UNIVERSIDAD NACIONAL
DE MAR DEL PLATA

Bases de datos

Mapeo objeto-relacional

Mg. Javier Bazzocco - javier.bazzocco@lifia.info.unlp.edu.ar

Contenido

- Introducción al mapeo
- Mapeo de cada uno de los conceptos OO
- Introducción a mapeadores
    - Hibernate
        - Configuración básica
        - Proceso de desarrollo
        - Archivos de mapeo
        - Ejemplos
- Criterios de diseño y arquitectura

Mapeo objeto - relacional

- Escenario de trabajo:
  - Contamos con un diseño orientado a objetos.
  - Contamos con una base de datos relacional.
  - Se intenta persistir la información administrada por el sistema O.O. en una base de datos relacional.

  - Hoy por hoy es una selección correcta de paradigmas?
  - Posibles inconvenientes?

Mapeo objeto - relacional

- Ventajas de la orientación a objetos:
  - Permite diseñar soluciones robustas y flexibles.
  - Contiene conceptos como abstracción, herencia, polimorfismo, encapsulamiento.
  - La mayoría de las plataformas de desarrollo hoy por hoy son o están tendiendo a ser orientadas a objetos.
  - Múltiples técnicas de diseño disponibles (MVC, double dispatching, patrones de diseño, etc).
  - Ideal para aplicaciones con mucho “comportamiento”.

Mapeo objeto - relacional

- Ventajas de las bases de datos relacionales:
  - Tecnología madura (+ de 40 años).
  - Gran base de instalaciones.
  - Mucha experiencia adquirida.
  - Lenguaje estándar: SQL 92.
  - Base mantemática (álgebra relacional y cálculo de tuplas).
  - Performance en aplicaciones orientadas a datos.

Mapeo objeto - relacional

- Recuento de los posibles inconvenientes:
  - Las bases de datos están orientadas a “datos”.
  - El paradigma OO se interesa por el “comportamiento”.
  - Los tipos soportados por las bases de datos relacionales son limitados.
  - Hay conceptos de bases de datos que no resultan naturales para el desarrollador OO (tx, concurrencia, performance).
  - Ambientes de prueba?
- Posible solución: “mapear” un sistema OO a una base de datos relacional.

Mapeo objeto - relacional

- Cualidades deseables de la integración?
  - Transparencia para el sistema OO de los detalles de la persistencia.
  - Control de todas las propiedades ACID.
  - Performance sin compromisos.
  - Integración fácil y sin limitaciones.
  - Soporte para múltiples bases de datos (sql 92).

Mapeo objeto - relacional

- Estrategias de mapeo
    - OIDs
        - Principio de unicidad
        - Sin valores del dominio
    - Mapeo de atributos a columnas
    - Mapeo de clases a tablas
    - Mapeo de relaciones
        - Uno a uno
        - Uno a muchos
        - Muchos a muchos
        - Asociación vs agregación (asociación fuerte)

Mapeo objeto - relacional

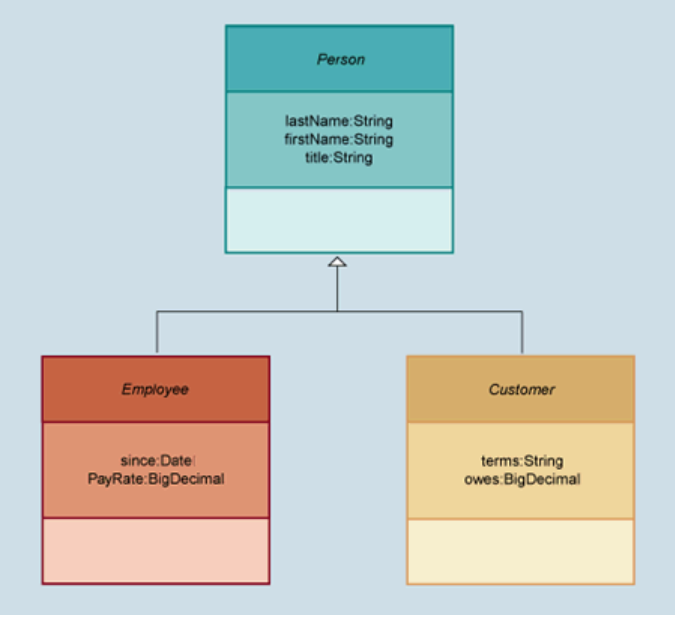
- Mapeo de OIDs
  - En general el OID de un objeto no es accesible dentro de los sistemas OO.
  - Al mapearlo al concepto de clave primaria de una BDR se debe considerar lo siguiente:
    - Debe ser único
    - No debe tener valores relacionados con el dominio.
  - Existen diferentes estrategias para su generación.
  - Impacto negativo en el diseño de objetos (aparece en las clases!!).

Mapeo objeto - relacional

- Mapeo de atributos a columnas
  - Los atributos de las clases se mapean a cero o más columnas de una o más tablas.
  - No todos los tipos son directamente mapeables.
  - Esto significa:
    - Existen atributos que no se desea persistir.
    - Existe la posibilidad de tener diferente granularidad entre el diseño OO y el de la base de datos.
    - Qué pasa con los atributos multivaluados (colecciones por ejemplo)?

Mapeo objeto - relacional

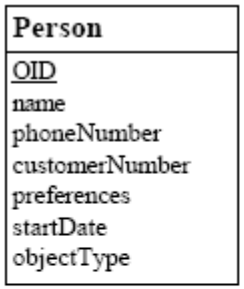
- Mapeo de clases a tablas
  - Es importante tratar de respetar el concepto de herencia.
  - Existen fundamentalmente tres alternativas diferentes:
    - Mapeo de toda la jerarquía a una sola tabla [1].
    - Una tabla por clase concreta [2].
    - Una tabla para cada clase [3].



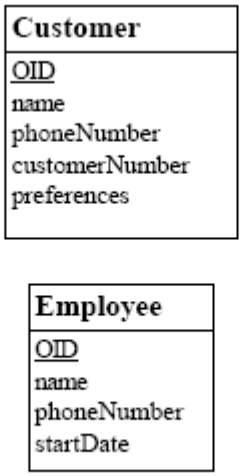
Mapeo objeto - relacional

- Mapeo de clases a tablas

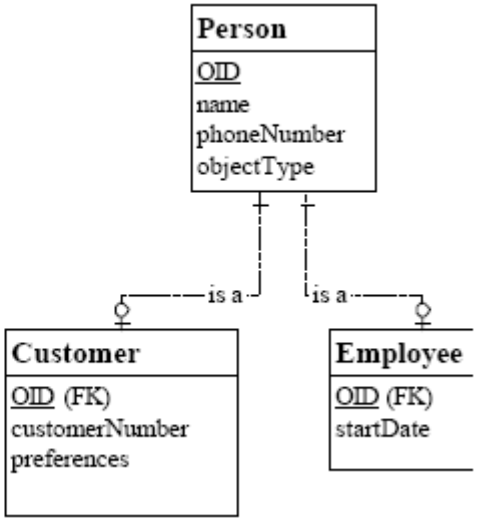
[1]



[2]



[3]



Mapeo objeto - relacional

- Mapeo de clases a tablas
  - Una tabla para toda la jerarquía
    - Ventajas:
      - Las consultas sobre todas las instancias son simples (no se requieren joins).
      - No se repite información inútilmente para instancias con 2 o más roles.
    - Desventajas:
      - Cada vez que se modifica cualquier clase hay que alterar todas las demás.
      - Se desperdicia mucho espacio de almacenamiento.

Mapeo objeto - relacional

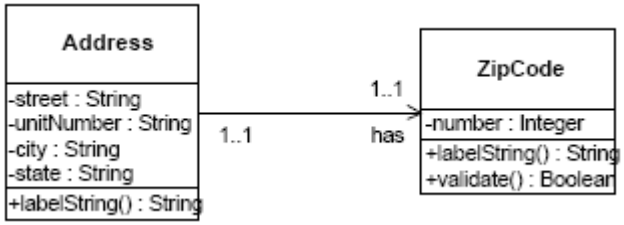
- Mapeo de clases a tablas
  - Una clase para cada clase concreta
    - Ventajas:
      - Los cambios en una clase no afectan a otras tablas mas que a la propia.
    - Desventajas:
      - Cada vez que se modifica una superclase hay que asegurarse de modificar cada una de las subclases.
      - Es complicado cuando hay instancias con varios roles.

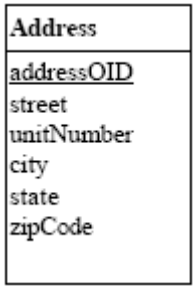
Mapeo objeto - relacional

- Mapeo de clases a tablas
  - Una tabla para cada clase
    - Ventajas:
      - Es la opción que más se asemeja al paradigma OO.
    - Desventajas:
      - Implica más tablas para mantener.
      - El acceso es más lento ya que siempre se requiere un join.
      - Es dificil soportar múltiples roles.

Mapeo objeto - relacional

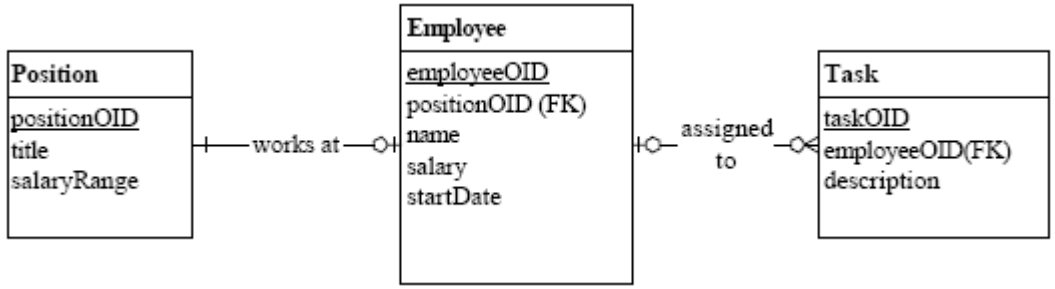
- Mapeo de clases a tablas
  - Existe una cuarta alternativa?
    - Mapeo de varias clases a una sola tabla.
    - En qué casos se puede llegar a utilizar?



