES DE DISEÑO 3

5.1 Patrones de diseño. 3
5.1.1 Introducción y definición. 4
5.1.2 Tipología y clasificación. 4
5.1.3 Consecuencias: Ventajas y desventajas. 7
5.1.4 Antipatrones. 7
5.2 El patrón Data Access Object. 8
5.2.1 Funcionamiento general. 8
5.2.2 Automatización de la generación DAO. 10
5.2.3 Consecuencias: Ventajas y desventajas. 12
5.3 El paquete PEAR:DB_DataObject. 13
5.3.1 El fichero de configuración/inicialización. 13
5.3.2 Autogeneración y esquema de la base de datos. 15
5.3.3 Fichero de enlace de tablas. 17
5.3.4 Procedimiento de autogeneración de esquemas y clases. 18
5.3.5 Métodos de la clase base. 18
5.3.6 Métodos de las clases extendidas. 18
5.3.7 Conversión de tipos: fechas, blobs y valor nulo 20
5.3.8 Métodos sobrecargables de las clases extendidas. 21
5.3.9 Ejemplos de uso de las clases extendidas. 22
5.3.10 Notas y aclaraciones. 23

CAPÍTULO 5. PATRONES DE DISEÑO

Tanto analistas como programadores lidiamos a diario con problemas usuales en el proceso de desarrollo de software. Para cada uno de estos problemas escogemos la solución más apropiada, que en la mayoría de ocasiones fueron a su vez soluciones para otros problemas similares en el pasado.

Conforme un programador o analista adquiere suficiente experiencia, este tipo de problemas, junto con sus soluciones, se producen con mayor frecuencia. De este hecho, podemos extraer un conjunto de problemas a los que se aplica una misma solución, de manera que, si documentamos dicha solución e ideamos una solución genérica, dispondremos de un mecanismo de resolución que podrá ser reutilizado con posterioridad.

De esta idea surgen los patrones de diseño, cuyo objetivo es proporcionar soluciones exitosas a problemas habituales.

5.1 Patrones de diseño.

Aunque existen multitud de definiciones, un patrón de diseño puede considerarse una solución a un problema de diseño no trivial que es efectiva y reusable.

Dentro de esta definición, se destaca que la solución debe ser **efectiva**. Es decir, que dicha solución ha sido probada con anterioridad, y que es **reusable**, por lo que dicha solución puede aplicarse a otros problemas distintos.

El auge de los patrones de diseño surgió a partir de la publicación del libro *Design Patterns: Elements of Reusable Object Oriented Software*, escrito por el *Gang of Four*, compuesto por Erich Gamma, Richard

Helm, Ralph Johnson y John Vlisides, en el que se recogían 23 patrones comunes.

5.1.1 Introducción y definición.

Un patrón de diseño es:

- Una solución estándar para un problema común de programación.
- Una abstracción de una solución en un nivel alto
- Una técnica para flexibilizar el código haciéndolo satisfacer ciertos criterios.
- Un proyecto o estructura de implementación que logra una finalidad determinada.
- Una manera práctica de describir ciertos aspectos de la organización de un programa.
- Conexiones entre componentes de programas.
- La forma de un diagrama de objeto o de un modelo de objeto.

Los patrones de diseño son una forma muy útil de reutilizar diseño ya que, además de abstraer e identificar aspectos claves de estructuras comunes de diseño, son documentados con gran precisión, haciendo sencilla su comprensión y aplicación para los desarrolladores.

5.1.2 Tipología y clasificación.

De manera clásica, los patrones de diseño se han clasificado en una serie de categorías:

- ***Creacionales***: tratan con la problemática propia de la creación de instancias de objetos. Su objetivo es abstraer el proceso de instanciación, ocultando sus detalles.
    - *Abstract Factory*. Proporciona una interfaz para crear familias de objetos o que dependen entre sí, sin especificar sus clases concretas.
    - *Builder*. Separa la construcción de un objeto complejo de su representación, de forma que el mismo proceso de construcción pueda crear diferentes representaciones.
    - *Factory Method*. 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.

- *Prototype*. Especifica los tipos de objetos a crear por medio de una instancia prototípica, y crear nuevos objetos copiando este prototipo.

- *Singleton*. Garantiza que una clase sólo tenga una instancia, y proporciona un punto de acceso global a ella.

- ***Estructurales***: describen como las clases y objetos pueden ser combinados para formar grandes estructuras y proporcionar nuevas funcionalidades.

  - *Adapter*. 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 sea más fácil de usar.

  - *Flyweight*. Usa el compartimiento para permitir un gran número de objetos de grano fino de forma eficiente.

  - *Proxy*. Proporciona un sustituto o representante de otro objeto para controlar el acceso a éste.

- ***Comportamiento***: ayudan a definir la comunicación e interacción entre los objetos de un sistema. Su propósito es reducir el acoplamiento entre 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.

- *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 las operaciones.

- *Interpreter*. Dado un lenguaje, define una representación de su gramática junto con un intérprete que usa dicha representación para interpretar las sentencias del lenguaje.

- *Iterator*. Proporciona un modo de acceder secuencialmente a los elementos de un objeto agregado sin exponer su representación interna.

- *Mediator*. Define un objeto que encapsula cómo interactúan un conjunto de objetos. Promueve un bajo acoplamiento al evitar que los objetos se refieran unos a otros explícitamente, y permite variar la interacción entre ellos de forma independiente.

- *Memento*. Representa y externaliza el estado interno de un objeto sin violar la encapsulación, de forma que éste puede volver a dicho estado más tarde.

- *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.

- *State*. Permite que un objeto modifique su comportamiento cada vez que cambia su estado interno. Parecerá que cambia la clase del objeto.

