Persistència en BDR-BDOR-BDOO

Josep Cañellas Bornas

Accés a dades

2. Eines de mapatge objecte-relacional (ORM)

Tot i que els sistemes de connexió a les bases de dades relacionals com ODBC o JDBC permeten una gran expressivitat, el que les converteix en eines molt potentssón eines de baix nivell que malauradament necessiten un nombre important de línies de codi per poder cobrir les necessitats de cada aplicació.

Els desenvolupadors han de continuar esmerçant-se en dissenyar dos models (relacional i orientat a objectes) i han de continuar creant eines de traducció entre ells, amb un cost realment important.

ORM són les sigles en anglès de Object-Relational Mapping.

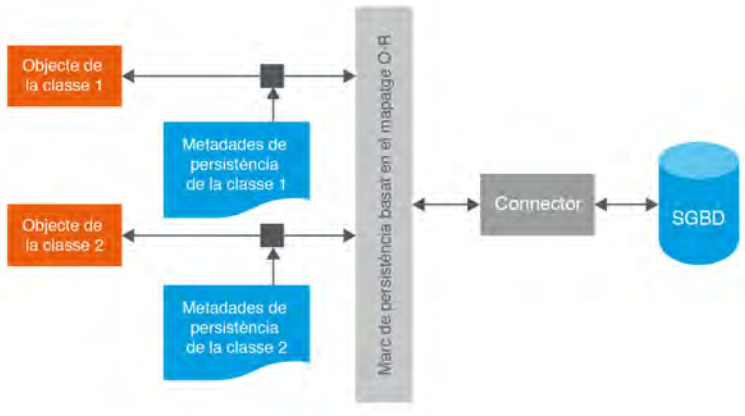
Les eines de mapatge objecte-relacional (ORM) intenten aprofitar la maduresa i l'eficiència de les bases de dades relacionals minimitzant tant com sigui possible el desfasament objecte-relacional. Es tracta de biblioteques i marcs de programació que defineixen un format per expressar múltiples situacions de transformació entre ambdós paradigmes. Això els permet automatitzar processos de traspàs d'un sistema a l'altre.

En certa forma podríem dir que implementen una base de dades orientada a objectes virtual perquè aporten característiques pròpies del paradigma OO, però el substrat on s'acaben emmagatzemant els objectes és un SGBD relacional.

2.1 Concepte de mapatge (objecte-relacional)

Com ja hem dit, les eines de mapatge objecte-relacional (O-R) automatitzen els processos necessaris d'intercanvi de dades entre sistemes OO i sistemes relacionals (figura 2.1).

Figura 2.1. Esquema que representa un marc de persistència basat en el mapatge O-R



L'automatització s'aconsegueix gràcies a un conjunt de metadades que descriuen quin procés cal utilitzar i quina correspondència hi ha entre les dades primitives d'ambdós sistemes i les estructures que les suporten.

Segurament la descripció més senzilla que han de suportar les metadades és la d'establir una correspondència directa entre classes i taules, i en darrer terme entre els atributs de tipus primitius i els camps o columnes. També caldrà identificar l'atribut corresponent al camp que actuarà de clau primària.

Malauradament, no sempre serà adequat establir aquest tipus de correspondències directes i caldrà que les metadades puguin expressar molta més complexitat.

A vegades pot interessar emmagatzemar un atribut en més d'una columna o diversos atributs en una columna única. Hi pot haver atributs que no s'emmagatzemin i camps emmagatzemats que no es reflecteixin en els objectes. Quan els atributs no siguin tipus de dades primitives caldrà saber també si haurem d'emmagatzemar les dades en una taula diferent o en la mateixa taula i, en cas d'emmagatzemar-les en taules diferents, quins atributs haurem de fer servir com a claus foranes, qui tindrà la responsabilitat de realitzar l'emmagatzematge, etc.

2.2 Eines de mapatge

El nombre d'eines de mapatge que podem trobar actualment és realment elevat. Existeixen eines de mapatge per a la majoria de llenguatges orientats a objectes PHP, Objective-C, C++, SmallTalk i evidentment Java.

Bàsicament són eines en què podem distingir tres aspectes conceptuals clarament diferenciats: un sistema per expressar el mapatge entre les classes i l'esquema de la base de dades, un llenguatge de consulta orientat a objectes per tal de neutralitzar el desfasament O-R i un nucli funcional que possibilita la sincronització dels objectes persistents de l'aplicació amb la base de dades.

2.2.1 Tècniques de mapatge

Entre les tècniques que aquestes eines fan servir per plasmar els mapes O-R distingirem les que incrusten les definicions dins el codi de les classes i les que emmagatzemen les definicions en fitxers independents. Les primeres solen ser tècniques molt vinculades al llenguatge de programació, així per exemple, en C++ se solen fer servir macros i en Java s'utilitzen anotacions. Les tècniques de definició basades en fitxers independents del codi solen sustentar-se en XML, perquè és un llenguatge molt expressiu i fàcilment extensible.

Normalment la majoria d'eines solen acceptar ambdues tècniques de mapatge i fins i tot permeten la convivència dins una mateixa aplicació. Les tècniques que usen el propi llenguatge de programació per incrustar el mapatge dins el codi

presenten una corba d'aprenentatge més baixa per desenvolupadors experimentats en el llenguatge amfitrió, però per contra no serà possible aprofitar les definicions si es decideix canviar de llenguatge de programació.

En canvi, fent servir les tècniques basades en XML sí que es poden reutilitzar les definicions per a diferents llenguatges, sempre que l'eina utilitzada estigui disponible en el llenguatge de programació requerit o bé si el format de les definicions segueix alguna especificació estàndard.

Diverses alternatives han intentat fer-se un lloc en el món dels estàndards, però a hores d'ara cap d'elles presenta un clar avantatge respecte les altres. Val a dir que la majoria d'aquests estàndards comparteixen força sintaxi quant a expressar les definicions del mapatge, però sempre hi ha diferències que no els fan del tot compatibles.

