Mapeo Objeto / Relacional (ORM)

Al desarrollar una aplicación siguiendo el paradigma orientado a objetos, los programadores se enfrentan, entre otros desafíos, al problema de la persistencia de los datos, debido a las diferencias que existen entre el modelo relacional y el modelo orientado a objetos. Este problema la mayoría de los casos es solucionado con la ayuda de ciertas herramientas que se encargan de generar de manera automática el acceso a datos, abstrayendo al programador de este problema.

El Modelo Relacional es un modelo de datos basado en la lógica de predicado y en la teoría de conjuntos para la gestión de una base de datos. Siguiendo este modelo se puede construir una base de datos relacional que no es más que un conjunto de una o más tablas estructuradas en registros (filas) y campos (columnas), que se vinculan entre sí por un campo en común. Sin embargo, en el Modelo Orientado a Objetos en una única entidad denominada objeto, se combinan las estructuras de datos con sus comportamientos. En este modelo se destacan conceptos básicos tales como objetos, clases y herencia.

Entre estos dos modelos existe una brecha denominada desajuste por impedancia dada por las diferencias entre uno y otro. Una de las diferencias se debe a que en los sistemas de bases de datos relacionales, los datos siempre se manejan en forma de tablas, formadas por un conjunto de filas o tuplas; mientras que en los entornos orientados a objetos los datos son manipulados como objetos, formados a su vez por objetos y tipos elementales.

Además en el modelo relacional no se puede modelar la herencia que aparece en el modelo orientado a objetos y existen también desajustes en los tipos de datos, ya que los tipos y denotaciones de tipos asumidos por las consultas y lenguajes de programación difieren. Esto concierne a tipos atómicos como integer, real, boolean, etc. La representación de tipos atómicos en lenguajes de programación y en bases de datos pueden ser significativamente diferentes, incluso si los tipos son denotados por la misma palabra reservada, ej.: integer. Esto ocurre también con tipos complejos como las tablas, un tipo de datos básico en SQL ausente en los lenguajes de programación.

Para atenuar los efectos del desajuste por impedancia entre ambos modelos existen varias técnicas y prácticas como los Objetos de Acceso a Datos (Data Acces Objects o DAOs), marcos de trabajo de persistencia (Persistence Frameworks), mapeadores Objeto/Relacionales (Object/Relational Mappers u ORM), consultas nativas (Native Queries), lenguajes integrados como PL-SQL de Oracle y T-SQL de SQL Server; mediadores, repositorios virtuales y bases de datos orientadas a objetos .

Mapeo Objeto/Relacional

El mapeo objeto-relacional es una técnica de programación para convertir datos del sistema de tipos utilizado en un lenguaje de programación orientado a objetos al utilizado en una base de datos relacional. En la práctica esto crea una base de datos virtual orientada a objetos sobre la base de datos relacional. Esto posibilita el uso de las características propias de la orientación a objetos (esencialmente la herencia y el polimorfismo).

Las bases de datos relacionales solo permiten guardar tipos de datos primitivos (enteros, cadenas de texto, etc.) por lo que no se pueden guardar de forma directa los objetos de la aplicación en las tablas, sino que estos se deben de convertir antes en registros, que por lo general afectan a varias tablas. En el momento de volver a recuperar los datos, hay que hacer el proceso contrario, se deben convertir los registros en objetos. Es entonces cuando ORM cobra importancia, ya que se encarga de forma automática de convertir los objetos en registros y viceversa, simulando así tener una base de datos orientada a objetos.

Entre las ventajas que ofrecen los ORM se encuentran: rapidez en el desarrollo, abstracción de la base de datos, reutilización, seguridad, mantenimiento del código, lenguaje propio para realizar las consultas. No obstante los ORM traen consigo algunas desventajas como el tiempo invertido en el aprendizaje. Este tipo de herramientas suelen ser complejas por lo que su correcta utilización requiere un espacio de tiempo a emplear en conocer su funcionamiento adecuado para posteriormente aprovechar todo el partido que se le puede sacar. Otra desventaja es que las aplicaciones suelen ser algo más lentas. Esto es debido a que todas las consultas que se hagan sobre la base de datos, el sistema primero deberá transformarlas al lenguaje propio de la herramienta, luego leer los registros y por último crear los objetos.

En el mapeo objeto-relacional encontramos el uso de algunos patrones de diseño como el Repository y el Active Record.

Patrón Repository

El patrón Repository utiliza un repositorio para separar la lógica que recupera los datos y los mapea al modelo de entidades, de la lógica del negocio que actúa en el modelo. El repositorio media entre la capa de fuente de datos y la capa de negocios de la aplicación; encuesta a la fuente de datos, mapea los datos obtenidos de la fuente de datos a la entidad de negocio y persisten los cambios de la entidad de negocio a la fuente de datos. En la figura 1 se muestra un esquema patrón Repository.

Los repositorios son puentes entre los datos y las operaciones que se encuentran en distintos dominios. Un repositorio elabora las consultas correctas a la fuente de datos y mapea los resultados a las entidades de negocio expuestas externamente. Los repositorios eliminan las dependencias a tecnologías específicas proveyendo acceso a datos de cualquier tipo [1].

El patrón de diseño Repository puede ayudar a separar las capas de una aplicación web ASP.NET ya que provee una arquitectura de 3 capas separadas, lo que mejora el mantenimiento de la aplicación y ayuda a reducir errores. Además facilita las pruebas unitarias.