- *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.

- *Template Method*. Define en una operación el esqueleto de un algoritmo, delegando en las subclases algunos de sus pasos. Permite que las subclases redefinan ciertos pasos del algoritmo sin cambiar su estructura.

- *Visitor*. Representa una operación sobre los elementos de una estructura de objetos. Permite definir una nueva operación sin cambiar las clases de los elementos sobre los que opera.

Aunque ésta es la distribución original de los primeros patrones descritos, en la actualidad existe un mayor número de patrones, y de categorías como por ejemplo los patrones de interacción.

5.1.3 Consecuencias: Ventajas y desventajas.

Entre los beneficios propios del uso de los patrones podemos encontrar:

- Reutilización de código. Se reducen los esfuerzos de desarrollo y mantenimiento, mejora la seguridad, eficiencia y consistencia de los diseños, ahorrando esfuerzos.
- Mejoran la flexibilidad, modularidad y extensibilidad.
- Aportan un gran nivel de abstracción.

El principal problema derivado del uso y análisis de patrones es su excesiva utilización, ya que en demasiadas ocasiones ocurre que se aplica la resolución mediante patrones sin el adecuado análisis del problema, lo que ocasiona un aumento de complejidad en el software sin haber aportado una solución real.

5.1.4 Antipatrones.

En contraposición a los patrones, que nos ofrecen una forma de resolver un problema típico, los antipatrones nos enseñan formas de enfrentarse a problemas con consecuencias negativas conocidas.

Los antipatrones se basan en la idea de que, ante un problema, puede resultar más fácil detectar fallos en una solución, y descartar los caminos erróneos, que encontrar directamente la solución apropiada, reduciendo de este modo el tiempo invertido para encontrar la mejor alternativa.

Los antipatrones pueden clasificarse en:

- **Antipatrones de desarrollo**: son aquellos patrones encontrados a bajo nivel del desarrollo del software, específicamente en programación y administración del código.
- **Antipatrones de arquitectura del software**: involucra problemas en la definición y mantenimiento en las estructuras de un sistema.

- **Antipatrones de gestión de proyectos**: está relacionado con la administración de los proyectos software, y el personal involucrado.

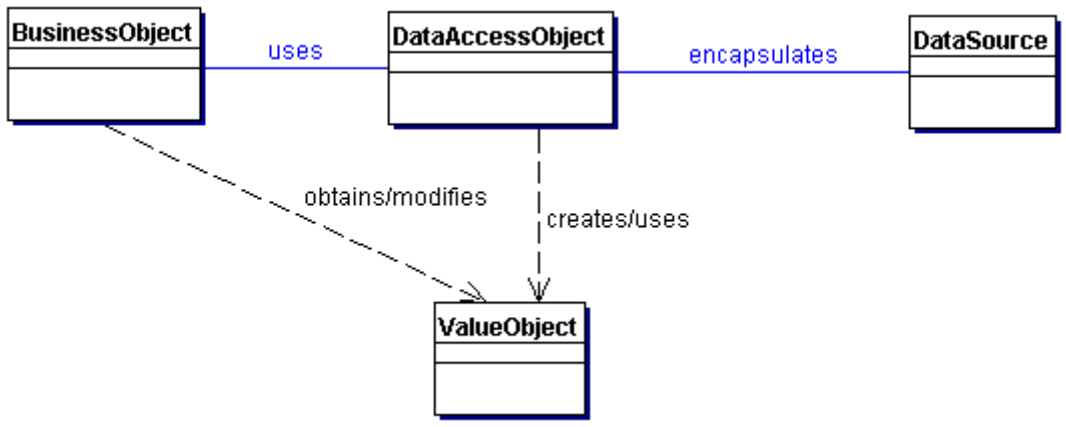
5.2 El patrón Data Access Object.

El objetivo del patrón *Data Access Object*, o en castellano *Objeto de Acceso a Datos*, en adelante *DAO*, es desacoplar la lógica de negocios de la lógica de acceso a datos, de manera que se pueda cambiar la fuente de datos fácilmente.

El beneficio inmediato al independizar el código de la lógica de negocio del código de acceso al origen de datos es la minimización de los costes de desarrollo al realizar cambios en el origen de datos. Al desacoplar la lógica de negocios de la lógica de acceso a datos, la sustitución de un origen de datos a otro distinto, como podría ser el caso de sustituir *mySQL* por *PostgreSQL*, no repercutiría sobre la lógica de negocios por lo que los únicos cambios a realizar serían el desarrollo de las librerías de acceso a *PostgreSQL*, y enlazarlas con el patrón *DAO*.

5.2.1 Funcionamiento general.

En el siguiente diagrama de clases podemos ver las distintas clases que participan en el patrón, y las relaciones existentes entre ellas.



Las clases participantes en este patrón son:

- **Business Object**. Representa los datos del cliente. Es el objeto que requiere el acceso a la fuente de datos para obtener y almacenar datos.