2.2.2 Llenguatge de consulta

ODMG

ODMG són les sigles d'Object Database Management Group, un consorci d'empreses amb l'objectiu de crear un estàndard per a la manipulació i creació d'objectes persistents, és a dir, emmagatzemats en una base de dades.

El llenguatge de consulta més utilitzat per la majoria d'eines és el llenguatge anomenat OQL (Object Query Language) o una variant del mateix. Es tracta d'un llenguatge especificat per l'ODMG. Presenta certa similitud amb SQL, atès que ambdós són llenguatges d'interrogació no procedimental, però l'OQL està totalment orientat a objectes. És a dir, els components de la consulta s'expressen fent servir la sintaxi pròpia dels objectes i els resultats obtinguts retornen objectes o col·leccions d'objectes.

Es tracta del llenguatge d'interrogació que també usen moltes bases de dades orientades a objecte, la qual cosa el fa un dels estàndards més populars i coneguts. Hi ha altres llenguatges de consulta orientats a objectes, però no tan estesos com OQL.

2.2.3 Tècniques de sincronització

La sincronització amb la base de dades és segurament un dels aspectes més crítics de les eines de mapatge. Solen ser processos força complexos, on trobem implicades sofisticades tècniques de programació destinades a descobrir els canvis que vagin patint els objectes, a crear i inicialitzar les noves instàncies que calgui posar en joc dins l'aplicació d'acord amb les dades emmagatzemades o també a extreure la informació dels objectes per revertir-la a les taules del'SGBD.

Aquest és probablement l'aspecte que més diferencia les eines entre si. Les principals tècniques emprades són la precompilació, la postcompilació i la reflexió. El llenguatge de programació suportat per l'eina influenciarà en gran mesura les solucions adoptades per resoldre la sincronització. Per exemple, C++ admet precompilació, però no pas reflexió, mentre que Java admet postcompilació i reflexió.

JDO és un estàndard de persistència per a Java desenvolupat per Apache. El projecte inclou, a més de l'especificació de l'estàndard, el desenvolupament d'un marc de persistència basat en la postcompilació. Aquesta tècnica consisteix en realitzar dues compilacions a les classes que requereixin persistència. La primera, realitzada pel jdk, generarà els *bytecode* estàndards. La segona compilació, fent servir les indicacions descrites en els documents de mapatge, modificarà els *bytecode* generats i hi afegirà nova funcionalitat necessària per dur a terme la persistència d'una forma transparent.

JDBC és també un estàndard de persistència. Es troba incorporat en el JDK des de la versió 5. Hi ha diverses biblioteques que el suporten, totes elles basades en el principi de la reflexió. Java és un llenguatge que en compilar les classes incorpora metadades amb informació pròpia de la classe, com ara l'estructura de dades interna, els mètodes que tingui disponibles, els paràmetres necessaris per invocar els mètodes, etc.

La màquina virtual permet fer servir aquestes metadades per accedir a la informació real emmagatzemada en els objectes, per invocar algun dels seus mètodes, per construir noves instàncies, etc.

La postcompilació, usada per JDO, presenta l'avantatge de ser més eficient degut al fet que incrusta codi compilat directament allà on cal, mentre que la tècnica de la reflexió ha d'anar llegint les metadades i interpretant la manera d'executar adequadament les ordres desitjades.

Malauradament, JDO no ha acabat de quallar i poques iniciatives comercials l'han seguit. Per contra JPA, que ha abraçat gran part de la tecnologia de dos gegants fortament implantats (Hibertante i EJB), ha estat rebut de forma molt més receptiva i ara per ara tot sembla apuntar que serà l'escollit per cobrir l'estandardització de la persistència en Java.

2.3 JPA (Java Persistence API)

Hem escollit JPA per estudiar en detall les eines de mapatge per diversos motius. En primer lloc, és l'estàndard incorporat a la màquina virtual. En segon lloc, forma part d'una especificació més gran i completa anomenada EJB3 (Enterprise Java Beans 3) que permet crear aplicacions distribuïdes realment complexes. D'altra banda, hi ha un elevat nombre d'eines que suporten aquest estàndard. Aquí farem servir el marc de persistència anomenat Hibernate.

2.3.1 Instal·lació de l'estàndard JPA implementat per Hibernate

JPA és una estàndard inclòs en en les darreres versions (5 en endavant). Es tracta d'una implementació parcial constituïda bàsicament per les interfícies que

garantiran la interoperabilitat de múltiples solucions completes.

És a dir, a més de la implementació inclosa en el JDK, necessitarem afegir una biblioteca amb les classes necessàries per completar la implementació. Podeu obtenir-ne una a l'adreça goo.gl/xRof5. Seleccioneu el fitxer comprimit hibernate-release-4.1.1.Final.zip o hibernate-release-4.1.1.Final.tgz, segons tingueu Windows o Linux, i descomprimiu el contingut en una carpeta que pugueu localitzar fàcilment. El contingut de la biblioteca que us acabeu de baixar està repartit en un nombre elevat de fitxers amb extensió *jar*, els quals hauríem d'afegir un a un en cada projecte en què precisséssim fer servir JPA.

NetBeans, però, facilita la creació de biblioteques enteses com a agrupacions de fitxers .jar. Cada biblioteca s'identifica per mitjà d'un nom, de manera que serà possible assignar el conjunt de fitxers en un sola acció.

2.3.2 Creació de la biblioteca JPA2 en el Netbeans

La major part de versions del NetBeans solen contenir, per defecte, una biblioteca Hibernateantiga corresponent a la primera versió del JPA. En aquesta unitat estudiarem la versió JPA2. Per això és necessari realitzar aquesta instal·lació.