Patrón Active Record

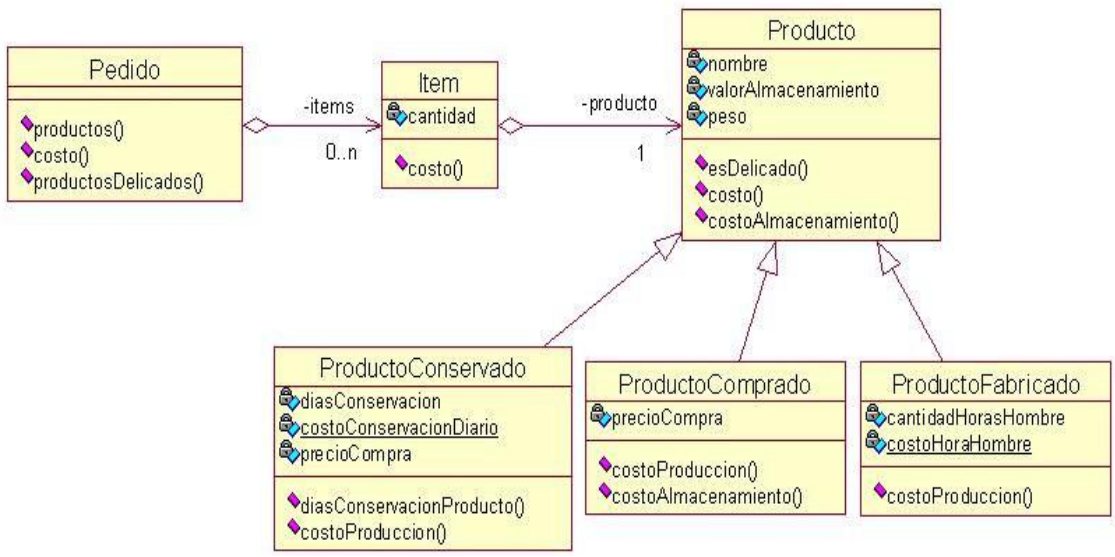
Active Record es un patrón en el cual, el objeto contiene los datos que representan a una fila (o tupla) de nuestra tabla o vista, además de encapsular la lógica necesaria para acceder a la base de datos. De esta forma el acceso a datos se presenta de manera uniforme a través de la aplicación (lógica de negocio + acceso a datos en una misma clase). En la figura 2 se muestra un esquema del patrón Active Record.

Una clase Active Record consiste en el conjunto de propiedades que representa las columnas de la tabla más los típicos métodos de acceso como las operaciones CRUD (Create, Read, Update, Delete), búsqueda (Find), validaciones, y métodos de negocio [5]. Este patrón constituye la aproximación más obvia, poniendo el acceso a la base datos en el propio objeto de negocio. De este modo es evidente como manipular la persistencia a través de él mismo. Gran parte de este patrón viene de un Domain Model y esto significa que las clases están muy cercanas a la representación en la base de datos. Cada Active Record es responsable de sí mismo, tanto en lo relacionado con persistencia como en su lógica de negocio .

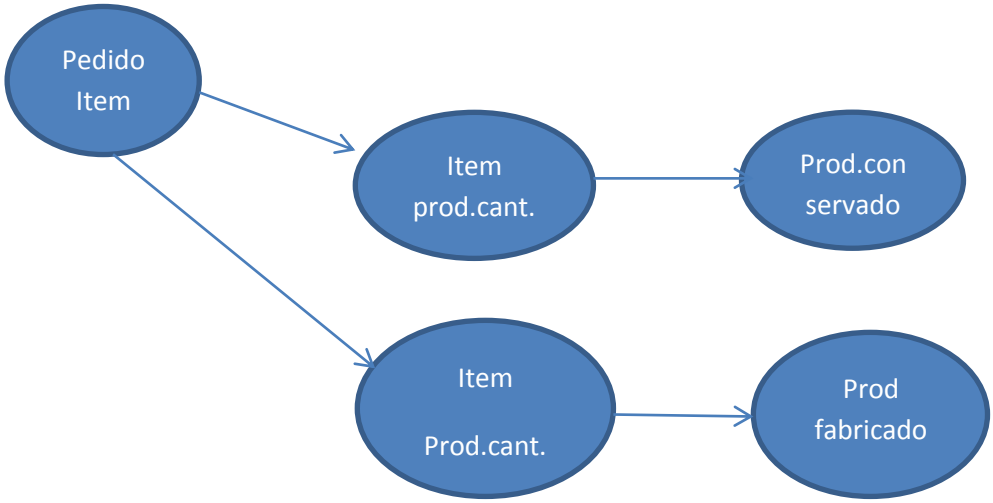
Desajuste por impedancia

Se entiende pro desajuste pro impedancia a las dificultades de traslación de un mundo a otro, es decir de un modelo orientado a objetos a una base de datos relacional.

Ejemplo

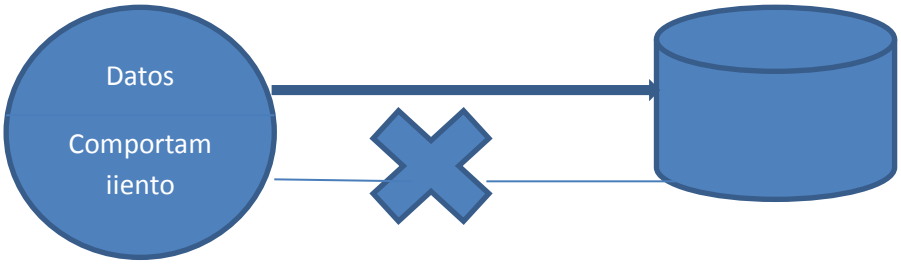


Los productos siguen en memoria una estructura de grafos



Estos objetos viven en el ambiente. El tema es que la memoria es limitada y necesito poder almacenar los objetos en un medio que me permita recuperarlos el día de mañana (concepto que llamaremos **persistencia**).
Entonces tengo distintas opciones:
- Puedo conservar la misma estructura en una base de objetos,
- o puedo utilizar una base de datos relacional como repositorio de información.
La pregunta es: este grafo que acabamos de dibujar ¿encaja justo en una estructura basada en el álgebra relacional?

• Originalmente podemos establecer una equivalencia entre el concepto de Tabla/Clase, y Registro/instancias de esa clase. Pero más adelante empezaremos a tener ciertas dificultades para que este mapping sea tan lineal.