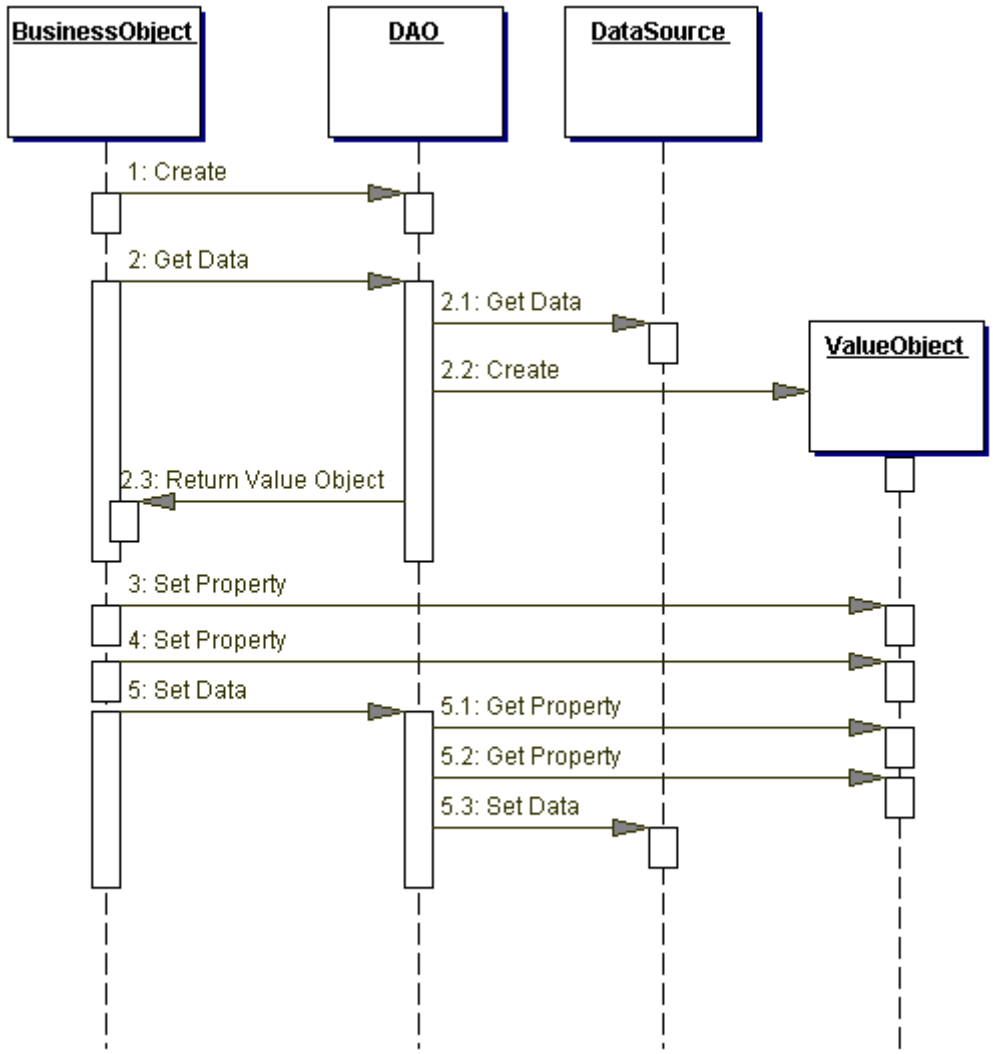
- **Data Access Object.** Es el principal objeto del patrón. Abstrae la implementación del acceso a datos subyacente a

businessObject permitiéndole un acceso transparente a la fuente de datos.

- **Data Source.** Representa a la propia fuente de datos. Al contrario que en otras generalizaciones, la fuente de datos no se ciñe a distintas bases de datos, sino que puede tratarse de otros sistemas, servicios o repositorios, como LDAP.

- **Value Object.** Más conocido como Transfer Object, u objeto de transferencia, se puede utilizar, opcionalmente, para encapsular y transportar la información entre distintas capas.

La siguiente figura muestra una posible secuencia de llamadas utilizando un *transfer object* para encapsular la información del origen de datos.



5.2.2 Automatización de la generación DAO.

Partiendo de la utilización del patrón *DAO*, y disponiendo de una base en el análisis y diseño de software podemos apreciar la equivalencia o similitud entre la lógica de negocios y la capa de datos. Al emplear el patrón *DAO* esta equivalencia se refleja en la similitud entre los objetos *BussinessObject* y los *TransferObject*. Debido a la equivalencia entre ambos objetos nos es posible la creación de herramientas semiautomáticas que nos ayuden en desarrollo del patrón *DAO*.

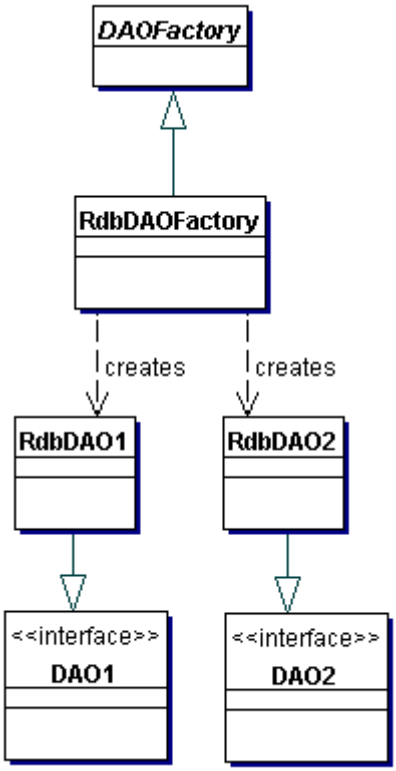
Por lo general, este tipo de herramientas permite a partir de una definición de datos desarrollar el código que relaciona ambas capas.

Llegados a este punto, disponemos de un mecanismo para obtener código uniforme para el acceso a datos, y el desarrollo del patrón *DAO*. El patrón DAO puede flexibilizarse mediante el uso de los patrones *Abstract Factory* y *Factory Method*.

5.2.2.1 El patrón Abstract Factory.

El objetivo del patrón *Factory Method*, o *Método de Fabricación*, es definir una interfaz para crear objetos, dejando a las subclases decidir la clase específica de objeto a crear, permitiendo delegar la responsabilidad de la instanciación a las subclases.