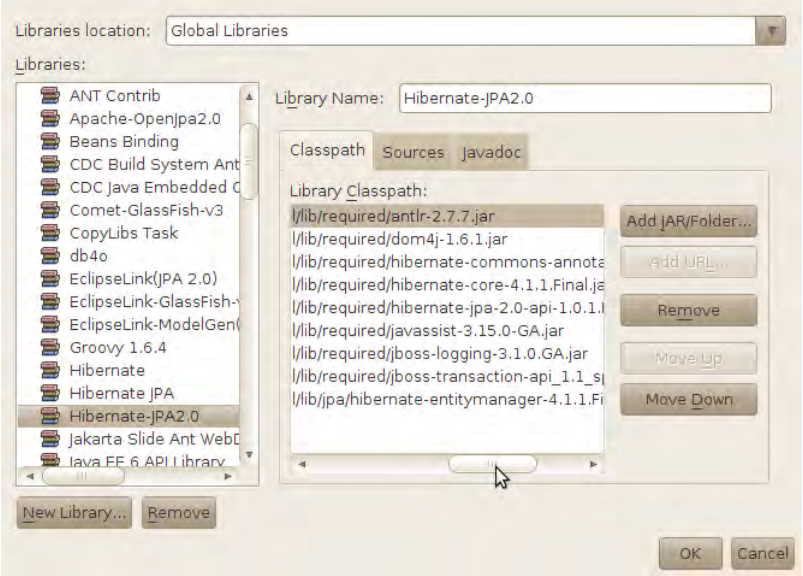
Per crear la biblioteca necessitarem seleccionar els fitxers .jar d'Hibernatei, per tant, és necessari que els tingueu descomprimits abans de continuar.

Obriu el NetBeans i seleccioneu l'opció *Libraries* del menú *Tools*. (*Tools–>Libraries*). Això us obrirà el quadre de diàleg del gestor de biblioteques (*Library Manager*). Cliqueu el botó *New Library*. Us demanarà el nom i el nom de la biblioteca i el tipus. És important posar un nom significatiu o descriptiu per recordar en futures ocasions quin és el seu contingut. Us proposem Hibernate-JPA2. En el tipus de biblioteca cal escollir *Class Library*. Premeu *OK*. Se us crearà la biblioteca *Hibernate-JPA2* buida. Seguidament cliqueu el botó *Add JAR/Folder*. Cal que navegueu fins a la carpeta on hagueu desplegat els arxius `.jar` d'Hibernate. Entreu a la carpeta *lib* i dins d'aquesta, la carpeta *required*. Seleccioneu tots els fitxers `.jar` de la carpeta (en total vuit fitxers) i cliqueu *Add Jar/Folder*. Repetiu l'operació per al `.jar` que que es troba a la carpeta *JPA* dins la carpeta *LIB* de *Hibernate*.

Al final hauríeu de tenir nou fitxers `.jar` agrupats en la biblioteca que esteu creant.

Un cop creada la biblioteca podrem afegir-la a qualsevol dels nostres projectes de la mateixa manera com si fos una biblioteca pròpia de NetBeans: des de la finestra de propietats o bé fent ús del menú de context, clicant amb el botó dret damunt l'etiqueta *Libraries* del projecte (figura 2.2).

FIGURA 2.2. Library Manager mostra tots els fitxers JAR que cal afegir a la biblioteca de Hibernate-JPA2 que esteu creant



2.3.3 Característiques generals del JPA

Abans de descriure els elements de JPA que ens permetin implementar aplicacions voldríem exposar algunes de les característiques més rellevants del JPA.

Dóna suport a la persistència de qualsevol tipus d'objecte. Només es requereix que els objectes persistents siguin seriables i, per tant, implementin la interfície `Serializable`.

El mapatge de les classes es pot configurar fent servir anotacions de Java o fitxers XML. Aquí veurem les dues formes. De fet, JPA pot fer servir ambdues configuracions al mateix temps. En cas de conflicte, JPA dóna prioritat a les definicions XML perquè es considera que les anotacions se solen fer servir durant la fase de desenvolupament i els fitxers XML durant les fases de manteniment i modificació de les aplicacions. Cal tenir en compte que les anotacions requereixen compilar el codi, mentre que les especificacions XML no.

JPA està basat en la tècnica de reflexió. Això li permet conèixer el nom i l'estructura de les classes, els noms amb què s'identifiquen els seus elements, etc. JPA utilitza aquesta informació com a metadades per defecte a l'hora de generar les bases de dades igual com si es tractés de les metadades descrites en el *mapatge*. JPA pot actuar per defecte en una gran majoria de casos, de manera que la configuració necessària per dur a terme el *mapatge* serà la mínima possible.

És el que es coneix com a *configuració per excepció*. És a dir, només cal plasmar el mapatge en cas que les opcions per defecte no coincideixin amb els resultats desitjats. Òbviament, aquest sistema pot estalviar molta feina als desenvolupadors.

2.4 Peces bàsiques del JPA

Abans d'entrar amb més detall sobre la implementació d'aplicacions basades en JPA, descriurem els conceptes més bàsics sobre els quals se sustenta JPA per tal d'oferir una visió general de com actuen les eines *JPA* i què cal tenir en compte en una implementació.

2.4.1 Entitats

El concepte d'entitat està molt relacionat amb els SGBD i els model relacionals, sobretot en les seves fases de disseny inicial amb el que s'anomena model *Entitat-Relació*. Per a JPA, les *entitats* són aquells *objectes* dels quals es desitja emmagatzemar el seu estat i que acabaran transformant-se en taules i relacions.

En JPA totes les entitats són persistents, però no tots els objectes ho són. Per fer que un objecte sigui persistent cal qualificar-lo d'entitat o bé ha de formar part de l'estat d'una entitat (en forma d'atribut, per exemple).

Totes les entitats s'han de poder identificar de forma única a partir del seu estat. Normalment, n'hi haurà prou amb una petita part dels seus atributs per aconseguir la identificació. La selecció d'atributs que compleixin aquest objectiu s'anomenen identificadors, i en l'SGBD actuaran com a clau primària.

2.4.2 El gestor d'entitats

JPA implementa una classe anomenada `EntityManager` que actuarà de gestor de les entitats de l'aplicació. Sobre aquesta classe recau tota la funcionalitat referida als processos de persistència i sincronització de les entitats. Es tracta, segurament, de la classe més important de la biblioteca JPA.