• Los objetos tienen comportamiento, las tablas sólo permiten habilitar ciertos controles de integridad (constraints) o pequeñas validaciones antes o después de una actualización (triggers). Los stored procedures no están asociados a una tabla, lo que genera el mismo problema que en la programación estructurada: toco un campo y necesito al menos recompilar todos los stored procedures asociados. Cada actualización impacta en los datos por un lado y en los programas que actualizan los datos (Find/Replace masivo).

• Los objetos encapsulan información, no para protegerse (siempre puedo hackearlo), sino para favorecer la abstracción del observador. Una tabla no tiene esa habilidad: si tengo una entidad con un atributo ACTIVO que es VARCHAR2(1), tengo que entrar al constraint check o bien tirar un SELECT para saber si el ACTIVO es 1 ó 0, “S” ó “N”, e inferir en el mejor de los casos qué significan los valores de ese campo. Puedo documentarlo en un diccionario de datos, algo que está ajeno a la base que estoy tocando y que necesita una sincronización propia de gente pulcra y obsesiva.

• En objetos puedo generar interfaces, que permiten establecer un contrato entre dos partes, quien publica un determinado servicio y quien usa ese servicio. En el álgebra relacional la interfaz no se convierte en ninguna entidad, con lo cual hay que contar con ciertos trucos para generar objetos que sólo encapsulan comportamiento.

• La herencia es una relación estática que se da entre clases, que favorece agrupar comportamiento y atributos en común. Cuando yo instancio un objeto recibo la definición propia de la clase y de todas las superclases de las cuales heredo. En el modelo lógico de un Diagrama Entidad/Relación yo tengo supertipos y subtipos, pero en la implementación física las tablas no tienen el concepto de herencia. Hay que hacer adaptaciones que se verán mas adelante.

• Al no existir el comportamiento en las tablas y no estar presente el concepto de interfaz _ no hay polimorfismo en el álgebra relacional: yo puedo recibir un objeto sin saber exactamente de qué clase es, y a mí no me interesa averiguarlo, sólo me concentro en lo que le puedo pedir

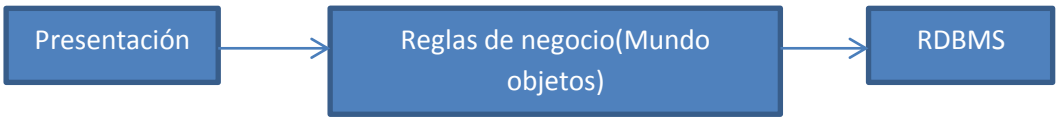
y en su contrato. Al tirar un query sobre una tabla, necesito saber de qué tabla se trata (claro, al menos en los queries tradicionales, que no acceden a los metadatos).

Todo esto es lo que se conoce como “Impedance mismatch” _ (dificultades por impedancia) las dificultades de traslación de los dos mundos.

De todas maneras, estamos usando la base de datos relacional sólo como repositorio para almacenar los datos de los objetos. El comportamiento sigue estando en
cuando se instancian. Por ese motivo tenemos lposibilidad de adaptar el modelo relacional al de objetos…

Algunas otras cosas a tener en cuenta:

• ¿Cómo manipulo los datos en una base relacional? A través de un lenguaje declarativo, el SQL donde digo qué quiero. Y muchas veces nos vamos a encontrar en la disyuntiva de algo muy fácil de hacer con un query/stored procedure vs. tener que armar una “regla de negocio” en objetos que implique muchas más líneas de código.
El tema es que muchas veces no puedo zafar de tener:

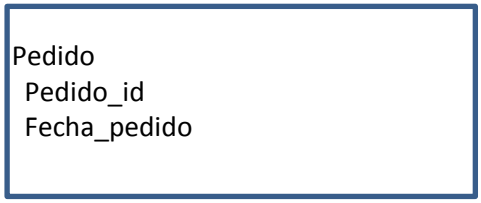


Eso tiene muchas contras:
• estoy mezclando tecnologías
• pierdo control cuando hay un cambio: es difícil así conservar el encapsulamiento.

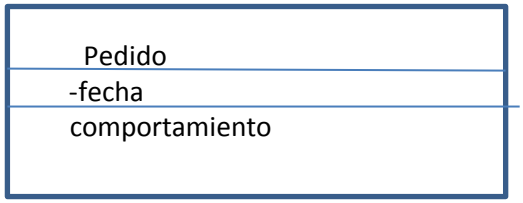
<u>Ejemplos de cómo se resolverían algunos casos</u>

Supongamos que Pedido tiene una fecha de pedido.

Modelo relacional : tabla pedido



Modelo objetos: clase pedido

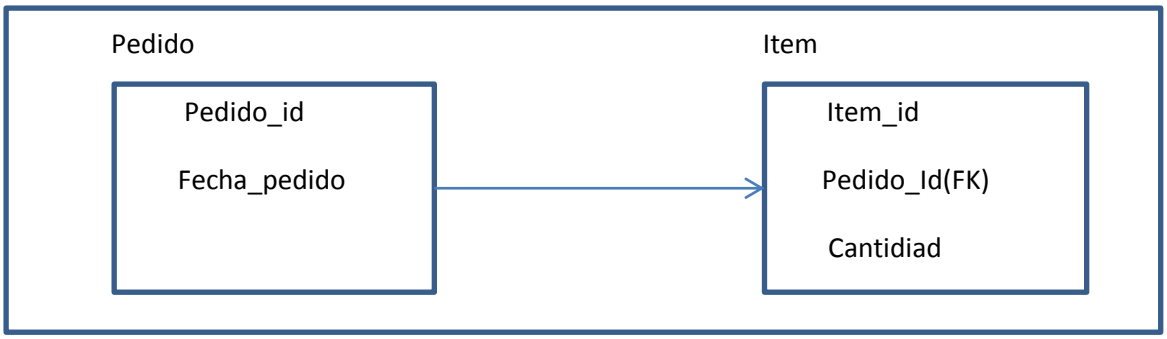


Fíjense qué diferencia tenemos en la tabla vs. la clase Pedido. Además del comportamiento (en la cajita de abajo), necesitamos una clave que identifique unívocamente a cada registro de la tabla Pedido.
Estamos hablando del mismo registro si tienen el mismo PEDIDO_ID.

