IES CHAN DO MONTE
C.S. de Desenvolveento de Aplicacions Multiplataforma

UNIDAD 4: HIBERNATE –ficheiros de mapeo

Índice

1. Estrutura dos ficheiros de Hibernate para o mapeo obxecto-relacional. Clases persistentes e mapeo

Hibernate ten dous ficheiros importantes que permiten o mapeo entre os obxectos Java e as táboas do SXBD relacional e son:

- **As clases Java** (a estas clases Hibernate refírese como `POJOS`), que representan os obxectos que teñen unha correspondencia coas táboas da base de datos relacional.
- **O ficheiro de mapeo** (`.hbm.xml`), que indica o mapeo entre os atributos dunha clase e os campos da táboa relacional coa que está relacionado.

Nota: Hai outra alternativa para especificar o mapeo entre os obxectos java e as táboas, que é utilizar Anotacións JPA sobre as clases persistentes, en vez do ficheiro xml de mapeo.

Nesta actividade imos utilizar os ficheiros de mapeo XML.

1.1 Clases persistentes

As clases persistentes son clases que nunha aplicación implementan as entidades do problema empresarial (por exemplo: Cliente, Pedido, etc). Representan os obxectos nunha aplicación que use Hibernate e correspóndense coa información almacenada nun SXBD relacional.

Hibernate traballa con estas clases, sempre e cando sexan *POJOS* (Plain Old Java Object.- Obxectos Simples). Estas clases deben cumprir as especificacións JavaBeans:

- Implementar un *construtor sen argumentos*.
- Proporcionar unha *propiedade identificadora* (opcional).
- Métodos *getter* e *setter* para cada atributo.
- Implementar a *interface Serializable*. Non é obrigatorio implementar esta interface, pero si recomendable.

Exemplo simple de POJO:

```
public class Persoa implements Serializable {
    private int id;
    private String nome;
    private String apelido1;
    private String apelido2;
    private String email;
    private String telefono;
    Persoa() {    }
    Persoa(int id, String nome, String apelido1, String apelido2,
            String email, String telefono) {
        this.id = id;
        this.nome = nome;
        this.apelido1 = apelido1;
        this.apelido2 = apelido2;
        this.email = email;
        this.telefono = telefono;
    }
    public String getApelido1() {      return apelido1;    }
    public void setApelido1(String apelido1) {  this.apelido1 = apelido1;  }
    public String getApelido2() {    return apelido2;    }
    public void setApelido2(String apelido2) {    this.apelido2 = apelido2;    }
    public String getEmail() {        return email;    }
    public void setEmail(String email) {       this.email = email;    }
    public int getId() {        return id;    }
    public void setId(int id) {  this.id = id;    }
    public String getNome() {        return nome;    }
    public void setNome(String nome) {        this.nome = nome;    }
    public String getTelefono() {        return telefono;    }
    public void setTelefono(String telefono) {        this.telefono = telefono;    }
}
```

O atributo `"id"` manterá un valor único que identificará a cada unha das instancias de `"Persoa"`.

Todas as clases de entidades persistentes deben ter unha propiedade que sirva como identificador, se queremos usar o conxunto completo de funcionalidades que nos ofrece Hibernate.

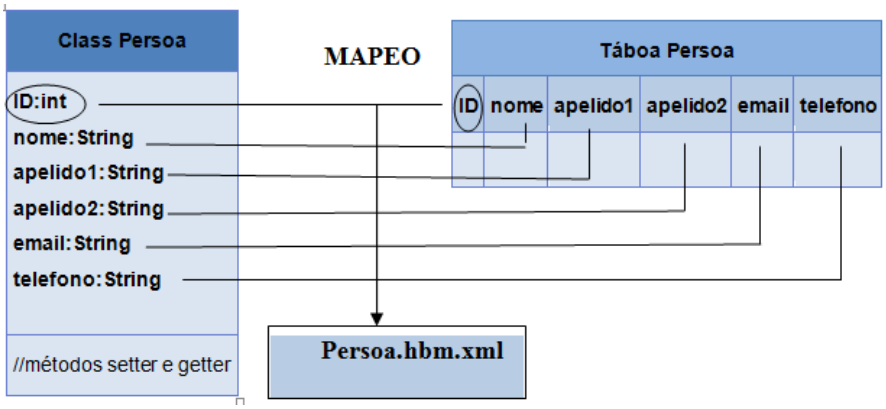
Na tarefa 3 faremos exercicios de creación das clases persistentes para poder representar os obxectos que teñan unha correspondencia coas táboas da base de datos relacional.

1.2 Ficheiro de mapeo

Establece a correspondencia entre os obxectos java e as táboas da base de datos. Asignaráselle o nome da clase, seguido da extensión `.hbm.xml`.

Os ficheiros de mapeo que se utilizan na aplicación, defínense en `hibernate.cfg.xml`, indicando a ruta onde se atopan estes.

Para cada clase que se queira facer persistente (por exemplo, a `clase Persoa`) na base de datos, crearase un ficheiro XML, coa información que permitirá mapear esta clase a unha táboa da base de datos relacional.



Exemplo simple de ficheiro `Persoa.hbm.xml`, que realiza o mapeo á táboa `Persoa` almacenada no SXBD SQL Server :

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class name="Persoahibernate.Persoa" table="Persoa" schema="dbo"
                catalog="Persoas">
        <id name="id" type="int">
            <column name="id" />
            <generator class="assigned" />
        </id>
        <property name="nome" type="string">
            <column name="nome" length="50" />
        </property>
        <property name="apelido1" type="string">
            <column name="apelido1" length="50" />
        </property>
        <property name="apelido2" type="string">
            <column name="apelido2" length="50" />
        </property>
        <property name="email" type="string">
            <column name="email" length="50" />
        </property>
        <property name="telefono" type="string">
            <column name="telefono" length="50" />
        </property>
    </class>
</hibernate-mapping>
```

Os elementos do ficheiro XML de mapeo, descríbense a continuación:

Doctype

Todos os mapeos XML deben declarar *o tipo de documento DTD*.

```
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
```

Mapeo de Hibernate: <Hibernate-mapping>

É o elemento raíz que contén todos os elementos `<class>`. Dentro del decláranse as clases dos obxectos persistentes.

Este elemento ten varios atributos opcionais que especifican os valores que por defecto terán os demais elementos.

```
<hibernate-mapping
      schema="nome_esquema "
      catalog="nome_catalog"
     default-cascade="estilo_cascade"
     default-access="field|property|ClassName"
     default-lazy="true|false"
     auto-import="true|false"
     package="nome_paquete"
/>
```

- **schema (opcional):** Nome dun esquema da base de datos. Se se especifica, os nomes das táboas cualifícanse co nome do esquema dado. Se se omite, os nomes das táboas non se cualifican.
- **catalog (opcional):** Nome dun catálogo da base de datos. Se se especifica, os nomes das táboas cualifícanse co nome do catálogo dado.

  estes valores aplicaranse a todas as clases mapeadas dentro dese bloque de <hibernate-mapping>. É dicir, son valores por defecto para todas as clases nese bloque.
- **default-cascade** (opcional, o valor por defecto é none): Estilo de cascada por defecto. Define como os cambios no obxecto afectan aos obxectos relacionados con el. Isto é especialmente relevante cando se traballa con obxectos que teñen relacións con outros obxectos, como as relacións un-a-moitos, moitos-a-un, etc.

  Os valores posibles para default-cascade son:

  - **none**: Non se propaga ningún cambio ao obxecto relacionado. Este é o valor por defecto.
  - **save-update**: Os cambios no obxecto principal propagaranse ao obxecto relacionado cando se garda ou se actualiza o obxecto principal.
  - **delete**: O obxecto relacionado será eliminado cando se elimine o obxecto principal.
  - **all**: Todos os cambios (gardar, actualizar, eliminar) no obxecto principal propagaranse ao obxecto relacionado.
  - **all-delete-orphan**: Similar a all, pero ademais, se un obxecto relacionado é eliminado da colección no obxecto principal, tamén será eliminado da base de datos.

    Por exemplo, imaxina que tes un obxecto Usuario que ten unha relación un-a-moitos con un obxecto Pedido. Se configuras default-cascade a all, cando gardas ou actualizas un Usuario, todos os Pedidos relacionados tamén se gardarán ou actualizarán. Ademais, cando eliminas un Usuario, todos os seus Pedidos relacionados tamén se eliminarán.
- **default-access** (opcional, o valor por defecto é property): Estratexia que Hibernate debe utilizar para acceder aos datos da clase. Hai dúas estratexias principais:

- **property**: Esta é a opción por defecto. Cando se usa property, Hibernate accede aos datos dunha clase a través dos métodos getter e setter. Por exemplo, se tes unha clase Usuario con un campo nome, Hibernate usará os métodos getNome() e setNome(String nome) para ler e escribir o valor do campo nome.
- **field**: Cando se usa field, Hibernate accede directamente aos campos dunha clase, ignorando os métodos getter e setter. No exemplo anterior, Hibernate accedería directamente ao campo nome da clase Usuario.

  A estratexia property é máis común porque permite encapsular a lóxica adicional nos métodos getter e setter. Por outro lado, a estratexia field pode ser útil se queres que Hibernate ignore esa lóxica adicional.

- **default-lazy** (opcional, o valor por defecto é true): Ao especificar unha relación nun mapeo de Hibernate, usamos o atributo "lazy" para definir cando Hibernate trae a información da táboa relacionada. Por defecto, lazy é igual a true (carga diferida), o que significa que a información da táboa relacionada non se recupera da base de datos ata que se realiza algunha operación sobre ela.
- **auto-import**: Cando esta opción está activada (o valor por defecto é true), pódese usar nomes de clases non cualificados na linguaxe de consulta. Por exemplo, unha clase chamada Usuario no código, pódese facer unha consulta como FROM Usuario en lugar de FROM com.mipaquete.Usuario.
- **package**: Esta opción permite especificar un prefixo de paquete que se debe usar para os nomes de clase non cualificados no documento de mapeo. Por exemplo, unha clase chamada Usuario no paquete com.mipaquete, pódese usar só Usuario no teu documento de mapeo se especificas package="com.mipaquete" na túa configuración de Hibernate.

Aínda que o elemento `hibernate-mapping` permite aniñar varios mapeos `<class>` persistentes, é unha boa práctica que se mapee somentes unha clase persistente, ou unha soa xerarquía de clases, nun arquivo de mapeo e nomealo como a superclase persistente. Por exemplo: Rectángulo.hbm.xml, Circulo.hbm.xml, ou se utiliza herdanza, Figura.hbm.xml.

Clase < class name>

O elemento class declara unha clase persistente que se vai almacenar na base de datos, os seus atributos e a súa relación coa táboa da base de datos.

```
<class name="Persoahibernate.Persoa" table="Persoa" schema="dbo"
       catalog="Persoas">
```

Algúns atributos (non están todos) que se poden declarar son os seguintes:

```
<class
    name="nome clase"
    table="nome táboa"
    mutable="true|false"
    schema="esquema"
    catalog="catálogo"
    proxy="interfaz Proxy"
    lazy="true|false"
    dynamic-update="true|false"
    dynamic-insert="true|false"
    select-before-update="true|false"
    entity-name="nome entidade"
    optimistic-lock="none|version|dirty|all"
    rowid="ROWID"
    subselect="expression SQL"
/>
```

- **name**: Nome completamente cualificado da clase Java. Por exemplo, com.mipaquete.Usuario.
- **table** (opcional). É o nome da táboa na base de datos á que estás mapeando a túa clase. Se non se especifica, Hibernate usará o nome da clase.
- **mutable** (opcional - true por defecto): Especifica se ás instancias son mutables ou non. Ás clases inmutables non se poden modificar nin eliminar. Por exemplo, unha clase HistorialUsuario que se quere que sexa inmutable, pdese configurar mutable a false
- **schema e catalog :** son opcións que se usan para especificar o esquema e o catálogo da base de datos, respectivamente
- **proxy** (opcional): Especifica unha interface ou clase a utilizar para os proxies de preguiceira (lazy). Pódese especificar o nome mesmo da clase. É moi útil cando se produce unha asociación entre dous obxectos (non é necesario traerse os dous), ou cando se carga un obxecto que ten coleccións (non me traio todos os datos da asociación). O obxecto proxy, o único que contén é o identificador.

  Cando se solicita un obxecto dunha base de datos, non necesariamente se obterá o obxecto real de inmediato. En cambio, podes obter un obxecto "proxy" que parece e se comporta como o obxecto real, pero que non carga os datos reais da base de datos ata que realmente necesites usar eses datos (isto é o que chamamos "carga preguiceira" ou "lazy loading").

  Por exemplo, imaxina que tes un obxecto Usuario que ten unha relación one-to-many con obxectos Pedido. Cando cargas un Usuario da base de datos, Hibernate pode crear proxies para os Pedidos relacionados en lugar de cargar todos os Pedidos de inmediato. Estes proxies dos Pedidos non cargarán os seus datos reais da base de datos ata que intentes acceder a algún dato dun Pedido (por exemplo, chamando a un método getter nun Pedido).
- **lazy** (opcional): Permite deshabilitar a recuperación preguiceira. Se é false, Hibernate cargará todos os datos relacionados cando cargues o obxecto principal.
- **dynamic-update** (opcional - false por defecto): A sentenza SQL UPDATE xérase en tempo de execución, e se é false somentes contén as columnas con valores que cambiasen.
- **dynamic-insert** (opcional - false por defecto): A sentenza SQL INSERT xérase en tempo de execución, e se é false somentes contén as columnas con valores distintos a null.
- **select-before-update** (opcional - false por defecto): Prodúcese unha sentenza SELECT antes de actualizar para comprobar se un obxecto foi modificado e a actualización é necesaria. Se non, non se produce a devandita actualización.
- **entity-name** (opcional - por defecto o nome da clase): Nome que se usa en lugar do nome de clase. Hibernate permite mapear unha mesma clase distintas veces (con distintas táboas), e este nome será o utilizado.
- **optimistic-lock** (opcional - versión por defecto): O bloqueo optimista é unha estratexia que Hibernate usa para manexar o problema de concorrencia na base de datos. A concorrencia ocorre cando múltiples transaccións tentan acceder e modificar os mesmos datos ao mesmo tempo.

  Na estratexia de bloqueo optimista, Hibernate permite que múltiples transaccións lean os mesmos datos sen bloqueos. Cando unha transacción quere modificar os datos, Hibernate comproba se algún outro modificou os datos desde que a transacción os leu. Se os datos foron modificados, Hibernate aborta a transacción e lanza unha excepción de concorrencia.
- **rowid** (opcional): Indica que identificador de columna debe ser usado. A opción rowid en Hibernate permite especificar un identificador de columna que debe ser usado.

  O rowid é un tipo de dato que moitas bases de datos usan para almacenar un valor único para cada fila nunha táboa. Este valor é xerado automaticamente pola base de datos e non pode ser modificado.

Podes usar o rowid para recuperar unha fila específica de forma moi eficiente. Non obstante, o uso de rowid é bastante específico da base de datos e non é portátil entre diferentes sistemas de xestión de bases de datos.

- **subselect** (opcional): Permite mapear unha entidade a unha subconsulta da base de datos. Isto é especialmente útil cando queres mapear a túa clase a unha vista en lugar dunha táboa.

```
<class name="com.mipaquete.Usuario" entity-name="UsuarioSubselect">
    <subselect>
        SELECT * FROM USUARIO WHERE ID > 100
    </subselect>
    <!-- máis configuracións aquí -->
</class>
```

A clase Usuario está mapeada a unha subconsulta que selecciona todos os usuarios cuxo ID é maior que 100. Isto significa que cando cargues instancias da clase Usuario, Hibernate executará esa subconsulta e só obterás os usuarios cuxo ID é maior que 100.

Un uso común para isto sería cando queres traballar con un subconxunto dos datos que cumpren certas condicións, pero non queres ou non podes crear unha vista na base de datos para iso. Neste caso, podes usar Subselect para definir a subconsulta directamente na túa clase de entidade.

Declaración columna de clave primaria: < id>

As clases mapeadas teñen que declarar a columna de clave primaria da táboa da base de datos.