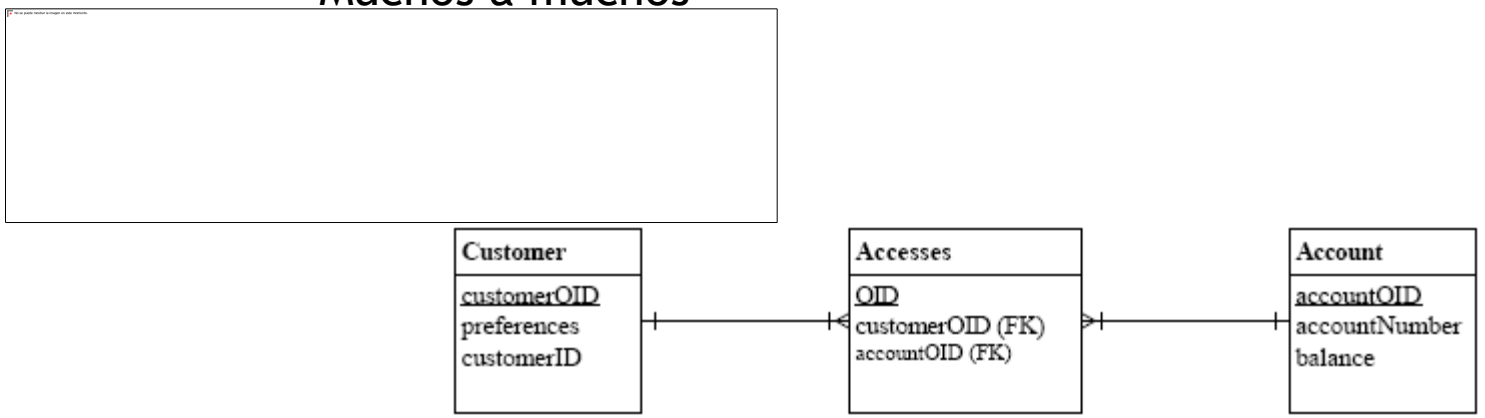


Mapeo objeto - relacional

- Mapeo de relaciones
  - En una base de datos relacional se mantienen mediante el uso de claves foráneas.
  - uno a muchos



  - Muchos a muchos



Mapeo objeto - relacional

- Asociación vs Agregación
  - La agregación es una asociación que incorpora el concepto de “parte de” en un diseño OO.

  - En términos de la base de datos, estructuralmente no hay diferencias.
  - Se requieren elementos adicionales (S.P., Triggers) para mantener la consistencia.

Mapeo objeto - relacional

- Otras consideraciones
  - Operaciones CRUD y el modelo de objetos.
  - Transacciones y las propiedades ACID. Atomicidad, Consistencia, Aislamiento y Durabilidad
  - Esquemas de Locking
    - Pesimista y optimista. Bloqueos por concurrencia

https://unpocodejava.wordpress.com/2011/01/10/tecnicas-de-bloqueo-sobre-base-de-datos-bloqueo-pesimista-y-bloqueo

  - Versionamiento.
  - Pseudo lenguajes de consulta.
  - Triggers.
  - Stored procedures.

Mapeo objeto - relacional

- Resumen
  - El paradigma OO y las BDR son la norma.
  - ODBC/JDBC/ADO no son suficientes.
  - Se requiere una capa de persistencia.
  - Harcodear SQl es una muy mala idea.
  - Se requiere acceso a datos “legacy”.
  - El modelo de datos no debería guiar el diseño.
  - Los joins son lentos.
  - Claves con significado en el dominio no son buenas.
  - Se requieren múltiples formas de mapeo de jerarquías.
  - Los stored procedures son una mala idea.

- Introducción
- Arquitectura
- Proceso de desarrollo
- Configuración
- Ejemplos
- Lenguajes de consulta
- Patrones de diseño
- Tips

Hibernate - Introducción

- ¿Qué es Hibernate?
  - Es una herramienta de mapeo objeto/relacional (ORM) para ambientes Java.
  - Framework de persistencia, basado en objetos, para bases de datos relacionales.
- ¿Por qué surge?
  - Necesidad de persistir objetos Java en bases de datos relacionales.

Hibernate

- ¿Qué características tiene?
  - Herencia
  - Polimorfismo
  - Relaciones uno a muchos, muchos a uno, uno a uno, muchos a muchos
  - Claves compuestas
  - Colecciones de datos
  - Cache
    - Varios cache providers
  - Transacciones
  - Lazy Initialization
  - HQL - SQL

ORM – Object Relational Mapping

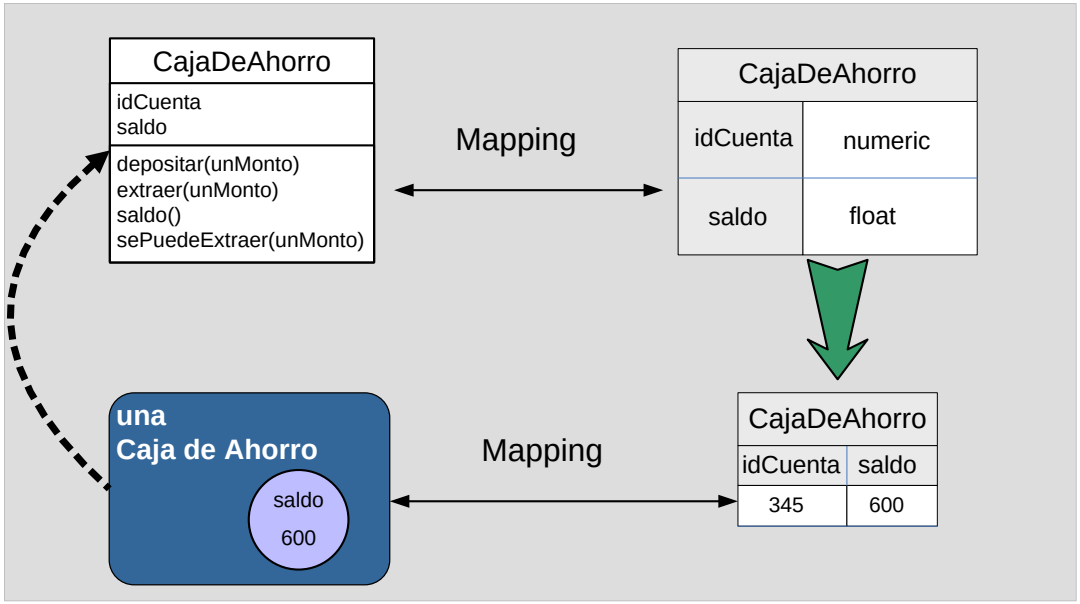
- ¿Qué es un mapping o mapeo?
  - Es un conjunto de reglas que establecen la forma en que los conceptos OO son llevados a conceptos en el modelo relacional y viceversa.

- ¿Cómo debe ser?
  - Idempotente.
  - Sin pérdida de información.

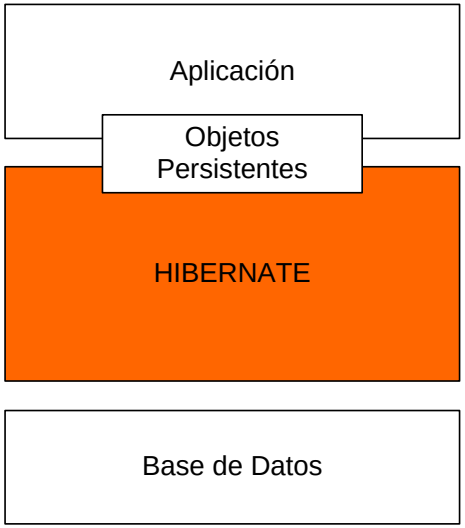
- ¿Cómo se establece el mapeo?
  - Hibernate utiliza comúnmente un archivo asociado a cada clase persistente **con extensión hbm.xml**; Ej. Consumidor.java -> Consumidor.hbm.xml.

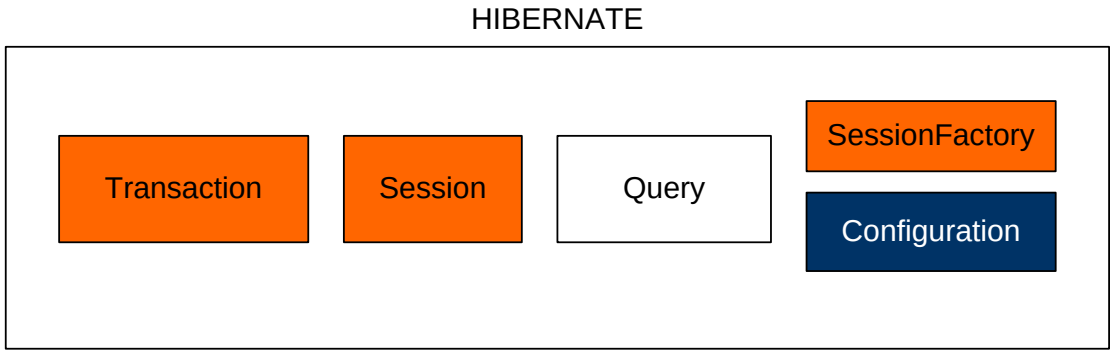
ORM - Object Relational Mapping

- Primer acercamiento
  - Clase = Tabla
  - Objeto = Tupla
  - Atributo del objeto = Columna de la tabla