En objetos se maneja el concepto de identidad: cada objeto sabe que es él y ningún otro objeto, entonces no es necesario trabajar con un identificador. Cuando yo referencio a un objeto, se que le estoy enviando un mensaje a ése objeto, no necesito identificarlo porque alguien me pasó la referencia a él. Si no lo conozco no le puedo pedir cosas.
Hablamos del mismo objeto si objeto1 == objeto2.

Agregamos el vínculo con Item

Modelo relacional



Fíjense que como el Pedido es la tabla madre y los ítems son hijos, la clave principal de ITEM se compone del Identificador del Pedido y el del Item.

Esto implica que si tengo el Pedido 1 con 2 ítems y el Pedido 2 con 2 ítems, la tabla Item tendría la siguiente información: (1,1) (1,2) (2,1) (2,2)

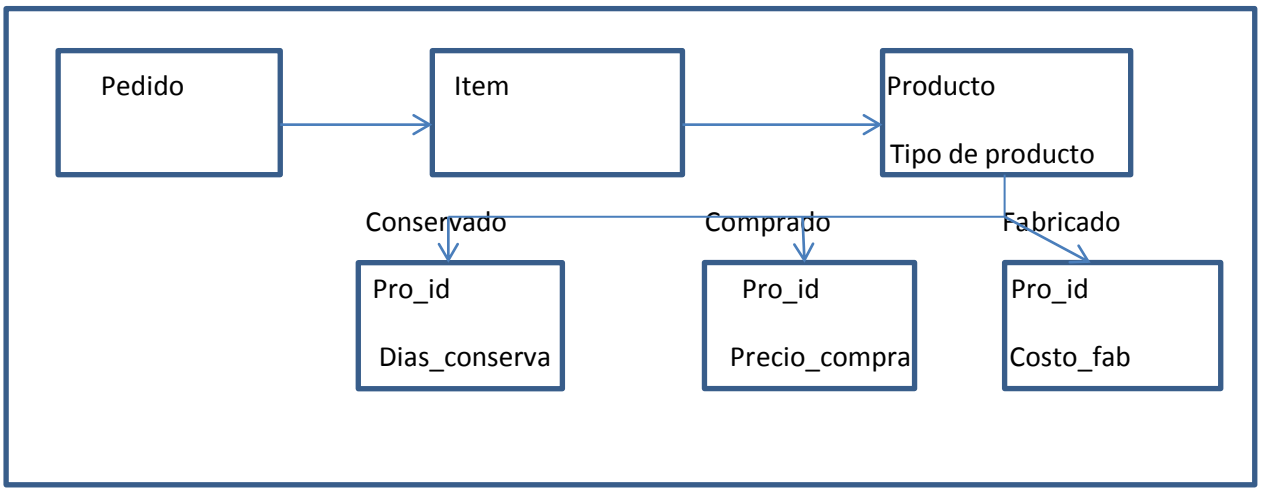
La otra opción es definir al ítem como autoincremental , en este caso los valores de las columnas serían (1,1) (2,1) (3,2) (4,2)

Muchas veces en Objetos se prefiere trabajar con claves autoincrementales, que dependan de una secuencia manejada por la base de datos. ¿Por qué? Porque es molesto definir un campo donde tenga que generar IDs incrementales según el pedido. Es más cómodo que la Base de Datos se encargue de generar ITEM_IDs por tabla, asegurándome por la misma Base que no se repiten (cosa mucho más compleja de asegurar cuando tengo más de una Virtual Machine/Ambiente dando vueltas).

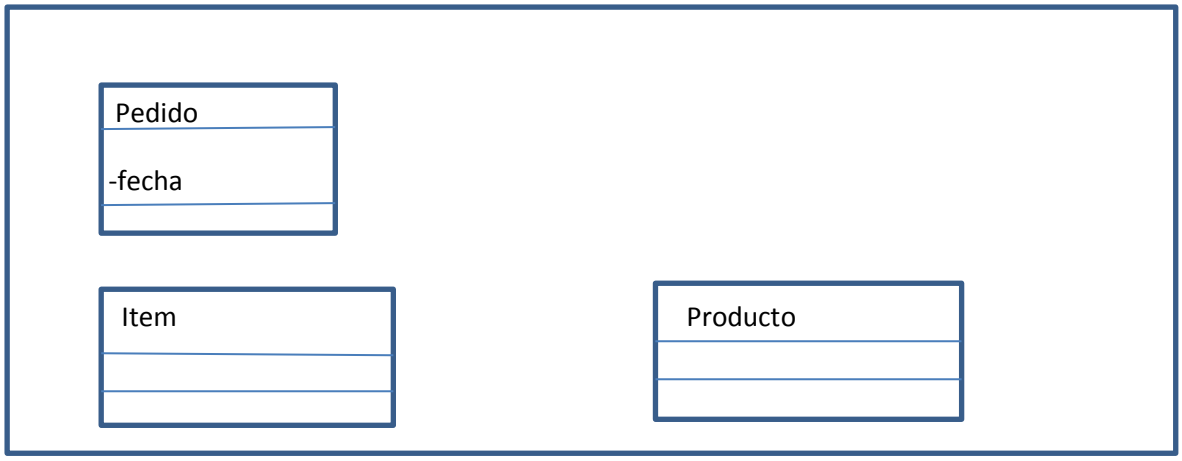
¿Cómo juega el ID dentro del objeto Pedido e Item? Bueno, quizás sí tenga un atributo Id en Pedido e Item pero si yo trabajo en objetos no quiero que el que use a ese objeto esté preguntando por el Id. Depende del contexto, el id puede ser un atributo privado que no tenga getter hacia fuera.

Agregamos el vínculo con Product

Modelo relacional



Modelo objetos



Donde la clase producto tiene subclases.

El modelo relacional de arriba se corresponde a la clase Producto con sus subclases. Entre el modelo lógico relacional y el modelo de objetos parece no haber mucha diferencia. Ahora bien, al implementar físicamente este modelo podemos elegir 3 opciones:
Implementar una tabla por cada clase, implementar una tabla pro subclase o implementar una única tabla.