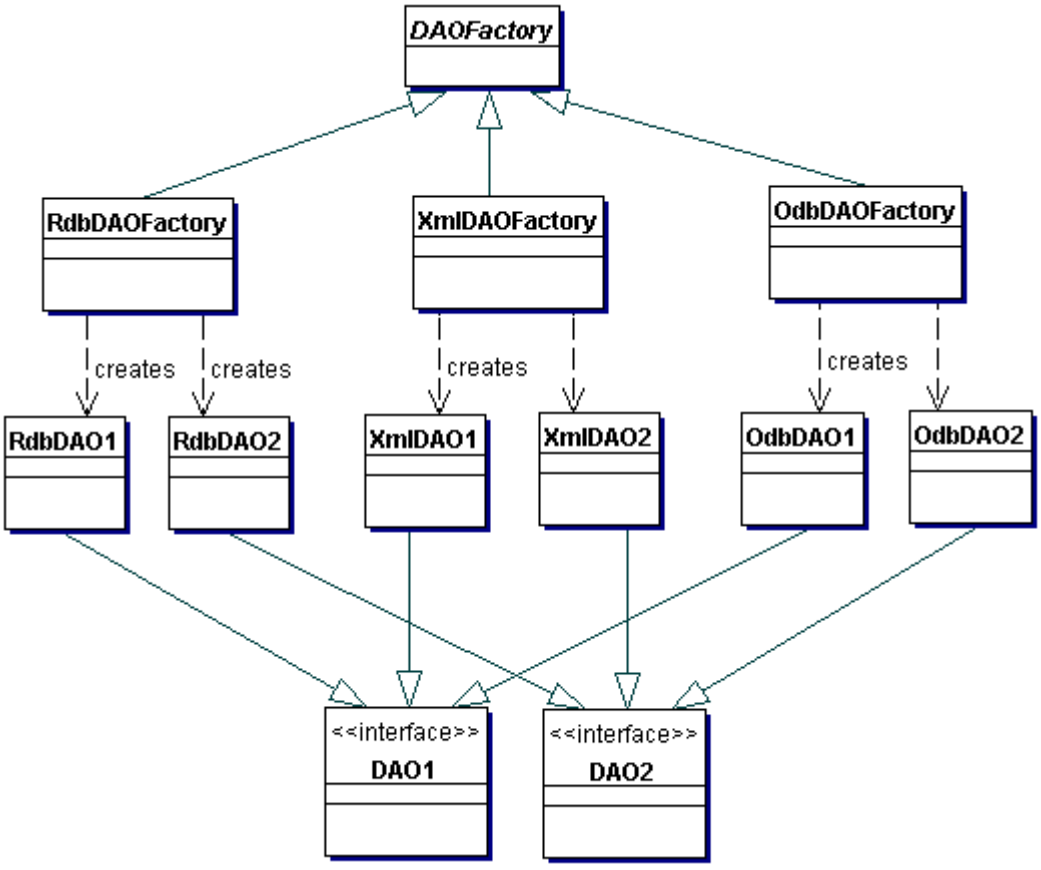
Mediante el uso de este patrón disponemos de distintos métodos para la creación de cada uno de los *DAOs*.



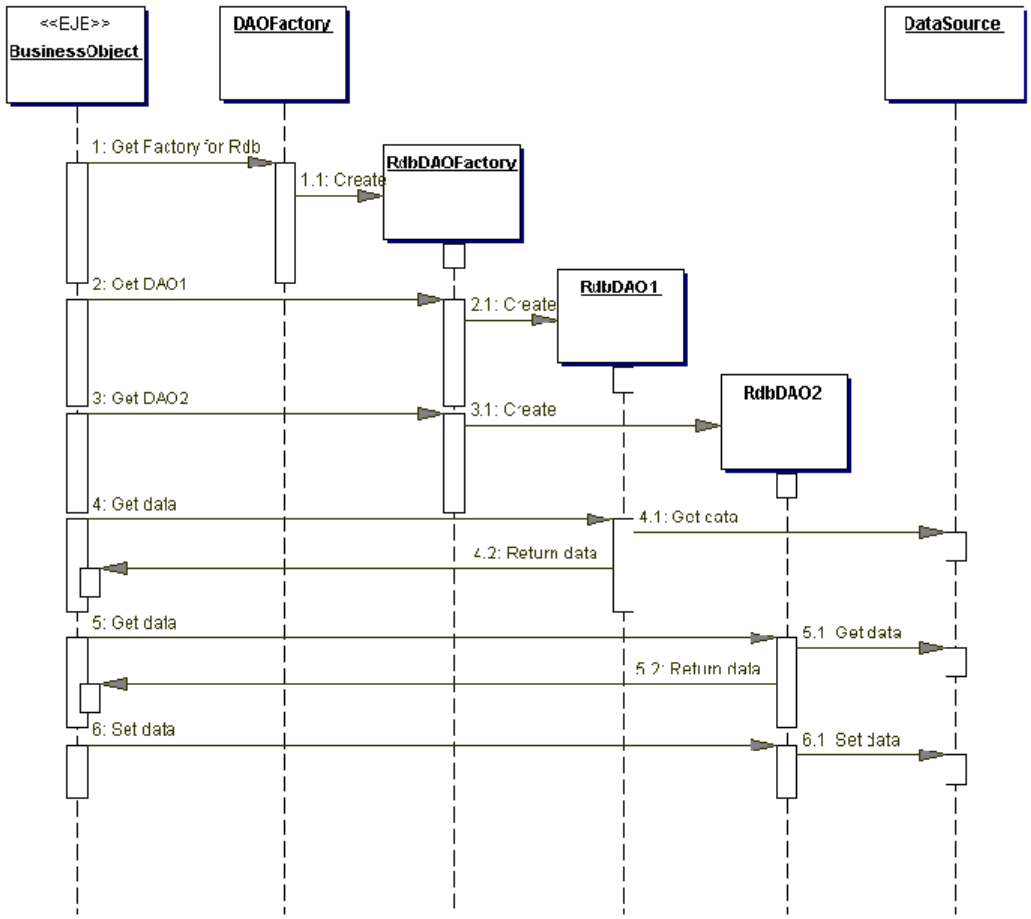
En segundo lugar, podemos emplear el patrón *Abstract Factory*. El problema que intenta solucionar el patrón *Abstract Factory*, o en castellano *Fábrica Abstracta*, es la creación de diferentes familias de objetos. Es recomendable su uso cuando se prevé la inclusión de nuevas familias de productos y puede resultar contraproducente su adición, o la modificación de las familias ya existentes.