Visión de alto nivel de la Arquitectura

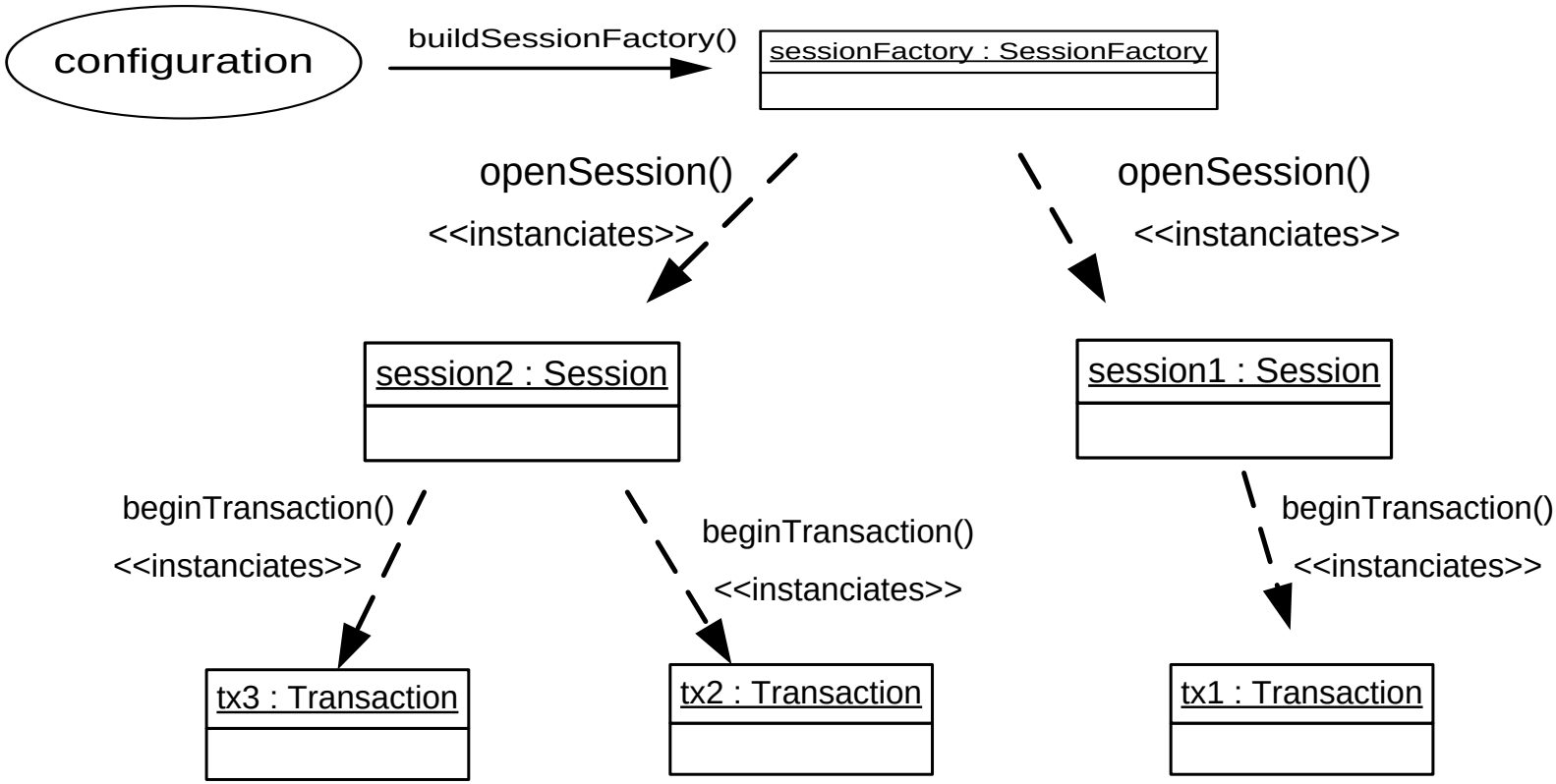




Clases principales de Hibernate

- SessionFactory
  - Es una caché (inmutable) de mappings compilados para una base de datos.
  - Es un factory para las sesiones .
- Session
  - Representa una “conversación” entre la aplicación y el almacenamiento persistente.
  - Es un factory para las transacciones .
  - Mantiene una caché de objetos persistentes, usada cuando se navega el grafo de objetos o para buscar objetos por identificador.
- Transaction
  - Usadas por la aplicación para especificar unidades de trabajo atómicas.

Arquitectura - Relaciones



Configuración de Hibernate

- ¿Qué se debe configurar en Hibernate?
  - Cómo obtener una conexión JDBC.
  - Dialecto de la base datos.
  - Manejo de transacciones.
  - Mappings de las clases persistentes.

- ¿Cómo se configura?
  - A través de un descriptor, el mismo puede ser:
    - Un archivo XML.
    - Un archivo de Propiedades.
    - Programáticamente, es decir, por código Java.

Configuración de Hibernate a través de un XML

- Archivo XML
  - El nombre por defecto es hibernate.cfg.xml

```
<hibernate-configuration>
     <session-factory>
        <property
       name="hibernate.connection.driver_class">org.hsqldb.jdbcDriver
        </property>
        <property name="hibernate.connection.url">jdbc:hsqldb:data/test
        </property>
        <property name="hibernate.connection.username">usuario</property>
        <property name="hibernate.connection.password">secreto</property>
        <property name="dialect">org.hibernate.dialect.HSQLDialect</property>
        <property name="show_sql">true</property>
        …
        <!- Mappings -->
        <mapping resource="paquete/miClase.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

  - Aquí vemos para cada propiedad cual es su valor correspondiente.
  - Inicialización desde Java:

```
Configuration cfg = new Configuration().configure();
SessionFactory sf = cfg.buildSessionFactory();
```

Configuración de Hibernate

RDBMS	Dialect
DB2	org.hibernate.dialect.DB2Dialect
DB2 AS/400	org.hibernate.dialect.DB2400Dialect
DB2 OS390	org.hibernate.dialect.DB2390Dialect
PostgreSQL	org.hibernate.dialect.PostgreSQLDialect
MySQL	org.hibernate.dialect.MySQLDialect
MySQL with InnoDB	org.hibernate.dialect.MySQLInnoDBDialect
MySQL with MyISAM	org.hibernate.dialect.MySQLMyISAMDialect
Oracle (any version)	org.hibernate.dialect.OracleDialect

Clases Persistentes

- Las clases persistentes son aquellas clases de las aplicaciones, que representan las entidades del negocio y cuyas instancias se pretenden almacenar.

- Para que Hibernate funcione correctamente, es recomendable que estas clases sigan un conjunto simple de reglas, conocidas como modelo de programación POJO (Plain Old Java Object).

Plain Old Java Object (POJO) & Hibernate

- Hay cuatro reglas principales a seguir:
  - Implementar un constructor sin argumentos.
  - Proveer un identificador (opcional).
  - Son preferibles las clases no-finales (opcional).
  - Declarar setters y getters para los campos persistentes (opcional).

Plain Old Java Object (POJO) & Hibernate

- Implementar constructor sin argumentos
  - Las clases persistentes deben tener el constructor por defecto (el cual puede ser no-público) para que Hibernate pueda instanciar los objetos usando Constructor.newInstance().
  - Se recomienda tener el constructor por defecto con al menos visibilidad “package”, para la generación en tiempo de ejecución de proxies en Hibernate

Plain Old Java Object (POJO) & Hibernate

- Proveer un identificador (opcional)
  - Esta propiedad se mapea a la columna de clave primaria de la tabla de la base de datos.
  - El identificador es opcional. Se puede omitir y dejar que Hibernate controle internamente los identificadores de los objetos. Sin embargo, no se recomienda esta práctica.
  - Algunas funcionalidades estarán disponibles solamente para las clases que declaran un identificador.
  - Se recomienda declarar identificadores con nombres significativos en las clases persistentes. También se recomienda usar tipos no primitivos (que puedan ser *null*).

Plain Old Java Object (POJO) & Hibernate

- Son preferibles las clases no-finales (opcional)
  - Una característica central de Hibernate son los proxies, y para poder utilizarlos es necesario que las clases persistentes sean *no-final* o que implementen una *interface* con todos los métodos públicos.
  - Se pueden persistir con Hibernate clases finales que no implementen una *interface*, pero entonces no se podrán usar proxies para las asociaciones lazy, lo cual limita las opciones a la hora de hacer ajustes de performance.
  - Se debe evitar también declarar métodos *public final* en las clases *no-finales*. Si se desea usar métodos *public final*, se deben explícitamente deshabilitar los proxies seteando lazy="false“.

Plain Old Java Object (POJO) & Hibernate

- Declarar setters y getters para los campos persistentes (opcional)
  - Es mejor tener una indirección entre el esquema relacional y las estructuras internas de datos de las clases.
  - Por defecto, Hibernate persiste propiedades del estilo *Javabeans*, y reconoce métodos de la forma getFoo(), isFoo() y setFoo(Foo foo).
  - Se puede utilizar el acceso directo a las propiedades de la clase, en caso de ser necesario.
  - No es necesario que las propiedades sean declaradas públicas. Hibernate puede persistir propiedades con getters y setters con visibilidad default, protected o private.

Mappings en Hibernate

- El mapeo objeto/relacional se define en un archivo XML.
- El lenguaje de mapeo esta centrado en la declaración de las clases persistentes, y no en la declaración de las tablas.
- Se pueden escribir los archivos XML de mapeo a mano o con alguna de las herramientas existentes, como ser XDoclet y otras.
- Es una buena práctica mapear solamente una clase persistente (o una sola jerarquía de clases persistentes) en un archivo de mapeo y llamar a éste como la superclase, por ejemplo: Cat.hbm.xml, Dog.hbm.xml, o si se utiliza herencia, Animal.hbm.xml.

Mappings en Hibernate

- La estructura básica es:

```
<?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>
[...]
</hibernate-mapping>
```

Ejemplo Mapping

```
package prueba;
public class Persona {
    private Long idPersona;
    private String nombre;
    private Date fechaNacimiento;
    public Date getFechaNacimiento() {
      return fechaNacimiento;
    }
    public void setFechaNacimiento(Date
      fechaNacimiento) {
      this.fechaNacimiento =
      fechaNacimiento;
    }
    public Long getIdPersona() {
      return idPersona;
    }
    public void setIdPersona(Long
      idPersona) {
      this.idPersona = idPersona;
    }
    public String getNombre() {
      return nombre;
    }
    public void setNombre(String
      nombre) {
      this.nombre = nombre;
    }
}
```

Persona.hbm.xml (Archivo Mapping)

```
<hibernate-mapping package="prueba">
  <class name="Persona" table="PERSONA">

    <id name="idPersona" column="ID_PERSONA">
            <generator class="native"/>
    </id>

    <property name="nombre" not-null="true"/>

    <property name="fechaNacimiento"/>

  </class>
</hibernate-mapping>
```

Tag <class>

- Se utiliza para mapear una clase persistente

```
<class name="ClassName"
        table="tableName"
        discriminator-value="discriminator_value"
        mutable="true|false"
        dynamic-update="true|false"
        dynamic-insert="true|false"
        select-before-update="true|false"
        where="arbitrary sql where condition"
                lazy="true|false"
        rowid="rowid"
        abstract="true|false"
        …
/>
```

Tag <class> - atributos

name:

- El nombre completo de la clase.

table (opcional – default → el nombre de la clase):

- El nombre de la tabla en la base.

discriminator-value (opcional – default → nombre de la clase):

- Un valor que distingue clases individuales, se usa para comportamiento polimórfico.

mutable (opcional, default → true):

- Especifica si las instancias son mutables o no.

dynamic-update (optional, default → false):

- Especifica si la sentencia SQL UPDATE sólo debe contener los campos que cambiaron.

dynamic-insert (optional, default → false):

- Especifica si la sentencia INSERT debe ser generada sólo con las columnas que son not-null .

Identificadores Simples - Elemento id

- Todas las clases persistentes deben declarar la columna de la clave primaria.
- Tendrán una variable que actuará como identificador.

```
<id       name="propertyName"
          type="typename"
          column="column_name"
          unsaved-value="null|any|none|undefined|id_value"
          access="field|property|ClassName">
          node="element-name|@attribute-
  name|element/@attribute|."

          <generator class="generatorClass"/>
</id>
```

name:

- El nombre de variable de instancia que se utiliza como identificador.

type (opcional):

- Un nombre que indica el tipo de hibernate a usar.

Identificadores Simples - Elemento id

column (opcional - default → nombre de la variable):

- El nombre de la columna de la clave primaria.

unsaved-value (opcional – default → valor por defecto de Java):

- Un valor de la variable de instancia que indica que una instancia es nueva (no ha sido guardada); se utiliza para distinguir entre una instancia desacoplada (deattached) que fueron guardadas o cargadas por sesiones anteriores.

access (opcional - defaults → property):

- La estrategia a utilizar para acceder el valor de la propiedad.

Elemento <property>

- Utilizando el tag `<property>` se mapean las propiedades estilo JavaBean de las clases.