Cuando trabajamos con objetos polimórficos tiene ventajas y desventajas

Concepto	A favor	En contra
1 tabla	□ Facilidad/> Performance para queries polimórficos □ Evito generar muchas tablas	□ Campos no utilizados
1 tabla por Cada clase (n+1)	□ Es el modelo “ideal” según las reglas de Normalización □ Permite trabajar con queries polimórficos (no necesito saber en qué tabla está la información) □ Permite establecer campos no nulos para cada subclase	□ Es la opc, que mas tablas crea
1 tabla por Subclase	□ Permite establecer campos no nulos y no req. Discriminar	□ Cada subclase repite campos heredados

Una vez adoptada la decisión de trabajar con bases relacionales, la pregunta es qué hacer primero:

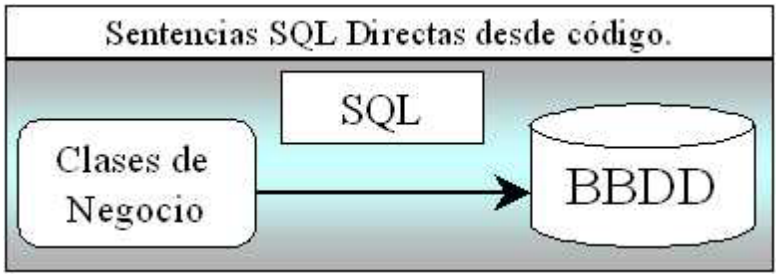
a) Generamos el modelo relacional y luego adaptamos el modelo de objetos en base a las tablas generadas
b) Generamos el modelo de objetos y en base a éste se crean las tablas.

La opción a) supone que es más importante la forma en que guardo los datos que las reglas de negocio que modifican esos datos. Pero a veces no nos queda otra chance, si partimos de un sistema construido o enlatado.
En la opción b) el desarrollo con objetos no se ve ensuciado por restricciones propias de otra tecnología.

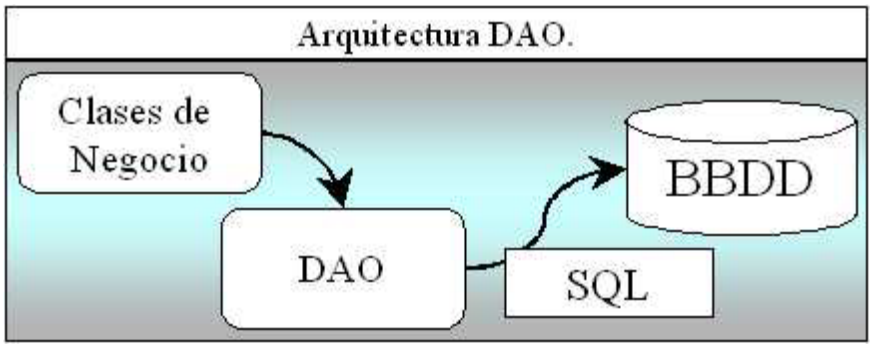
HIBERNATE

Persistencia de los datos y posibilidades que estas ofrecen

En el diseño de una aplicación una parte muy importante es la manera en la cual accedemos a nuestros datos en la base de datos (en adelante BBDD) determinar esta parte se convierte en un punto crítico para el futuro desarrollo.
La manera tradicional de acceder seria a través de JDBC directamente conectado a la BBDD mediante ejecuciones de sentencias SQL:

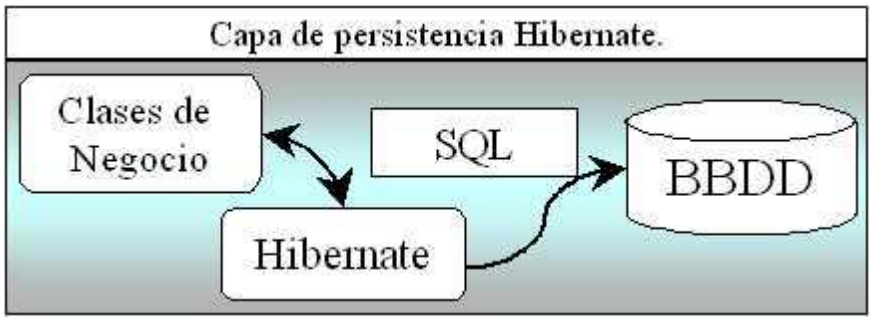


Esta primera aproximación puede ser útil para proyectos o arquitecturas sin casi clases de negocio, ya que el mantenimiento del código está altamente ligado a los cambios en el modelo de datos relacional de la BBDD, un mínimo cambio implica la revisión de casi todo el código así como su compilación y nueva instalación en el cliente.
Aunque no podemos desechar su utilidad. El acceso a través de SQL directas puede ser utilizado de manera puntual para realizar operaciones a través del lenguaje SQL lo cual seria mucho mas efectivo que la carga de gran cantidad de objetos en memoria. Si bien un buen motor de persistencia debería implementar mecanismos para ejecutar estas operaciones masivas sin necesidad de acceder a este nivel.
Una aproximación mas avanzada seria la creación de unas clases de acceso a datos (
DAO Data Acces Object). De esta manera nuestra capa de negocio interactuaría con la capa DAO y esta sería la encargada de realizar las operaciones sobre la BBDD.



Los problemas de esta implementación siguen siendo el mantenimiento de la misma así como su portabilidad. Lo único que podemos decir es que tenemos el código de transacciones encapsulado en las clases DAO. Un ejemplo de esta arquitectura podría ser Microsoft ActiveX Data Object (ADO).

Y como encaja Hibernate en todo esto?. Lo que parece claro es que debemos separar el código de nuestras clases de negocio de la realización de nuestras sentencias SQL contra la BBDD. Por lo tanto Hibernate es el puente entre nuestra aplicación y la BBDD, sus funciones van desde la ejecución de sentencias SQL a través de JDBC hasta la creación, modificación y eliminación de objetos persistentes.