Mediante la combinación de estas estrategias disponemos de un sistema *DAO* en el que se pueden añadir o sustituir el acceso a distintas bases de datos con un coste reducido y un impacto mínimo.

De este modo, y como se aprecia en la siguiente figura, el acceso a datos se realiza a través de una figura única, el *DAOFactory*, encargado de crear una fábrica *DAO* específica para el origen de datos especificado, y como cada fábrica *DAO* específica está encargada de crear los objetos *DAO*, que mediante la inclusión de interfaces unificados permiten la separación entre la capas de lógica de negocios y la capa de acceso a datos. Cada *DAO* concreto es responsable de conectar con la fuente de datos y de obtener y manipular los datos para el objeto de negocio que soporta.



Finalmente, podemos observar el diagrama de secuencia para esta estrategia:



5.2.3 Consecuencias: Ventajas y desventajas.

Las consecuencias más relevantes del uso del patrón *DAO* son:

- **Transparencia**, los objetos de negocio pueden utilizar la fuente de datos sin conocer los detalles específicos de su implementación.

- **Facilita la migración**, al realizar un desacoplamiento de capas la migración afecta únicamente a la capa *DAO*. Si además empleamos un sistema de fábricas, únicamente es necesario el desarrollo de una nueva familia de acceso a datos, o dicho de otra manera, de una nueva fabrica.

- **Reduce la complejidad de la capa de negocios**, al excluir el acceso a datos de la capa de negocios se reduce simplifica el código de éste, así como el de cualquier otro servicio que empleará el acceso a datos. Además, mejora la lectura del código y la productividad del desarrollo.

- **Centraliza los accesos a datos en una capa independiente**, al reunir todas las operaciones de acceso a datos en los *DAOs*. Esta centralización hace que el software sea más sencillo de

mantener y manejar, pero a cambio se paga con un esfuerzo adicional para la creación y mantenimiento de esta nueva capa.

- **Necesidad de diseñar un árbol de clases**, al utilizar una estrategia de fábricas necesitamos diseñar e implementar el árbol de fábricas, y el árbol de productos concretos producidos por cada fábrica, lo que incrementa la complejidad del diseño.

- **Aumento de la complejidad**. Al igual que la adición de cualquier nueva funcionalidad, el uso de Data Objects produce un aumento en la complejidad del desarrollo, así como una mayor inversión inicial en el desarrollo que para proyectos de pequeñas dimensiones puede no verse compensado.

5.3 El paquete PEAR:DB_DataObject.

El paquete *DB_DataObject* es una capa de modelado de datos construida, inicialmente, sobre *PEAR:DB*. Los principales propósitos de dicho paquete son:

- Construir sentencias SQL basadas en los atributos de los objetos, y ejecutarlas.

- Proporcionar una API sencilla y consistente para acceder y manipular los datos.

Explicado de una manera más generalista, el paquete *DB_DataObject* establece una equivalencia entre algunos objetos de la capa de negocio, y los datos almacenados en la base de datos. De esta manera, es posible simplificar el intercambio de información entre las capas de negocio y el almacenamiento de datos.

5.3.1 El fichero de configuración/inicialización.

Aunque *DB_DataObject* es capaz de autogenerar las clases equivalentes a la base de datos, es necesario configurarlo previamente. El mecanismo más sencillo para realizar dicha configuración es mediante el uso de ficheros ***ini***.

En el fichero de configuración podemos encontrar varios tipos de opciones de configuración:

- Obligatorias.

- Opcionales.

- De múltiples bases de datos: permiten definir el acceso a diversas bases de datos.
- De generación: permiten definir el modo en que se hará la autogeneración de clases y esquemas.

Entre las opciones obligatorias podemos encontrar:

TÉRMINO	DESCRIPCIÓN
database <DSN>	Cadena DSN a utilizar para acceder a la base de datos.
schema_location <dir>	Directorio donde se encuentra el fichero del esquema de la BD.
require_prefix <dir>	Ruta absoluta o relativa donde se encuentra la ruta de inclusión por defecto. En esta ruta se buscan las clases extendidas. Lo utilizan los métodos *staticGet()* y *getLinks()*.
class_prefix <string>	Prefijo a aplicar a las clases DAO generadas.