Un `EntityManager` assumeix tota la funcionalitat que una aplicació pugui necessitar, però únicament a nivell local. JPA és un estàndard de caire general que es pot fer servir en qualsevol tipus d'aplicació, fins i tot en aplicacions distribuïdes. Per aconseguir una sincronització adequada JPA no permet instanciar els `EntityManager` directament, sinó que obliga a instanciar-los des d'un `EntityManagerFactory`, el qual a la seva vegada només podrà ser instanciat per la classe `Persistence`.

La responsabilitat de l'`EntityManagerFactory` està restringida a la creació de gestors d'entitats capaços de compartir un context de persistència de forma coordinada.

En una aplicació, també en les distribuïdes, només hi pot haver una única

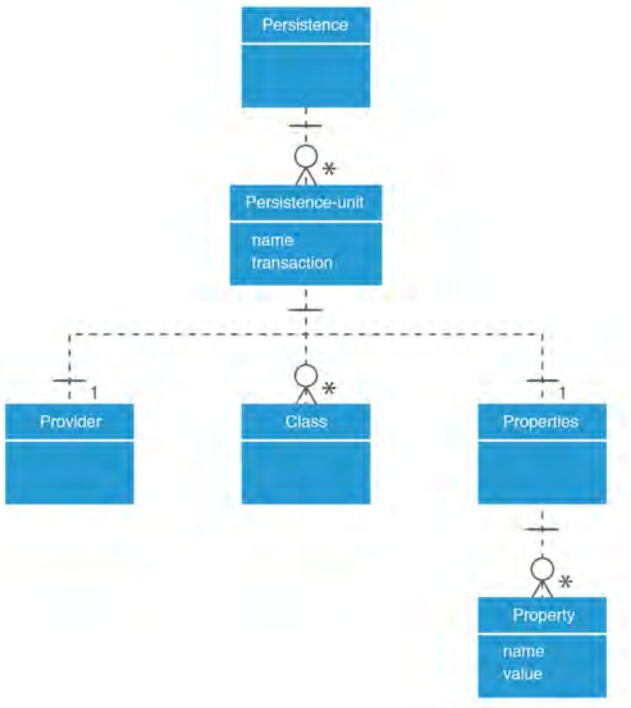
instància de `EntityManagerFactory` per cada SGBD que calgui controlar. Qualsevol intent de duplicar l'`EntityManagerFactory` podria donar resultats inconsistents i totalment inesperats. És per això que JPA obliga a instanciar els `EntityManagerFactory` usant el mètode estàtic de la classe `Persistence` anomenat `createEntityMangerFactory`.

El primer cop que s'instanciï un `EntityManger` es connectarà al'SGDB i comprovarà si existeixen totes les taules necessàries per mantenir la persistència de les entitats que aquest `EntityManger` controli. En cas que falti alguna, es generaran les sentències de creació adequades d'acord amb les metadades llegides del mapatge.

2.4.3 Unitats de persistència

La configuració de cada `EntityManagerFactory` s'aconsegueix a través d'un fitxer XML anomenat persistence.xml (figura 2.3). Es troba situat en un directori de l'aplicació anomenat *META-INF*. Dins d'aquest fitxer hi escriurem totes les configuracions de connexió necessàries per a cada SGBD. Cada configuració constituirà el que anomenem una *unitat de persistència.*

Figura 2.3. Esquema dels elements principals del fitxer de configuració Persistence.xml



Les unitats de persistència s'identifiquen per mitjà d'un nom, el qual passarem com a paràmetre al mètode `createEntityMangerFactory` de la classe `Persistence`, de manera que l'`EntityManagerFactory` creat estarà configurat per connectar-se a un SGBD específic.

El format XML del fitxer segueix l'esquema que podeu veure a la figura. De l'element arrel anomenat `Persistence` es poden descriure tants `Persistence-Unit` com calgui.

Dins d'un `Persistence-Unit` trobem l'element `Provider`, que contindrà la classe principal de l'eina que implementarà JPA. En el cas d'Hibernate, la classe és `org.hibernate.ejb.HibernatePersistence`. També hi podem incloure el conjunt de classes de la nostra aplicació que cal considerar*entitats* i que seran els objectes de la persistència. Finalment, l'esquema presenta una manera de parametrizar la configuració en funció dels diferents *providers* o eines d'implementació de JPA. Ens referim a l'element `Properties`.

Dins l'element `Propertiespodem` ubicar-hi un conjunt variable de propietats (elements `Property`), entenent aquestes com una parella *nom-valor* que assignarem a partir dels atributs del mateix nom.

Vegem un exemple amb dues unitats de persistència:

```
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation=
  "http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
  <persistence-unit name="UnitatDePersistenciaPersistenciaEmpreses"
  transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <class>ioc.dam.m6.exemples.comandes.Comercial</class>
    <class>ioc.dam.m6.exemples.comandes.Client</class>
    <properties>
      <property name="javax.persistence.jdbc.url"
    value="jdbc:postgresql://localhost:5432/ioc_comandes"/>
      <property name="javax.persistence.jdbc.password"
                                              value="ioc"/>
      <property name="javax.persistence.jdbc.driver"                    value="
          org.postgresql.Driver"/>
      <property name="javax.persistence.jdbc.user" value="ioc"/>
    </properties>
  </persistence-unit>
  <persistence-unit
        name="UnitatDePersistenciaPersistenciaProductes"
    transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <class>ioc.dam.m6.exemples.comandes.Producte</class>
    <class>ioc.dam.m6.exemples.comandes.Magatzem</class>
    <properties>
      <property name="javax.persistence.jdbc.url"
        value="jdbc:postgresql://localhost:5432/ioc_comandes"/>
      <property name="javax.persistence.jdbc.password"
            value="ioc"/>
      <property name="javax.persistence.jdbc.driver"
    value="org.postgresql.Driver"/>
      <property name="javax.persistence.jdbc.user" value="ioc"/>
    </properties>
  </persistence-unit>
</persistence>
```