Con la creación de la capa de persistencia se consigue que los desarrolladores no necesiten conocer nada acerca del esquema utilizado en la BBDD. Tan solo conocerán el interface proporcionado por el motor de persistencia. De esta manera conseguimos separar de manera clara y definida, la lógica de negocios de la aplicación con el diseño de la BBDD.
Esta arquitectura conllevará un proceso de desarrollo más costoso pero una vez se encuentre implementada las ventajas que conlleva merecerán la pena. Es en este punto donde entra en juego Hibernate. Como capa de persistencia desarrollada tan solo tenemos que adaptarla a nuestra arquitectura.

Qué es hibernate

Hibernate es una capa de persistencia objeto/relacional y un generador de sentencias sql. Te permite diseñar objetos persistentes que podrán incluir polimorfismo, relaciones, colecciones, y un gran número de tipos de datos. De una manera muy rápida y optimizada podremos generar BBDD en cualquiera de los entornos soportados : Oracle, DB2, MySql, etc.. Y lo más importante de todo, es **open source**, lo que supone, entre otras cosas, que no tenemos que pagar nada por adquirirlo.
Uno de los posibles procesos de desarrollo consiste en, una vez tengamos el diseño de datos realizado, mapear este a ficheros XML siguiendo la DTD de mapeo de Hibernate.
Desde estos podremos generar el código de nuestros objetos persistentes en clases Java y también crear BBDD independientemente del entorno escogido.
Hibernate se integra en cualquier tipo de aplicación justo por encima del contenedor de datos.
Una posible configuración básica de hibernate es la siguiente:

Podemos observar como Hibernate utiliza la BBDD y la configuración de los datos para proporcionar servicios y objetos persistentes a la aplicación que se encuentre justo por arriba de él.

Mapas de objetos relacionales en ficheros XML

Antes de empezar a escribir una línea se debe tener realizado el análisis de la aplicación y la estructura de datos necesaria para su implementación.
El objetivo es conseguir desde el diseño de un objeto relacional un fichero XML bien formado de acuerdo a la especificación Hibernate. El trabajo consistirá en *traducir* las propiedades, relaciones y componentes a el formato XML de Hibernate.
Asumiremos que objeto persistente, registro de tabla en BBDD y objeto relacional son la misma entidad.

Estructura del fichero XML

Esquema general de un fichero XML de mapeo es algo como esto:
<Encabezado XML>
<Declaración de la DTD>
<class - Definición de la clase persistente>
<id - Identificador>
<generator - Clase de apoyo a la generación automática de OID's>
<component - Componentes, son las columnas de la tabla>

A continuación se detallan las características, parámetros y definición de las etiquetas arriba utilizadas así como de algunas otras que nos serán de utilidad a la hora de pasar nuestro esquema relacional a ficheros de mapeo XML.

Declaración de la DTD.
El documento DTD que usaremos en nuestros ficheros XML se encuentra en cada distribución de Hibernate en el propio **.jar** o en el directorio src.

Elemento Raíz **<hibernate-mapping>.** Dentro de él se declaran las clases de nuestros objetos persistentes. Aunque es posible declarar más de un elemento <class> en un mismo fichero XML, no debería hacerse ya que aporta una mayor claridad a nuestra aplicación realizar un documento XML por clase de objeto persistente.

<class> Este es el tag donde declaramos nuestra clase persistente. Una clase persistente equivale a una tabla en la base de datos, y un registro o línea de esta tabla es un objeto persistente de esta clase. Entre sus posibles atributos destacaremos:
1. *name* : Nombre completo de la clase o interface persistente. Deberemos incluir el package dentro del nombre.
2. *table* : Nombre de la tabla en la BBDD referenciada. En esta tabla se realizará las operaciones de transacciones de datos. Se guardarán, modificarán o borrarán registros según la lógica de negocio de la aplicación.
3. *discriminator-value* : Permite diferenciar dos sub-clases. Utilizado para el polimorfismo.
4. *proxy* : Nombre de la clase Proxy cuando esta sea requerida.

<id> Permite definir el identificador del objeto. Se corresponderá con la clave principal de la tabla en la BBDD. Es interesante definir en este momento lo que será para nuestra aplicación un OID(Identificador de Objeto). Tenemos que asignar identificadores únicos a nuestros objetos persistentes, en un primer diseño podríamos estar tentados a asumir un dato con

significado dentro de la capa de negocios del propio objeto fuese el identificador, pero esta no sería una buena elección. Imaginemos una tabla de personas con su clave primaria N.I.F.. Si el tamaño, estructura o composición del campo cambiase deberíamos realizar este cambio en cada una de las tablas relacionadas con la nuestra y eventualmente en todo el código de la aplicación.
Utilizando OID's (Identificadores de Objetos) tanto a nivel de código como en BBDD simplificamos mucho la complejidad de nuestra aplicación y podemos programar partes de la misma como código genérico.
El único problema en la utilización de OID es determinar el nivel al cual los identificadores han de ser únicos. Puede ser a nivel de clase, jerarquía de clases o para todas la aplicación, la elección de uno u otro dependerá del tamaño del esquema relacional.
1. *name* : Nombre lógico del identificador.
2. *column* : Columna de la tabla asociada en la cual almacenaremos su valor.
3. *type* : Tipo de dato.
4. *unsaved-value* ("any|none|null|id_value"): Valor que contendrá el identificador de la clase si el objeto persistente todavía no se ha guardado en la BBDD.
5. *generator* : clase que utilizaremos para generar los oid's. Si requiere de algún parámetro este se informa utilizando el elemento <paramater name="nombre del parámetro">.