Los parámetros opcionales son:

TÉRMINO	DESCRIPCIÓN
*sequence_*{table} <string>	Obliga a establece un campo determinado como clave primaria de la tabla.
ignore_sequence_keys <string>	Evita el uso de la función *nextval()* en las tablas indicadas.
debug <integer>	Nivel de debug. 0=off, 1=log sql básico, 2=log de resultados y 3=todo
debug_ignore_updates <boolean>	A falso ignora las acciones de actualización. (Def: FALSE)
dont_die <boolean>	Evita la interrupción inmediata por causa de error. Permite capturar los errores ocurridos (Def: FALSE)
quote_identifiers <boolean>	Con valor '1' obliga a introducir nombres de campos y tablas entre comillas simples.
proxy <string>	Con valor 'full' genera los esquemas y las clases durante la misma ejecución.

Las opciones para utilizar múltiples bases de datos son:

TÉRMINO	DESCRIPCIÓN
*database_** <string>	Permite establecer una cadena *DSN* para cada base de datos.
*table_** <string>	Permite mapear tablas individuales con diferentes bases de datos.

Los parámetros de generación son:

TÉRMINO	DESCRIPCIÓN
class_location <dir>	Directorio donde se encuentran las clases. Usado en la autogeneración de clases.
extends <string>	Nombre de la clase base para la generación. Normalmente: *DB_DataObject*.
extends_location <dir>	Directorio donde se encuentra la clase base.
generator_class_rewrite <*b*oolean>	Al activarse, la regeneración de clases afecta a los campos extendidos de las clases.
build_views <*b*oolean>	En algunos *SGBDs*, permite acceder a las vistas como si se tratara de tablas.
generator_include_regex <string>	Sólo se generan las tablas que coinciden con la expresión regular indicada.
generator_exclude_regex <string>	No se generan las tablas que coinciden con la expresión regular indicada.

Un fichero de configuración/inicialización básico con las principales opciones sería como el que se muestra a continuación:

```
[DB_DataObject]
database        = mysql://user:password@localhost/nombre_bd
schema_location = C:\proyecto\dao
class_location  = C:\proyecto\dao
require_prefix  = C:\proyecto\dao
class_prefix    = dao_
```

En este fichero se establecen los parámetros obligatorios, así como las opciones de generación automática. Así pues, se especifica la conexión con la base de datos, las rutas en las que se encuentran los DataObjects del proyecto, los esquemas de clases y donde se crearán los anteriores si utilizamos la autogeneración. Además especificamos que los nombres de los DataObjects irán precedidos del prefijo "dao_".

5.3.2 Autogeneración y esquema de la base de datos.

El paquete *DB_DataObject* emplea dos ficheros para establecer la equivalencia entre las clases y las tablas. Estos ficheros son: el fichero de definición de clase, y el fichero del esquema de la base de datos.

5.3.2.1 Fichero de definición de clases.

La definición de la clase es, en realidad, una clase que hereda, directa o indirectamente, de la clase base DB_*DataObject*. En dicha clase se almacenan los atributos de la clase, que corresponden a columnas de la tabla asociada, así como un método que participa en la creación

de instancias de dicha clase. La estructura de una de estas clases podría ser la siguiente:

```php
<?php
/*
* Table Definition for group
*/

class DataObjects_Grp extends DB_DataObject {

  ###START_AUTOCODE
  /* the code below is auto generated do not remove the above tag */
  var $__table='group';                // table name
  var $id;                             // int primary_key
  var $name;                           // string
  var $grp_owner;                      // int
  var $official;                       // string

  /* Static get */
  function staticGet($k,$v=NULL) {
    return DB_DataObject::staticGet('DataObjects_Grp',$k,$v);
  }

  /* the code above is auto generated do not remove the tag below */
  ###END_AUTOCODE
}
?>
```

Tal y como se observa en el código, existe una zona delimitada entre las etiquetas "###START_AUTOCODE" y "###END_AUTOCODE" que contiene código autogenerado. Por este motivo, cualquier adición de código que se efectúe, deberá realizarse en el exterior de esta zona para que futuros cambios sobre la base de datos no corrompan las modificaciones manuales.

5.3.2.2 El fichero de esquema de la base de datos.

En segundo lugar, tenemos el fichero de esquema. Este fichero debe ubicarse en la ruta señalada por el parámetro "schema_location", y deberá tener el mismo nombre que la base de datos a la que represente.

Dicho fichero deberá generarse de nuevo cada vez que la base de datos asociada sufra una modificación en su estructura o en su nombre. Aunque este tipo de ficheros **no** deben ser modificados a mano, ya que la regeneración eliminaría dichas modificaciones, es posible realizar modificaciones en la definición de claves ajenas mediante dos mecanismos. El primero es mediante el uso de la opción de configuración "sequence_{table} = key", y la segunda mediante la definición del método *sequenceKey()* en la propia clase afectada.

A continuación, podemos observar un fichero de esquema de ejemplo, en él se aprecia la estructura general del mismo:

```
[group]
id = 129
name = 130
grp_owner = 129
official = 130

[group__keys]
id = N
```

Aunque existe una alternativa al uso de los ficheros de esquema, dicha alternativa es completamente manual por lo que no interesa a los objetivos de este curso. Para quien pueda estar interesado, dejamos un enlace a dicha información.

5.3.3 Fichero de enlace de tablas.

A partir de la versión 0.3 de *DB_DataObject* se decidió profundizar en la capacidad de mapear columnas entre tablas, o lo que es lo mismo, profundizar en la funcionalidad aportada por las claves ajenas. Por ello se optó por permitir la creación de un fichero que almacenara estas relaciones entre tablas.

De este modo, al igual que disponemos del fichero del esquema cuyo nombre coincide con el nombre de la propia base de datos, "*base_de_datos.ini*", disponemos de un segundo fichero, que deberá ubicarse en la misma ruta que en anterior, y cuyo nombre será "*base_de_datos.links.ini*".