```
<property name="propertyName"
        column="column_name"
        type="typename"
        update="true|false" insert="true|false"
        formula="arbitrary SQL expression"
        access="field|property|ClassName"
        lazy="true|false"
        unique="true|false"
        not-null="true|false"
        generated="never|insert|always"
        …
  />
```

Atributos - Elemento <property>

name:

- El nombre de la propiedad.

column (opcional – default → nombre de la propiedad):

- El nombre de la columna a la que mapeará la propiedad.

type (opcional):

- El nombre que indica el tipo a usar (básicos o custom)

update, insert (opcional – default → true)

- Especifica si la(s) columna(s) deben ser incluidas en las sentencia SQL UPDATE y/o INSERT.

formula (opcional):

- Una expresión SQL que define el valor para una propiedad calculada. Dichas propiedades no poseen una columna

Atributos - Elemento <property>

access (opcional – default → property):

- La estrategia que debe usar hibernate para acceder a la propiedad.

lazy (opcional – default → false):

- Especifica si la propiedad debe ser cargada lazy cuando la variable de instancia es accedida por primera vez .

unique (opcional):

- Permite la generación de un restricción unique.

not-null (optional):

- Permite la generación de una restricción not-null para la columna.

generated (opcional – default → never):

- Especifica si el valor de la propiedad es generada por la base de datos.

Tipos Predefinidos

- Hibernate mapea los tipos estándar de java.

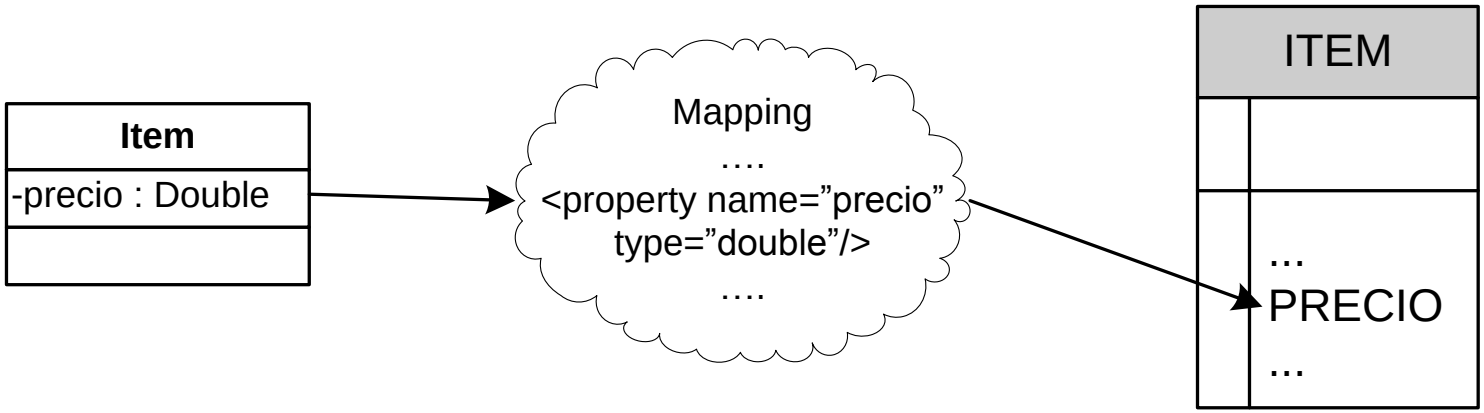
Mapping type	Java type	Standard SQL built-in type
integer	int or java.lang.Integer	INTEGER
long	long or java.lang.Long	BIGINT
short	short or java.lang.Short	SMALLINT
float	float or java.lang.Float	FLOAT
double	double or java.lang.Double	DOUBLE
big_decimal	java.math.BigDecimal	NUMERIC
character	java.lang.String	CHAR(1)
string	java.lang.String	VARCHAR
byte	byte or java.lang.Byte	TINYINT
boolean	boolean or java.lang.Boolean	BIT
yes_no	boolean or java.lang.Boolean	`CHAR(1) ('Y' or 'N')`
true_false	boolean or java.lang.Boolean	`CHAR(1) ('T' or 'F')`

Tipos Definidos por el usuario (Custom Types)

- Hibernate provee dos interfaces para definir nuevos tipos
  - `UserType`
  - `CompositeUserType`
- La clase del “tipo” no es el tipo de la propiedad, sino que es una clase que sabe como serializar instancias de otra clase desde y hacia JDBC.
- Existen para que el usuario mapee elegantemente sus tipos comunes
- ¿En qué se diferencian las interfaces?
  - Un tipo que implementa CompositeUserType puede ser usado en queries, mientras que la que implementa UserType no

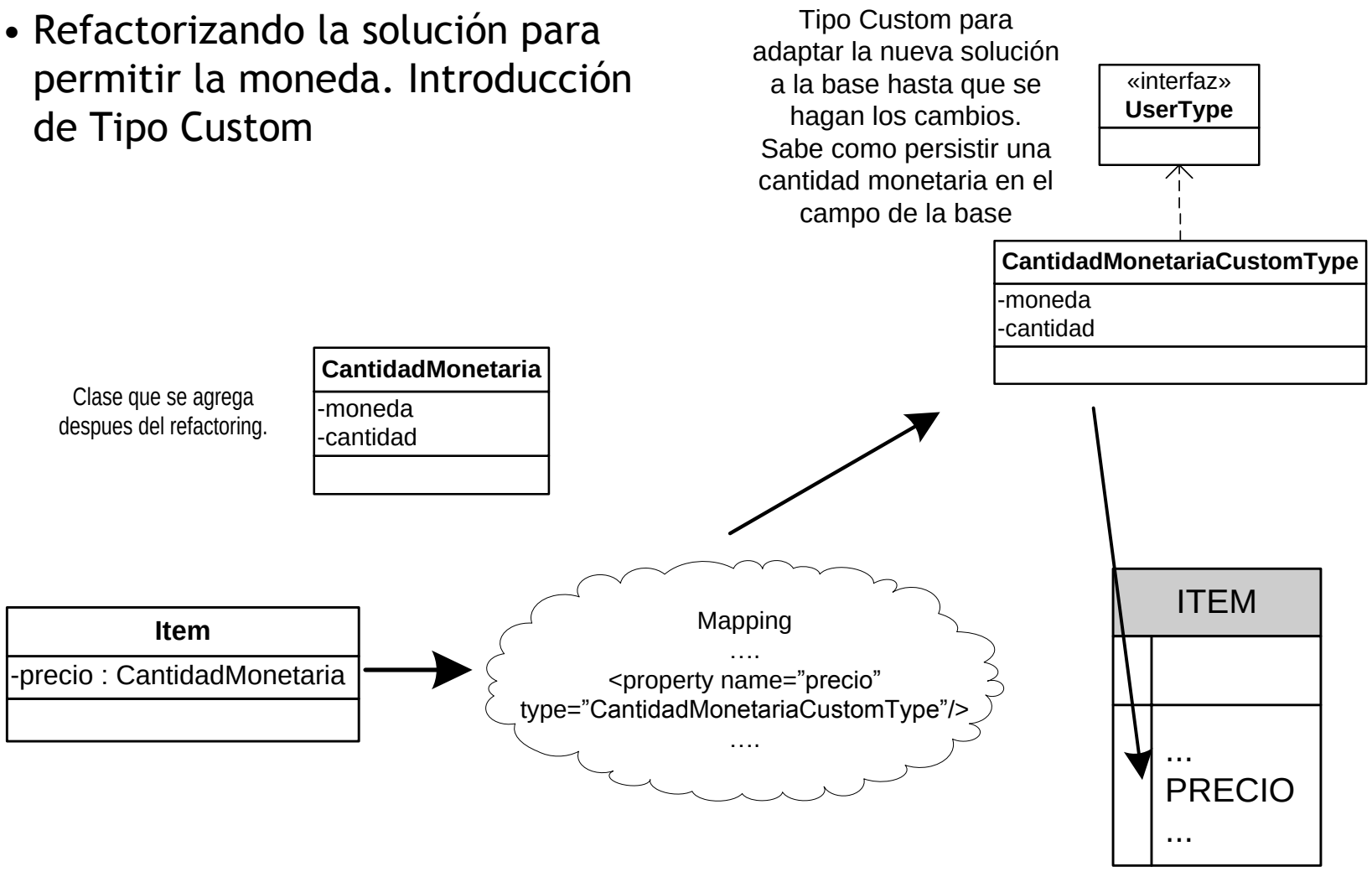
Custom Types - Ejemplo

- Con tipo predefinido

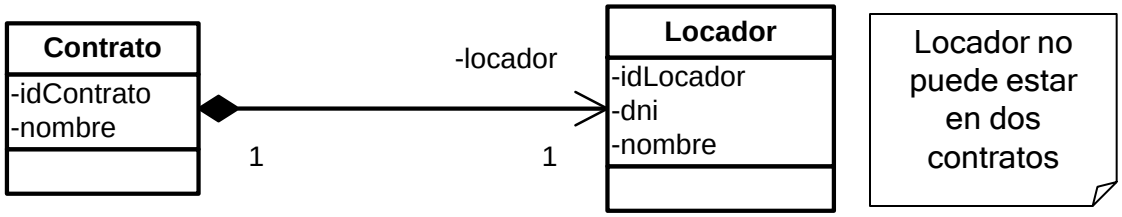


Custom Types - Ejemplo

- Refactorizando la solución para permitir la moneda. Introducción de Tipo Custom



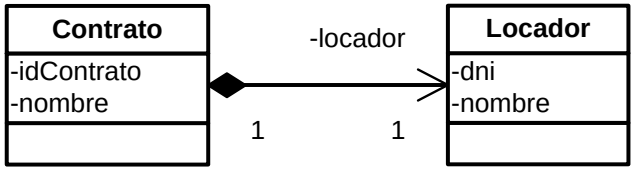
Componentes



- La agregación es una asociación fuerte entre dos clases, el “todo” y la “parte”, esto define una semántica adicional sobre los objetos y su ciclo de vida.
- En el ejemplo: Locador “es una parte” de un Contrato y no tiene sentido sin el Contrato (el “todo”).

- Hibernate tiene dos formas de representar estas agregaciones, una de ellas es definiendo a la “parte” como un componente del “todo”.
- En nuestro caso Locador como componente de Contrato.

Componentes



- Definiendo una clase como componente de otra, le indica a Hibernate que se persistirá en la misma tabla.
- Un componente no define una clave primaria.