Hibernate proporciona clases que generan automaticamente estos OID evitando al programador recurrir a trucos como coger la fecha/hora del sistema. Entre ellas caben destacar :

1.*vm* : Genera identificadores de tipo long, short o int. Que serán únicos dentro de una JVM.
2. *sequence* : Utiliza el generador de secuencias de las bases de datos DB2, PostgreSQL, Oracle, SAP DB, McKoi o un generador en Interbase. El tipo puede ser long, short o int.
3. *hilo* : Utiliza un algoritmo hi/lo para generar identificadores del tipo long, short o int. Estos OID's son únicos para la base de datos en la cual se generan. En realidad solo se trata de un contador en una tabla que se crea en la BBDD. El nombre y la columna de la tabla a utilizar son pasados como parámetros, y lo único que hace es incrementar/decrementar el contador de la columna con cada nueva creación de un nuevo registro. Así, si por ejemplo decimos tener un identificador único por clase de objetos persistentes, deberíamos pasar como parámetro tabla *table_OID* y como columna el nombre de la clase *myclass_OID*.
4. *uuid.string* : Algoritmo UUID para generar códigos ASCII de 16 caracteres.
5. *assigned* : Por si después de todo queremos que los identificadores los gestione la propia aplicación.

<discriminator>Cuando una clase declara un discriminador es necesaria una columna en la tabla que contendrá el valor de la marca del discriminador. Los conjuntos de valores que puede tomar este campo son definidos en la cada una de las clases o sub-clases a través de la propiedad <discriminator-value>. Los tipos permitidos son string, character, integer, byte, short, boolean, yes_no, true_false.
1. *column*: El nombre de la columna del discriminador en la tabla.
2. *type*: El tipo del discriminador.

<property> Declara una propiedad persistente de la clase , que se corresponde con una columna.
1. *name*: Nombre lógico de la propiedad.
2. *column*: Columna en la tabla.

3. *type*: Indica el tipo de los datos almacenados. Mirar **Tipos de datos en Hibernate** para ver todos las posibilidades de tipos existentes en Hibernate.

Tipos de relaciones.(Componentes y Colecciones) En todo diseño relacional los objetos se referencian unos a otros a través de relaciones, las típicas son : uno a uno **1-1**, uno a muchos **1_n**, muchos a muchos **n-m**, muchos a uno **n-1**. De los cuatro tipos, dos de ellas **n-m** y **1-n** son colecciones de objetos las cuales tendrán su propio y extenso apartado, mientras que a las relaciones **1-1** y **n-1** son en realidad componentes de un objeto persistente cuyo tipo es otro objeto persistente.
<many-to-one>La relación **n-1** necesita en la tabla un identificador de referencia, el ejemplo clásico es la relación entre padre - hijos. Un hijo necesita un identificador en su tabla para indicar cual es su padre. Pero en objetos en realidad no es un identificador si no el propio objeto padre, por lo tanto el componente n-1 es en realidad el propio objeto padre y no simplemente su identificador. (Aunque en la tabla se guarde el identificador)
1. *name* : El nombre de la propiedad.
2. *column* : Columna de la tabla donde se guardara el identificador del objeto asociado.
3. *class*: Nombre de la clase asociada. Hay que escribir todo el package.
4. *cascade* ("all|none|save-update|delete"): Especifica que operaciones se realizaran en cascada desde el objeto padre.

<one-to-one>Asociacion entre dos clases persistentes, en la cual no es necesaria otra columna extra. Los OID's de las dos clases serán idénticos.
1. *name* : El nombre de la propiedad.
2. *class* : La clase persistente del objeto asociado
3. *cascade* ("all|none|save-update|delete") : Operaciones en cascada a partir de la asociación.
4. *constrained* ("true"|"false") .

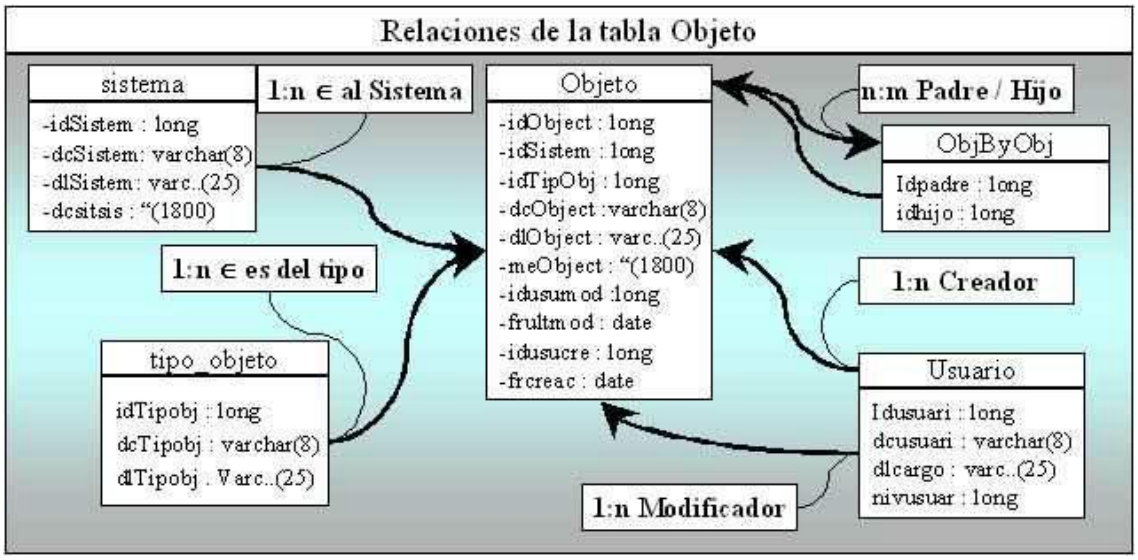
Tipos de datos en Hibernate.
» *Son tipos básicos*: integer, long, short, float, double, character, byte, boolean, yes_no, true_false.
» *string* : Mapea un java.lang.String a un VARCHAR en la base de datos.
» *date, time, timestamp* : Tipos que mapean un java.util.Date y sus subclases a los tipo SQL : DATE, TIME, TIMESTAMP.
» *calendar, calendar_date* : Desde java.util.Calendar mapea los tipos SQL TIMESTAMP y DATE
» *big_decimal* : Tipo para NUMERIC desde java.math.BigDecimal.
» *locale, timezone, currency* : Tipos desde las clases **java.util.Locale**, **java.util.TimeZone** y **java.util.Currency**. Se corresponden con un VARCHAR.
Las instancias de Locale y Currency son mapeadas a sus respectivos códigos ISO y las de TimeZone a su ID.
» *class* : Guarda en un VARCHAR el nombre completo de la clase referenciada.
» *binary*: Mapea un array de bytes al apropiado tipo de SQL.
» *serializable* : Desde una clase que implementa el interface Serializable al tipo binario SQL.
» *clob, blob*: Mapean clases del tipo java.sql.Clob y java.sql.Blob
» *Tipos enumerados persistentes (Persistente Enum Types)*: Un tipo enumerado es una clase de java que contiene la enumeración a utilizar(ej:Meses, Dias de la semana, etc..). Estas clases han de implementar el interface **net.sf.hibernate.PersistentEnum** y definir las operaciones **toInt()** y **fromInt()**. El tipo enumerado persistente es simplemente el nombre de la clase completo. Ejemplo de clase persistente :

```
package com.hecsua.enumerations;
import net.sf.hibernate.PersistentEnum
public class Meses implements PersistentEnum {
private final int code;
```

```
        private Meses(int code) {
                this..code = code;
}
public static final Meses Enero = new Meses(0);
public static final Meses Febrero = new Meses(1);
public static final Meses Marzo = new Meses(2);
...
public int to Int() {return code;}
public static Meses fromInt(int code){
case 0: return Enero;
case 1: return Febrero;
case 2: return Marzo;
...
default: throw new RuntimeException("Mes no valido.");
                }
}
```