En resum,per crear un `EntityManager` cal tenir un fitxer anomenat Persistence.xml amb el format que s'acaba de descriure. A més, cal crear un `EntityManagerFactory` configurant-lo a partir d'una unitat de persistència inclosa al fitxer persistence.xml, el qual ens permetrà obtenir l'`EntityManager`.

```
EntityManagerFactory emfProd =
        Persistece.createEntityManagerFactory(
```

```
                "UnitatDePersistenciaPersistenciaProductes");

EntityManager emProd = emfProd.CreateEntityManager();
```

2.4.4 El fitxer XML que conté el mapatge

En cas que s'opti per descriure el mapatge en un format XML, aquestes podran estar contingudes en un o més fitxers. JPA descobrirà on es troben els fitxers de mapatge si ho indiquem en el fitxer principal persistence.xml fent servir l'element mapping-file dins de `Persistence-unit`.Per exemple:

```
<persistence ...>
   <persistence-unit>
        ...
        <mapping-file>META-INF/comandes.xml</mapping-file>
        ...
   </persistence-unit>
</persistence>
```

L'arrel dels fitxers de mapatge és entity-mappings, que contindrà el mapatge d'una o més classes. La capçalera del fitxer tindrà la forma següent:

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings
  xmlns="http://java.sun.com/xml/ns/persistence/orm"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
      http://java.sun.com/xml/ns/persistence/orm_2_0.xsd"
  version="2.0">

  ...


</entity-mappings>
```

2.4.5 Transaccions i excepcions

En aplicacions locals `EntityManager` disposa del mètode `getTransaction` per obtenir la transacció en curs, si n'hi ha, o per crear-ne una en cas contrari. Un cop creada, la transacció s'activa invocant el mètode `begin` i finalitza quan s'invoca `commit`.

No és necessari invocar `rollback` en cas d'error. JPA invoca automàticament la revocació de les accions, quan es llanci qualsevol excepció, a partir de l'última invocació de *begin*.

Totes les excepcions generades per JPA són de tipus `RuntimeException`. Aquest tipus d'excepció presenta la particularitat que no s'ha de declarar en la signatura del mètode i per tant, l'ús de *try-catch* no és obligatori.

Aquest tipus de transaccions presenten l'avantatge de poder escriure un codi més net (sense sentències *try-catch* intermèdies), però per contra el desenvolupador ha d'anar molt més en compte de no oblidar-se de fer el tractament de les excepcions. Per facilitar aquest tractament, totes les excepcions JPA hereten d'un antecessor comú anomenat `PersistenceException`.

2.5 Metadades per definir la persistència

En aquest apartat estudiarem les principals metadades usades per mapar les classes persistents. Com ja hem dit, JPA admet dos sistemes: les anotacions Java i els fitxers XML.

És possible també fer servir ambdós sistemes al mateix temps. De fet, el mapatge XML sol ser considerat una forma de sobreescriure les anotacions quan sigui necessari fer canvis sense tocar el codi. És a dir, les descripcions XML prevalen sobre les anotacions.

És per això que estudiarem les dues versions. Cal recordar, de tota manera, que l'ús de les anotacions precisa disposar de les fonts del model per poder mapar-les (les anotacions s'escriuen en el codi) i això no sempre és possible. En canvi, la versió XML no les requereix. En aquest sentit, el sistema XML podria considerar-se més independent.

El cert, però, és que les anotacions tenen cada cop més pes en les implementacions actuals perquè són molt fàcils de compaginar amb el codi tot i que no són intrusives. És a dir, són totalment transparents al desenvolupador que hagi de fer servir el model de classes. A més, el fet que es permeti sobreescriure usant XML fa que siguin un candidat idoni per mapar les classes durant la creació i el disseny del model.

En cas que les modificacions siguin tan estructurals que sigui preferible escriure tot el mapatge, existeix la possibilitat d'anul·lar els efectes de les anotacions escrivint en el primer element de l'arrel entity-mappings els elements següents:

```
<entity-mappings ...>
    <persistence-unit-metadata>
        <xml-mapping-metadata-complete>
    </persistence-unit-metadata>

...

</entity-mappings>
```

2.5.1 Marcant entitats

`Entity` és una anotació de classe que permet marcar les entitats que necessitem controlar. Qualsevol classe que contingui aquesta anotació serà susceptible de

fer-se persistent amb l'`EntityManager`.

Ja s'ha comentat que totes les entitats han de tenir un identificador. Per poder marcar un atribut de la classe com un valor d'identificació disposem de l'anotació *Id*. Més endavant estudiarem formes més complexes d'expressar claus primàries compostes, però de moment farem servir l'anotació *Id* per marcar un atribut de tipus primitiu com a identificador. En el model E-R, el seus valors es faran servir com a clau primària.

En cas que no es desitgés emmagatzemar algun dels atributs, caldrà marcar-los com a `Transient`. Els atributs que no estiguin marcats així es consideraran persistents i, si no s'indica res més, es considerarà que cal emmagatzemar-los en una columna que tingui el mateix nom que l'atribut.

Imaginem que hem d'implementar una aplicació de gestió comercial. Imaginem que el model dissenyat per a aquesta aplicació té una classe anomenada `UnitatDeMesura` que simbolitza les diferents unitats de mesura que podem aplicar als productes de l'aplicació que es compren o es venen. Bàsicament tindrà dos atributs, el símbol de la unitat (m, kg, mm^3, cm^2, cl, etc.) que farà d'identificador i la descripció de la mateixa.

Funcions de dispersió

Les funcions de dispersió (hash functions en anglès) assignen per mitjà d'un complex càlcul un valor numèric (anomenat valor de dispersió). El valor es calcula en funció de la seqüència de lletres que componen la cadena.

Imaginem que necessitem fer servir molt sovint el valor de dispersió del símbol. En lloc de calcular-lo cada vegada, decidim calcular-lo només una única vegada i assignar-lo a un atribut temporal. No volem emmagatzemar aquest valor i, per tant, el declararem `Transient`.

Exemple usant anotacions:

```
@Entity
public class UnitatDeMesura implements Serializable {
    @Id
    private String simbol;
    private String descripcio;
    @Transient
    private Integer hashSimbol;

    public UnitatDeMesura() {
    }
    public UnitatDeMesura(String simbol, String descripcio) {
        this.simbol = simbol;
        this.descripcio = descripcio;
    }

  ...
}
```

Exemple usant XML:

```
<entity-mappings ...>

  ...