```
<hibernate-mapping package="prueba">
<class name="Contrato">

  <id name="idContrato" column="ID_CONTRATO">
        <generator class="native"/>
  </id>
  <property name="nombre" not-null="true"/>

  <component name="locador" class="Locador">
      <property     name="dni"
                    column="DNI"
                    not-null="true" />
      <property     name="nombre"
                    column="NOMBRE_LOCADOR"
                    not-null="true"/>
  </component>


</class>
</hibernate-mapping>
```

CONTRATO	
PK	ID_CONTRATO
	NOMBRE DNI NOMBRE_LOCADOR

Colecciones

- Hibernate permite mapear distintos tipos de colecciones:
  - Colecciones indexadas
    - List → `<list>`
    - Map → `<map>`
  - Colecciones no indexadas:
    - Set → `<set>`: no permite repetidos
    - List → `<bag>` (una lista en la que el orden no se mantiene)
      → `<idbag>`
  - Colecciones Ordenadas (Sortered)
    - SortedSet → `<set order="orden"/>`
    - SortedMap → `<map order="orden"/>`

Colecciones Indexadas

- Todos los mapping de colecciones, salvo aquellos con la semántica del `Set` y `Bag`, necesitan una columna índice en la tabla de la colección.

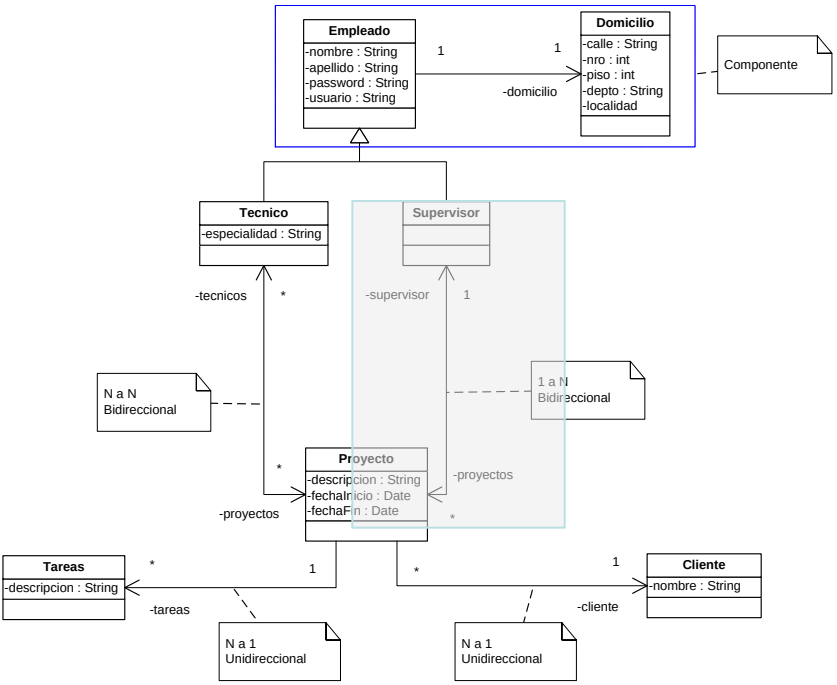
- ¿Qué es la columna índice?
  - El índice de un arreglo.
  - El índice de una lista (List).
  - Las claves de una Map (Map key).

- El índice de un arreglo o una lista
  - Siempre es de tipo integer → `<list-index>`

- El índice de una Map puede ser:
  - Cualquier tipo básico → `<map-key>`
  - Un componente → `<composite-map-key>`
  - Una referencia a una entidad → `<map-key-many-to-many>`

Asociaciones - Uno a muchos <one-to-many>

- Relaciona las tablas de dos clases a través de una clave foránea sin la intervención de una tabla intermedia.
- Se utiliza dentro de una colección.




```
<one-to-many
class="ClassName"
not-found="ignore|exception"
entity-name="EntityName"
node="element-name"
embed-xml="true|false"
/>
```

```
<set name="projects"
        inverse="true"
        cascade="all">
        <key column="oid_emp" />
        <one-to-many class="Project" />
</set>
```

<one-to-many>

class (requerido):

- El nombre de la clase asociada

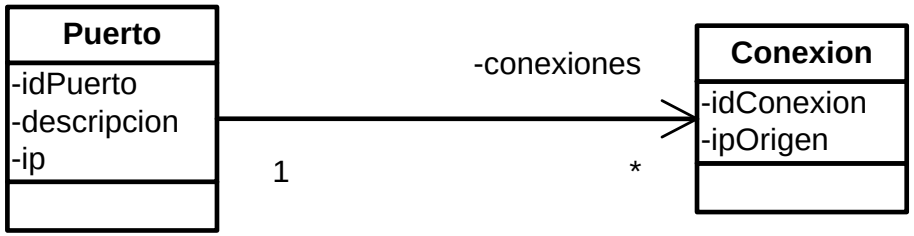
not-found (opcional - default → exception):

- Especifica cómo se manejan los identificadores que referencian a filas inexistentes; ignore tratará las filas inexistentes como asociaciones nulas.

- Se lleva la clave primaria de la clase que contiene el `<one-to-many>` a la clase indicada en class.

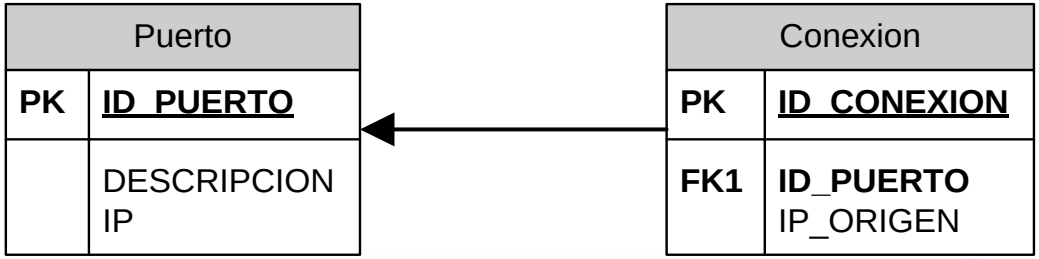
- `<one-to-many>` no necesita declarar ninguna columna, tampoco es necesario especificar ninguna tabla.

Unidireccionales: Uno a Muchos (Mapping)




```
<hibernate-mapping
  package="com.lifia.hibernate.model.uni">

<class name="Puerto">

    <id name="idPuerto"column="ID_PUERTO">
        <generator class="native"/>
    </id>

    <set name="conexiones" cascade="all">
        <key column="ID_PUERTO"
             not-null="true"/>
        <one-to-many class="Conexion"/>
    </set>

  <property name="descripcion" />
  <property name="ip" />

</class>
</hibernate-mapping>
```

```
<hibernate-mapping
  package="com.lifia.hibernate.model.uni">

<class name="Conexion">

<id name="idConexion" column="ID_CONEXION">
    <generator class="native"/>
</id>

<property name="ipOrigen" />

</class>
</hibernate-mapping>
```



Asociaciones – Muchos a uno <many-to-one>

- Este elemento permite asociar dos clases persistentes.
- Crea una referencia de integridad entre dos tablas.

```
<many-to-one
        name="propertyName"
        column="column_name"
        class="ClassName"
        cascade="cascade_style"
        property-ref="propertyNameFromAssociatedClass"
        unique="true|false"
        not-null="true|false"
        lazy="proxy|no-proxy|false"
        not-found="ignore|exception"  … />
```

name (obligatorio):

- El nombre de la propiedad .

column:

- El nombre de la columna de la clave foránea.

Asociaciones – Muchos a uno <many-to-one> (cont.)

class (default → el tipo de la propiedad determinado por reflexión):

- El nombre de la clase asociada.

cascade:

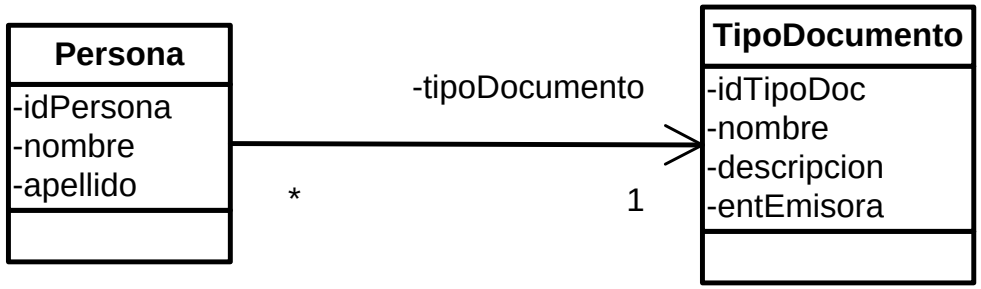
- Especifica cuales operaciones deben ser propagadas desde el objeto padre al objeto asociado.

unique:

- Si se especifica la relación de transforma en uno a uno (por defecto tiene valor false).

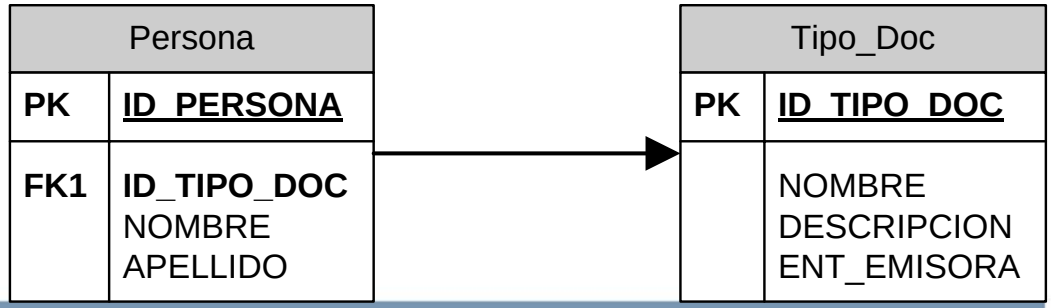
- Se puede usar o no en una colección.

Unidireccionales: Muchos a Uno (Mapping)

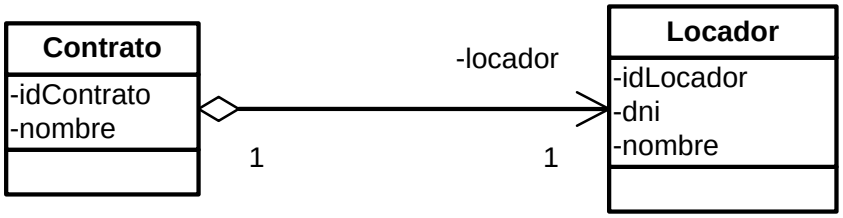



```
<hibernate-mapping package="model.uni">

<class name="Persona">

  <id name="idPersona"column="ID_PERSONA">
          <generator class="native"/>
  </id>

  <many-to-one name="tipoDocumento"
               column="ID_TIPO_DOC"
               not-null="true"
               cascade="all"/>

  <property name="nombre"/>
  <property name="apellido"/>

</class>
</hibernate-mapping>
```

```
<hibernate-mapping package="model.uni">
<class name="TipoDocumento"
  table="Tipo_Doc">