```
<id name="id" type="int">
    <column name="id" />
    <generator class="assigned" />
</id>
```

Hibernate necesita coñecer cal é a *estratexia elixida para a xeración de claves primarias*. Moitas bases de datos usan claves primarias naturais, claves que teñen un significado no mundo real (CIF, NSS, ...). Nalgúns casos as claves naturais están compostas de varias columnas, o que fai o mantemento, consultas e a evolución do sistema máis difícil. Unha recomendación é utilizar claves artificiais (ou suplentes), que non teñen significado para a aplicación e que son xeradas automaticamente polo sistema, e definir restricións de unicidade nas claves naturais.

O elemento que define o mapeo da columna de clave primaria é a etiqueta `<id>`. Sintaxe:

```
<id
    name="nome do atributo"
    type="tipo de datos"
    column="nome  da columna"
    access="field|property|ClassName">
    <generator class="generatorClass"/>
</id>
```

- **name** (opcional): Nome do atributo da clase persistente que vai ser un identificador. Si se omite name, asúmese que a clase non ten propiedade identificadora.
- **type** (opcional): Nome que indica o tipo de datos de Hibernate.
- **column** (opcional - por defecto é o nome do atributo): Nome da columna de clave primaria na base de datos.
- **access** (opcional - property por defecto): Estratexia de acceso ao valor da propiedade.
- **generator** (opcional): Ao establecer unha clave primaria, débese especificar a forma en que esta se xera. Hibernate ten implementada varias formas de xerar a clave primaria, pero non todas as formas son portables a todos os SXBD. Todos os xeradores implementan a interface `org.hibernate.id.IdentifierGenerator`. Algúns xeradores incorporados son os seguintes:

assigned: Esta estratexia permite á túa aplicación asignar o valor do identificador. Deixa á aplicación asignar un identificador ao obxecto antes de que se chame ao método save(). Esta é a estratexia por defecto se non se especifica un elemento <generator>.Isto é útil cando os teus identificadores teñen un significado no dominio do teu problema. Por exemplo, se estás mapeando unha clase Usuario e usas o nome de usuario como identificador, podes usar a estratexia assigned.

increment: Esta estratexia xera identificadores incrementais. Cada vez que necesitas un novo identificador, Hibernate incrementa o valor máximo actual na columna do identificador. Esta estratexia é simple e eficiente, pero só debe usarse cando a túa aplicación ten acceso exclusivo á base de datos.

identity: Esta estratexia usa columnas de identidade proporcionadas pola base de datos. Moitas bases de datos soportan columnas de identidade que xeran automaticamente un valor único cada vez que se inserta unha nova fila. Esta estratexia é moi eficiente e permite á base de datos encargarse da xeración do identificador.

sequence: Esta estratexia usa secuencias de base de datos para xerar identificadores. Unha secuencia é un obxecto de base de datos que xera unha secuencia de números únicos. Cando necesitas un novo identificador, solicitas o seguinte valor da secuencia.

hilo e seqhilo: Estas estratexias usan un algoritmo "high/low" para xerar identificadores de forma eficiente. O algoritmo high/low xera identificadores que son únicos para a base de datos. Os valores high obtéñense dunha fonte global e fanse únicos engadindo un valor low. A estratexia seqhilo é similar, pero usa unha secuencia de base de datos para xerar os valores high.

uuid e guid: Estas estratexias xeran identificadores únicos a nivel global. uuid usa un algoritmo UUID para xerar un identificador de 128 bits, mentres que guid usa unha cadea GUID xerada pola base de datos.

native: Esta estratexia permite a Hibernate escoller a mellor estratexia baseándose nas capacidades da base de datos.

select e foreign: Estas estratexias son un pouco diferentes. select recupera un identificador asignado por un disparador da base de datos, mentres que foreign usa o identificador dun obxecto asociado.

Claves compostas : <Composite-id>

Define claves compostas por varios campos. Hai varias alternativas para definir un identificador composto e a máis recomendada é a de implementalo como unha clase compoñente aparte. Os atributos descritos a continuación aplícanse só a este enfoque alternativo:

Algúns atributos (non están todos) que se poden declarar son os seguintes

```
<composite-id
    name=" nome atributo"
    class="nome da clase"
    access="field|property|ClassName"
    <key-property name="nome atributo" type="tipo de datos" column="nome da columna"/>
        ......
</composite-id>
```

- **name** (obrigatorio para esta alternativa): Nome da propiedade que contén o identificador composto.
- **class** (opcional): A forma de implementar unha clave composta en Hibernate é a través dunha clase. Esta propiedade indica o nome da devandita clase que conterá atributos para cada unha das columnas que compoñen a clave.

- **access** (optativo, por defecto, property): Estratexia que Hibernate debería usar para acceder ao valor das propiedades.
- **key-property**: Dentro da etiqueta composite-id, indica cada un dos campos que forman a clave e que corresponden ás claves foráneas doutras táboas.

  name: Nome da propiedade da clase que implementa o identificador.

  type (opcional): Tipo Hibernate da propiedade.

  column (opcional - nome da propiedade por defecto): Nome da columna.

Exemplo: A clave de `EmpregadoProxecto` é composta por `nssEmpregado` e `numProxecto`, creamos unha clase a parte para implementar este identificador composto.

```
public class EmpregadoProxecto  implements java.io.Serializable {
     private EmpregadoProxectoId id;
     private Empregado empregado;
     private Proxecto proxecto;
     private int horasSemanais;

    public EmpregadoProxecto() {    }
   //métodos getter e setter
        …….
        …….
}

public class EmpregadoProxectoId  implements java.io.Serializable {
     private String nssEmpregado;
     private int numProxecto;

    public EmpregadoProxectoId() {   }

   //métodos getter e setter

        …….
        …….
}
```

No ficheiro `EmpregadoProxecto.hbm.xml`, a clave sería declarada como:

```
<composite-id name="id" class="empresahibernate.EmpregadoProxectoId">
        <key-property name="nssEmpregado" type="string">
           <column name="NSS_Empregado" length="15" />
        </key-property>
        <key-property name="numProxecto" type="int">
           <column name="Num_proxecto" />
        </key-property>
 </composite-id>
```

Non se pode utilizar un xerador de claves para xerar claves compostas. En cambio, a aplicación debe asignar os seus propios identificadores.

Propiedade. - <Property>

O elemento <property> declara unha propiedade dunha clase persistente, estilo JavaBean da clase.

```
<property name="nome" type="string">
            <column name="nome" length="50" />
 </property>
```

Algúns dos seus atributos son:

```
<property
   name="nome atributo"
   type="tipo de datos"
   column="nome da columna"
   update="true|false"
   insert="true|false"
   formula=" expresión SQL "
   lazy="true|false"
   unique="true|false"
   not-null="true|false"
   generated="never|insert|always"
   unique_key="unique_key_id"
/>
```

- **name**: Nome da propiedade, coa letra inicial en minúscula.
- **column** (opcional - por defecto é o nome da propiedade): Nome da columna da táboa de base de datos mapeada.
- **type** (opcional): Nome que indica o tipo de datos de Hibernate.
- **update, insert** (opcional - por defecto é true): Especifica que as columnas mapeadas deben ser incluídas nas declaracións SQL UPDATE e/ou INSERT. Especificando ambas as dúas como `false`, permite unha propiedade "derivada", o seu valor iníciase dende algunha outra propiedade que mapee á mesma columna (ou columnas), por un disparador ou por outra aplicación.
- **formula** (opcional): Expresión SQL que define o valor para unha propiedade calculada. As propiedades calculadas non teñen unha columna mapeada propia.
- **lazy** (opcional - por defecto é false): Especifica que se debe recuperar preguiceiramente esta propiedade cando se acceda por primeira vez á variable de instancia.
- **unique** (opcional): Restrición de unicidade para a columna.
- **not-null** (opcional): Restrición de nulabilidade para a columna.
- **generated** (opcional - por defecto é never): Especifica que o valor da propiedade é xerado pola base de datos.

Tipos de datos Hibernate

El mapeo implica a conversión entre tipos de datos de JAVA e tipos de datos SQL. Hibernate xestiona esta conversión por medio dos seus propios tipos de datos para mapeo. Cada tipo Hibernate ten un equivalente en JAVA e en ANSI SQL.

Mediante o "dialecto SQL" do SXBD, Hibernate adaptará os seus tipos de nativos aos tipos de datos SQL. Así consíguese independencia fronte a un SXBD específico. Tamén se poden usar tipos Java, ou mesmo non indicalos, pero é menos convinte.

Tipo	Tipo Java	Tipo ANSI SQL
integer	int / java.lang.Integer	INTEGER
long	long / java.lang.Long	BIGINT
short	short / java.lang.Short	SMALLINT
float	float / java.lang.float	FLOAT
double	double / java.lang.Double	DOUBLE
big_decimal	java.math.BigDecimal	NUMERIC
character	java.lang.String	CHAR(1)
string	java.lang.String	VARCHAR
byte	byte / java.lang.Byte	TINYINT
boolean	boolean / java.lang.Boolean	BIT
yes_no	boolean / java.lang.Boolean	CHAR(1) ('Y' o 'N')
true_false	boolean / java.lang.Boolean	CHAR(1) ('T' o 'F')
date	java.util.Date / java.sql.Date	DATE
time	java.util.Date / java.sql.Time	TIME
date	java.util.Date / java.sql.Date	DATE
time	java.util.Date / java.sql.Time	TIME
timestamp	java.util.Date / java.sql.Timestamp	TIMESTAMP
calendar	java.util.Calendar	TIMESTAMP
calendar_date	java.util.Calendar	DATE

2. Estratexias no mapeo. Entidades e Tipos Valor

Un obxectivo principal de Hibernate é o apoio aos *modelos de obxectos de gran fino*, como un dos requisito máis importante para os mapeos de datos. É unha razón pola cal se traballa con POJOS. O termo de gran fino significa "*máis clases que táboas*". Isto significa que nas filas dunha táboa da base de datos pódese representar máis dun único obxecto e que non todas as clases que teñamos nun proxecto van ter correspondencia unívoca cunha táboa na base de datos.

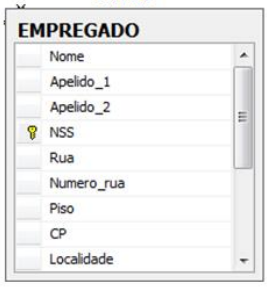
Exemplo: unha clase Cliente con propiedades para o enderezo de facturación (rúa,cidade, código postal, localidade) e para o envío. Pode que estas informacións estean na base de datos nunha única táboa Cliente pero que dende o punto de vista da codificación en Java, pode ser convinte ter unha clase Enderezo con estes datos, ou ter a propiedade dirección de correo electrónico, en vez dunha propiedade de tipo cadea, como unha propiedade dunha clase Email, o que pode engadir un comportamento más sofisticado, ao poder definilo mediante métodos, por exemplo, pode ofrecer un método sendEmail().

No seguinte exemplo, amósanse *dúas clases que se mapean nunha única táboa* da base de datos:

```
public class Empregado  implements java.io.Serializable {
     private String nss;
     private Empregado empregado;
     private String nome;
     private String apelido1;
     private String apelido2;
     private Enderezo enderezo;

 public class Enderezo implements Serializable{
      private String rua;
      private int numeroRua;
      private String piso;
      private String cp;
      private String localidade;
```

Táboa



Este problema de granularidade lévanos a facer unha distinción importante dentro de Hibernate. Si pensamos nun deseño de gran fino ("máis clases que táboas"), unha fila representa a múltiples instancias de diferentes clases. Dado que a identidade dunha táboa da base de datos é implementada mediante a clave primaria, entón algúns obxectos persistentes non teñen a súa propia identidade, é dicir, un dos obxectos representados na fila ten a súa propia identidade (exemplo anterior sería empregado), e outros dependen deste e non teñen un valor identidade (exemplo enderezo). Hibernate fai a distinción esencial seguinte:

- Entidades.
- Tipo Valor (compoñente).

2.1 Entidades

- Un obxecto de tipo de entidade ten unha identidade propia na base de datos (valor de clave principal).
- Unha referencia de obxecto a unha instancia de entidade é persistido como unha referencia na base de datos (un valor de clave externa).
- A entidade ten o seu propio ciclo de vida, e pode existir independentemente de calquera outra entidade.

- O estado persistente dunha entidade consiste en referencias a outras entidades, e a instancias do que en Hibernate se denomina "value types" (tipos "valor").

  Exemplo: Departamento, Empregado, Proxecto.

2.2 Tipo Valor

- Un obxecto de tipo valor non ten unha identidade de base de datos, senón que pertence a unha instancia dunha entidade e o seu estado persistente incrústase na fila da táboa á que pertence. Obxectos persistentes sen identidade propia (nin táboa propia).
- Os tipos de valor non teñen identificadores ou propiedades identificadoras.
- A vida útil dunha instancia de tipo de valor está limitada pola vida útil da pertenza a instancias de entidade.
- Un tipo de valor non é compatible con referencias compartidas.

Exemplo: Enderezo.

2.3 Mapeamento de coleccións

Hibernate ofrece distintas formas para almacenar coleccións. Cando Hibernate almacena unha colección, almacena tamén toda a semántica asociada ao seu interface. Por exemplo, cando unha lista (interface List) é almacenada, o índice que identifica a orde tamén é almacenado. Desa forma cando esta sexa recuperada da base de datos o seu comportamento será idéntico ao inicial.

Hibernate soporta as coleccións que se amosan na seguinte táboa:

Colección (interface)	Etiqueta de Hibernate	Instancia requirida (clase Java que implementa a interface)	Observacións
java.util.Set	**<set>**	java.util.HashSet.	**Colección non ordenada de valores de obxectos sen repeticións.** A orde dos elementos non é preservada. Non se permiten elementos repetidos.
java.util.SortedSet	**<set>**	java.util.TreeSet	Set non permite ordenar os seus elementos. Para que **os elementos dun conxunto teñan orde**, Java proporciona un tipo SortedSet que herda de Set. A clase que implementar SortedSet chámase TreeSet
java.util.List	**<list>**	java.util.ArrayList	**Colección ordenada de valores de obxectos, admite repetidos**. Respecta a orde dos elementos sempre e cando se especifique unha columna que se utilice como índice.
java.util.Collection	**<bag> ó <idbag>.**	java.util.ArrayList	Java non conta cunha interface ou implementación para bag, aínda que java.util.Collection permite **utilizar a semántica das bags permitindo duplicados** e a orde non se mantén entre os elementos.
java.util.Map	**<map>**	java.util.HashMap	Un mapa de java pode ser utilizado preservando os **pares (clave, valor).**

Os arrays están soportados por Hibernate con `<primitive-array>` (para tipos primitivos Java) e `<array>` (para outros) pero úsanse raramente.

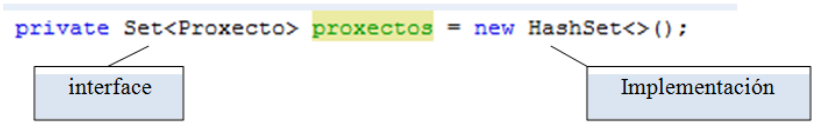
Hibernate recomenda usar interfaces nas declaracións de tipo das coleccións Java para poder realizar o mapeo das coleccións correctamente. O procedemento recomendado é:

- Usar unha interface para declarar o tipo da colección (ex: Set).
- Elixir unha clase que implementar á interface (ex: HashSet) e iniciala na declaración sen esperar a facelo no construtor ou nalgún método setter. Este é o método recomendado.

A sintase para declarar unha propiedade de colección é:

```
privada < Interface > nome_coleccion = new < Implementación> ();
...
/ / Os métodos getter e setter
```

Exemplo:



As coleccións poden conter*: tipos básicos, entidades e compoñentes*. Non se poden crear coleccións de coleccións.

2.3.1 Atributos comúns dos elementos de colección

A maioría dos elementos de colección comparten atributos que se mostran a continuación:

```
tag set|map|list|bag|idbag>|
  name="nome propiedade"
  table="nome táboa"
  schema="nome esquema"
  lazy="true|extra|false"
  inverse="true|false"
  cascade="all|none|save-update|delete|all-delete-orphan|delete-orphan"
  order-by="nome columna asc|desc"
  where="condición SQL where "
  fetch="join|select|subselect"
  batch-size="numero"
  access="field|property|ClassName"
  optimistic-lock="true|false"
  mutable="true|false"
</tag >
```

- **name**: Nome da propiedade colección.
- **table**: Nome da táboa da base de datos que contén a relación.
- **schema**: Nome do esquema da base de datos.
- **lazy** (opcional - por defecto é false): Especifica que se debe recuperar preguiceiramente esta propiedade cando se acceda por primeira vez á variable de instancia.
- **inverse** (defecto false): Declárase nas relacións un a moitos ou moitos a moitos. Sirve para decidir cal das dúas relacións é a propietaria para manexar a relación e facer as operacións de inserción e actualización na columna foránea. No caso das relacións moitos a moitos é necesario poñer un lado da relación co valor true e o outro co valor false.

- **cascade** (opcional- por defecto none): Indica a acción a determinar cos obxectos relacionados cando no obxecto pai se realiza unha operación de actualización (inserción, borrado ou modificación).
- **order-by** (opcional): Nome de columna que define a orde de iteración.
- **where** (opcional): Condición WHERE a utilizar cando se recuperan ou eliminan obxectos da colección.
- **fetch** (opcional - select por defecto): Elixe entre busca mediante outer join ou por select secuencial.
- **batch-size** (opcional -por defecto1): Especifica o número de elementos da colección que se poden recuperar cando se instancia a clase pola propiedade identificadora.
- **access** (opcional - property por defecto): Estratexia utilizada para acceder á propiedade.
- **optimistic-lock** (opcional - true por defecto): Establece estratexia de bloqueo optimista.
- **mutable** (opcional - true por defecto): Establece si os elementos da colección poden cambiar ou non.

2.3.2 Etiqueta Key. Mapeo da clave foránea

Indica a *clave foránea* da táboa referenciada. Os seus atributos son:

```
<key
      column="nome columna"
      on-delete="noaction|cascade"
      property-ref="nome propiedade"
      not-null="true|false"
      update="true|false"
      unique="true|false"
/>
```

- **column**: Nome da columna da clave foránea.
- **on-delete** (opcional - por defecto é no action): Co valor a true especifícase que a restrición de clave foránea ten o borrado en "`cascade`" activado a nivel de base de datos.
- **property-ref** (opcional): Especifica que a clave foránea referencia columnas que non son a clave principal da táboa orixinal. Proporciónase para os datos herdados.
- **not-null** (opcional): Co valor a true especifícase que as columnas da clave foránea son non nulables. Isto aplícase cando a clave foránea tamén é parte da clave principal. Tamén hai que declaralo a true nunha relación un a moitos unidireccional si a clave foránea non admite valores nulos.
- **update** (opcional): Co valor a false especifícase que a clave foránea nunca se debe actualizar. Isto aplícase cando a clave foránea tamén é parte da clave principal.
- **unique** (opcional): Co valor a true especifícase que a clave foránea debe ter unha restrición de unicidade.

2.3.3 Mapeo dos elementos da colección

As coleccións poden conter: tipos básicos, compoñentes e referencias a outras entidades.

Un elemento nunha colección pode ser manexado como:

- Tipo valor : o seu ciclo de vida depende completamente do obxecto propietario da colección. Os tipos básicos, os personalizados e os compoñentes mapéanse como este tipo.

  Estes elementos son mapeados por `<element>` para os tipos básicos ou `<composite-element>` para os compoñentes.

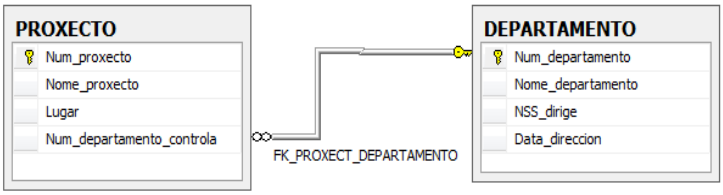
- Referencia a outra entidade: teñen o seu propio ciclo de vida. Son as asociacións entre obxectos.

  Son mapeados con `<one-to-many>` ou `<many-to-many>`.

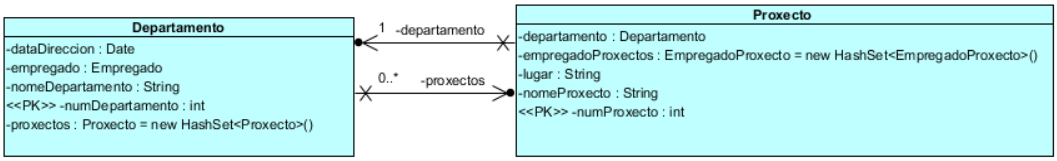
2.3.4 Mapeo dun Set

Unha colección de entidades Set mapéase co elemento `<set>`. Esta colección non permite elementos duplicados e non é necesario un índice xa que os elementos están sen ordenar.

Exemplo: Na base de datos temos a seguintes relacións:



O diagrama de clases UML é:



As relacións entre clase, na programación OO, represéntanse mediante *referencias a obxectos* e *coleccións de referencias a obxectos*. A través destas referencias, podemos navegar dende un obxecto a outro. Neste exemplo, estamos a utilizar direccionalidade nos dous sentidos, por que ambas as dúas clases teñen referencia unha coa outra.

Como un departamento pode controlar de 0 a N proxectos, para representar está relación temos que engadir na clase `Departamento`, unha referencia mediante unha colección á clase`Proxecto`. Para realizar esta referencia, podemos utilizar unha colección `Set`, xa que neste caso non pode haber elementos repetidos e sen orde establecida.

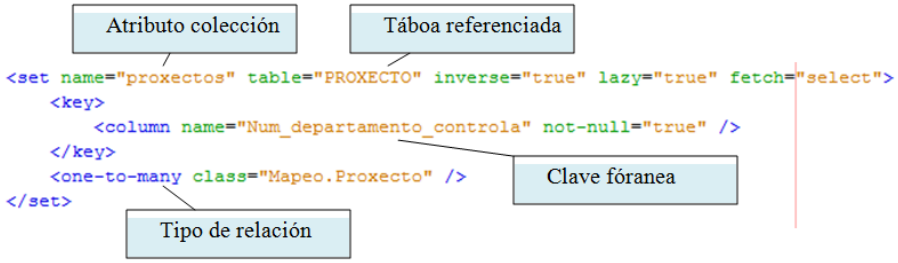
```
public class Departamento  implements java.io.Serializable {
     private int numDepartamento;
     private Empregado empregado;
     private String nomeDepartamento;
     private Date dataDireccion;
     private Set<Proxecto> proxectos = new HashSet<Proxecto>();
```

Na clase Proxecto engadimos unha referencia á clase Departamento, para representar que un proxecto ten que ser controlado por un só departamento.

```
public class Proxecto  implements java.io.Serializable {
     private int numProxecto;
     private Departamento departamento;
     private String nomeProxecto;
     private String lugar;
```

As instancias dunha colección son diferenciadas na base de datos mediante *unha clave foránea* do obxecto relacional ao que pertence. Esta clave é denominada a clave da colección. Esta clave será mapeada co `tag < key>`.

– No ficheiro de mapeo `Departamento.hbm.xml` temos que mapear a colección set.



2.3.5 Mapeo dunha lista

As listas poden conter elementos duplicados e están indexadas, utilízase un índice para acceder e buscar elementos na lista a través da súa posición.

Unha propiedade dunha colección de tipo lista mapéase con `<list>` e a propiedade ten que ser inicializada con `ArrayList`.

O índice da lista tense que manter nunha columna adicional, por tanto *hai que engadir esta columna extra á táboa*. Esta columna de índice define a posición do elemento na colección e mediante ela, Hibernate é capaz de preservar a orde dos elementos da colección.

Para definir a columna que almacena o índice utilízase o elemento <`list-index`>.

Para definir a columna que almacena o *índice* (neste exemplo Idx) utilízase a etiqueta `<list-index>` e para a columna de *clave foránea* utilízase `<key>`.

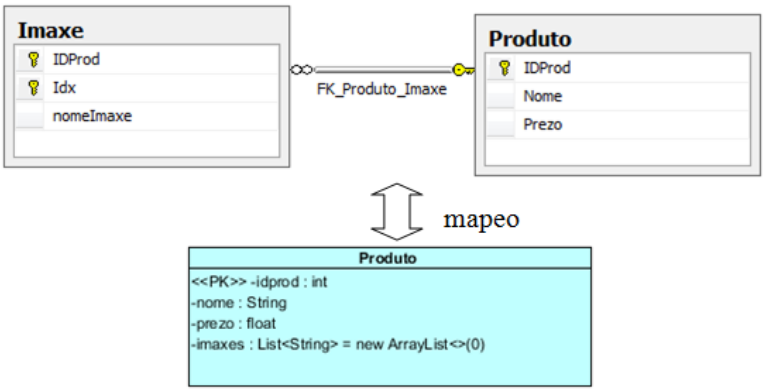
```
<List-index
    column="nome_columna"
    base="0|1|..."
/>
```

- column (requirido): Nome da columna que ten os valores do índice da colección.
- base (opcional - por defecto é 0): Valor da columna índice que corresponde ao primeiro elemento da lista ou do array.

Exemplo:

Amósase unha clase que se mapea con dúas táboas. Na clase defínese unha lista de strings para almacenar os nomes das imaxes que se corresponden cun determinado produto.

Neste caso os elementos da lista son de *Tipo Valor*, xa que o seu ciclo de vida depende completamente dun determinado produto propietario da colección e os elementos da lista non teñen referencias a outros obxectos entidades. Por tanto, o elemento `nomeImaxe` da lista será mapeado coa etiqueta `<element>`.

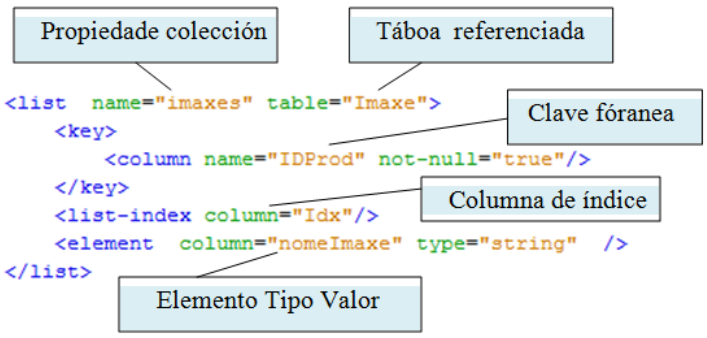


O ficheiro de mapeo `Produto.hbm.xml` quedará:

```
<hibernate-mapping>
    <class catalog="Produtos" name="hbproduto.Produto" schema="dbo" table="Produto">
        <id name="idprod" type="int">
            <column name="IDProd"/>
            <generator class="assigned"/>
        </id>
        <property name="nome" type="string">
            <column length="25" name="Nome"  />
        </property>
        <property name="prezo" type="java.lang.Float">
            <column name="Prezo" precision="53" scale="0"/>
        </property>
        <list  name="imaxes" table="Imaxe">
            <key>
                <column name="IDProd" not-null="true"/>
            </key>
            <list-index column="Idx"/>
            <element  column="nomeImaxe" type="string"  />
        </list>
    </class>
</hibernate-mapping>
```

A continuación detállase a etiqueta `<list>` que corresponde ao mapeamento da propiedade imaxes da clase produto.

```
private List<String> imaxes= new ArrayList<>();
```



O índice da lista comeza en cero. Utilizando <list-index base"1".../ > podemos facer que empece por un.

2.3.6 Mapeo dun Map

Un `<map>` almacena as súas entradas como parellas `clave/valor`, non pode almacenar claves duplicadas, pero si almacenar datos idénticos no seu campo valor, non ten unha orde establecida. Aos elementos `valor` do map accédese especificando a `clave`.

A propiedade dun `<map>` ten que ser inicializada coa interface `HashMap()`. A interface Map proporciona entre outros os seguintes métodos:

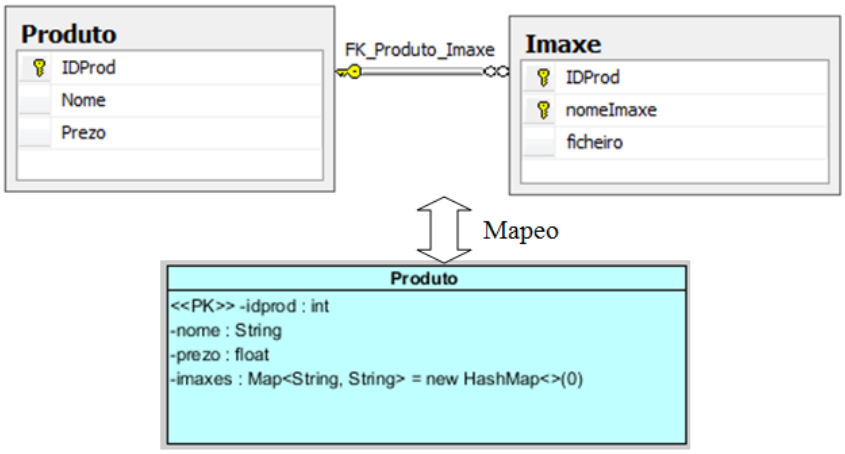
- `Object Map.get(Object clave)`
  - Retorna o obxecto que actúa como valor e que está asociado coa clave que se pasa como parámetro (se esa clave ten un obxecto na colección).
- `Map.put(Object clave, Object valor)`
  - Insire un parella clave/obxecto na colección.

Unha colección Map mapéase coa etiqueta `<map>`. Para definir a columna que almacena a clave utilízase a etiqueta `<map-key>`

```
<Map-key
   column="nome_columna"
   formula="expresión SQL"
   type="Tipo de datos"
/ >
```

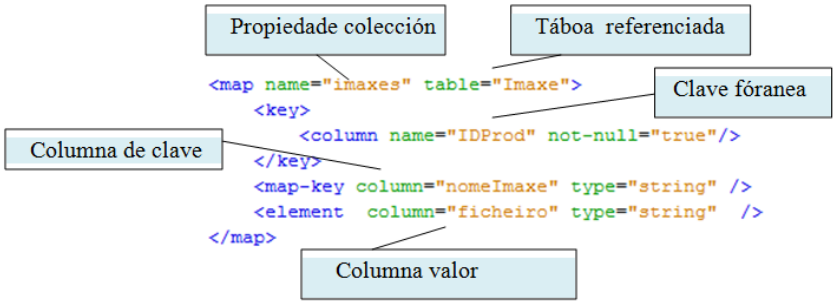
- column (opcional): Nome da columna que ten os valores da clave da colección.
- formula (opcional): Fórmula SQL que se usa para avaliar a clave do map.
- type (requirido): Tipo de datos da clave do map.

Exemplo: supoñamos que agora, as imaxes dun produto teñen un índice non numérico, mediante unha cadea: "Imaxe1", "Imaxe2", etc..., e tamén se garda o nome do arquivo asociado á imaxe. Unha forma de modelar isto é cun `map` que ten o nome da imaxe como a `clave` e o nome do ficheiro como o `valor`.



Mapeamento do map no fichero de mapeo `Produto.hbm.xml` que corresponde ao mapeamento da propiedade imaxes da clase produto.

```
private Map<String,String> imaxes= new HashMap<>(0)
```



2.3.7 Mapeamento dun bag e ibag

Unha colección <bag> é unha colección que permite elementos duplicados e sen orden.

Non está implementada na API de coleccións de Java pero pódense utilizar as interfaces `java.util.List` ou `java.util.Collection` para mapear os `<bag>`. Recoméndase utilizar mellor a interface Collection. As dúas interfaces inicialízanse con `ArrayList`.

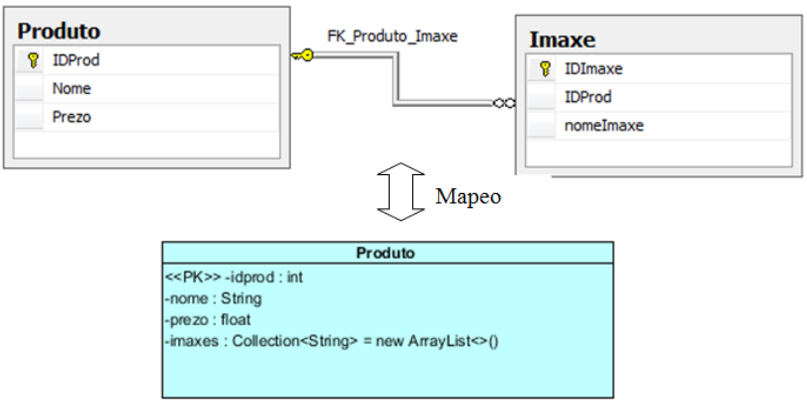
Unha colección <idbag> é similar a un <bag> , pero no mapeo engade unha columna clave primaria "artificial" na táboa da colección, que se xerará automaticamente por Hibernate. O xerador asigna unha clave "artificial" diferente a cada fila da colección. Hibernate pode localizar filas individuais eficientemente e actualizalas ou borralas individualmente, do mesmo xeito que si fose unha lista, mapa ou conxunto.

Para definir a columna que almacena a clave primaria xerada, utilízase a etiqueta `< collection-id>`:

```
< collection-id
   column="nome_columna"
   type="Tipo de datos" >
  <generator class="clase_xeradora"/>
< /collection-id>
```

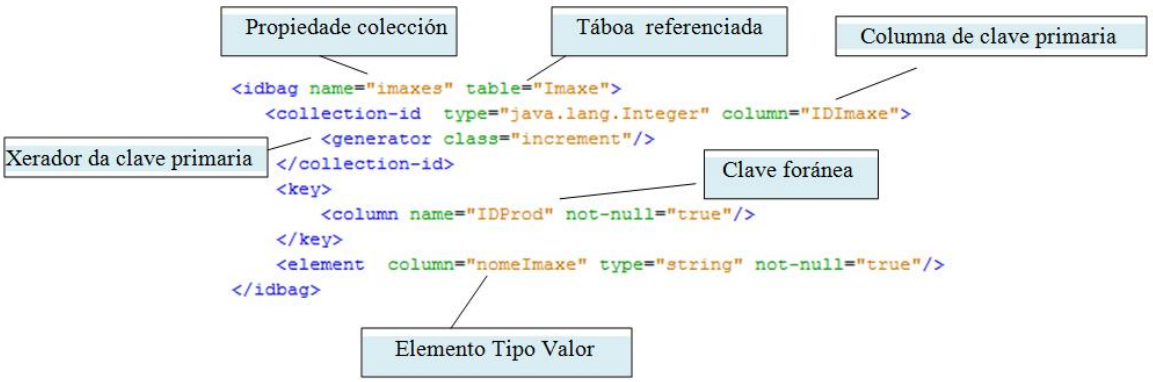
- column : Nome da columna que ten a clave primaria xerada da colección.
- type: Tipo de datos da clave principal.
- generator (opcional): Forma en que a clave primaria se xera. Na implementación actual, a estratexia de xeración de identificador native non se atopa soportada para identificadores de coleccións .

No seguinte exemplo, engadimos na táboa Imaxe unha clave primaria "artificial" ,`IDImaxe`, que será xerada automaticamente por Hibernate e así permítense valores duplicados na colección, é dicir dous produtos poden ter a mesma imaxe.



Mapeamento do <ibag> no fichero de mapeo `Produto.hbm.xml` que corresponde ao mapeamento da propiedade imaxes da clase produto.

