Manual Hibernate

Fecha de creación: 21.03.2003

Revisión 1.0 (22.04.2003)

Héctor Suárez González (hsg arroba telecable punto es)

http://www.javaHispano.org

Copyright (c) 2003, Héctor Suárez González. Este documento puede ser distribuido solo bajo los términos y condiciones de la licencia de Documentación de javaHispano v1.0 o posterior (la última versión se encuentra en http://www.javahispano.org/licencias/).

Manual Hibernate

Presentación del tutorial.

. Logo de Hibernate

La intención de este tutorial es introducir al lector en el uso del framework Hibernate. A través del mismo se irán desarrollando diversos ejemplos y se propondrán ejercicios para su resolución. El objetivo último es que el lector pueda desarrollar un pequeña aplicación donde la capa de persistencia se utilizará para el acceso a la base de datos. Este objetivo se encuentra dividido en pequeños pasos :

1. *Capas de persistencia y posibilidades que estas ofrecen.*
2. *Que es hibernate. Estructura de una arquitectura base.*
3. *Mapas de objetos relacionales en ficheros XML. Hasta aqui trataremos en este primer documento, posteriormente afrontaremos los demas puntos.*
4. *Generación de Código y Bases de Datos.*
5. *Utilización del API Hibernate.*
6. *HSQL. El lenguaje sql de Hibernate*
7. *Posibles arquitecturas y ejemplos implementados*
8. *Conclusiones*
9. *Posibles ejercicios*

1. Capítulos.

Se esperá que el lector de este documento tenga conocimientos básicos de Hiberante. Dichos conocimientos se pueden obtener a través de la Guía hacia Hibernate [2] disponible en javaHispano.

1. Capas de persistencia y posibilidades que estas ofrecen.

NOTA: Esta primera parte es de introducción al concepto de capa de persistencia, por lo tanto es bastante teórica.

En el diseño de una aplicación (me referiré a una aplicación a desarrollar utilizando Java) 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.

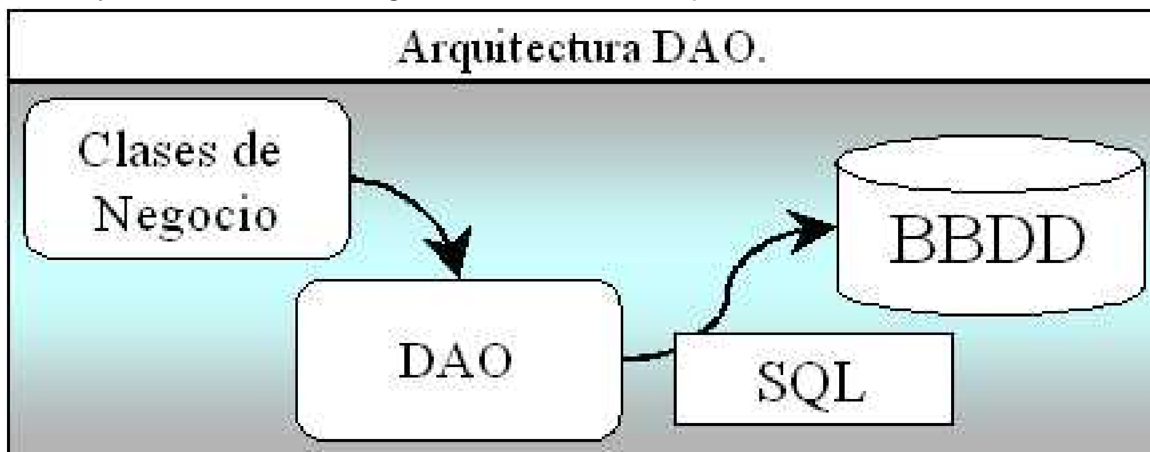
La manera tradicional de acceder seria a través de JDBC directamente conectado a la BBDD mediante ejecuciones de sentencias SQL:

1. Sentencias SQL directas

Esta primera aproximación puede ser útil para proyectos o arquitecturas sin casi clases de negocio, ya que el mantenimiento del código esta 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.