Creación de Ficheros XML

El siguiente paso es aplicar todo lo anteriormente explicado en un ejemplo. Partiendo del siguiente diseño, crearemos los ficheros XML de acuerdo a las especificaciones anteriormente explicadas:



Traducir este diseño relacional a ficheros XML siguiendo las especificaciones Hibernate será mucho más sencillo de lo que pueda parecer a primera vista
Para crear el mapeo de **Objeto**[3], comenzaremos con la definición del fichero XML[4], mediante las dos líneas siguientes :

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping
  DTD//EN"
    "..\..\hibernate\hibernate-mapping-2.0.dtd">
```

En estas dos líneas declaramos el tipo de codificación del fichero XML así como su DTD.

Después comenzamos a definir nuestro objeto persistente, abrimos el tag

<hibernate-mapping>. Una muy buena costumbre es definir un objeto por fichero, ya que hace mucho más legible el código y simplifica su modificación.
El siguiente tag será el **class**:

```
<!- Clase persistente que representa un Objeto dentro del proyecto este puede ser una pantalla, un procedimiento un algoritmo, dependerá del proyecto en cuestión-->
    <class name="com.hecsua.bean.Objeto" table="objeto">
```

Como podéis observar la clase se denominará **Objeto** y estará dentro del package **com.hecsua.bean**. Esta clase hará referencia a la tabla objeto dentro de la BBDD.

El siguiente paso es definir el identificador, el nombre lógico de la propiedad queremos que sea **id**, que haga referencia a la columna **idObject** y es un dato de tipo **long**, el resultado es el siguiente :

```
    <id name="id" column="idobject" type="long">
```

Para obtener los OID's utilizaremos la clase **hilo** sobre la tabla **uid_table** y la columna **next_hi_value** :

```
    <generator class="hilo">
    <param name="table">uid_table</param>
    <param name="column">next_hi_value</param>
    </generator>
```

Las propiedades se definen de manera parecida al identificador, con el nombre lógico de la propiedad (aquí podremos ser tan descriptivos como deseemos), la columna dentro de la tabla a la que hace referencia y el tipo de dato. En el caso de que este necesite parámetros de longitud u otro tipo también se declararán aquí mediante el parámetro adecuado.

```
    <property name="descor" column="dcobject" type="string" length="8" />
    <property name="deslon" column="dlobject" type="string" length="25" />
    <property name="texto" column="meobject" type="string" length="1500"/>
    <property name="frCreacion" column="frcreac" type="date" />
    <property name="frUltimaModificacion" column="frultmod" type="date" />
```

Ahora definimos las relaciones, como se puede ver son todas del tipo muchos a uno, por lo que en realidad son columnas de identificadores ajenos a nuestra tabla. Se definen utilizando la etiqueta many-to-one, el nombre lógico de la propiedad nos ha de recordar el rol de la relación, el parámetro class deberá ser la clase del objeto asociado y la columna que almacenara su identificador.

```
  <!- Relación con la clase persistente Sistema a través de la columna idsistem -->
  <many-to-one name="sistema" class="com.hecsua.bean.Sistema" column="idsistem" />
  <!- Relación con la clase persistente Usuario realizando el rol de Usuario Creador del objeto
      a través de la columna idusucre -->
  <many-to-one name="creador" class="com.hecsua.bean.Usuario" column="idusucre" />
  <!- Relación con la clase persistente Usuario con el rol de Ultimo Usuario que Modifico el
      Objeto a través de la columna idusumod-->
   <many-to-one name="modificador" class="com.hecsua.bean.Usuario"
    column="idusumod" />
```

Por último tenemos que mapear la relación n-m, esta es una relación circular entre registros de la tabla objeto. Para mapear esta relación utilizaremos la siguiente estructura, dejando como se puede observar los identificadores relacionados en la tabla Obj_By_Obj:

```
    <set name="padres" table="obj_by_obj" lazy="true">
    <key column="idobject" />
    <many-to-many column="idobject" class="com.hecsua.bean.Objeto"
     not-null="true" />
```

```
</set>
<set name="hijos" table="obj_by_obj" lazy="true" inverse="true">
<key column="idobject" />
<many-to-many column="idobject" class="com.hecsua.bean.Objeto"
  not-null="true" />
</set>
```

En estas dos relaciones observamos que ambas utilizan la tabla obj_by_obj donde se guardarán las parejas de identificadores relacionados. Una de ellas ha de ser "inverse", con esta etiqueta declaramos cual es el final de la relación circular.

Finalmente cerramos las etiquetas, y acabamos de crear nuestro primer mapeo de un objeto relacional.