  <id     name="idTipoDoc"
          column="ID_TIPO_DOC">
      <generator class="native"/>
  </id>
  <property name="nombre" />
  <property name="descripcion" />
  <property name="entEmisora"
  column="ENT_EMISORA"/>
</class>
</hibernate-mapping>
```

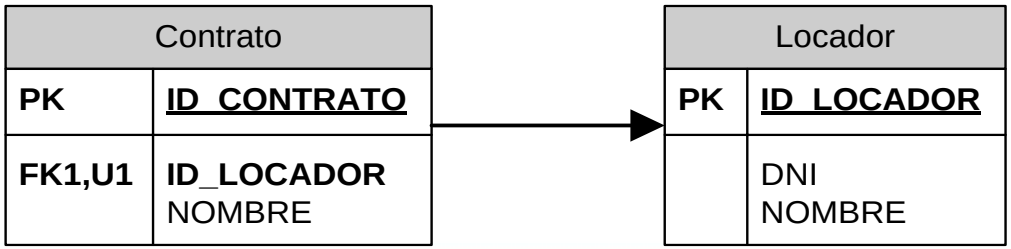


Unidireccionales: Uno a Uno (Mapping)




```
<hibernate-mapping
  package="com.lifia.hibernate.model.uni">

<class name="Contrato" ">

  <id name="idContrato"column="ID_CONTRATO">
          <generator class="native"/>
  </id>

  <many-to-one name="locador"
               column="ID_LOCADOR"
               unique="true"
               not-null="true"
               cascade="all" />

  <property name="nombre" column="NOMBRE"/>

</class>
</hibernate-mapping>
```

```
<hibernate-mapping
  package="com.lifia.hibernate.model.uni">

<class name="Locador">

  <id name="idLocador" column="ID_LOCADOR">
          <generator class="native"/>
  </id>

  <property name="dni" column="DNI" />
  <property name="nombre" column="NOMBRE"/>


</class>
</hibernate-mapping>
```



Asociaciones – Muchos a Muchos <many-to-many>

- Se mapean con el elemento `<many-to-many>`.
- Llevan una tabla dedicada.
- Se utiliza dentro de una colección

```
<many-to-many column="column_name"
              class="ClassName"
              unique="true|false"
              … />
```

class (obligatorio):

- El nombre de la clase asociada.

column:

- El nombre de la columna de la clave foránea del elemento.

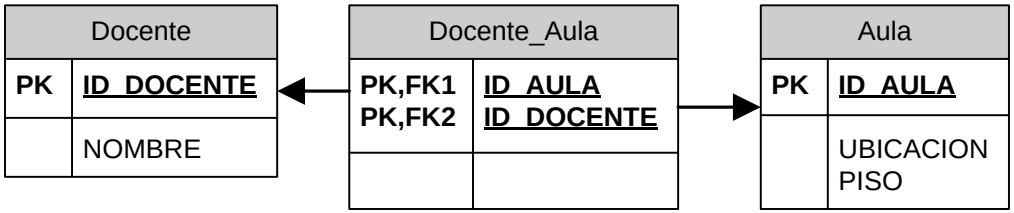
unique:

- Si se pone en “true” la asociación se transforma en uno a muchos.

Unidireccionales: Muchos a Muchos (Mapping)


```
<hibernate-mapping>
<class name="Docente" >

  <id name="idDocente"
  column="ID_DOCENTE">
          <generator class="native"/>
    </id>

    <set name="aulas"
         table="DOCENTE_AULA"
         cascade="all">
      <key column="ID_DOCENTE" />
      <many-to-many column="ID_AULA"
                    class="Aula" />
    </set>

  <property name="nombre" />
</class>
</hibernate-mapping>
```

```
<hibernate-mapping>
<class name="Aula" >

  <id name="idAula" column="ID_AULA" >
        <generator class="native"/>
    </id>

  <property name="ubicacion" />
  <property name="piso" />

</class>
</hibernate-mapping>
```

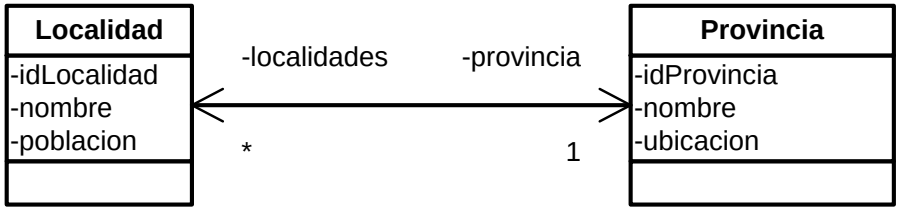


Asociaciones Bidireccionales

- Una *asociación bidireccional* permite la navegación desde ambos lados de la asociación. Se soportan dos tipos:
  - uno a muchos (one-to-many)
    - Conjuntos de valores (set o bag) en un lado y un valor simple en el otro.
  - Muchos a muchos (many-to-many)
    - Conjuntos de valores (set o bag) en ambos lados.

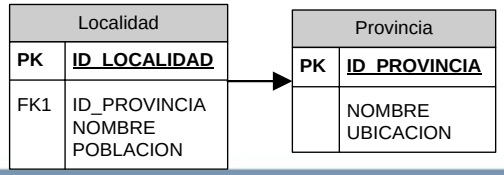
- ¿Cómo se declara?
  - Se especifica mapeando dos asociaciones a la misma tabla de la base y declarando una como `inverse`.

Bidireccionales: Muchos a uno (Mapping)

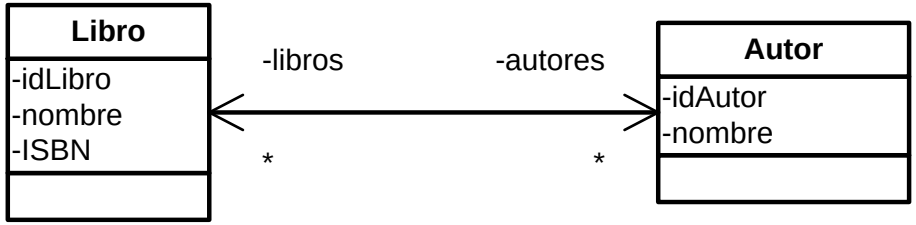



```
<hibernate-mapping package="model.bi">
<class name="Localidad" >

  <id name="idLocalidad"
      column="ID_LOCALIDAD">
      <generator class="native"/>
  </id>

  <many-to-one name="provincia"
               column="ID_PROVINCIA"
               not-null="true"
               cascade="all"/>

  <property name="nombre" />
  <property name="poblacion" />

</class>
</hibernate-mapping>
```

```
<hibernate-mapping package="model.bi">
<class name="Provincia" >

  <id name="idProvincia"
      column="ID_PROVINCIA">
        <generator class="native"/>
    </id>

  <set name="localidades"
       inverse="true" cascade="all">

      <key column="ID_PROVINCIA"/>
      <one-to-many class="Localidad"/>
  </set>

  <property name="nombre" />
  <property name="ubicacion" />

</class>
</hibernate-mapping>
```

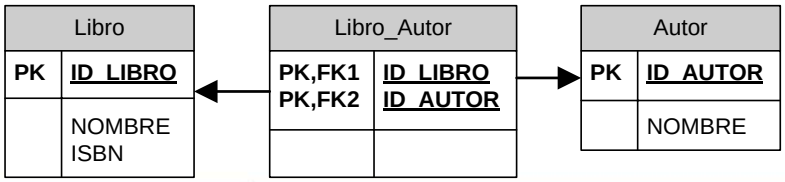


Bidireccionales: Muchos a Muchos (Mapping)

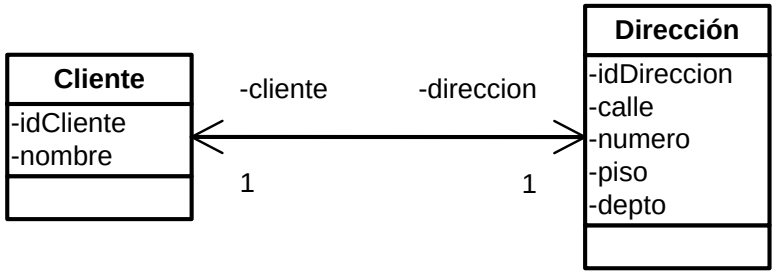



```
<hibernate-mapping package="model">
<class name="Libro">

  <id name="idLibro" column="ID_LIBRO">
        <generator class="native"/>
    </id>
  <set name="autores"
       table="LIBRO_AUTOR"
       cascade="all">

     <key column="ID_LIBRO" />
     <many-to-many column="ID_AUTOR"
                   class="Autor"/>
  </set>

</class>
</hibernate-mapping>
```

```
<hibernate-mapping package="model">
<class name="Autor">

  <id name="idAutor" column="ID_AUTOR">
        <generator class="native"/>
  </id>

  <set name="libros"
       table="LIBRO_AUTOR"
       inverse="true" >

     <key column="ID_AUTOR" />
     <many-to-many column="ID_LIBRO"
                   class="Libro"/>
  </set>

</class>
</hibernate-mapping>
```

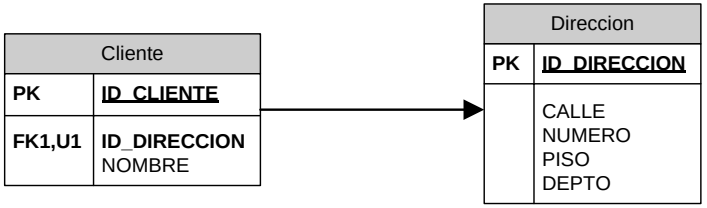


Bidireccionales: Uno a Uno (Mapping)




```
<hibernate-mapping >
<class name="Cliente">

 <id name="idCliente"
     column="ID_CLIENTE">
          <generator class="native"/>
  </id>
  <property name="nombre"/>

  <many-to-one name="direccion"
               column="ID_DIRECCION"
               unique="true"
               not-null="true"
               cascade="all" />
</class>
</hibernate-mapping>
```

```
<hibernate-mapping>
<class name="Direccion" >

  <id name="idDireccion"
      column="ID_DIRECCION">
          <generator class="native"/>
  </id>
  <property name="calle" />
  <property name="numero" />
   …
  <one-to-one name="cliente"
          property-ref="direccion" />