```
private Collection <String> imaxes= new ArrayList<>(0);
```



Coleccións ordenadas

Hai dúas formas de ordenar unha colección ao mapealas:

- Colección clasificada(*Sorted collection*): Utilizando as funcións de ordenación proporcionadas polas coleccións de Java. Os datos lense dende a base de datos e realízase a clasificación na memoria da máquina virtual Java (JVM). Este tipo de ordenación non é eficiente para grandes coleccións. Só as coleccións `<set> e <map>` apoian este tipo de ordenación.
- Colección ordenada (*Ordered collection*): Ordenar unha colección a nivel de consulta mediante a cláusula order-by. Os datos cárganse na colección xa ordenados. Este é o xeito máis eficaz se a colección é moi grande.

Colección clasificada(*Sorted collection)*

Utilizan o atributo `sort` para definir a clasificación nas coleccións `< map>` ou `<set>`.

```
< set|map
        name="nome propiedade"
        table="nome da táboa"
        sort="unsorted|natural|nome clase">
        ..........................
</set|map>
```

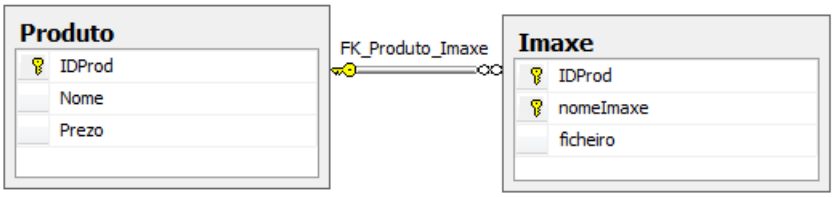
Os valores permitidos do atributo sort son `unsorted, natural` e o `nome dunha clase` que implementar `java.util.Comparator`.

Si se especifica `sort="natural"`, a ordenación realízase usando ou método `compareTo()` definido na interface `java.lang.Comparable` do tipo de datos correspondente. Moitos tipos de datos básicos, como String, Integer e Double implementan esta interface. Se queremos outros algoritmos (por exemplo, orde alfabética inversa), temos que especificar unha clase que implemente `java.util.Comparator`.

Hibernate soporta ás coleccións clasificadas implementando:

- `Para as coleccións <map>: java.util.SortedMap e` inicialízanse `con TreeMap.`
- `Para` as coleccións `<set>: java.util.SortedSet` e inicialízanse con `TreeSet`.

Exemplo: Temos as seguintes táboas que se van mapear coa clase produto utilizando unha colección clasificada `<map>`, utilizando o campo clave `nomeImaxe` para establecer a ordenación de forma alfabética. O mesmo proceso sería válido para unha colección <set>.




```
public class Produto  implements java.io.Serializable {
     private int idprod;
     private String nome;
     private float prezo;
     private SortedMap<String,String> imaxes= new TreeMap();
```

Como a ordenación se vai realizar por orde alfabético polo campo clave `nomeImaxe`, no atributo `sort` especifícase a opción `"natural"`. Neste caso a ordenacion faríase usando o método `compareTo()` definido na interface `java.lang.Comparable` do tipo de datos `String`, e que para estes tipos de datos é a orde alfabética si non se sobrescribe este método.

```
<map name="imaxes" table="Imaxe" sort="natural">
    <key>
        <column name="IDProd" not-null="true"/>
    </key>
    <map-key column="nomeImaxe" type="string" />
    <element  column="ficheiro" type="string"  />
</map>
```

Ao recuperar un produto da base de datos, a súa colección de imaxes estaría ordenada alfabeticamente polo campo `nomeImaxe`.

```
Session sesion = HibernateUtil.getSessionFactory().openSession();
Produto p=(Produto)sesion.get(Produto.class, 1);
for (String i: p.getImaxes().keySet()){
        System.out.println(i);
}
```

Si quixeramos outra ordenación diferente, teriamos que especificar unha clase que implemente `java.util.Comparator` e sobrescriba o método `CompareTo`. Por exemplo, para obter os nomes das imáxenes na orde alfabética inversa, implementamos a seguinte clase:

```
public class ComparadorInverso implements Comparator<String> {
      @Override
  public  int compare(String o1, String o2) {
            return (o2.compareTo(o1));
    }
}
```

No atributo sort da colección teriamos que indicar que utilice agora este comparador.

```
<map name="imaxes" table="Imaxe" sort="hbproduto.ComparadorInverso">
    <key>
        <column name="IDProd" not-null="true"/>
    </key>
    <map-key column="nomeImaxe" type="string" />
    <element  column="ficheiro" type="string"  />
</map>
```

2.3.8 Colección ordenada (*Ordered collection*)

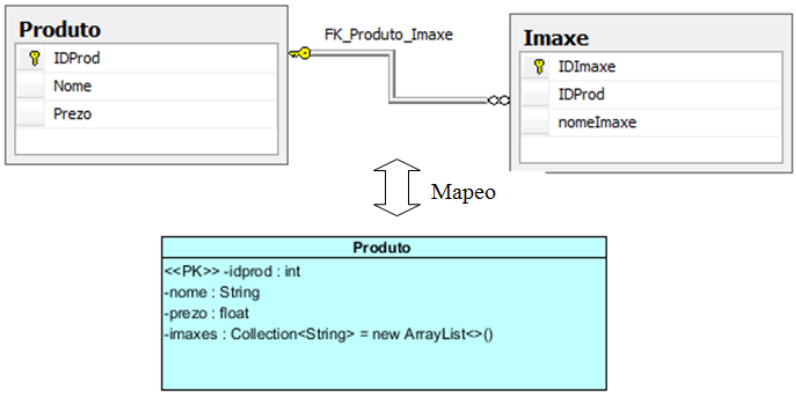
Si queremos que a mesma base de datos ordene os elementos da colección, utilízase o atributo `order-by` nos mapeos `set, bag, idbag` ou `map`. A ordenación realízase mediante a linguaxe SQL. Esta solución está dispoñible somentes baixo o JDK 1.4 ou superior.

```
<set|map|bag|idbag>
    name="nome propiedade"
    table="nome dá táboa"
    order-by="columna1 [,columna2..] [asc|desc]">
    ..........................
</set|map|bag|idbag >
```

O atributo `order-by` debe ser unha columna da base de datos (ou columnas), non unha propiedade e podemos especificar a orde (`asc` ou `desc`). Podemos ordenar por calquera columna da táboa de colección.

Nos ficheiros das clases, as coleccións impleméntanse da forma habitual segundo o tipo de colección (`Set / HashSet`, `Map/HashMap` ou `Collection|List/ArrayList`), pero interna-amente, Hibernate utiliza variacións destas coleccións (`LinkedHashSe`t ou `LinkedHash-Map`) que conservan a orde de inserción dos elementos clave.

Exemplo:



No mapeamento do `<ibag>` no fichero de mapeo `Produto.hbm.xml` que corresponde ao mapeamento da propiedade imaxes da clase produto, engadimos o atributo `order-by` para ordenar polo campo `nomeImaxe` en orde descendente cando se recupere a colección.

– No ficheiro produto.java.

```
private Collection <String> imaxes= new ArrayList<>(0);
```

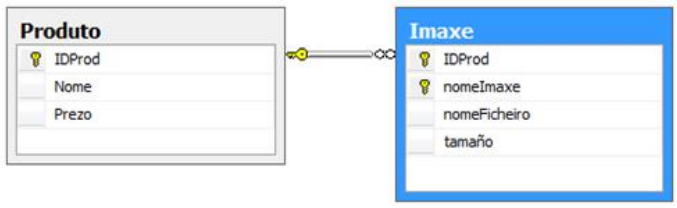
– No ficheiro produto.hbm.xml:

```
<idbag name="imaxes" table="Imaxe" order-by="nomeImaxe desc">
   <collection-id  type="java.lang.Integer" column="IDImaxe" >
        <generator class="increment"/>
    </collection-id>
    <key>
        <column name="IDProd" not-null="true"/>
    </key>
    <element  column="nomeImaxe" type="string" not-null="true"/>
</idbag>
```

2.3.9 Colección de compoñentes

Ata agora mapeamos coleccións dun único dato. Podemos ter varios datos como propiedades nunha `clase compoñente` e ter unha colección delas. A clase compoñente impleméntase como un POJO sen identificador.

Imaxinemos por exemplo que temos o seguinte modelo de datos.




```
public class Produto  implements java.io.Serializable
    private int idprod;
    private String nome;
    private float prezo;
    private Set<Imaxe> imaxes= new HashSet<>(0);
```

```
public class Imaxe {
    private String nomeImaxe ;
    private String nomeFicheiro;
    private float tamaño;
```

Hibernate proporciona `<composite-element>` para o mapeo dunha colección de compoñentes. Nas coleccións `<set>`, `<list>`, `<map>`, `<bag>` e `<idbag>` pódese empregar esta etiqueta.

```
<set|list|map|bag|idbag>
  name="nome da propiedade " table="nome da táboa" ">
  <key column="columna clave foránea"/>
  <composite-element class="nome clase compoñente">
    <property name="nome da propiedade"/>
    ..........................................................
  </composite-element>
</set|list|map|bag|idbag
```

Si definimos unha colección de compoñentes, e sobre todo nos `Set`, é moi importante implementar os métodos `equals()` e `hashCode()`, de xeito correcto na clase compoñente. Hibernate necesita estes métodos para comparar e chequear instancias en caso de modificacións.

```
public int hashCode() {
   int hash = 7;
   hash = 31 * hash + Objects.hashCode(this.nomeImaxe);
   hash = 31 * hash + Objects.hashCode(this.nomeFicheiro);
   hash = 31 * hash + Float.floatToIntBits(this.tamaño);
   return hash;
         }
          @Override
  public boolean equals(Object obj) {
    if (obj == null) {
            return false;
     }
    if (getClass() != obj.getClass()) {
            return false;
     }
    final Imaxe other = (Imaxe) obj;
    if (!Objects.equals(this.nomeImaxe, other.nomeImaxe)) {
            return false;
```

```
    }
    if (!Objects.equals(this.nomeFicheiro, other.nomeFicheiro)) {
            return false;
    }
    if (Float.floatToIntBits(this.tamaño) != Float.floatToIntBits(other.tamaño))
     {
            return false;
    }
       return true;
    }
```

O mapeamento do ficheiro `Produto.hbm.xml` é similar ao visto antes pero usamos `<composite-element>` en lugar de `<element>`.

```
<set name="imaxes" table="Imaxe">
    <key>
        <column name="IDProd" not-null="true"/>
    </key>
    <composite-element class="hbproduto.Imaxe">
        <property name="nomeImaxe" column="nomeImaxe" type="string" not-null="true"/>
        <property name="nomeFicheiro" column="nomeFicheiro" type="string" />
        <property name="tamaño" column="tamaño" type="java.lang.Float" />
    </composite-element>
</set>
```

2.3.10 Navegación bidireccional nas coleccións de compoñentes

O elemento `<composite-element>` permite un subelemento `<parent>` que mapea unha propiedade da clase do compoñente como unha referencia á entidade propietaria.

```
<parent name="nome da propiedade propietaria"/>
```

No exemplo anterior, a navegación de `Produto` a `Imaxe` é unidireccional, é dicir podemos obter as imaxes dun produto mediante `imaxes.getImaxes.iterator`. Si quixeramos obter a partir dunha imaxe o seu produto co método `prod.getProduto`, teriamos que engadir no XML o subelemento `<parent>`.

```
public class Produto  implements java.io.Serializable {
    private int idprod;
    private String nome;
    private float prezo;
    private Set<Imaxe> imaxes= new HashSet<>(0);
```

```
public class Imaxe {
    private String nomeImaxe ;
    private String nomeFicheiro;
    private float tamaño;
    private Produto produto;
```

```
<set name="imaxes" table="Imaxe">
    <key>
        <column name="IDProd" not-null="true"/>
    </key>
    <composite-element class="hbproduto.Imaxe">
        <parent name="produto"/>
        <property name="nomeImaxe" column="nomeImaxe" type="string" not-null="true"/>
        <property name="nomeFicheiro" column="nomeFicheiro" type="string" />
        <property name="tamaño" column="tamaño" type="java.lang.Float" />
    </composite-element>
</set>
```

Na tarefa 7 faremos exercicios de como mapear os elementos dunha colección utilizando diferentes interfaces de coleccións soportadas por Hibernate.

2.4 Mapeo de clase compoñente: Composición.

Hibernate denomina "compoñente" á clase que traslada o seu estado persistente á táboa da súa clase "propietaria". A clase "compoñente" persite como un tipo valor, non como unha referencia de entidade.

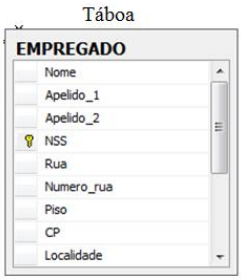
En programación orientada a obxectos, o termo "compoñente" fai referencia ao concepto de composición.

A clase compoñente non necesita un atributo identificador e tampouco o seu propio ficheiro `hbm.xml`. Mapéase no mesmo ficheiro de mapeo da súa clase propietaria utilizando a etiqueta `<component>`:

```
public class Empregado  implements java.io.Serializable {
     private String nss;
     private Empregado empregado;
     private String nome;
     private String apelido1;
     private String apelido2;
     private Enderezo enderezo;

 public class Enderezo implements Serializable{
      private String rua;
      private int numeroRua;
      private String piso;
      private String cp;
      private String localidade;
```

Táboa



Clase propietaria

Táboa

Clase compoñente

```
<hibernate-mapping>
    <class name="Mapeo.Empregado" table="EMPREGADO" schema="dbo" catalog="EmpresaHB">
        <id name="nss" type="string">
            <column name="NSS" length="15" />
            <generator class="assigned" />
        </id>

        //.....................................
        <component name="enderezo" class="Mapeo.Enderezo">
            <property name="rua" type="string">
                <column name="Rua" length="30" not-null="true" />
            </property>
            <property name="numeroRua" type="int">
                <column name="Numero_rua" not-null="true" />
            </property>
            <property name="piso" type="string">
                <column name="Piso" length="4" />
            </property>
            <property name="cp" type="string">
                <column name="CP" length="5" />
            </property>
            <property name="localidade" type="string">
                <column name="Localidade" length="25" not-null="true" />
            </property>
        </component>
```

Tamén se permite a navegación bidireccional, igual que no mapeo das coleccións de compoñentes vistos anteriormente e segue a mesma lóxica. O elemento `<component>`, igual que o elemento `<composite-element>` para o mapeo das coleccións de compoñentes, permite un subelemento `<parent>` que mapea unha propiedade da clase do compoñente como unha referencia á entidade propietaria.