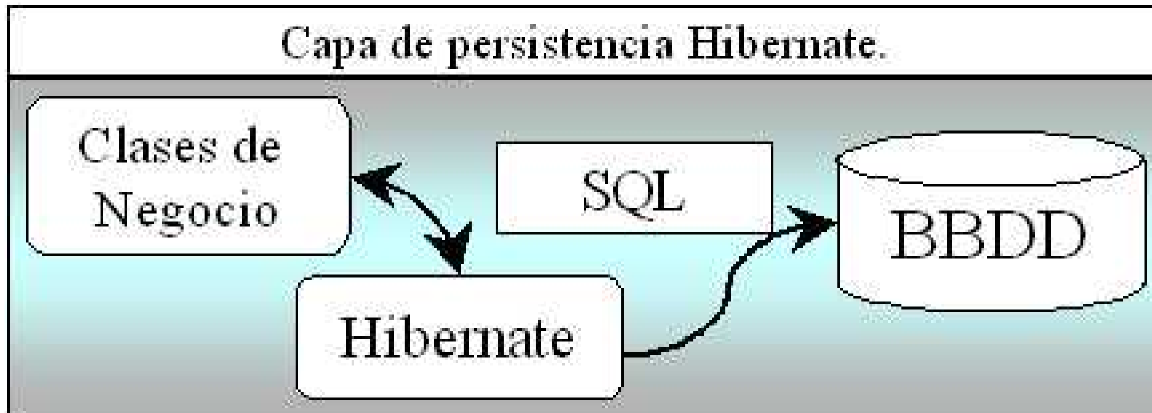


2. Ejemplo de DAO (Data Access Object

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.



3. Persistencia con Hibernate

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 nuestro 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.

2. Qué es hibernate

Hibernate[1] 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:

4. Arquitectura Base

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.

3. Mapas de objetos relacionales en ficheros XML

Antes de comenzar de verdad este tutorial es recomendable que miréis por lo menos el tutorial de Java Hispano una Guía hacia Hibernate[2]. En el se configura las propiedades de Hibernate para poder acceder a una BBDD determinada y se ejecuta un pequeño ejemplo. Si ya lo habéis hecho o queréis saltarlo, pues nada, adelante.

NOTA: Antes de empezar a escribir una línea debemos tener realizado el análisis de nuestra aplicación y la estructura de datos necesaria para su implementación.

Objetivos : Conseguir desde el diseño de un objeto relacional un fichero XML bien formado de acuerdo a la especificación Hibernate. Nuestro trabajo consistirá en *traducir* las propiedades, relaciones y componentes a el formato XML de Hibernate.

A partir de ahora asumiremos que objeto persistente, registro de tabla en BBDD y objeto relacional son la misma entidad.

3.1. 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>
```

```
                                            .
                                            .
                <one-to-many / many-to-many - Posibles relaciones con
otras entidades persistentes>
                                            .
                                            .
```

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

3.1.1. 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.

3.1.2. <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 una 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 realizaraá 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.

2. Atributos del elemento class

3.1.3.<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 seria 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">.

3. Atributos del elemento id

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 :

NOTA: En Hibernate se pueden definir identificadores compuestos, esta parte se tratará en el tutorial avanzado.

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.

4. Generadores de OID's presentes en Hibernate

3.1.4. <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.

5. Atributos del elemento discriminator

3.1.5. <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 la sección **3.1.7 Tipos de datos en Hibernate** para ver todos las posibilidades de tipos existentes en Hibernate.

6. Atributos del elemento property

3.1.6. 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.

3.1.6.1. <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.

8. Atributos de las relaciones n-1

3.1.6.2. <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") :

9. Atributos de las relaciones 1-1

3.1.7. 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 enumerción 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.");

        }
}
```

10. Tipos en Hibernate.

3.2. Una primera aproximación hacia las colecciones

NOTA: Es necesario el conocimiento del framework Collections de java

Enlace a java.lang.Collection[5]

Las colecciones de elementos que Hibernate puede tratar como persistentes son : **java.util.Set**, **java.util.SortedMap**, **java.util.SortedSet**, **java.util.List**, y cualquier array de elementos o valores persistentes. Propiedades del tipo **java.util.Collection** o **java.util.List** pueden ser persistentes utilizando la semántica de *bag*.