</class>
</hibernate-mapping>
```



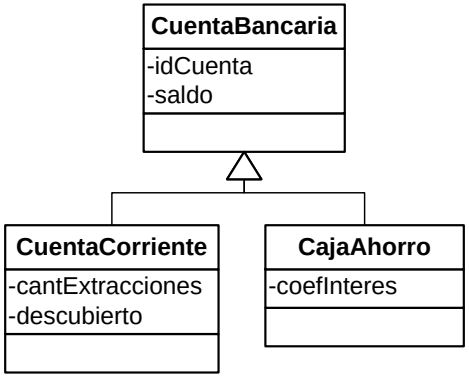
Resumen

		Clase A	Clase B
Unidireccionales	1 a 1	<many-to-one unique="true"/>	-
	1 a N	<one-to-many>	-
	N a 1	<many-to-one>	-
	N a N	<many-to-many>	-
Bidireccionales	1 a 1	<many-to-one unique="true"/>	<one-to-one>
	1 a N	<one-to-many>	<many-to-one>
	N a N	<many-to-many>	<many-to-many>

Herencia - Jerarquías - Mappings

- Hibernate soporta tres estrategias para mapear Jerarquías
  - Tabla por jerarquía.
  - Tabla por subclase.
  - Tabla por clase concreta.

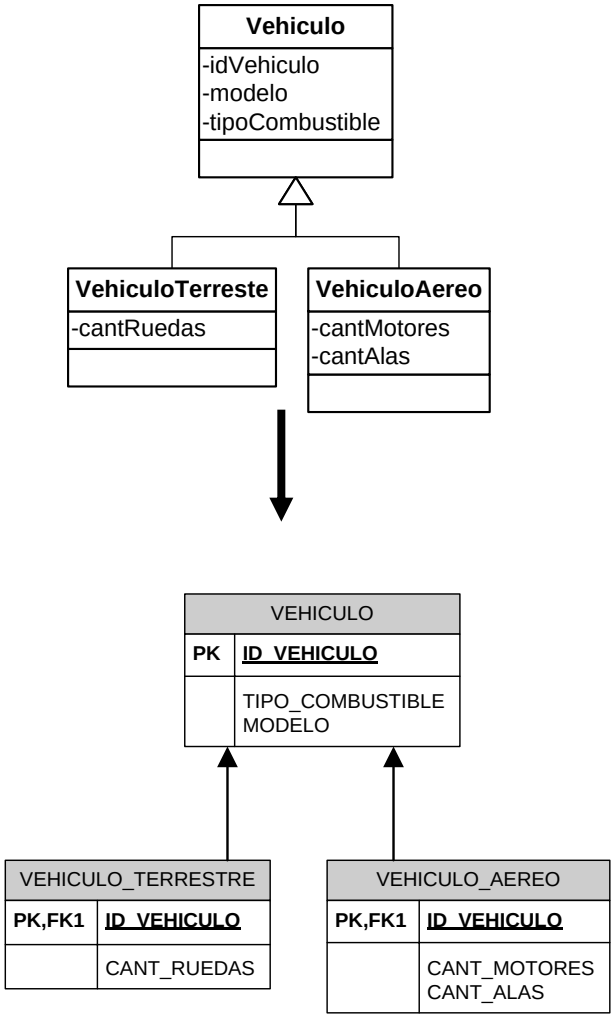
Tabla por Jerarquía



CUENTA_BANCARIA	
PK	**ID_CUENTA**
	TIPO_CUENTA SALDO CANT_EXTRACCIONES COEF_INTERES

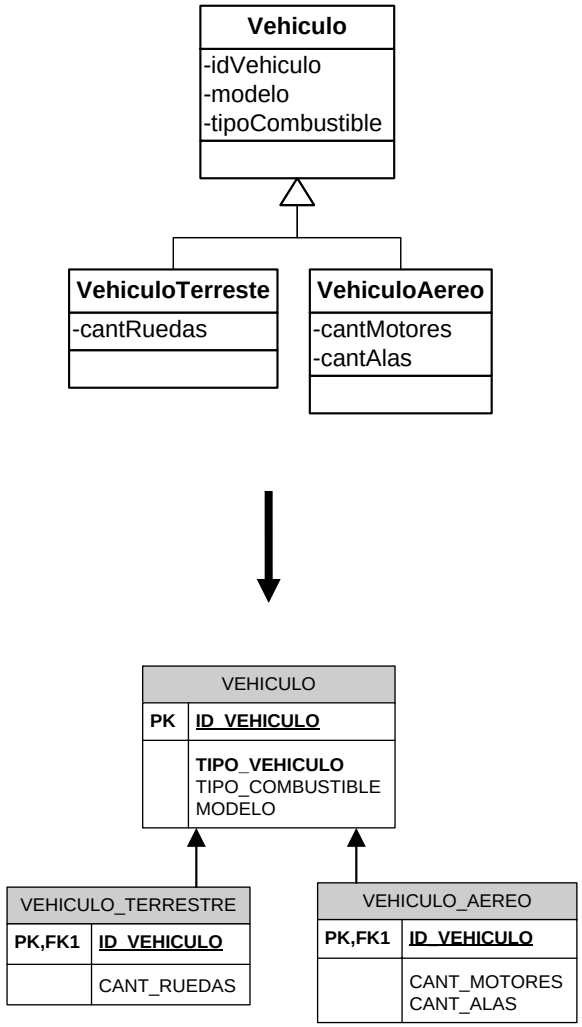
```
<hibernate-mapping
package="com.lifia.hibernate.model.hierarchy">

<class    name="CuentaBancaria"
          table="CUENTA_BANCARIA">
  <id name="idCuenta" column="ID_CUENTA">
        <generator class="native"/>
  </id>
  <discriminator column="TIPO_CUENTA"
                 type="string"/>
  <property name="saldo" column="SALDO"/>
  <subclass name="CuentaCorriente"
            discriminator-value="CC">
      <property name="cantExtracciones"
                column="CANT_EXTRACCIONES"/>
      <property name="descubierto"
                column="DESCUBIERTO"/>
  </subclass>
  <subclass name="CajaAhorro"
            discriminator-value="CA">
      <property name="coefInteres"
                column="COEF_INTERES"/>
  </subclass>
</class>
</hibernate-mapping>
```

Tabla por Subclase



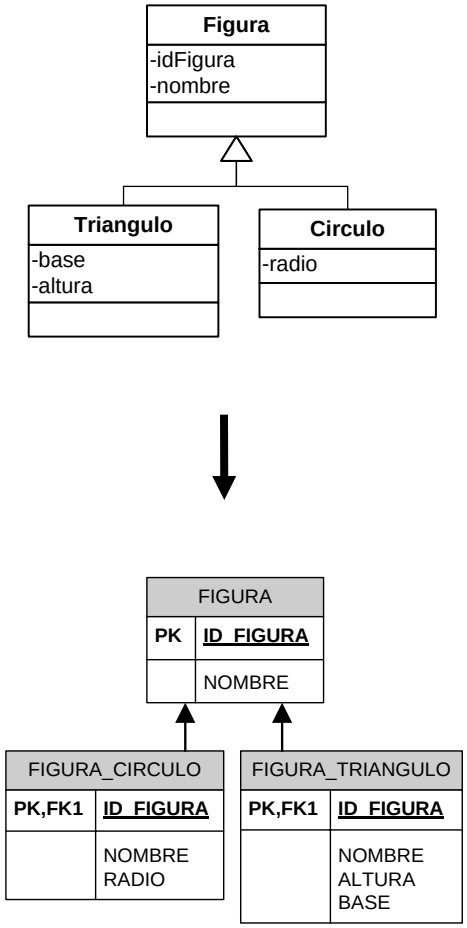

```
<hibernate-mapping
package="com.lifia.hibernate.model.hierarchy“>
<class name="Vehiculo" table="VEHICULO" >
   <id name="idVehiculo" column="ID_VEHICULO">
        <generator class="native"/>
   </id>
   <property name="modelo"/>
   <property name="tipoCombustible"
             column="TIPO_COMBUSTIBLE"/>
   <joined-subclass name="VehiculoTerrestre"
                    table="VEHICULO_TERRESTRE">
        <key column="ID_VEHICULO"/>
        <property name="cantRuedas"
                  column="CANT_RUEDAS"/>
   </joined-subclass>
   <joined-subclass name="VehiculoAereo"
                    table="VEHICULO_AEREO">
        <key column="ID_VEHICULO"/>
        <property name="cantMotores"
                  column="CANT_MOTORES"/>
        <property name="cantAlas"
                  column="CANT_ALAS"/>
    </joined-subclass>
</class>
</hibernate-mapping>
```

Tabla por Subclase con discriminante




```
<hibernate-mapping
package="com.lifia.hibernate.model.hierarchy">
<class name="Vehiculo" table="VEHICULO" >
   <id name="idVehiculo"column="ID_VEHICULO">
        <generator class="native"/>
   </id>
   <discriminator column="TIPO_VEHICULO‟ type="string"/>
   <property name="modelo"/>
   <property name="tipoCombustible"
             column="TIPO_COMBUSTIBLE"/>
   <subclass name="VehiculoTerrestre"
discriminator-value="VT">
          <join table="VEHICULO_TERRESTRE">
             <key column="ID_VEHICULO"/>
             <property name="cantRuedas"
column="CANT_RUEDAS"/>
          </join>
   </subclass>
   <subclass name="VehiculoAereo"
             discriminator-value="VA">
          <join table="VEHICULO_AEREO">
             <key column="ID_VEHICULO"/>
             <property name="cantMotores"
                       column="CANT_MOTORES"/>
             <property name="cantAlas‟ column="CANT_ALAS"/>
          </join>
   </subclass>
</class>
</hibernate-mapping>
```

Tabla por Clase Concreta




```
<hibernate-mapping
package="com.lifia.hibernate.model.hierarchy">

<class name="Figura" table="FIGURA">

    <id name="idFigura" column="ID_FIGURA">
        <generator class="hilo"/>
    </id>

    <property name="nombre"/>

    <union-subclass name="Circulo"
                    table="FIGURA_CIRCULO">
          <property name="radio"/>
    </union-subclass>

    <union-subclass name="Triangulo"
                    table="FIGURA_TRIANGULO">
          <property name="altura"/>
          <property name="base"/>
    </union-subclass>

</class>
</hibernate-mapping>
```

Lazy Initialization

- Cuando una colección se mapea usando `lazy="true"`, esto le indica a Hibernate que dicha colección no se leerá de la base hasta que sea accedida.
- En Java, en ejecución se reemplaza la colección por un proxy que representa la colección mientras la colección no se use.
- Cuando se accede a la misma se leen los datos de la base, se crea la colección y se reemplaza el proxy.
- Todo este proceso es totalmente transparente para el desarrollador OO (o casi).

Inicialización Lazy

- Ejemplo:

```
Session sess = sessionFactory.openSession();
Servidor servidor = (Servidor) sess.load(Servidor.class,id);
servidor.getClientes() // en este momento Hibernate carga los clientes
sess.close();

Session sess = sessionFactory.openSession();
Servidor servidor = (Servidor) sess.load(Servidor.class,id);
sess.close();
servidor.getClientes() // Se levanta una excepción porque los clientes no
  están, sino que hay un proxy en vez de la colección y no estoy en una
  sesión abierta.
```

Recuperar un objeto

Métodos para Recuperar:

- session.load()
  - Recupera el objeto con un identificador, sino levanta una excepción