```
<component name:……  >
   <parent name="nome da propiedade propietaria"/>
   <property name:…./>
</component>
```

Na tarefa 8 faremos exercicios de como mapear as clases compoñente que non teñen unha táboa na base de datos asociadas.

2.5 Relacións entre entidades: Asociacións

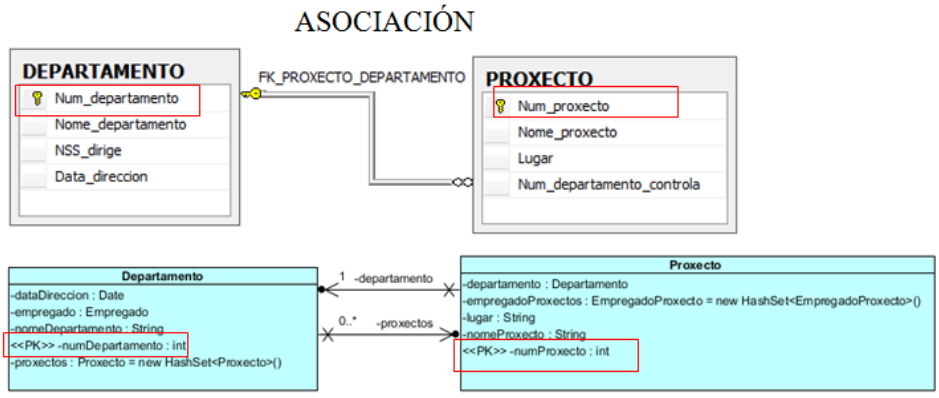
As clases, ao igual que os obxectos, non existen de modo illado. Por esta razón existirán relacións entre clases e entre obxectos. Unha relación representa o vínculo entre dúas clases e represéntase cunha referencia aos obxectos.

Ante un deseño orientado a obxectos, é importante coñecer as diferentes relacións que se poden establecer entre as clases.

Imos considerar só as relacións entre "`clases entidades`", chamada asociacións.

O mapeo de asociacións entre clases é unha das partes fundamentais de calquera ferramenta ORM.

- *Todas as clases participantes deben ter unha clave primaria na base de datos e polo tanto, unha táboa propia*.



- *Cada instancia das clases entidades ten un completo ciclo de vida independente*. Os novos obxectos son transitorios e teñen que facerse persistentes cada un, se queremos almacenalos na base de datos. A súa relación non inflúe no seu ciclo de vida, son independientes un do outro. Se a clase fose un compoñente de tipo valor, o estado dunha instancia súa sería o mesmo que o estado da súa entidade propietaria. Neste caso, non obstante, cada clase entidade é unha entidade separada co seu propio estado completamente independente.

2.5.1 Características das relacións

A relación entre clases ten as seguintes características:

- **Cardinalidade**: indica *cantas instancias poden existir* en cada lado da relación. Por tanto, limita o número de obxectos relacionados.

  Hibernate define catro tipos de relacións segundo a súa cardinalidade:

  – Un a Un
  – Un a Moitos / Moitos a Un
  – Moitos a Moitos

  Hibernate non ten en conta a cardinalidade mínima, é irrelevante.

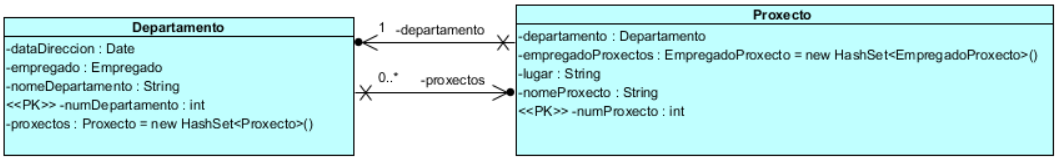
- **Direccionalidad**e (multiplicidade): define a *forma en que se navega entre as entidades* que manteñen unha relación:

  – **Unidireccional**: se só unha entidade referencia á outra, é dicir sabe con que obxectos está relacionado, pero os devanditos obxectos non coñecen o obxecto orixinal.

– **Bidireccional**: se cada entidade referencia á outra, é dicir cando os obxectos en ambos os dous extremos da relación saben do obxecto no extremo contrario. Hibernate soporta 2 tipos de asociacións bidireccionais: `one-to-many` e `many-to-many`.

No seguinte exemplo:

– A direccionalidade é bidireccional, xa que cada entidade referencia á outra.
– A relación dende Proxecto a Departamento é moitos a un.
– A relación inversa dende Departamento a Proxecto é un a moitos.



2.5.2 Estratexias de carga dos obxectos.

Hibernate permite utilizar diferentes estratexias de carga para recuperar ou cargar un obxecto na memoria, coas súas coleccións e asociacións.

Por exemplo: si a clase `Persoa` mantén unha colección de `Actividades` que realiza. Cando carguemos unha `Persoa` en memoria, Hibernate podería cargar xa os obxectos correspondentes ás súas `Actividades`: *estratexia "inmediata" (ou temperá).* Unha única consulta (*cun join*) permitiría a Hibernate traer a información da persoa e das súas actividades e crear os obxectos.

Pero tamén podemos cargar só a `Persoa` en memoria, e cargar máis adiante as `Actividades`: *estratexia "preguiceira* ". Nese caso, Hibernate necesitaría polo menos dúas consultas (unha para a persoa e outra para as súas actividades).

- A *estratexia temperá (EAGER),* indica que no momento de obter a entidade mestra se deben obter as entidades fillas que estean asociadas, mentres que a estratexia *preguiceira (LAZY)* só obtén a entidade mestra e os datos das entidades fillas obtéñense ao forzar a súa consulta.
- Si usamos a estratexia *lazy*, podemos ter problemas ao acceso da colección fóra da sesión. Hibernate xerará unha excepción.
- Na estratexia *eager*, si accedemos a colección con moitos elementos, podemos ter problemas de memoria.

Hibernate soporta diferentes estratexias para decidir `COMO` cargar os datos (unha consulta, varias....) e `CANDO` facelo (todo, á vez, por partes...).

Estratexia de como cargar os obxectos. Fetch mode

Catro tipos de estratexias definen COMO se cargan os datos:

- **Recuperación por unión** (*join fetching*): recupérase a instancia asociada a colección cun só SELECT, usando join.
– Exemplo: ao recuperar un departamento, coa mesma consulta Hibernate tráese ao seu director.
- **Recuperación por selección** (*select fetching*): úsase un segundo SELECT para recuperar a entidade ou colección asociada.
– Exemplo: recupérase un departamento cun primeiro SELECT, máis tarde recupérase o seu director cunha nova SELECT.

- **Recuperación por subselección** (*subselect fetching*): úsase un segundo SELECT para recuperar a entidade ou colección asociada dun conxunto de entidades recuperadas previamente.
  - Exemplo: recupérase unha lista de departamentos cun primeiro SELECT, máis tarde recupéranse os seus directores cunha nova SELECT (única).
- **Recuperación por lotes**: as entidades ou coleccións asociadas vanse recuperando en bloques (a partir dunha lista de identificadores).
  - Exemplo: un primeiro SELECT recupera a lista de departamentos, un segundo SELECT recupera os directores dos 15 primeiros departamentos, un terceiro SELECT recupera os 15 seguintes, etc..

Estratexia de cando cargar os obxectos. Fetch type

- **Recuperación inmediata**: a entidade ou colección asociada cárgase inmediatamente cando se carga o obxecto "propietario".
  - Exemplo: ao cargar o departamento, xa temos os seus directores cargados en memoria.
- **Recuperación preguiceira de coleccións**: a colección asociada non se carga ata que se invoca unha operación sobre a colección.
  - Exemplo: cargamos o departamento, pero os seus membros non se traen a memoria ata que intentamos acceder ao primeiro, entón tráense todos.
- **Recuperación aínda máis preguiceira de coleccións**: cada membro da colección asociada cárgase por separado, cando se necesite.
  - Exemplo: cargamos o departamento; os seus membros tráense a memoria un por un, a medida que os imos usando.
- **Recuperación por proxy**: a entidade asociada non se trae a memoria ata que non accedemos a unha propiedade súa, distinta do identificador.
  - Exemplo: cargamos o departamento, o director non se carga ata que non pedimos o seu nome.
- **Recuperación "non-proxy"**: a entidade asociada non se trae a memoria ata que non se usa a variable da instancia.
  - Exemplo: cargamos o departamento, o director cárgase cando o asignamos como director doutro departamento.

Podemos cambiar esas estratexias no mapeo para unha propiedade ou colección concreta.

Estratexias por defecto

Por defecto, Hibernate usa:

- **Para referencias a entidades** (asociacións monovaluadas): recuperación por `proxy de forma preguiceira`.
  - Exemplo: cando recuperemos un departamento, o seu director será un proxy. Cando pidamos o nome do director, o proxy lanzará unha SELECT para cargar o director en memoria.
- **Para referencias a coleccións** (e asociacións multivaluadas): recuperación `preguiceira por selección`.
  - Exemplo: cando recuperemos un departamento, a súa colección de empregados non se carga. Cando usemos esa lista, lánzase unha SELECT para traela completa á memoria.

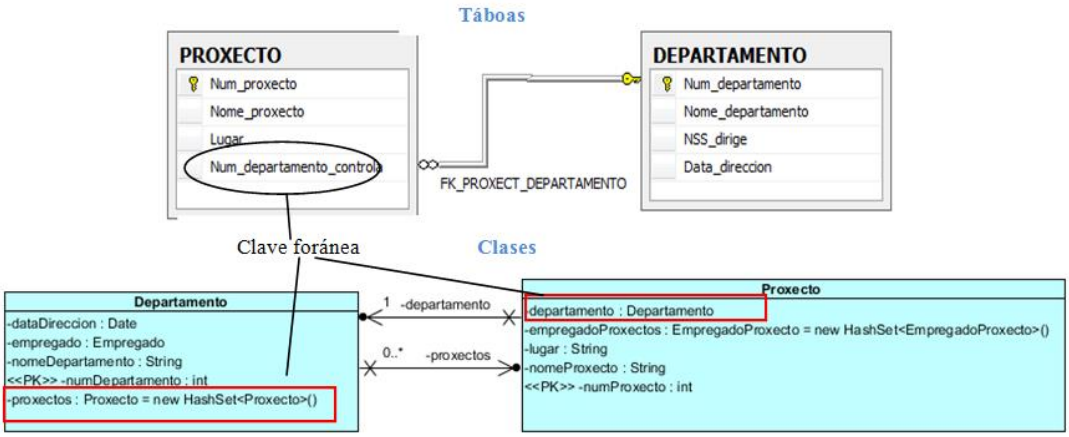
Perigo de usar estratexia lazy: se accedemos a colección fóra da sesión, Hibernate xerará unha excepción.

Perigo de usar estratexia non lazy: se accedemos a colección con moitos elementos, podemos ter problemas de memoria.

2.5.3 Representación das asociacións

A asociación represéntase:

- Nas clases Java, mediante *referencias a obxectos* e poden ser *coleccións*.
- Na base de datos, mediante *claves foráneas*.
- Si a navegación é bidireccional, dúas propiedades de dúas clases diferentes están representadas pola mesma clave foránea na base de datos.



2.5.4 Asociación One-to-One

A relación un a un en Hibernate consiste simplemente en que un obxecto teña unha referencia a outro obxecto de forma que ao persistirse o primeiro obxecto tamén se persista o segundo.

Estas relacións poden ser expresadas na base de datos utilizando a *mesma clave primaria* (e ou mesmo valor) para as dúas táboas ou ben mediante *unha clave allea* dunha dás táboas á outra.

Unha asociación de `un a un` a outra clase persistente declarase usando o elemento `one-to-one`.

```
<one-to-one
   name="nome da propiedade"
   class="nome da clase"
   cascade="estilo de cascada"
   constrained="true|false"
   fetch="join|select"
   property-ref="nome da propiedade referenciada"
   access="field|property|ClassName"
   formula="expresión SQL"
   lazy="proxy|no-proxy|false"
   entity-name="Nome da entidades"
   foreign-key="nome clave foránea"
/>
```

- name: nome da propiedade.
- class (opcional): nome da clase asociada.
- cascade (opcional): especifica que operacións deben ir en cascada dende o obxecto pai ata o obxecto asociado.
- constrained (opcional): especifica que unha restrición de clave foránea na clave principal da táboa mapeada referencia á táboa da clase asociada.

- fetch (opcional - por defecto é select): escolle entre a recuperación de unión exterior (outer-join) ou a recuperación por selección secuencial.
- property-ref (opcional): nome dunha propiedade da clase asociada que estea unida á clave principal desta clase. Si non se especifica utilízase a clave principal da clase asociada.
- access (opcional - por defecto é property): estratexia que Hibernate utiliza para acceder ao valor da propiedade.
- formula (opcional): case todas as asociacións un-a-un mapean á clave principal da entidade propietaria. Si este non é o caso, pode especificar outra/s columna/s, ou unha expresión para unir utilizando unha fórmula SQL.
- lazy (opcional - por defecto é `proxy`): por defecto as asociacións recupéranse preguiceiramente cando se chama o método `get`. Ao especificar `lazy="non-proxy"`, indicamos que esta propiedade debe ser recuperada cando se acceda por primeira vez á variable de instancia. Con `lazy="false"` especifícase que a asociación sempre será recuperada de forma temperá. Se `constrained="false"`, a aplicación de proxies é imposible e Hibernate recuperará de forma temperá a asociación.
- entity-name (opcional): nome da entidade da clase asociada.
- foreign-key (opcional): nome da clave foránea.

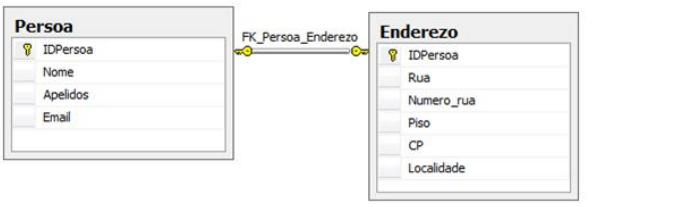
Clave primaria compartida entre ambas

As asociacións de claves principais non necesitan unha columna extra na táboa. Se dúas filas están relacionadas pola asociación entón as dúas filas de táboas comparten o mesmo valor de clave principal.

Nunha *asociación unidireccional un a un*, só unha entidade referencia a outra.

```
public class Persoa implements java.io.Serializable {
    private int idpersoa;
    private String nome;
    private String apelidos;
    private String email;
    private Enderezo enderezo;
```

```
public class Enderezo {
    private int idpersoa;
    private String rua;
    private int numero;
    private String piso;
    private String cp;
    private String localidade;
```



A clase Persoa ten unha propiedade chamada enderezo da clase Enderezo, mentres que a clase Enderezo non posúe ningunha referencia a Persoa, xa que definimos unha direccionalidade desde Persoa cara a Enderezo pero non ao revés.

Para mapear a propiedade `enderezo` no ficheiro `Persoa.hbm.xml` engadimos:

```
<one-to-one  class="hbpersoaonetoone.Enderezo" name="enderezo"  cascade="all"/>
```

O atributo `cascade` indica a Hibernate como debe actuar cando realicemos as operacións de persistencia de gardar, borrar, ler, etc. No exemplo o valor é `all` indicando que deberemos realizar a mesma operación en `Persoa` que en `Enderezo`.

No fichero `Dirección.hbm.xml` non haberá ningunha información relativa á relación `un a un` posto que no exemplo a relación `un a un` ten unha direccionalidade desde `Persoa` ata `Enderezo`, por tanto Enderezo non sabe nada sobre Persoa.

O seguinte exemplo mostra o código para inserir unha nova Persoa. Como podemos ver, só persistimos a clase Persoa e debido á relación `one-to-one` e a operación en `cascade="all"`, creáronse as filas tanto na táboa Persoa coma na táboa Enderezo.

```
Persoa pers=new Persoa (7,"Ana","Sanchez","ana@gmail.com");
Enderezo enderezo=new Enderezo (7,"Paz",3,"4A","36008","Poio");
pers.setEnderezo(enderezo);
sesion.save(pers);
tx.commit();
```

Nesta asociación unidireccionalidad un a un entre Persoa e Enderezo, non podemos asegurar que o obxecto Enderezo teña a mesma clave primaria que Persoa.

Podemos facer que a clave de Enderezo sexa a mesma que a clave de Pesoa, facendo a asociación *bidireccional* e *xerando automaticamente o valor da clave* de Enderezo a partir do valor introducido na clave de Persoa.

Agora na clase Enderezo temos que inserir un campo que faga referencia á clase Persoa, para poder utilizar a bidireccionalidade.

```
public class Persoa implements java.io.Serializable {
    private int idpersoa;
    private String nome;
    private String apelidos;
    private String email;
    private Enderezo enderezo;
```

```
public class Enderezo {
    private int idpersoa;
    private String rua;
    private int numero;
    private String piso;
    private String cp;
    private String localidade;
    private Persoa persoa;
```

O arquivo `Persoa.hbm.xml` queda como o exemplo anterior. Agora temos que modificar o arquivo `Enderezo.hbm.xml` e engadir:

```
<hibernate-mapping>
    <class name="hbpersoaonetoone.Enderezo" table="Enderezo" schema="dbo" catalog="Persoa4">
        <id name="idpersoa" type="int">
            <column name="IDPersoa" />
            <generator class="foreign">
                <param name="property"> persoa</param>
            </generator>
        </id>
        <one-to-one  class="hbpersoaonetoone.Persoa" name="persoa" constrained="true"/>
```

Coa etiqueta `<Generator class="foreign " >  <Param name="property">persoa`

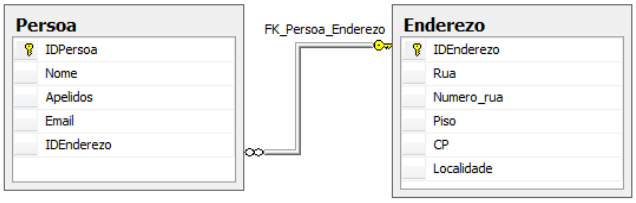
`</param ></Generator>` , asígnaselle o mesmo valor de clave primaria que á instancia de Persoa.

No seguinte código créase unha instancia de `Persoa` e outra de `Enderezo` e posteriormente enlázanse de xeito *bidireccional* utilizando os métodos `setXXX` de ambas clases. Na táboa da base de datos, insírese unha fila na táboa Persoa e outra fila na táboa dirección e nesta tamén se insire a clave primaria da táboa Persoa.

```
Persoa pers=new Persoa (7,"Ana","Sánchez","ana@gmail.com");
Enderezo enderezo=new Enderezo ("Paz",3,"4A","36008","Poio");
pers.setEnderezo(enderezo);
enderezo.setPersoa(pers);  //por ser bidireccional
sesion.save(pers);
tx.commit();
```

Asociacións un a un de clave foránea única

Agora cada entidade ten a súa propia clave e necesítase unha columna extra nunha das táboas para establecer a clave foránea. As relacións vanse facer a través desta clave foránea.



Para mapear a propiedade `enderezo` no ficheiro `Persoa.hbm.xml` engadimos:

```
<many-to-one  class="hbpersoaonetoone.Enderezo" name="enderezo" column="IDEnderezo"
              cascade="all" unique="true"/>
```

Para establecer a relación das entidades a través da clave foránea temos que utilizar `<many-to-one>` en lugar de `<one-to-one>`,por que temos unha entidade `Persoa` que ten unha clave foránea `IdEnderezo` apuntando a outra entidade `Enderezo`. Co atributo `unique="true"` impedimos que dúas instancias de Persoa compartan a mesma instancia Enderezo.

Finalmente, si queremos que a asociación sexa bidireccional, no ficheiro `Enderezo.hbm.xml` poñeriamos:

```
<one-to-one name="persoa" class="hbpersoaonetoone.Persoa" property-ref="enderezo"/>
```

2.5.5 Asociación One-to-Many ou Many-to-One

As relacións de *un a moitos* ou *moitos a un* destacan pola súa frecuencia de aparición. Nas relacións *un a moitos*, un obxecto da entidade "A" (lado un) está relacionado con moitos obxectos da entidade "B" (lado moitos) e *moitos a un*, moitos obxectos da entidade "B" (lado moitos) están relacionados cun obxecto da entidade "A" (lado un).

Impleméntase en Java, cunha *propiedade* (referencia a un obxecto) na clase do *"lado moitos*" e /ou unha *colección* na do "*lado un*". Si a relación é bidireccional, a implementación é dos dous lados, en cambio si é unidireccional só se implementa nun dos lados.

Temos tres posibilidades de implementación, mediante:

- **Unha asociación unidireccional moitos a un**. *A implementación da relación só se realiza na clase do"lado moitos" mediante unha propiedade que referencia ao "lado un"*. Na outra clase do "lado un" non se implementa ningunha relación.