  <entity class="ioc.dam.m6.exemples.comandes.UnitatDeMesura"
    metadata-complete="true">
    <attributes>
      <id name="simbol"/>
      <transient name="hashSimbol"/>
    </attributes>
  </entity>
```

```
12
13    ...
14
15  </entity-mappings>
```

En les declaracions XML, metadata-complete implica que s'anul·laran totalment les anotacions de la classe `UnitatDeMesura`. Si no s'especifica *metadata-complete* o se li dóna un valor fals, la metadada final s'obtindrà de la unió de les anotacions i les descripcions XML. En cas de conflicte, prevaldran sempre les descripcions XML.

Aquesta metadada indica que les entitats de tipus `UnitatDeMesura` s'emmagatzemaran en una taula anomenada *unitatdemesura* composta de dues columnes, la columna *simbol* que serà clau primaria, i la columna *descripcio*, ambdues de tipus VARCHAR.

2.5.2 Generació automàtica de la clau primària

És molt probable que en el tractament d'algunes entitats desitgem despreocupar-nos del valor de la clau primària i decidim generar-la de forma automàtica. Si l'aplicació que realitzem, per exemple, hagués de generar documents de comandes identificats amb un número que seguís un ordre seqüencial, resultaria força tediós haver de recordar el darrer número entrat. És més fàcil deixar que sigui el sistema qui s'encarregui de generar de forma automàtica el valor corresponent.

La generació automàtica de la clau primària també pot ser una forma d'independitzar el model E-R dels objectes de l'aplicació. El model OO no necessita l'existència de claus primàries i la generació automàtica podria permetre ignorar totalment el requeriment imposat pels sistemes relacionals.

La majoria d'SGBD disposen d'eines per generar claus primàries de forma automàtica. El problema és que no tots els SGBD comparteixen les mateixes eines. Mentre alguns SGBD fan servir seqüències, altres utilitzen un tipus de columna específic que autoincrementarà el valor cada cop que hi hagi una inserció. També hi ha SGBD que no disposen de cap eina.

Per tal d'adaptar-se a tots els SGBD, JPA disposa de quatre estratègies per aconseguir la generació automàtica de la clau. Dues d'elles fan servir una de les eines esmentades dels SGBD. Les altres dues són específiques de JPA.

Per tal de seleccionar una de les estratègies n'hi haurà prou amb indicar quina estratègia volem fer servir usant la notació `GeneretedValue`abans de l'atribut identificador. Aquesta notació admet un paràmetre anomenat *strategy*.

Els 4 possibles valors es troben definits com a constants de la classe `GenerationType`. Les constants són: `auto`, `table`, `sequence` i `identity`.

L'estratègia `GenerationType.AUTO` és útil per treballar durant la fase de desenvolupament. En realitat, cada implementació JPA pot gestionar aquesta estratègia de

diferent manera. Normalment s'acostuma a basar en un registre de la memòria RAM que es va incrementant a cada inserció i que s'actualitza cada cop que comencem a usar JPA. És útil durant la fase de desenvolupament perquè és una estratègia molt ràpida que no requereix cap manipulació del'SGBD, però no s'aconsella fer-la servir en la fase d'explotació ja que no hi ha garantia real que la clau generada sigui certament única.

Si fem servir anotacions, caldrà especificar:

```
@id
@GeneratedValue(strategy=GererationType.AUTO);
private long atributId;
```

I en format XML:

```
...

<id name="atributId">
    <generarted–value strategy="AUTO"/>
</id>

...
```

L'estratègia `GenerationType.TABLE` està basada en una taula normal del'SGBD capaç d'emmagatzemar un o diversos comptadors que ajudaran a generar un valor únic cada cop que sigui necessari. Es tracta d'una estratègia molt flexible que s'adapta a qualsevol SGBD. La taula es crea de forma automàtica juntament amb la resta de taules i conté dues columnes. La primera és una *cadena de caràcters* que prendrà el nom de la classe on calgui generar el valor de la clau primària i s'usa com a identificador del comptador. La segona columna és de tipus *enter* i emmagatzema el següent valor amb el qual es generarà la nova clau primària.

Usant anotacions, especificarem:

```
@id
@GeneratedValue(strategy=GenerationType.TABLE);
private long atributId;
```

I en format XML:

```
...

<id name="atributId">
    <generated–value strategy="TABLE"/>
</id>

...
```

L'estratègia `GenerationType.SEQUENCE` està basada en l'ús de seqüències pròpies dels SGBD. Això significa que només es pot usar en aquells sistemes gestors que suportin seqüències.

En la majoria de sistemes de bases de dades, les seqüències tenen associats un únic comptador per seqüència i s'identifiquen per mitjà d'un nom.

Usant anotacions, ho indicarem fent servir l'anotació `SequenceGenerator`.

Aquesta es pot parametritzar indicant un nom que identificarà la seqüència que es vol fer servir per a la classe anotada.

`SequenceGenerator` haurà d'identificar el nom de la seqüència i el nom del generador de claus primàries utilitzat.

```
@Id
@SequenceGenerator(name="nomGenerador", sequenceName="nomSequencia")
@GeneratedValue(strategy= GenerationType.SEQUENCE, generator="nomGenerador")
private long atributId;
```

La versió XML de les seqüències serà:

```
...

<id name="atributId">
    <generated-value strategy="SEQUENCE"
        generator="nomGenerador"/>
    <sequence-generator name="nomGenerador"
        sequence-name="nomSequencia"/>
</id>

...
```

Finalment, l'estratègia `GenerationType.IDENTITY` és també una estratègia que depèn del'SGBD. Està pensada específicament per a aquells SGBD que disposen d'un tipus autoincremental com MySQL, Access o d'altres SGBD. No precisa de cap element extra de configuració. N'hi ha prou d'indicar-ho i que l'SGBD ho suporti. Vegem ambdues versions. Amb anotacions:

```
@Id
@GeneratedValue(strategy= GenerationType.IDENTITY)
private long atributId;
```

I amb format XML:

```
...