Las colecciones persistentes no retienen ninguna semántica añadida por la clase implementada de la interface de colección (ej :iteradores ordenados de **LinkedHashSet**). La propiedad persistente que contenga una colección a de ser un interface del tipo **Map**, **Set** o **List**; nunca **HashMap**, **TreeSet** o **ArrayList**.Esta restricción es debida a que Hibernate reemplaza las instancias de Map, Set y List con instancias de sus propias implementaciones de Map, Set o List.

NOTA: Hay que tener cuidado al comparar las colecciones directamente con ==

Debido al modelo relacional existente por debajo, no soportan valores nulos.

NOTA: Hibernate no distingue entre una colección nula y una colección vacía

Las instancias de una colección son diferenciadas en la BBDD mediante una clave ajena del objeto relacional al cual pertenecen. Esta clave es denominada la clave de la colección. Esta clave será mapeada con el tag <key>.Las colecciones pueden contener : tipos basicos, entidades y componentes. No se pueden crear colecciones de colecciones.

Hay muchos tipos de mapeados de colecciones, que podrán ser útiles en un desarrollo mas avanzado, pero en este primer tutorial nos centraremos en las relaciones entre las clases persistentes. Las colecciones one-to-many y many-to-many.

3.2.1. Mapeando una colección

Las colecciones son declaradas utilizando <set>, <list>, <map>, <bag>, <array> y

<primitive-array>. Los posibles parámetros y sus valores son

1. *name*: El nombre lógico de la propiedad. Es útil poner un nombre que nos recuerde el rol de la colección (ej: Administradores, MultasSinPagar, etc..)

2. *table*: Nombre de la tabla de la colección.

   NOTA: No se utiliza en asociaciones one-to-many

3. *lazy ("true"|"false")*: Permite el uso de inicialización "lazy". Este tipo de inicialización hace que los objetos de la colección sean solicitados en demanda y no se carguen todos a la vez. Esto es especialmente útil para optimizar búsquedas, etc...

   NOTA: No se puede utilizar con arrays

4. *inverse*: Señala esta colección como el fin de una asociación bidireccional. Utilizada en relaciones many-to-many sobre todo.

5. *cascade*: Permite las operaciones en cascada hacia los entidades hijas.

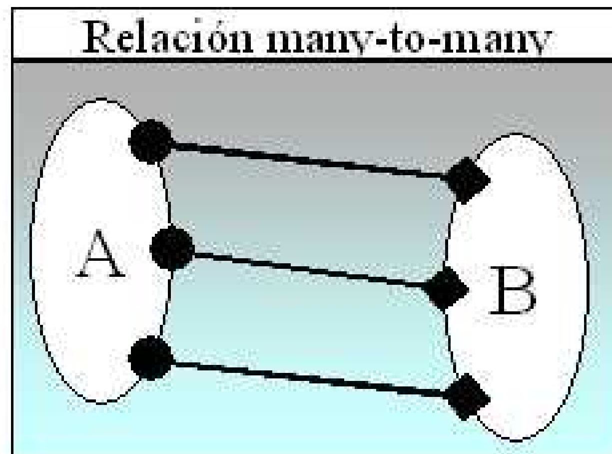
6. *sort*: Especifica una colección con una ordenación natural o con una clase comparadora dada.

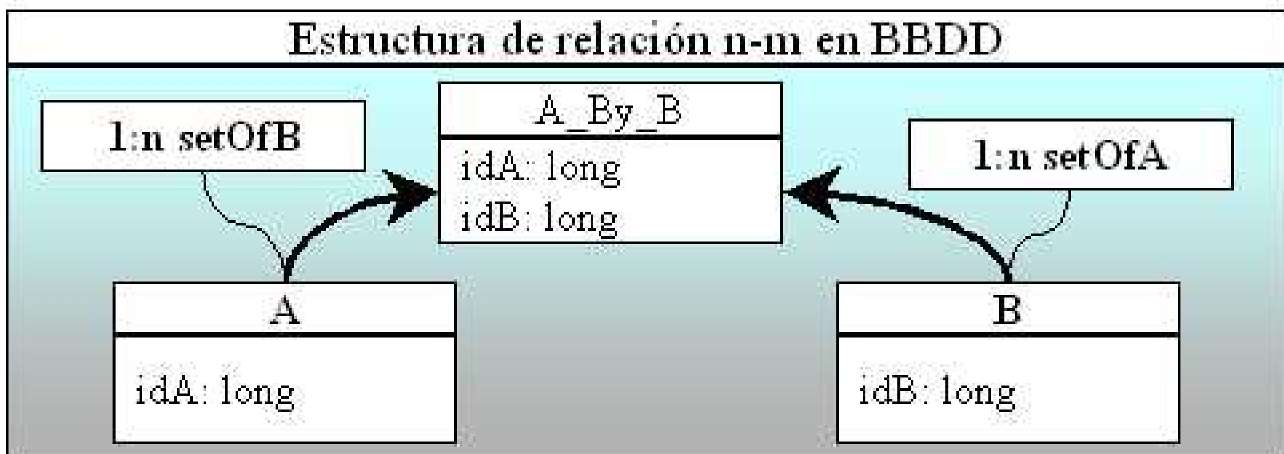
7. *order-by*: Columna\s de la tabla que definen el orden de iteración. Puede ser ascendente o descendente.