```
<class name="Clase-lado-moitos">
 <id name="identificador " column="clave primaria">
     <generator class="estratexia de xeración da clave"/>
 </id>
  <propertry name=...../>
 <many-to-one name="clase-lado-un"  column="clave foránea"
       not-null="true"/>  // obrigatorio si a clave foránea no admite nulos.
 </class>
<class name="clase-lado-un ">
    <id name="identificador" " column="clave primaria">
          <generator class="estratexia de xeración da clave"/>
      </id>
    </id>
```

```
  <propertry name=...../>
</class
```

- **Unha asociación unidireccional un-a-moitos** nunha clave foránea: é un caso moi inusual e non se recomenda. *A implementación da relación só se realiza na clase do "lado un" mediante unha colección que referencia ao "lado moitos".* Na outra clase do "lado moitos" non se implementa ningunha relación.

```
<class name="Clase-lado-moitos">
 <id name="identificador " column="clave primaria">
     <generator class="estratexia de xeración da clave"/>
 </id>
  <propertry name...../>
</class>
<class name="clase-lado-un ">
  <id name="identificador" " column="clave primaria">
      <generator class="estratexia de xeración da clave"/>
  </id>
  <propertry name...../>
  <set|bag|map|list
    name="nome da propiedade de colección" table="táboa referenciada".....>
    <key>
      <column name="clave foránea"  />
    </key>
    <one-to-many class="clase-lado-moitos" />
  </set|bag|map|list>
</class
```

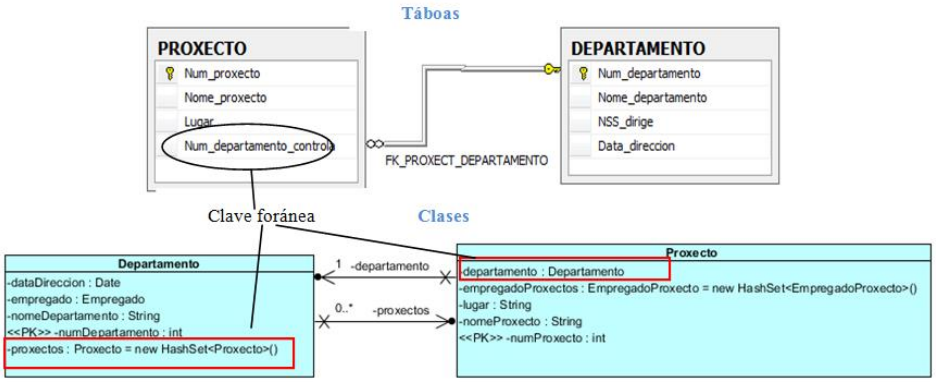
- **Unha asociación bidireccional:** *A implementación da relación realízase nos dous lados, na clase do "lado moitos" mediante unha propiedade que referencia ao "lado un" e na clase do" lado un" mediante unha colección que referencia ao "lado moitos".*

```
<class name="clase-lado-moitos">
 <id name="identificador " column="clave primaria">
     <generator class="estratexia de xeración da clave"/>
 </id>
  <propertry name...../>
 <many-to-one name="clase-lado-un"  column="clave foránea"
       not-null="true"/>   // obrigatorio si a clave foránea no admite nulos.
 </class>
<class name="clase-lado-un ">
  <id name="identificador" " column="clave primaria">
      <generator class="estratexia de xeración da clave"/>
  </id>
  <propertry name...../>
  <set|bag
    name="nome da propiedade de colección" table="táboa referenciada"
    inverse="true"  .....>
    <key>
      <column name="clave foránea"  />
    </key>
    <one-to-many class="clase-lado-moitos" />
  </set|bag>
</class
```

A etiqueta `inverse="true"`, hai que *engadila sempre que implementemos unha asociación bidireccional e sempre no "lado un"*, é dicir na colección. Na asociación unidireccional non se engade.

A razón pola que sempre temos que engadir inverse="true" nas relacións bidireccionais, é porque agora temos dúas asociacións unidireccionais que mapean sobre a mesma columna (a clave foránea), entón: ¿Qué lado controla esta columna?. En tempo de execución hai dúas representacións en memoria do mesmo valor, a propiedade do lado moitos e un elemento da colección do lado un. Nun momento dado, pódese modificar ese valor e Hibernate detectaría dous cambios en memoria nas instancias persistentes. Hibernate non detecta o feito de que os dous cambios se refiren a mesma columna, xa que temos mapeada a clave foránea dúas veces e Hibernate necesita coñecer este feito

por medio do atributo inverse. Neste caso, Hibernate ignora o cambio realizado na colección cando vai actualizar na base de datos.



No exemplo anterior amósase unha relación *bidireccional* e a cardinalidade é `One to Many` entre Departamento e Proxecto e `Many to One` entre Proxecto e Departamento. Utilizouse a interface Set para as coleccións pero outras opcións serían igualmente válidas.

- Mapeo da propiedade do "lado moitos".

```
<many-to-one
   name="Nome propiedade"
   column="Nome columna"
   class="Nome da clase"
   cascade=" all|none|save-update|delete|all-delete-orphan|delete-orphan"
  fetch="join|select"
   update="true|false"
   insert="true|false"
   property-ref="propertyNameFromAssociatedClass"
   access="field|property|ClassName"
   unique="true|false"
   not-null="true|false"
   optimistic-lock="true|false"
   lazy="proxy|no-proxy|false"
   not-found="ignore|exception"
   entity-name="Nome da entidade"
   formula="Expresión SQL"
   index="index_name"
   unique_key="unique_key_id"
   foreign-key="foreign_key_name "
/>
```

- name: nome da propiedade.
- column (opcional): nome da columna da clave foránea.
- class (opcional): nome da clase asociada.
- cascade (opcional – por defecto none): especifica que operacións deben ir en cascada desde o obxecto pai ata o obxecto asociado.
- fetch (opcional - por defecto é select): escolle entre facer varias consultas Select para recuperación dos datos ou facer só unha utilizando unha consulta de unión (outer-join).
- update, insert (opcional - por defecto é true): especifica que as columnas mapeadas deben ser incluídas nas declaracións SQL UPDATE e/ou INSERT. Ao establecer ambas como false permite unha asociación "derivada" onde o seu valor é inicializado desde algunha outra propiedade que mapea á mesma columna (ou columnas), por un disparador ou por outra aplicación.
- property-ref: (opcional): nome dunha propiedade da clase asociada que se encontra unida á súa clave foránea. Si non se especifica, utilízase a clave principal da clase asociada.

- access (opcional - por defecto é `property`): estratexia que Hibernate utiliza para acceder ao valor da propiedade.
- unique (opcional): activa a xeración DDL dunha restrición de unicidade para a columna de clave foránea. Amais permite que este sexa o obxectivo dunha property-ref. Pode facer que a asociación sexa de multiplicidade un-a-un.
- not-null (opcional): activa a xeración DDL dunha restrición de nulabilidade para as columnas de clave foránea.
- optimistic-lock (opcional - por defecto é true): especifica que as actualizacións a esta propiedade requiren ou non da obtención dun bloqueo optimista.
- lazy (opcional - por defecto é `proxy`): por defecto as asociacións recupéranse preguiceira mente cando se chama o método `get`. `lazy="non-proxy"` especifica que esta propiedade debe ser recuperada cando se acceda por primeira vez á variable de instancia. `lazy="false"` especifica que a asociación sempre será recuperada de xeito temperán.
- not-found (opcional - por defecto é `exception`): especifica como se manexarán as claves foráneas que referencian as filas que faltan. `ignore` tratará unha fila perdida como unha asociación nula.
- entity-name (opcional): nome de entidade da clase asociada.
- formula (opcional): unha expresión SQL que define o valor para unha clave foránea computada.

Exemplo: para mapear o exemplo anterior, no ficheiro Proxecto.hbm.xml engadimos:


```
<hibernate-mapping>
    <class name="Mapeo.Proxecto" table="PROXECTO" schema="dbo" catalog="EmpresaHB">
        <id name="numProxecto" type="int">
            <column name="Num_proxecto" />
            <generator class="assigned" />
        </id>

        <many-to-one name="departamento" class="Mapeo.Departamento" fetch="select">
            <column name="Num_departamento_controla" not-null="true" />
        </many-to-one>
```

- Usamos `<many-to-one>` en lugar de `<property>` para que Hibernate saiba que a propiedade non contén un valor, senón *unha referencia a unha entidade* (instancia doutra clase).
- A columna `Num_departamento_controla` é a columna da táboa `Departamento` que actúa como *clave foránea.*

- Mapeo da propiedade do "lado un".

```
<one-to-many
   class="Nome clase asociada"
   not-found="ignore|exception"
   entity-name="nome da entidade"
/>
```

- class (obrigatorio): nome da clase asociada.
- entity-name (opcional): nome de entidade da clase asociada.

- not-found (opcional - por defecto é `exception`): especifica como se manexarán as claves foráneas que referencian as filas que faltan. `ignore` tratará unha fila perdida como unha asociación nula.

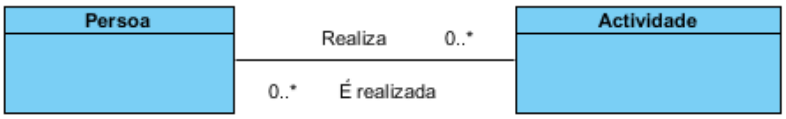
Exemplo:

- No "`lado un`" temos unha colección e mapeámola utilizando a etiqueta `<set>` porque se trata dunha colección que utiliza a interface Set.
- Coa etiqueta `<one-to-many>` indicamos que a colección non contén valores, senón referencias a entidades doutra clase (Proxecto).
- Coa etiqueta `<key>` indicamos que a lista de elementos da colección débese obter revisando a columna `Num_departamento_controla` da táboa da clase Proxecto e que ademais é a clave foránea nesta táboa.
- A etiqueta `inverse="true"`, hai que *engadila sempre que implementemos unha asociación bidireccional e sempre no "lado un"*, é dicir na colección. Na asociación unidireccional non se engade.
- Hibernate manexa por defecto as relacións en modo "`lazy`", é dicir non fai a consulta sobre a táboa Proxecto ata que non pedimos os proxectos que controla o departamento. En modo lazy as consultas SQL lazy son disparadas polos métodos get. Si se accede aos atributos directamente obterase null xa que desta forma non se dispara a consulta SQL.

  Conclusión: recoméndase poñer os atributos private para evitar accesos directos que non disparen as consultas SQL en modo lazy.
- Se o atributo `fetch` toma o valor `select` (valor por defecto), fará unha consulta para obter os datos do departamento e outra consulta para obter os proxectos relacionados. Se polo contrario fetch toma o valor join, hibernate obtén cunha soa consulta (left outer join) os datos do departamento e os seus proxectos.

2.5.6 Asociación Many-to-Many

A relación moitos a moitos consiste en que unha instancia da clase A pode estar relacionada con varias instancias da clase B e unha instancia da clase B pode estar relacionada con varias instancias da clase A .



As relacións moitos a moitos no modelo relacional impleméntanse cunha táboa intermedia.

Exemplo:

As asociacións moitos a moitos represéntanse mediante a etiqueta `<many-to-many>`.

```
<many-to-many
  column="nome da columna"
  formula="expresión SQL"
  class="nome da clase"
  fetch="select|join"
  unique="true|false"
  not-found="ignore|exception"
  entity-name="nome da entidade"
  property-ref="nome da clase referencia"
/>
```

- No caso de que a relación sexa *unidireccional*:
  - Só a entidade "A" ten unha referencia aos obxectos de tipo "B".
  - Esta relación está representada por unha colección e a entidade "A" pode acceder a cada un dos obxectos de tipo "B" desa colección.

```
public class Persoa  implements java.io.Serializable {
     private int idpersoa;
     private String nome;
     private String apelidos;
     private String email;
     private Set<Actividade> actividades = new HashSet<>(0);
.
public class Actividade  implements java.io.Serializable {
    private int idactiv;
    private String nomeActiv;
    private Double prezo;
```

Para mapear a propiedade da colección, no ficheiro `Persoa.hbm.xml` engadimos:

```
<set name="actividades" table="PersoaActividade" >
    <key>
        <column name="IDPersoa" not-null="true" />
    </key>
    <many-to-many entity-name="hbpersoamanytomany.Actividade">
        <column name="IDActiv" not-null="true" />
    </many-to-many>
</set>
```

- No caso de que a relación sexa *bidireccional*:
  - A entidade "A" ten unha referencia aos obxectos de tipo "B" e entidade “B" ten unha referencia aos obxectos de tipo "A".

- Esta relación está representada por unha colección tanto na entidade "A" como na entidade B.
- Como sempre, nunha *asociación bidireccional requírese que se estableza un dos extremos con inverse="true"*.

> Como xa comentamos, que un lado dunha asociación bidireccional ten que mapearse como inverse porque temos nomeada a clave foránea dúas veces. O mesmo principio rexe nas asociacións moitos-a-moitos: *cada fila da táboa intermedia (de enlace) está representada por dous elementos de colección*. Unha asociación entre la entidade A e B está representada en memoria por unha instancia A da colección de B, pero tamén por unha instancia B na colección de A.

```
public class Persoa  implements java.io.Serializable {
     private int idpersoa;
     private String nome;
     private String apelidos;
     private String email;
     private Set<Actividade> actividades = new HashSet<>(0);

public class Actividade  implements java.io.Serializable {
     private int idactiv;
     private String nomeActiv;
     private Double prezo;
     private Set<Persoa> persoas = new HashSet<>(0);
```

Para mapear a propiedade de colección `Actividades`, no ficheiro `Persoa.hbm.xml` engadimos:

```
<set name="actividades" table="PersoaActividade" inverse="true" lazy="true" fetch="select">
    <key>
        <column name="IDPersoa" not-null="true" />
    </key>
    <many-to-many entity-name="hbpersoamanytomany.Actividade">
        <column name="IDActiv" not-null="true" />
    </many-to-many>
</set>
```

Para mapear a propiedade de colección `Persoas`, no ficheiro `Actividade.hbm.xml` engadimos:

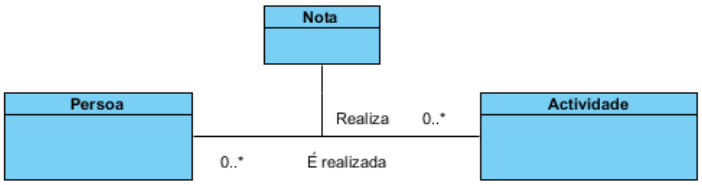
```
<set name="persoas" table="PersoaActividade" inverse="false" lazy="true" fetch="select">
    <key>
        <column name="IDActiv" not-null="true" />
    </key>
    <many-to-many entity-name="hbpersoamanytomany.Persoa">
        <column name="IDPersoa" not-null="true" />
    </many-to-many>
</set>
```

O `inverse="true"` na colección `actividades` do ficheiro `persoa.hbm.xml`, dille a Hibernate que ignore os cambios feitos na colección `actividades` e utilice o outro extremo da asociación, a colección `persoas` para a sincronización coa base de datos.

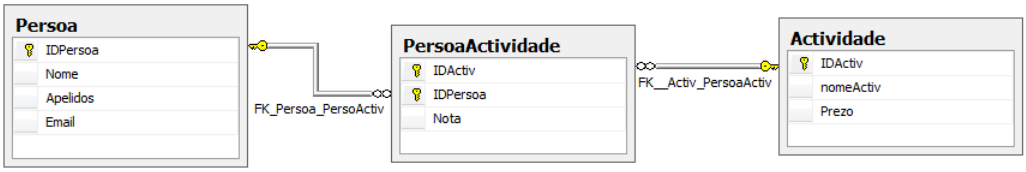
Nas asociacións moitos-a-moitos bidireccionais, non temos porque poñer o mesmo tipo de colección en ambos lados. Por exemplo, podemos poñer unha colección <list> no lado non inverso e un <bag> no lado inverso. Hai que ter en conta que as coleccións indexadas (listas e maps) non funcionan no lado inverso porque Hibernate non inicializará ou manterá a columna do índice destas coleccións.

2.5.7 Asociación Many-to-Many con atributos propios na relación

Supoñamos na relación anterior entre `Actividades` e `Persoas`, que queiramos rexistrar a `nota` obtida de cada persoa na realización da actividade.



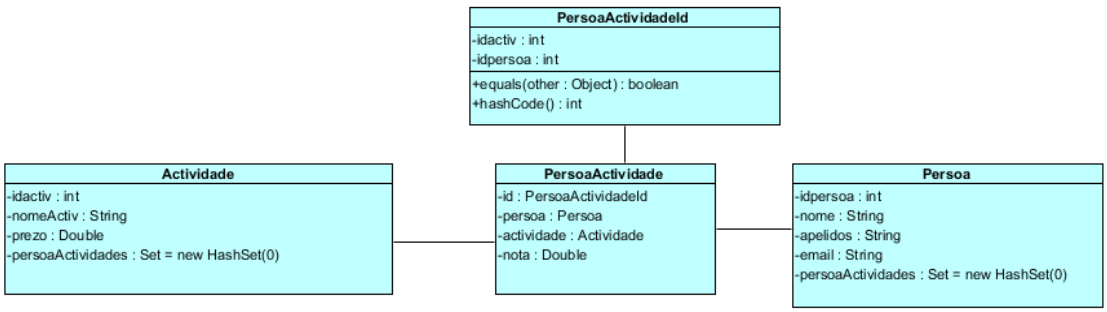
Na base de datos, o atributo `nota` engadiríase á táboa intermedia (join), como se mostra a continuación:



Hai varias estratexias para implementar isto.

Mapeado da táboa join nunha entidade intermedia

Unha opción consiste en establecer entre as entidades `Persoa` e `Actividade`, unha entidade `PersoaActividade`. O identificador de esta entidade é a composición de `IDActiv` e `IdPersoa`. O diagrama de clases amósase a continuación:



```xml
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<!-- Generated 23-mar-2014 21:04:41 by Hibernate Tools 3.6.0 -->
<hibernate-mapping>
    <class name="hbpersoamanytomany.PersoaActividade" table="PersoaActividade" schema="dbo" catalog="Persoa5">
        <composite-id name="id" class="hbpersoamanytomany.PersoaActividadeId">
            <key-property name="idactiv" type="int">
                <column name="IDActiv" />
            </key-property>
            <key-property name="idpersoa" type="int">
                <column name="IDPersoa" />
            </key-property>
        </composite-id>
        <many-to-one name="persoa" class="hbpersoamanytomany.Persoa" update="false" insert="false" fetch="select">
            <column name="IDPersoa" not-null="true" />
        </many-to-one>
        <many-to-one name="actividade" class="hbpersoamanytomany.Actividade" update="false" insert="false" fetch="select">
            <column name="IDActiv" not-null="true" />
        </many-to-one>
        <property name="nota" type="java.lang.Double">
            <column name="Nota" precision="53" scale="0" />
        </property>
    </class>
</hibernate-mapping>
```

As dúas asociacións `<many-to-one>` son só de lectura; `insert` e `update` están postos a `false`. Isto é necesario porque as columnas están mapeadas dúas veces, unha na clave composta (que é responsable da inserción dos valores) e outra nas asociacións many-to-many.

As entidades `Actividade` e `Persoa` teñen unha asociación `one-to-many` coa entidade `PersoaActividade`.

No ficheiro `Persoa.hbm.xml`, engadimos:

```xml
<set name="persoaActividades" table="PersoaActividade" inverse="true" lazy="true" fetch="select">
    <key>
        <column name="IDPersoa" not-null="true" />
    </key>
    <one-to-many class="hbpersoamanytomany.PersoaActividade" />
</set>
```

No ficheiro `Actividade.hbm.xml`, engadimos:

```xml
<set name="persoaActividades" table="PersoaActividade" inverse="true" lazy="true" fetch="select">
    <key>
        <column name="IDActiv" not-null="true" />
    </key>
    <one-to-many class="hbpersoamanytomany.PersoaActividade" />
</set>
```

Unha vantaxe desta estratexia é a posibilidade da navegación bidireccional. O inconvinte é que o código é máis complexo para xestionar as instancias de `PersoaActividade`, para crear e borrar as asociacións (teñen que gravarse e borrarse independentemente) e para a xestión da clave composta. Con todo, podemos permitir persistencia transitiva coas opcións de cascade nas coleccións de Persoa e actividade.

Mapeado da táboa join cunha colección de compoñentes

Outra alternativa é facer a clase `PersoaActividade` de tipo valor, sen un identificador. Esta clase compoñente ten que estar posuída por unha entidade. Por exemplo, imos facer que a propietaria sexa `Persoa`, entón esta identidade ten que ter unha colección de compoñentes.

```
public class PersoaActividade  implements java.io.Serializable {
     private Persoa persoa;
     private Actividade actividade;
     private Double nota;

public class Persoa  implements java.io.Serializable {
     private int idpersoa;
     private String nome;
     private String apelidos;
     private String email;
     private Set persoaActividades = new HashSet(0);
```

No ficheiro `Persoa.hbm.xml`, engadimos o mapeo da clase compoñente.

```
<set name="persoaActividades" table="PersoaActividade" >
    <key>
        <column name="IDPersoa" not-null="true" />
    </key>
    <composite-element class="hbpersoamanytomany.PersoaActividade">
        <parent name="Persoa"/>
        <many-to-one name="actividade" column="IDActiv" not-null="true"
                     class="hbpersoamanytomany.Actividade"/>
        <property name="nota" type="java.lang.Double">
            <column name="Nota" precision="53" scale="0" />
        </property>
    </composite-element>
</set>
```

O inconvinte é que non hai modo de permitir navegación bidireccional: un compoñente como `PersoaActividade` non pode por definición ter referencias compartidas. Non podemos navegar dende `Actividade` a `PersoaActividade`. Non obstante, podemos executar unha consulta para atopar os obxectos que necesitemos.

2.5.8 Persistencia transitiva

Por *defecto* Hibernate, *non navega polas asociacións*, co que operacións de inserción, borrado ou modificación non teñen efecto sobre as entidades asociadas. Pode chegar a ser bastante incómodo gardar, borrar, ou modificar obxectos individuais, especialmente si tratamos cun grafo de obxectos asociados.

As instancias das entidades asociadas teñen ciclos de vida independentes e soportan referencias compartidas. Eliminar unha entidade dunha colección non significa que se borre esta da base de datos. Estas instancias son transient e teñen que ser persistidas si as queremos gardar, borrar ou modificar na base de datos. Entón temos dúas opcións:

- Persistir cada unha das instancias de forma individual. Isto implica máis código.
- Usar a característica de persistencia transitiva, que permite aforrar liñas de código e manexar automaticamente o ciclo de vida das instancias de entidades asociadas.

Para usar a característica de persistencia transitiva, Hibernate permite a configuración de cada unha das asociacións. Para elo usamos o atributo `cascade` nas asociacións de entidades, que especifica que operacións deben ir en cascada dende o obxecto pai ata o obxecto asociado.

Para cada operación básica da sesión de Hibernate - incluíndo persist(), merge(), saveOrUpdate(), delete(), lock(), refresh(), evict(), replicate() - existe un estilo de cascada correspondente. Si se quere que unha operación sexa tratada en cascada ao longo dunha asociación, debemos indicalo no documento de mapeo.

A continuación, preséntanse as opcións que o atributo `cascade` pode aceptar:

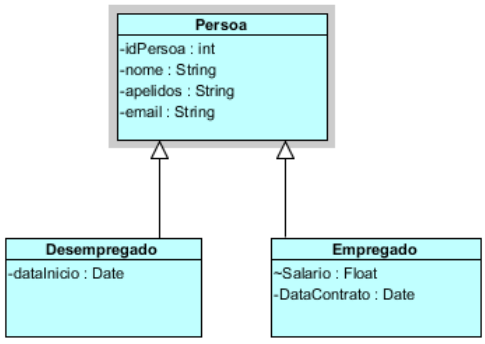
- **none** ->Ignora a asociación e non se propaga ningunha acción. Opción por defecto.
- **save-update** -> Navega a asociación cando a sesión é sincronizada e cando o obxecto executa algún dos métodos `save()`,`saveorupdate` ou `update()`, propaga estás operacións ás entidades asociadas. Os obxectos das entides asociadas poden estar `transient` ou `detached`, e pasan a `persistent`.
- **delete** -> Borra as entidades asociadas en estado persistente.
- **persist** ->Hibernate persiste calquera instancia `transient` asociada cando se chama ao método `persist()` do obxecto.
- **merge** -> Navega a asociación e propaga a operación `merge()`. As instancias das entidades asociadas en estado `detached`, replicanse como `persitent` e as `transient` pasan a `persistent`.
- **Lock** -> Incorpora ao contexto de persistencia aquelas instancias de entidades asociadas que estean en estado detached. O modo de bloqueo (LockMode) non é propagado.
- **evict()** -> Borra da caché todas as instancias de entidades asociadas.
- **refresh** ->Recupérase o estado dos obxectos asociados da base de datos.
- **all** -> Todas as opcións mostradas anteriormente.
- **delete-orphan** -> Borra as entidades asociadas cando son eliminadas da asociación, é dicir, provoca o borrado dos obxectos con só sacalos da colección do pai. Úsase cando a entidade borrada non ten referencias compartidas. Aplícase só as asociacións un-a-moitos.

Recomendacións:

- Usualmente non ten sentido habilitar o tratamento en cascada nunha asociación `<many-to-one>` ou `<many-to-many>` e si se habilita, a opción `saveupdate` sería a que ten sentido. O tratamento en cascada é frecuentemente útil para as asociacións `<one-to-one>` e `<one-to-many>`.
- Se o período de vida dos obxectos fillos está ligado ao período de vida do obxecto pai, unha opción para aforrar código é especificar `cascade="all,delete-orphan"`.

2.6 Mapeamento da herdanza (xerarquías)

As clases java que queremos facer persistentes con Hibernate poden ter herdanza, é dicir, unhas clases herdan doutras. Por exemplo:



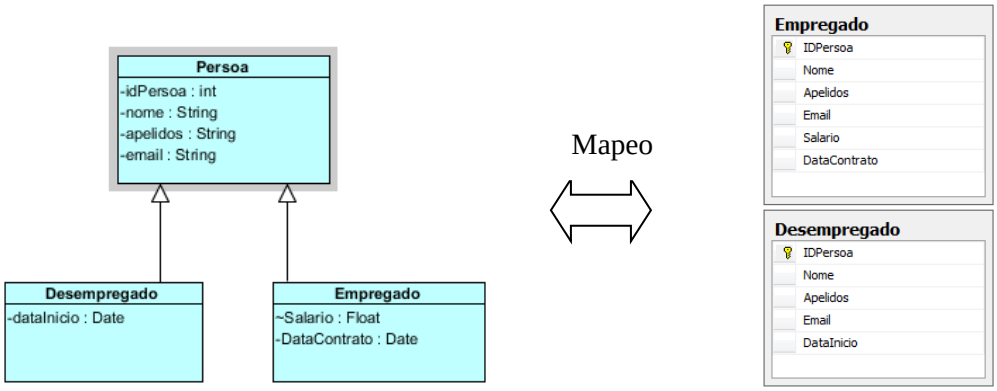
A herdanza é un dos desaxustes estruturais entre a orientación a obxectos e as bases de datos relacionais. Os SXBD non soportan herdanza en forma nativa e polo tanto, é necesario mapear.

Catro posibles estratexias de mapeo:

- Táboa por cada subclase concreta.
- Táboa por subclase concreta con unións.
- Táboa por xerarquía.
- Táboa por clase.

2.6.1 Táboa por cada suclase concreta

Créase unha táboa por cada subclase concreta, redundando todos os atributos.



Mapeamos cada subclase concreta á súa táboa correspondente, do xeito usual e por separado. A superclase `Persoa` non se mapea.

- Mapeamos a clase `Empregado` no ficheiro `Empregado.hbm.xml`:.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
  "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class catalog="Persoa1" name="hbpersoa.Empregado" schema="dbo" table="Empregado">
        <id name="idPersoa" type="int">
            <column name="IDPersoa"/>
            <generator class="assigned"/>
        </id>
        <property name="nome" type="string">
            <column length="15" name="Nome"/>
        </property>
        <property name="apelidos" type="string">
            <column name="Apelidos" length="30"/>
        </property>
        <property name="email" type="string">
            <column name="Email" length="30"/>
        </property>
        <property name="salario" type="java.lang.Double">
            <column name="Salario" precision="53" scale="0" />
        </property>
        <property name="dataContrato" type="date">
            <column name="DataContrato" />
        </property>
    </class>
</hibernate-mapping>
```

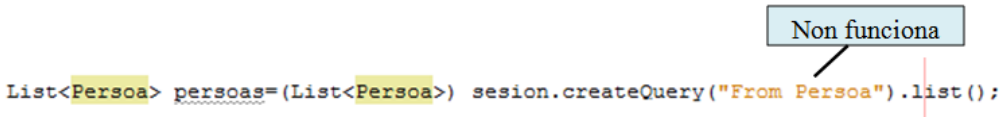
- Mapeamos a clase Desempregado no ficheiro `Desempregado.hbm.xml`:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class catalog="Persoa1" name="hbpersoa.Desempregado" schema="dbo" table="Desempregado">
    <id name="idPersoa" type="int">
      <column name="IDPersoa"/>
      <generator class="assigned"/>
    </id>
    <property name="nome" type="string">
      <column length="15" name="Nome"/>
    </property>
    <property name="apelidos" type="string">
      <column name="Apelidos" length="30"/>
    </property>
    <property name="email" type="string">
      <column name="Email" length="30"/>
    </property>
    <property name="dataInicio" type="date">
      <column name="DataInicio" />
    </property>
  </class>
</hibernate-mapping>
```

Inconvintes :

- Hai que repetir o mapeo das propiedades da subclase en todos os ficheiros de mapeo.
- Non soporta consultas polimórficas. Unha consulta contra a superclase debe executarse como varias SELECTs ou consultas HQL, unha por cada subclase concreta. A superclase non se pode consultar directamente.

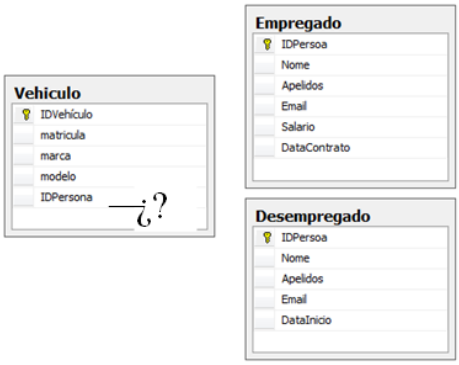
– Ex: non podemos recuperar dunha vez todas as persoas.



- Non da soporte ás asociacións polimórficas. As asociacións na base de datos represéntanse habitualmente como claves foráneas. Na figura anterior, se mapeamos as subclases en diferentes táboas, unha asociación polimórfica coa súa superclase `Persoa` non se pode representar como unha relación de clave foránea. Isto sería problemático no noso modelo si a clase `Persoa` tivera unha relación con outra entidade, por exemplo, un a moitos con `Vehículo`, esta táboa `Vehículo` necesitaría unha columna cunha soa clave foránea que se tería que referenciar con ambas táboas das subclases e isto non é posible.

– Ex: A clase persistente `Vehículo` ten a súa propia táboa asociada.

– ¿A que táboa debe apuntar a clave foránea `IDPersoa`? Aquí o maior problema é que non podemos engadir unha restrición de clave foránea á columna `IDPersoa`, porque algunhas filas corresponden coa táboa `Desempregado` e outras coa táboa `Empregados`. Hai que implementar outra forma de asegurar a integridade (un trigger, por exemplo).



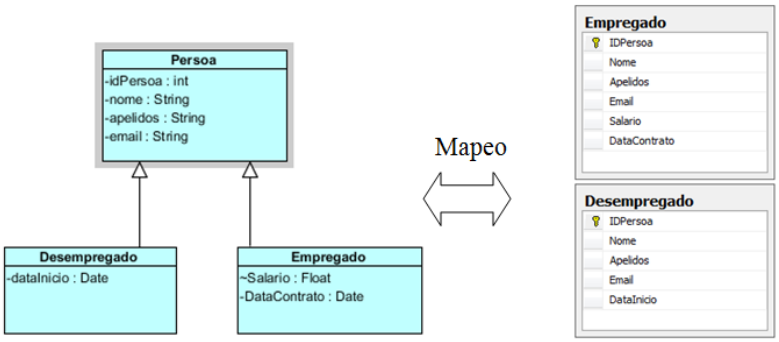
- Un problema conceptual adicional con esta estratexia de mapeamento é que algunhas columnas distintas en distintas táboas comparten a mesma semántica. Isto fai o sistema máis complexo. Por exemplo, un cambio nunha propiedade da superclase implica cambios en múltiples columnas o que fai moito máis difícil implementar restricións de integridade na base de datos.

Recoméndase esta estratexia só no nivel superior da xerarquía onde o polimorfismo non é requirido e as modificacións das superclases non vai ser probable.

2.6.2 Táboa por clase concreta con unións

Como a anterior, cada clase concreta asóciase a unha táboa con todos os atributos. Pero agora definimos *un único ficheiro de mapeo*, onde se especifica:

- Cal é a superclase abstracta, as súas propiedades e como se mapean.
- Cales son as subclases concretas, as súas propiedades e como se mapean nas táboas correspondentes.



Seguimos tendo dúas táboas con columnas da superclase duplicadas pero o que é novo e un mapeamento especial de Hibernate que agora se inclúe na superclase.

As subclases indícanse coa etiqueta:

```
<union-subclass name="nome da subclase" table="nome táboa">
      <property name="nome propiedade " type="tipo de datos">
         <column name="nome da columna" />
      </property>
                .....
 </union-subclass>
```


```
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
  "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class catalog="Persoa1" name="hbpersoa.Persoa" schema="dbo" abstract="true" >
        <id name="idPersoa" type="int">
            <column name="IDPersoa"/>
            <generator class="assigned"/>
        </id>
        <property name="nome" type="string">
            <column length="15" name="Nome"/>
        </property>
        <property name="apelidos" type="string">
            <column name="Apelidos" length="30"/>
        </property>
        <property name="email" type="string">
            <column name="Email" length="30"/>
        </property>
        <union-subclass name="hbpersoa.Empregado" table="Empregado">
            <property name="salario" type="java.lang.Float">
                <column name="Salario" />
            </property>
            <property name="dataContrato" type="date">
                <column name="DataContrato" />
            </property>
        </union-subclass>
        <union-subclass name="hbpersoa.Desempregado" table="Desempregado">
            <property name="dataInicio" type="date">
                <column name="DataInicio" />
            </property>
        </union-subclass>
    </class>
</hibernate-mapping>
```

Como `Persoa` é unha superclase abstracta, ten que declararse como `abstract="true";` noutro caso sería necesaria *unha táboa para as instancias da superclase*.

O mapeamento do identificador está compartido para todas as subclases concretas da xerarquía.

As propiedades da superclase decláranse en `property` e son herdadas por todas as clases concretas. Isto evita a duplicidade.

Cada subclase concreta é mapeada a unha táboa; a táboa herda o identificador da superclase e as propiedades.

Vantaxes:

- Ao usar un ficheiro de mapeo único, a definición das propiedades comúns xa non se duplica.
- As consultas polimórficas xa funcionan. Podemos recuperar instancias de `Empregado` e de `Desempregado`, pero tamén de `Persoa`. Agora si poderíamos realizar a seguinte consulta:

```
List<Persoa> persoas = (List<Persoa>) sesion.createQuery("From Persoa").list();
for (Persoa i : persoas) {
            System.out.println(i.getNome()+" "+i.getApelidos());
}
```

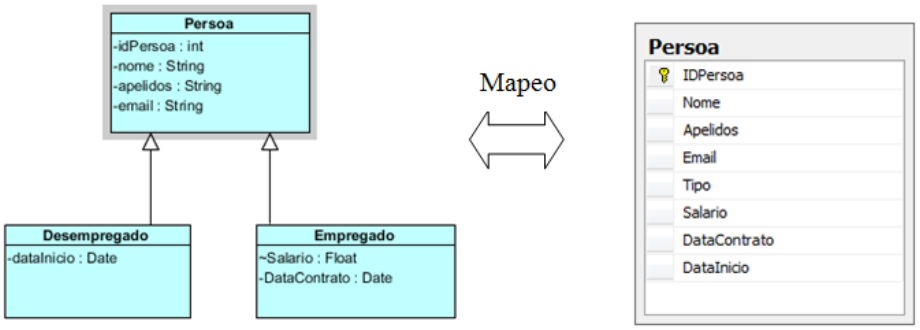
- Pódense chegar a soportar asociacións polimórficas; por exemplo, un mapeo da asociación de Vehículo a Persoa é posible. Hibernate pode usar unha consulta UNION para simular unha única táboa como obxectivo no mapeo da asociación.

2.6.3 Táboa por xerarquía

Agora temos unha *única táboa con todas as propiedades* e toda a xerarquía mapease nesta táboa.

Na táboa debemos incluír unha columna para *o discriminante*, que permite determinar a subclase asociada a cada fila e úsaa internamente Hibernate. Sirve para distinguir entre as clases, non é unha propiedade.

En cada fila, as columnas correspondentes a propiedades non aplicables énchense con nulos.



Para definir a columna da táboa que se vai utilizar para o discriminante, úsase a etiqueta `<discriminator>`.

```
<discriminator column="nome da columna" type="tipo de datos"/>
```

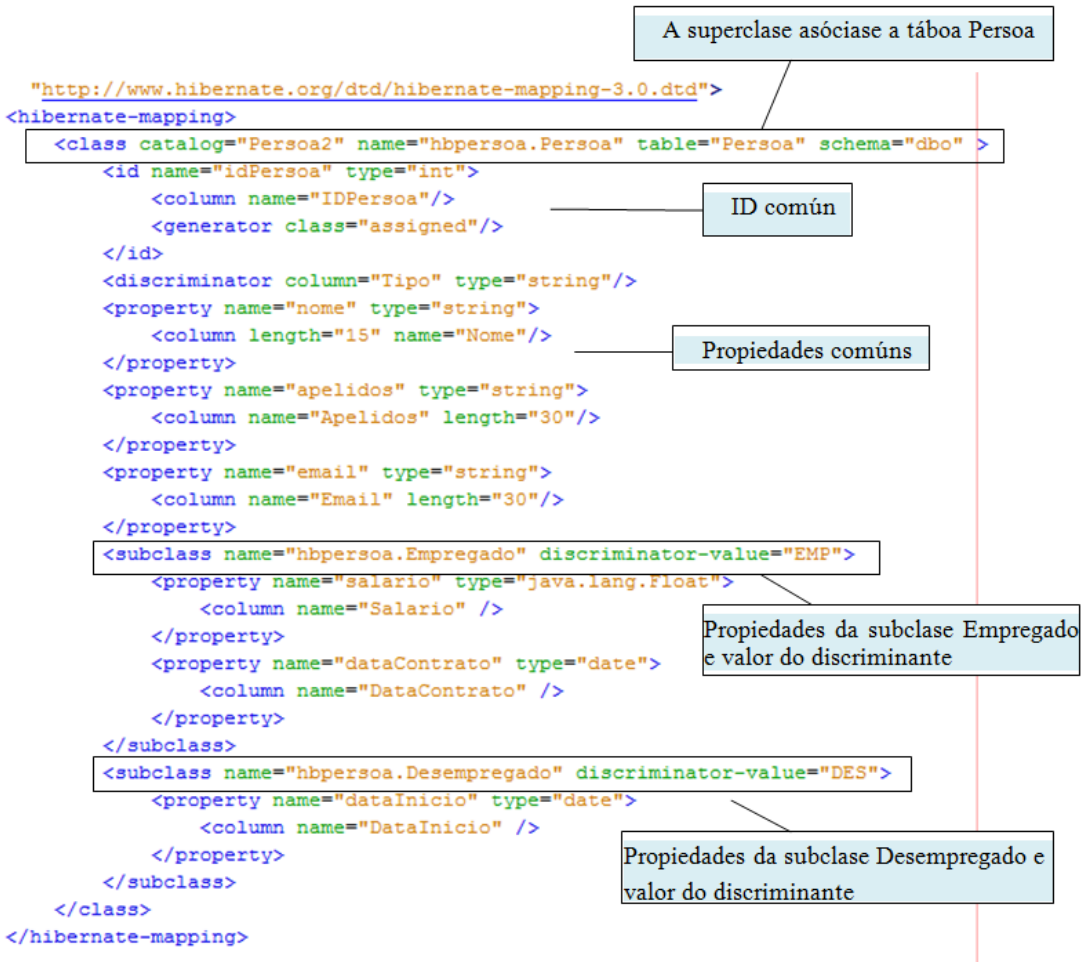
Para mapear cada unha das subclases úsase a etiqueta `<subclass>`.

```
<subclass name="nome da subclase" discriminator-value="valor">
   <property name="nome da propiedade" type="tipo de datos">
        <column name="nome da columna" />
    </property>
……..
</subclass>
```

Cada subclase ten o seu elemento `<subclass>`. As propiedades da superclase mapéanse como sempre cun elemento `<property>`.

Un elemento `<subclass>` pode ter á súa vez outros elementos `<subclass>` para mapear unha xerarquía completa nunha táboa.

No atributo `discriminator-value="valor"` póñense os valores da columna discriminante que serve para diferenciar ás subclases.



A clase raíz Persoas da xerarquía mapéase coa táboa Persoas. Na base de datos engadíuse a columna Tipo que é o discriminante da clase e os valores que pode tomar son "EMP" ou "DESC". Nas columnas da subclase non están permitidas as restricións NOT NULL.

As veces, non temos a liberdade para incluír unha columna discriminante. Neste caso, podemos aplicar *unha fórmula* para calcular o discriminador en cada fila. Por exemplo:

```
<discriminator
        formula="case when salario is not null then 'EMP' else 'DESC' end"
        type="string"/>
...
<subclass name="hbpersoa.Desempregado" discriminator-value="DES">
```

Esta estratexia é a mellor en termos de simplicidade e rendemento. Presenta o mellor rendemento para representar o polimorfismo e máis fácil de implementar. É posible unha consulta ad-hoc sen joins e unións complexas. Facilita a evolución do esquema.

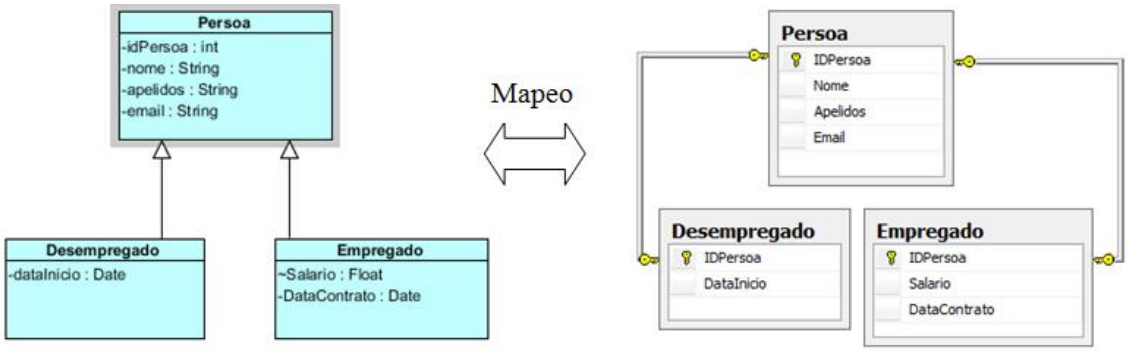
Pero presenta algúns problemas maiores:

- Pode ser un serio problema o feito de que todas as columnas das subclases deben admitir nulos e o modelo ten subclases con propiedades coa restrición de que non pode ser nulo.
- Outro problema é a normalización. Temos dependencias entre columnas que non son clave violando a terceira forma normal.
- Complícase manter a integridade dos datos, sobre todo no caso de que se queira impoñer unha restrición só ás ocorrencias dunha subclase, por exemplo, se quixésemos que só as persoas que son empregados poidan ser propietarios dun coche.

2.6.4 Táboa por clase

Tanto a superclase como as subclases son persistentes e teñen a súa táboa. As relacións de herdanza represéntanse mediante claves foráneas. Todas comparten a clave primaria.

Cada táboa que representa as subclases só ten columnas para as propiedades non herdadas (cada clase, as súas) xunto coa clave primaria que é unha clave foránea da táboa da superclase.



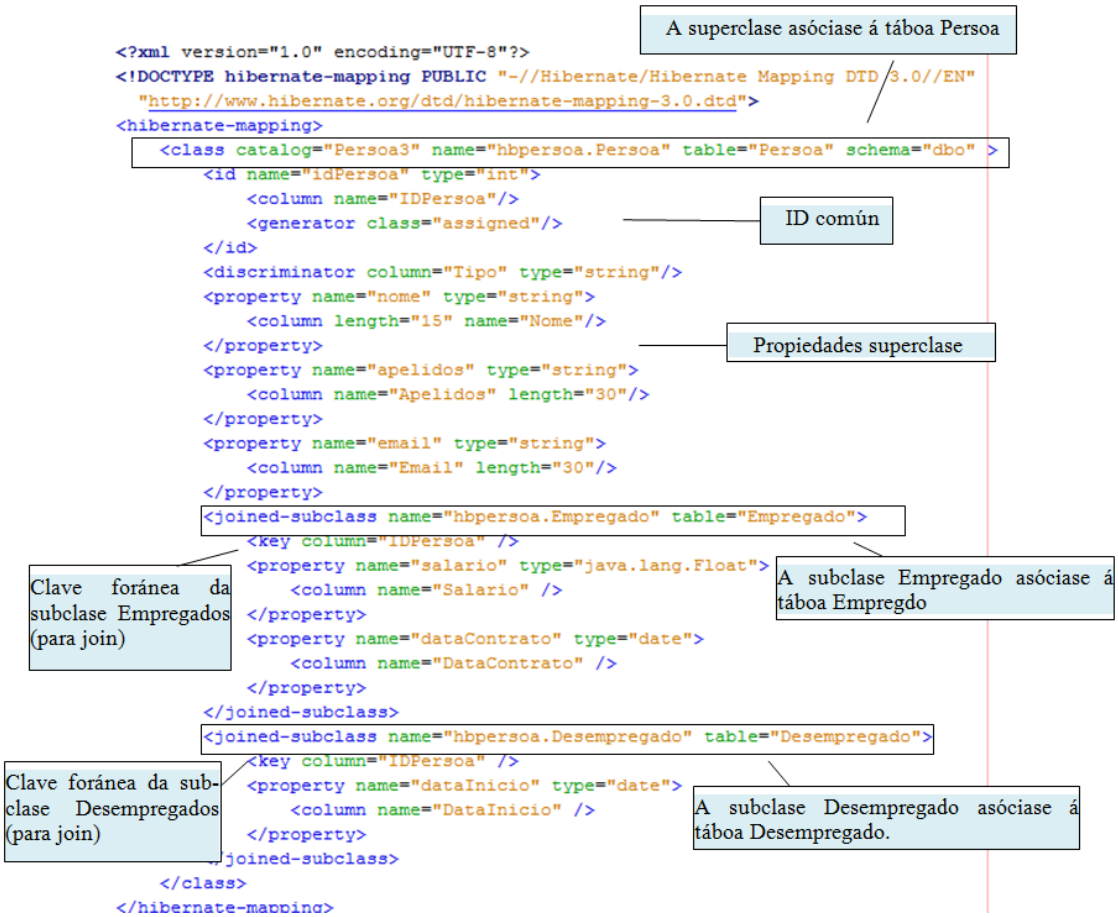
Se unha instancia da subclase `Empregado` é feita persistente, os valores das propiedades declaradas na superclase `Persoa` son persistidos nunha nova fila da táboa `Persoa`. Só os valores das propiedades declaradas pola subclase son persistidos nunha nova fila da táboa `Empregado` e a súas filas quedan enlazadas polo valor da súa clave primaria. Despois, a instancia de subclase pódese obter da base de datos facendo join coa táboa da superclase.

En Hibernate, usamos o elemento `<joined-subclass>` para crear unha táboa por subclase:

```
<joined-subclass name="nome da subclase" table="nome da táboa">
        <key column="clave foránea" />
        <property name="nome propiedade" type="tipo de datos">
            <column name="nome da columna" />
        </property>
          ……..
 </joined-subclass>
```

O elemento `<joined-subclass>` mapea unha subclase cara unha táboa e todas as propiedades declaradas na subclase son mapeadas nesta táboa. Unha clave primaria requírese para a táboa que mapea a subclase. Esta columna tamén ten unha restricción de clave foránea coa clave primaria da táboa que mapea á superclase.

Un elemento <joined-subclass> pode conter outros elementos <joined-subclass> para mapear toda a xerarquía.




```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
  "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class catalog="Persoa3" name="hbpersoa.Persoa" table="Persoa" schema="dbo" >
        <id name="idPersoa" type="int">
            <column name="IDPersoa"/>
            <generator class="assigned"/>
        </id>
        <discriminator column="Tipo" type="string"/>
        <property name="nome" type="string">
            <column length="15" name="Nome"/>
        </property>
        <property name="apelidos" type="string">
            <column name="Apelidos" length="30"/>
        </property>
        <property name="email" type="string">
            <column name="Email" length="30"/>
        </property>
        <joined-subclass name="hbpersoa.Empregado" table="Empregado">
            <key column="IDPersoa" />
            <property name="salario" type="java.lang.Float">
                <column name="Salario" />
            </property>
            <property name="dataContrato" type="date">
                <column name="DataContrato" />
            </property>
        </joined-subclass>
        <joined-subclass name="hbpersoa.Desempregado" table="Desempregado">
            <key column="IDPersoa" />
            <property name="dataInicio" type="date">
                <column name="DataInicio" />
            </property>
        </joined-subclass>
    </class>
</hibernate-mapping>
```

Unha vantaxe desta estratexia é que o esquema SQL está normalizado e a evolución do esquema e a definición de restricións é fácil. Unha asociación polimórfica para unha subclaseparticular pódese representar como unha clave foránea referenciando a táboa desa subclase particular.

Cando facemos unha consulta na superclase, internamente Hibernate ten que realizar unha consulta máis complexa, baseada en outer join. O rendemento pode verse reducido considerablemente nunha xerarquía complexa. As consultas requiren `joins` entre moitas táboas e moitas lecturas secuenciais.