<id name="atributId">
    <generated-value strategy="IDENTITY"/>
</id>

...
```

Ara, vegem un exemple complet usant l'estratègia `taula`. Es tracta de la classe `Envas`, emmarcada també dins l'aplicació de comandes esmentada. Aquesta classe representa l'envàs d'un producte, caracteritzat pel tipus (paquet, bric, bossa, sac, ampolla, llauna, caixa, etc.), la quantitat de producte i les unitats en què es mesura aquesta quantitat (500 ml, per exemple).

Per evitar de crear una clau composta dels tres atributs, s'ha decidit incorporar un identificador numèric anomenat *id* que serà generat automàticament pel sistema durant la inserció de les seves instàncies.

```
@Entity
public class Envas implements Serializable {
    @Id
    @GeneratedValue(strategy= GenerationType.TABLE)
```

```
    private Long id;
    private String tipus;
    private double quantitat;
    @ManyToOne
    private UnitatDeMesura unitat;

    /** Constructor per defecte
     */
    public Envas() {
    }

    public Envas(String tipus, double quantitat,
                             UnitatDeMesura unitat) {
        this.tipus = tipus;
        this.quantitat = quantitat;
        this.unitat = unitat;
    }

    protected Long getId() {
        return id;
    }

    protected void setId(Long id) {
        this.id = id;
    }

    public String getTipus() {
        return tipus;
    }

    public void setTipus(String tipus) {
        this.tipus = tipus;
    }

    public double getQuantitat() {
        return quantitat;
    }

    public void setQuantitat(double quantitat) {
        this.quantitat = quantitat;
    }

    public UnitatDeMesura getUnitat() {
        return unitat;
    }

    public void setUnitat(UnitatDeMesura unitat) {
        this.unitat = unitat;
    }
}
```

Accessors

En Java, s'anomenen accessors d'atributs privats aquells mètodes que permeten llegir i escriure el contingut de l'atribut. Les convencions Java recomanen que l'accessor de lectura s'anomeni com l'atribut,anteposant-hi però el prefix get i canviant la primera inicial del nom de l'atribut a majúscula. L'accessor d'escriptura seguiria el mateix patró, però caldrà anteposar-hi el prefix set. Exemple: getId i setId seran els noms dels accessors de l'atribut id.

Fixeu-vos que hem declarat *private* l'atribut *id* i *protected* els seus accessors. Volem que la clau d'aquesta classe sigui totalment transparent al model. És a dir, que puguem ignorar-la sense cap conseqüència.

L'equivalent XML seria:

```
<entity-mappings ...>

...

    <entity class="ioc.dam.m6.exemples.comandes.Envas"
                metadata-complete="true">
        <attributes>
            <id name="id">
                <generated-value strategy="TABLE"/>
            </id>
        </attributes>
```

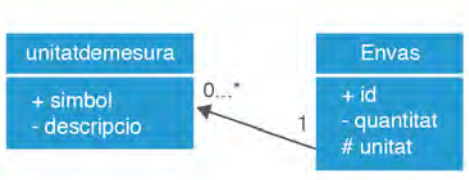
```
    </entity>

...

</entity_mappings>
```

2.5.3 Marcant relacions

JPA obliga també a marcar les relacions que puguin haver entre diferents entitats d'acord amb la seva cardinalitat. Es preveuen quatre tipus de cardinalitat: *ManyToOne*, *OneToOne*, *OneToMany* o *ManyToMany*. Les dues primeres s'anomenen també de valor únic perquè es corresponen amb un atribut que referencia un únic objecte. A l'exemple anterior de la classe `Envas` podeu veure una relació *ManyToOne* per a l'atribut unitat. Fixeu-vos que l'atribut referencia una única unitat.

```
@ManyToOne
private UnitatDeMesura unitat;
```

Figura 2.4. Relació Molts és a 1 entre la taula Envas i la taula unitatdemesura



Per emmagatzemar la *unitat* a la taula *ENVAS* només caldrà guardar la clau primària, ja que la resta de dades es guardaran a la taula *unitatdemesura* (vegeu la figura 2.4).

La diferència entre *OneToOne* i *ManyToOne* és més conceptual que pràctica, ja que ambdós (utilitzats per defecte) produeixen la mateixa conversió: una clau forana a la taula on s'emmagatzemarà la relació.

Considereu ara la classe `Comercial`. A banda de les dades personals, imagineu que els comercials tenen assignada una zona i que cada zona pertany exclusivament a un comercial (figura 2.5). Aquí direm que la relació serà *OneToOne*.

```
@Entity
public class Comercial implements Serializable {
    @Id
    private String nif;
    ...
    @OneToOne
    private Zona zona;
    ...
}
```

Figura 2.5. Esquema E-R de les taules Comercial i Zona

La versió XML dels dos exemples anteriors és la següent:

```
<entity-mappings ...>

    ...

    <entity class="ioc.dam.m6.exemples.comandes.Envas"
                metadata-complete="true">
        <attributes>
            <id name="id">
                <generated-value strategy="TABLE"/>
            </id>
            <many-to-one name="unitat"/>
        </attributes>
    </entity>

    ...

    <entity class="ioc.dam.m6.exemples.comandes.Comercial"
                 metadata-complete="true">
        <attributes>
            <id name="nif"/>
            <one-to-one name="zona"/>
        </attributes>
    </entity>

...