11. Atributos de una colección genérica

3.2.1.1. Asociaciones many-to-many

En esta asociación tenemos dos clases A y B. Un elemento de A tiene un conjunto de elementos de B hijos, y un elemento de B tiene otro conjunto distinto o igual de elementos de A.





5. Relación n-m

Esta estructura se puede diseñar creando una tabla intermedia que relacione los códigos de los elementos de A con los elementos de B. Queda claro por tanto que una colección muchos a muchos se ha de mapear en una tabla a parte con las claves de las dos tablas como claves ajenas.

Esto lo podríamos mapear como sigue :

```
<set role="setOfB" table="A_By_B">
<key column="A_id"/>
<many-to-many column="B_id" class="elementOfB"/>
</set>
```

En este punto no tenemos una columna extra en B que diga los elementos de B que le corresponden a un elemento de A. En vez de eso tenemos una tabla nueva A_By_B que contiene los pares de claves relacionados tanto de A hacia B como de B hacia A. Para que sea bidireccional tiene que ser declara en el mapeo de la clase B como sigue, de paso la definimos como el fin de la relación entre las dos tablas. Cualquier otro parámetro, posible para una colección puede ser utilizado aquí ej: lazy, cascade, etc...

```
<set role="setOfA" table="A_By_B" inverse="true">
<key column="B_id"/>
<many-to-many column="A_id" class="elementOfA"/>
</set>
```

3.2.1.2. Asociaciones One-to-Many

Esta relación pierde algunos conceptos semánticos de los colecciones de Java:

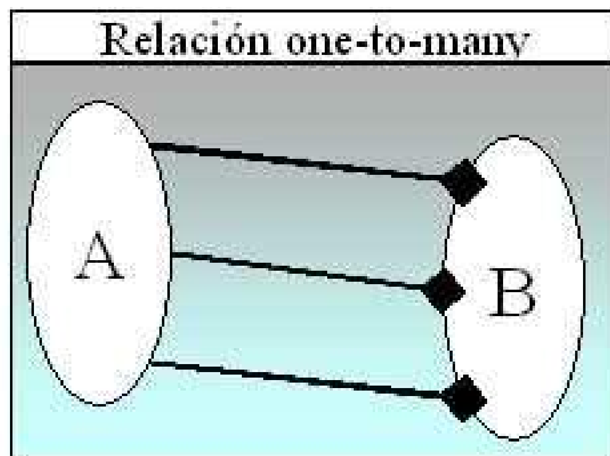
» Ningún valor nulo puede ser contenido en un map o set

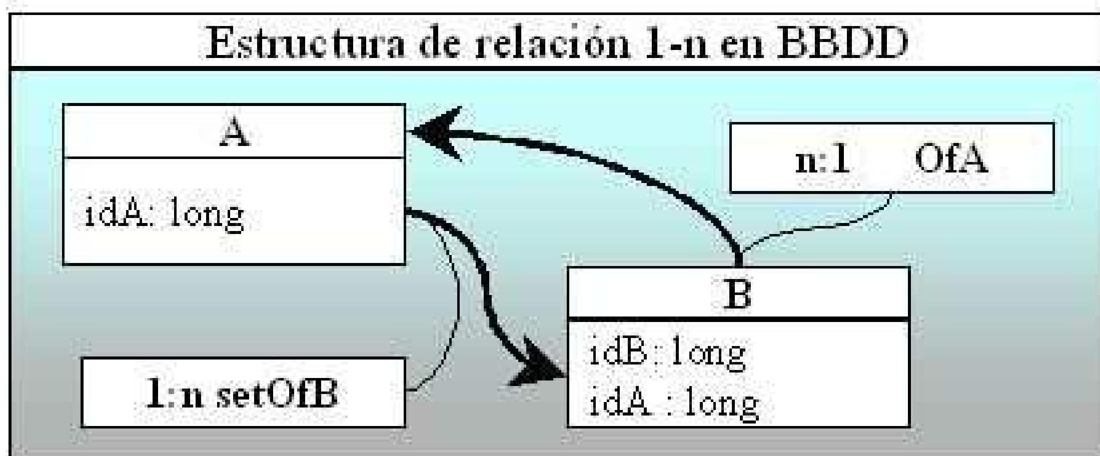
» Una instancia del contenedor no puede pertenecer a mas de una instancia de la colección