```
Persona persona = (Persona)
session.load(Persona.Class,identificador);
```

- session.get()
  - Recupera el objeto con un identificador, si no existe retorna null

```
Persona persona = (Persona)
session.get(Persona.Class,identificador);
```

Borrar Objetos

- Se utiliza el método `delete()` de la `Session`
  - Borra el objeto de la base.

- Ejemplo

```
// la session se obtuvo previamente
Persona persona = (Persona)
          session.load(Persona.class, idPersona);

session.delete(persona);
```

Control optimista de concurrencia

- Cuando se tienen ambientes de alta concurrencia y escalabilidad, se utiliza control de concurrencia optimista y versionamiento.

- Chequeo de versiones para detectar conflictos de actualización, se usa:
  - Números de versión.
  - Timestamps.

- Hibernate posee tres acercamientos:
  - Control de versiones por aplicación.
  - Sesión extendida y versionamiento automático.
  - Objetos desacoplados y versionamiento automático.

Control de versiones por aplicación

- Hibernate casi no interviene, se encarga el programador de hacer el control de versiones

```
// foo es una instancia cargada en una Session previa
session = factory.openSession();
Transaction t = session.beginTransaction();
int oldVersion = foo.getVersion();
// carga el estado actual
session.load( foo, foo.getKey() );
if ( oldVersion!=foo.getVersion() )
        throw new StaleObjectStateException();
foo.setProperty("bar");
t.commit();
session.close();
```

- Sólo sirve para casos muy triviales, cuando hay grafos complicados no sirve.
- La propiedad versión se utiliza usando el tag `<version>` y hibernate se encarga de actualizarlo durante un `flush()`

Versionamiento

- El elemento `<version>` es opcional e indica que la tabla contiene datos versionados.
- Es útil si se planea usar transacciones largas.

```
<version column="version_column"
         name="propertyName"
         type="typename"
         access="field|property|ClassName"
         unsaved-value="null|negative|undefined"
         generated="never|always"
         insert="true|false"
/>
```

column (opcional – default → valor de la propiedad):

- El nombre de la columna que contiene el número de versión.

name:

- El nombre de la propiedad persistente.

type (optional – default → integer):

- El tipo del número de versión.

Queries

- Hibernate tiene varias formas de consultar a la base.

- Existen tres formas de escribir un query:
  - En HQL (Hibernate Query Language)
    - `session.createQuery( "from Persona p where p.nombre like "%Pablo%");`
  - Con un Criteria (consultas dinámicas)
    - `session.createCriteria(Persona.class) .add(Expression.like("nombre","%Pablo%");`
  - Con SQL
    - `session.createSQLQuery( "select {p.*} from PERSONA {p} where nombre like "%Pablo%", p, Persona.class)}`

Interfaces Query y Criteria

- Las interfaces `Query` y `Criteria` definen varios métodos para controlar la ejecución de una consulta.
- `Query` provee métodos para hacer binding entre parámetros concretos y los parámetros de la consulta.
- Para ejecutar una `Query` es necesario obtener una instancia de estas interfaces usando `Session`.
- `Criteria` se utiliza para generar consultas dinámicas.

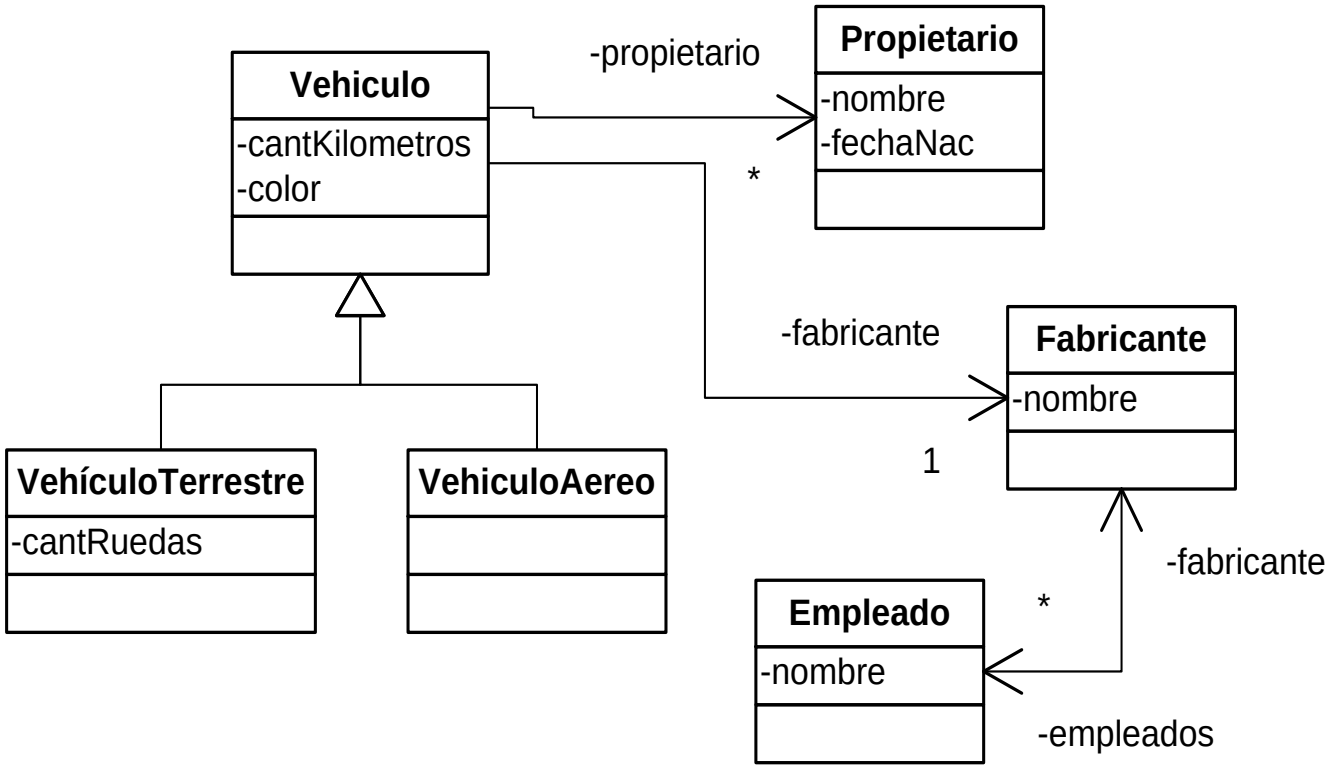
Instanciando Query

- Para crear una nueva instancia de `Query`, se debe invocar a:
  - `createQuery()`: prepara una consulta con HQL
    - `Query hqlQuery = session.createQuery("from Persona");`
  - `createSQLQuery()`: crea una consulta SQL usando la sintaxis de la base subyacente
    - `Query sqlQuery = session.createSQLQuery("select {p.*} from PERSONA {p}", "p", Persona,class}`

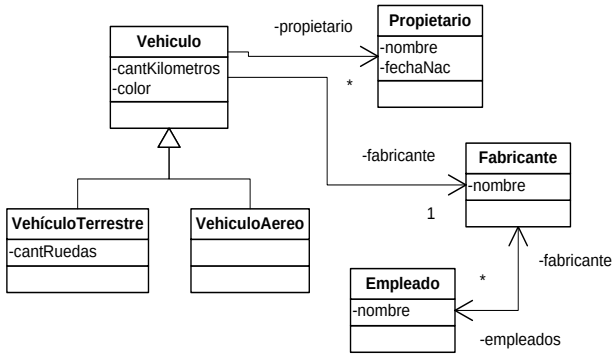
HQL – Hibernate Query Language

- Lenguaje de consultas:
  - from
    - “from Persona p”; selecciona las instancias de Persona, p alias
  - where
  - order by
  - group by
  - Joins
    - inner join
    - left outer join
    - right outer join
    - full join (no es muy útil)
  - Funciones
    - avg()
    - sum()
    - min()
    - max()
    - count()

Ejemplos


Vehiculo
-cantKilometros
-color
-propietario
*
Propietario
-nombre
-fechaNac
-fabricante
1
Fabricante
-nombre
VehículoTerrestre
-cantRuedas
VehiculoAereo
Empleado
-nombre
*
-fabricante
-empleados

Consultas



- Listar todos los vehículos

```
from Vehiculo v
```

- Listar todos los vehículos rojos

```
from Vehiculo v
where v.color like “rojo”
```

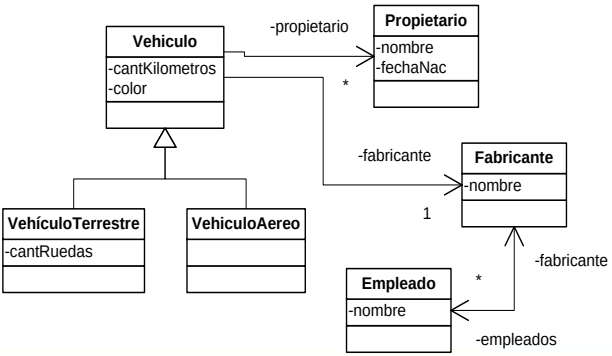
- Listar todos los vehículos del fabricante con nombre honda

```
select v
from vehiculo v
     join v.fabricante as f
where f.nombre like "honda"
```

Consultas

- Listar todos los vehículos rojos de más de 20000 kilómetros que pertenecen al fabricante honda y cuyo propietario se llama Juan.

```
select v
from   Vehiculo as v
       join v.propietario as p
       join v.fabricante as f
where v.color like "rojo" and
       v.cantKilometros > 20000 and
       f.nombre like "honda" and
       p.nombre like "juan"
```



Consultas

- Listar todos los vehículos terrestres rojos cuyo kilometraje es mayor al promedio.

```
select v
from VehiculoTerrestre as v
where
        v.color like “rojo”
        v.cantKilometros > (
                select avg(v1.cantKilometros)
                from Vehiculo as v1 )
```

Externalización de consultas

- Hibernate permite guardar consultas en un archivo xml y recuperarlas por un nombre.
- Esto se conoce como consultas nombradas
  - Ejemplo:

```
session.getNamedQuery("buscarPersonasPorNombre")
       .setString("nombre", nombre)
       .list();
```

  - Donde "buscarPersonasPorNombre" está en Persona.hbm.xml usando el elemento <query>

```
<query name="buscarPersonasPorNombre">
        <![CDATA[from Persona persona
        where persona.nombre like :nombre
        ]]>
</query>
```

No necesariamente tiene que ser HQL, puede ser SQL.

Referencias

- Mapping objects to relational databases
  - Scott Ambler
  - http://www.AmbySoft.com/mappingObjects.pdf

- www.hibernate.org

- HSQLDB
  - www.hsqldb.org