A continuación se muestra un ejemplo del contenido de dicho fichero:

```
[person]
eyecolor = colors:name
owner = grp:id
picture = attachments:id

[sales]
car_id = car:id
car_id.partnum = part_numbers:car_id
```

Por ejemplo, la línea que hace referencia al atributo *eyecolor* debe interpretarse como que existe una clave ajena del campo *eyecolor* de la tabla *person* hacia el campo *name* de la tabla *colors*, representada en *SQL* mediante la siguiente instrucción:

ALTER TABLE ***person***

*ADD FOREIGN KEY (****eyecolor****) REFERENCES* ***colors****(****name****)*

Como se puede deducir del ejemplo anterior, este fichero dispone de una sección para cada tabla. Dentro de la sección de la tabla podemos especificar la relación que existe entre una columna de la tabla especificada con una columna de una tabla externa. Es decir, cada línea establece una clave ajena.

Mediante este fichero podemos hacer uso de método como *getLink()*, *getLinks()*, *joinAdd()* o incluso *selectAs()* para utilizar y/o acceder a estas relaciones, o modificar alguno de sus valores por defecto.

5.3.4 Procedimiento de autogeneración de esquemas y clases.

Este procedimiento utiliza como herramienta principal el fichero de script "*createTables.php*" ubicado en el subdirectorio de *PEAR* "DB\DataObject".

Dicho script se basa en la información contenida en un fichero de configuración/inicialización, para acceder a la base de datos y crear los ficheros siguiendo las especificaciones de dicho fichero de configuración.

Para lanzar la autogeneración debemos ubicarnos en el mismo directorio en que se encuentra el script "*createTables.php*" y, teniendo en cuenta las ubicaciones de los distintos ficheros, adaptaríamos el siguiente comando:

php.exe createTables.php myconfig.ini

Una vez lanzado el comando, y si todo funciona correctamente, podremos observar que se han creado diversos ficheros en las carpetas indicadas.

5.3.5 Métodos de la clase base.

Los métodos principales de la clase *DB_DataObject* son los siguientes:

MÉTODO	DEVUELVE	DESCRIPCIÓN
factory	Mixed / false	Instancia una clase a partir del nombre de tabla
staticGet	Mixed / false	Mezcla de *factory()* y *get()*. Carga la clase, y devuelve la instancia con el *key* indicado.
debugLevel		Establece el nivel de debug. Es decir, el detalle del log.
raiseError	Object	Lanza el error especificado.

5.3.6 Métodos de las clases extendidas.

Aunque oficialmente no existe ninguna clasificación sobre los métodos de las clases extendidas de *DB_DataObject*, vamos a distribuirlas en distintos grupos funcionales con el objetivo de establecer algún orden.

En la siguiente tabla podemos encontrar los métodos básicos de operación:

MÉTODO	DEVUELVE	DESCRIPCIÓN
staticGet	Mixed / false	Devuelve la instancia con el *key* indicado. El tipo de objeto lo extrae de la propia clase.
get	Nº filas	Obtiene un resultado interno con las tuplas que coinciden con la clave / premisa establecida.
find	Nº filas / 1/ true	Ejecuta la consulta, y almacena internamente el resultado obtenido.
fetch	true / false	Extrae el siguiente elemento del resultado.
count	Nº filas / false	Cuenta el número de filas resultantes de la petición.
insert	ID / key	Inserta la información del objeto actual en la *BD*.
update	Nº filas / false	Actualiza la información del objeto actual sobre la *BD*.
delete	Nº filas / false	Elimina de la *BD* usando la clave o la cláusula Where asociada.
query		Establece una consulta *SQL*.
free		Libera los recursos de los resultados (tras get/find/query).

A continuación tenemos los métodos de enlace entre tablas:

MÉTODO	DEVUELVE	DESCRIPCIÓN
getLink	Mixed / false	Devuelve el objeto asociado a una columna del estilo clave ajena.
GetLinks	True / false	Carga todos los objetos asociados con el objeto actual.
joinAdd		Realiza, internamente, un *join* con el objeto indicado.

También dispondríamos de modificadores sobre la acción:

MÉTODO	DEVUELVE	DESCRIPCIÓN
limit		Permite establecer límites a las consultas.
selectAdd		Añade la selección de una columna. Por defecto se devuelven todas las columnas (*). Al invocar la función sin parámetros se eliminan todas las columnas.
whereAdd		Añade una cláusula Where a la petición pendiente.
escape		Escapa una cadena para su uso en sentencias de tipo LIKE.
orderBy		Establece el orden de los resultados. Sin argumentos cancela.
groupBy		Establece grupos. Sin argumentos cancela.
selectAs		Realiza un renombrado de las columnas o tablas indicadas.

Métodos de acceso, transferencia y validación de los datos contenidos en el objeto:

MÉTODO	DEVUELVE	DESCRIPCIÓN
set* / get*		Permiten asignar o comprobar el valor de un determinado atributo. Pueden ser sobrecargados por el programador en la clase extendida.
setFrom	Array / true	Crea una instancia de la clase extendida a partir de vector.
toArray	Array	Crea un array a partir de los datos de la instancia actual.
validate	Array / true	Comprueba la validez de los datos de la instancia actual mediante el uso de los métodos *Validate*()*, que haya definido el programador.

Métodos de acceso a la información general de la base de datos asociada:

MÉTODO	DEVUELVE	DESCRIPCIÓN
tableName	String	Devuelve o establece el nombre de tabla de un objeto.
database	String	Devuelve o establece el nombre de base de datos que usa el objeto.
table	Array	Devuelve o establece el esquema de la tabla del objeto.
keys	Array	Devuelve o establece las claves (*keys*) del objeto.