» Una instancia de las entidades contenidas no pueden aparecer en mas de una vez en el índice de la colección

14. Diferencias entre las las colecciones de Hibernate y el API Collections

Como en el caso anterior si queremos tener una asociación uno a muchos entre dos tablas, deberemos mapear correctamente las dos. En una crearemos una relación one-to-many y en la otra una many-to-one. Una asociación one-to-many de A hacia B requerirá un nuevo campo en B con el valor del índice de A al que se encuentra asociado. En la tabla A no será necesario ningún nuevo campo, como observamos en la siguiente imagen :





6. Relación 1-n

```
<set name="setOfB" table="B">
<key column="A_id"/>
<one-to-many class="B"/>
</set>
```

3.2.1.3. Consideraciones comunes a las colecciones.

Se ha de destacar ciertas características de las colecciones en Hibernate:

1. **Inicialización "Lazy"**: Cuando definimos una colección como "lazy" conseguimos que la carga de datos desde la BBDD sea en demanda de los mismos. Es decir, si de una colección de 50000 objetos solo necesitamos buscar en los 10 primeros no tiene sentido cargarlos todos no?. Por eso los objetos que son devueltos al iterar sobre la colección son cargados a medida que se accede a ellos. Aunque esto puede producir ciertos problemas que serán abordados con posterioridad.

2. **Colecciones ordenadas**: Hibernate soporta la implementación de colecciones ordenadas a través de los interfaces java.util.SortedMap y java.util.SortedSet. Si queremos podemos definir un comparador en la definición de la colección. Los valores permitidos son natural, unsorted y el nombre de la clase que implementa java.util.Comparator.

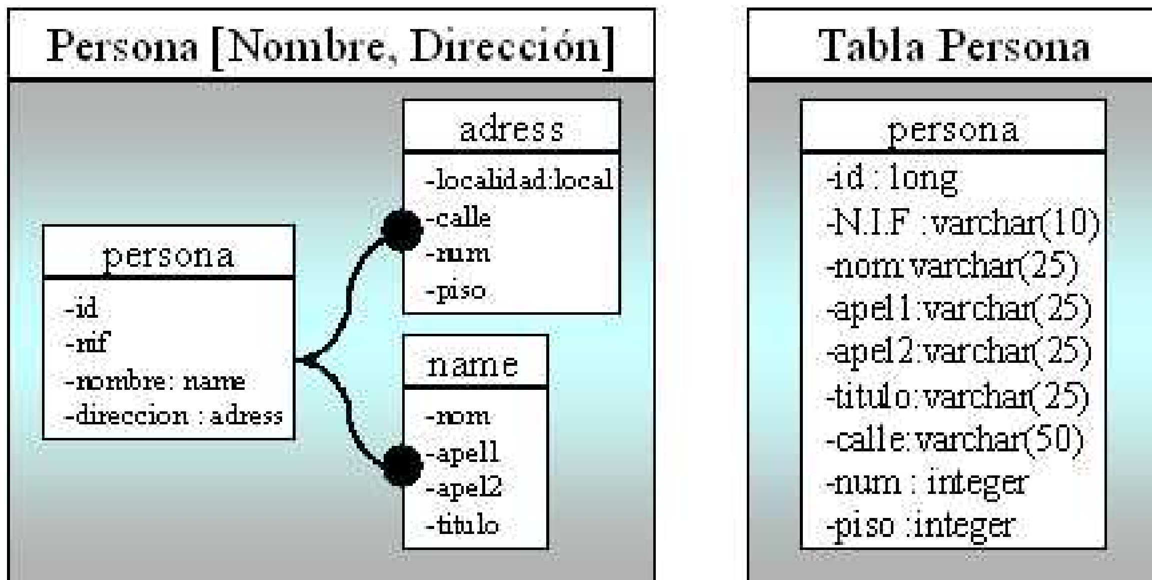
3. **El colector de basura de las colecciones**: Las colecciones son automáticamente persistidas cuando son referenciadas por un objeto persistente y también son borradas automáticamente cuando dejan de serlo.