Y, finalmente, los métodos de acceso a información relativa a la propia conexión, y de debug:

MÉTODO	DEVUELVE	DESCRIPCIÓN
getDatabaseConnection	Conexión	Devuelve la conexión a base de datos que usa el objeto.
getDatabaseResult		Devuelve el resultado de la consulta. Puede usarse en conjunción con *Pager* o *HTML_Select*.
debug		Sirve para definir la información de salida de debug.

5.3.7 Conversión de tipos: fechas, blobs y valor nulo.

Aunque se trata de una ***funcionalidad experimental***, vamos a explicar los mecanismos que prevé *DB_DataObject* para la conversión de valores.

Estos mecanismos de conversión de tipos se sustentan en el uso de la clase *DB_DataObject_Cast* cuyos métodos permiten realizar

conversiones específicas. El propósito de este objeto es utilizar de un modo sencillo representantes de tipos de datos poco comunes.

El funcionamiento de esta clase se puede observar en los ejemplos mostrados a continuación:

```
$pers->birthday = DB_DataObject_Cast::date(año, mes, día);
$pers->birthday = DB_DataObject_Cast::date("dia/mes/año");
$pers->birthday = DB_DataObject_Cast::date("año-mes-dia");

$pers->photo = DB_DataObject_Cast::blob(file_get_content(<fichero>));
$pers->xmldocs = DB_DataObject_Cast::string(file_get_content(<fichero>));

$pers->brother = DB_DataObject_Cast::sql('NULL');
```

5.3.8 Métodos sobrecargables de las clases extendidas.

Antes de explicar este punto es importante hacer hincapié en que los métodos explicados en esta sección utilizan, de uno u otro modo, componentes que se encuentran en ***fase experimental***, por lo que su comportamiento puede variar en versiones posteriores.

Las clases extendidas disponen básicamente de dos tipos de métodos sobrecargables. Éstos son los métodos de acceso a la información del objeto, y los métodos de validación.

Los métodos de acceso a la información son del tipo *get*()* y *set*()*, sustituyendo el asterisco por el nombre del campo a acceder, y teniendo en cuenta que el primer carácter pasa a ser una letra mayúscula. Los métodos de validación adquieren el nombre *validate*()*, aplicando la misma teoría en cuanto a nombres de atributos que el anterior.

De este modo, los métodos de acceso a la información permiten realizar ciertas transformaciones a los datos durante su transferencia, como puede ser el cambio de formato en las fechas.

A continuación se muestra un ejemplo de su sobrecarga:

```
class DataObjects_Person extends DB_DataObject {
  var $id;
  var $name;
  var $date_of_birth;

  function getDate_of_birth() {
    return date('d M Y', strtotime($this->date_of_birth));
  }
}
```

Por otro lado, existen los métodos de validación. Estos métodos de validación sirven, entre otras cosas, para dar soporte al método *validate()*, que se ha descrito anteriormente.

El funcionamiento del método *validate()* es el siguiente: al invocarse recorre todos los métodos de validación específicos. Supuestamente existirá un campo de comprobación por cada atributo interno del objeto. Acumulará los errores producidos y los devolverá finalmente en un array, siempre y cuando se haya producido algún error.

Para invocar a los métodos específicos de validación, éstos deberán haber sido definidos previamente por el programador. A continuación se muestran algunos de estos métodos:

```
/* Dentro de la clase DataObject_Person */
function validateEmail() {
  return Validate::email($this->email, true);
}

function validateHomepage() {
  return Validate::url($this->homepage, true);
}

function validateDate_of_birth() {
  return Validate::date($this->date_of_birth, "%d-%m-%Y", array(01,01,1970),
    array(01,01,2030));
}
```

Como se puede observar, todos los métodos invocan a la clase *Validate*. Se trata de una clase ***experimental***, cuyo objetivo es convertirse en un repositorio de comprobaciones sobre diversos tipos de datos. La invocación interna a dicha clase es completamente opcional. Lo que sí que resulta imprescindible es la devolución de una cadena de error en caso de producirse cualquier deficiencia durante la validación del campo.

5.3.9 Ejemplos de uso de las clases extendidas.

5.3.9.1 Filtrado y búsqueda de tuplas.

En el siguiente ejemplo, se calcula cuantas personas pelirrojas hay que llevan gafas, y se almacenan sus datos en un vector informando por pantalla de los nombres de las personas que ya han sido transferidas.

```
$person = new DataObjects_Person;

$person->hair = 'red';
$person->has_glasses = 1;

$number_of_rows = $person->find();

$people = array();
while ($person->fetch()) {
    $people[] = clone($person);
    echo "GOT {$person->name}<BR>";
}
```

5.3.9.2 Selección por clave primaria y modificación.

En el próximo ejemplo se selecciona a la persona cuyo identificador es 12, y se cambia su nombre por Fred.

```
$person = new DataObjects_Person;
$person->get(12);
$person->name='fred';
if ($person->update() == FALSE)
  die('Error en la actualización');
```

5.3.9.3 Acceso a datos enlazados.

En el último ejemplo, se muestra el nombre del grupo al que pertenece una persona.

```
$person = new DataObjects_Person;
$person->get(12);
$group = $person->getLink('group_owner');
echo $group->name;
```

5.3.10 Notas y aclaraciones.

El objetivo de este tema no es exponer el funcionamiento de *DB_DataObject* en su totalidad ya que, para ello, disponemos de la documentación oficial del proyecto, sino dar una visión general sobre las funcionalidades y posibilidades de este paquete.

Para profundizar, así como para documentarse con más ejemplos, se aconseja el estudio de la documentación oficial.