12. Características de las colecciones

Por último si una colección es pasada de un objeto persistente a otro, sus elementos son movidos de una tabla a la otra. Lo bueno de Hibernate es que no nos tenemos que preocupar de esto, debemos utilizar las colecciones como normalmente lo hemos hecho. Desechándolas cuando sea necesario, Hibernate se ocupará de su gestión.

3.3. Componentes.

Un buen diseño relacional necesitará de la composición de objetos. El nombre de la persona, la dirección, una localidad etc.. son todo estructuras de datos que componen objetos más grandes. Una componente en Hibernate es un objeto persistente contenido dentro de la misma tabla de su propietario. ej: Clásico ejemplo del nombre/dirección de una persona :



7. Clase Persona y su composición.

El diseño del primer recuadro a parte de ser mucho más claro es muy sencillo de mapear, procedemos de la siguiente manera :

```
<class name="com.hecsua.bean.Person" table="persona">
        <id name="id" column="id" type="long">
                <gen....
        </id>
        <property name="nif" column="nif" type="string" length="10"/>
                <component name="nombre" class="com.hecsua.bean.Name">
                        <property name="nom" column="nombre"
type="string" length="25"/>
                        <property name="apel1" column="apel1"
type="string" length="25"/>
                        <property name="apel2" column="apel2"
type="string" length="25"/>
                        <property name="titulo" column="titulo"
type="string" length="25"/>
                </component>
                <component name="direccion"
class="com.hecsua.bean.Adress">
                        <property name="calle" column="calle"
type="string" length="50"/>
                        <property name="num" column="numero"
type="integer"/>
                        <property name="piso" column="piso"
type="integer"/>
                        <componente name="localidad" />
                                                ...
                </component>
</class>
```

La tabla persona tiene que contener los elementos correspondientes a los objetos name y adress. Así como al resto de las propiedades de persona. Los tipos de estas propiedades pueden ser cualquier tipo de los soportados por Hibernate, incluso más componentes

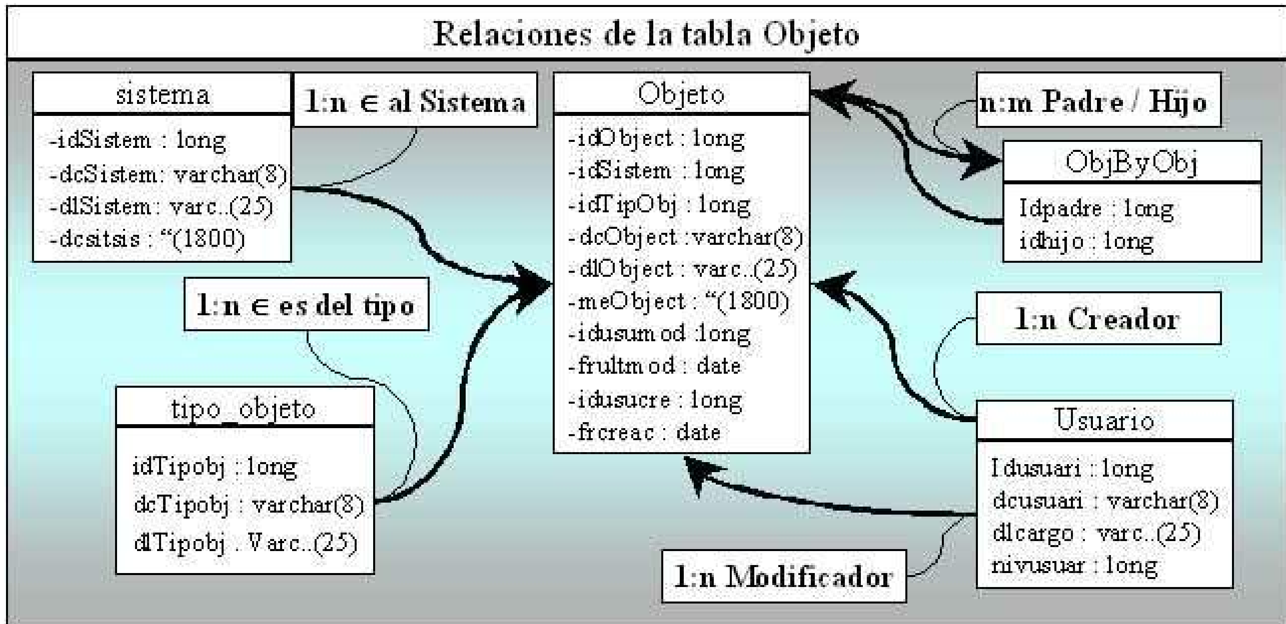
NOTA: Un buen ejercicio, antes de continuar con el siguiente capitulo, podría consistir en realizar el mapeo de Localidad (País, Provincia o Estado, Localidad, código) como una componente de Dirección.

, colecciones y relaciones. Por último destacar que cuando un componente es nulo todas sus propiedades lo son a la hora de guardarlo en la BBDD. En el caso contrario cuando

cargamos un objeto a memoria, si todas las propiedades del elemento son nulas el elemento es nulo.

3.4. 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:



8. Ejemplo de Diseño Relacional

3.4.1. 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.

3.4.2. 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.

3.4.3. 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>
```

3.4.4. 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" />
```

3.4.5. 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" />
```

3.4.6. 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, como podéis observar no es tan complicado como parece, eso sí, siempre que partamos de un buen diseño será mucho mas fácil. Ahora tan solo resta definir el resto de los objetos persistentes en sus correspondientes ficheros XML , generar las clases persistentes asociadas, y comenzar a utilizar Hibernate. Todo esto en el siguiente capitulo Herramientas, Configuración y Uso de Hibernate.

Aún nos faltarían cosas por hacer, que podeis realizar como ejercicio:

1. Crear y definir ficheros Usuario.hbm.xml, Sistema.hbm.xml y TipoObjeto.hbm.xml.
2. Modificar Usuario.hbm.xml para que contenga el nombre del usuario, la dirección (esta a su vez la localidad) como componentes.
3. Definir una relación entre Usuario y Sistema. Un Usuario tiene que pertenecer a un Sistema.
4. Validar mediante DTD de Hibernate que son ficheros XML bien formados
5. Crear carpetas donde dejarlos, en nuestro caso ./com/hecsua/bean.

11. Tareas a realizar.

Conclusión

No hemos utilizado ni la mitad de las posibilidades de Hibernate. Este abarca tal

cantidad de parámetros, configuraciones, etc.. que lo único que se conseguiría seria perderse entre todo. Por lo tanto tan solo me he limitado a explicar las características básicas del mapeo para crear una pequeña aplicación por la cual empezar. En posteriores versiones o anexos de este documento iré explicando todas las posibilidades, e incluso vosotros mismos me podréis comentar vuestros propios trucos o eso al menos espero.

Recursos y referencias

[1] Hibernate, *http://hibernate.bluemars.net*
[2] Guía hacia Hibernate, *http://www.javahispano.org/articles.article.action?id=80*
[3] Objeto.hbm.xml, Objeto.hbm.xml.sampleObjeto.hbm.xml
[4] Documentación sobre XML, *http://www.xml.org*
[5] Framework Collection de java, *http://java.sun.com*

Acerca del autor

Héctor es un estudiante de 4 de Informática en Gijón. Gracias al proyecto Fin de carrera comenzó a programar en Java. A parte de trabajar y de estudiar en sus ratos libres se dedica a salir de juerga siempre que puede, hacer algo de deporte e intentar en la medida de lo posible facilitar la iniciación en Java a todo el que pueda, al igual que hicieron con él.

Copyright (c) 2003, Héctor Suárez González. Este documento puede ser distribuido solo bajo los términos y condiciones de la licencia de Documentación de javaHispano v1.0 o posterior (la última versión se encuentra en http://www.javahispano.org/licencias/).