</entity-mappings>
```

Les relacions *OneToMany* i *ManyToMany* s'anomenen també multiavaluades, perquè es corresponen amb atributs de tipus *array*, llista, conjunt, mapa, etc.

Hi ha diferències conceptuals entre ambdues: en les relacions *OneToMany*cada objecte relacionat es troba associat a un i només un objecte, mentre que les *ManyToMany* no. Això significa que les relacions *OnteToMany* poden plasmar-se en un model E-R fent servir només dues taules, mentre que les *ManyToMany* precisaran sempre una taula extra.

En programació orientada a l'objecte, a diferència del paradigma relacional, no és gaire rellevant tipificar la relació d'un o altre tipus perquè les relacions es troben segmentades sempre en petites col·leccions associades a un objecte, i des del punt de vista de l'objecte aquella relació podria considerar-se sempre de tipus *un-és-a-molts,* malgrat que pel conjunt calgués tipificar-la de *molts-és-a-molts*.

Per aquesta raó, JPA, malgrat que disposa de marques per distingir entre ambdós tipus de relacions, a la pràctica la implementació per defecte que s'obté és idèntica en els dos casos i respon sempre a l'estil *molts-és-a-molts* que implica la creació d'una taula extra.

Òbviament, és possible evitar l'ús de taules extres a relacions *OneToMany* reals, però caldrà afegir metadades específiques.

Cerquem, de moment, un exemple *ManyToMany*. Imagineu que l'aplicació de comandes classifica els clients en sectors professionals. A més, per poder crear promocions i campanyes, l'aplicació manté informació de quins sectors professionals gasten determinats productes. La relació que s'estableix entre productes i sectors té una cardinalitat de *molts a molts,* ja que un sector pot gastar molts productes, però un mateix producte el poden comprar les empreses de diversos sectors.

Anem a plasmar aquesta relació a la classe `Sector`. Desitgem mantenir, per a cada sector, una llista de tots els productes consumits per les empreses que pertanyin al sector.

Primer fent servir anotacions:

```
@Entity
public class Sector implements Serializable {
    @Id
    private String id;
    private String descripcio;
    @ManyToMany
    private List<Producte> productes=new ArrayList<Producte>();

    protected Sector() {
    }

    public Sector(String id) {
        this.id = id;
    }

    public Sector(String id, String descripcio) {
        this.id = id;
        this.descripcio = descripcio;
    }

    public String getId() {
        return id;
    }

    protected void setId(String id) {
        this.id = id;
    }

    public String getDescripcio() {
        return descripcio;
    }

    public void setDescripcio(String descripcio) {
        this.descripcio = descripcio;
    }
}
```

Les relacions multivalents es concreten sempre en algun tipus de col·lecció. JPA necessita que els atributs es declarin fent servir les interfícies corresponents. Mai s'ha de declarar un atribut que representi una relació multivalent fent servir una classe concreta. Fixeu-vos que l'atribut *productes* està declarat de tipus `List` (interfície) en comptes d'`ArrayList` (classe).

Passem a veure ara el format XML equivalent:

```
<entity-mappings ...>

    ...

    <entity class="ioc.dam.m6.exemples.comandes.Sector"
                metadata-complete="true">
        <attributes>
            <id name="id"/>
            <many-to-many name="productes"/>
        </attributes>
    </entity>

    ...

</entity-mappings>
```

2.5.4 Modificant les opcions per defecte de JPA

Amb les metadades estudiades fins el moment, segurament podem mapar la majoria de models orientats a objecte que siguem capaços de dissenyar. Ara bé, probablement les taules generades no es correspondrien pas amb el que nosaltres haguéssim dissenyat.

Potser les taules generades no tindran la forma més eficient d'organitzar les dades, o és possible que partim d'una base de dades ja existent i necessitem adaptar JPA a un disseny de les taules diferent al generat per defecte.

Indicar característiques pròpies del sistema relacional

Per defecte, ja s'ha comentat que les classes qualificades d'*entitats* generen una taula amb el mateix nom que la classe. Aquesta opció és prou bona sempre que partim de zero i ens calgui generar totes les taules; però si haguéssim de treballar amb taules ja existents provinents d'altres aplicacions i no poguéssim caviar-ne el nom, hauríem d'adaptar les metadades per reflectir l'estructura existent.

Table

L'element `Table` permet indicar el nom de la taula on s'emmagatzemaran les *entitats*, de manera que sigui possible treballar amb una taula que tingui un nom diferent al de *l'entitat*. També es pot especificar el catàleg i/o l'esquema del'SGBD al qual pertanyi la taula. Ambdós són opcionals i només caldrà especificar-los en cas que l'esquema o catàleg de la taula no coincideixi amb el que es trobi configurat per defecte a la `PersistenceUnit`.

```
@Entity
@Table(name="UNITAT_DE_MESURA", schema="COMU")
public class UnitatDeMesura implements Serializable {
    @Id
    private String simbol;
    private String descripcio;

    public UnitatDeMesura() {
```

```
    }
    public UnitatDeMesura(String simbol, String descripcio) {
        this.simbol = simbol;
        this.descripcio = descripcio;
    }

    ...
}
```

En l'exemple podem veure com s'especifica que els objectes de la classe `UnitatDeMesura` s'emmagatzemin a la taula *UNITAT_DE_MESURA* ubicada a l'*schema* anomenat *COMU*.

Seguint la mateixa lògica, el format XML tindrà la forma següent:

```
<entity_mappings ...>

    ...

    <entity class="ioc.dam.m6.exemples.comandes.UnitatDeMesura"
                metadata_complete="true">
        <table name="UNITAT_DE_MESURA" schema="COMU" />
        <attributes>
             <id name="simbol"/>
        </attributes>
    </entity>

...

</entity_mappings>
```

L'element `Table` pot fer-se servir també per generar restriccions de valors únics durant la creació de les taules. Es poden especificar un nombre variable de restriccions, a cada una de les quals hi podran intervenir un nombre variable de columnes.

L'atribut de `Table` que caldrà especificar és `uniqueConstraints`, el qual contindrà una col·lecció de valors d'una altra anotació anomenada `UniqueConstraint`. Aquesta admet un paràmetre amb la col·lecció de noms de columnes que intervindran en la restricció de valor únic.

Vegem un exemple. Tornem a la classe `Envas`. Recordeu que hem considerat que un envàs és un tipus, una quantitat i una unitat. A més, per simplificar, es va decidir treballar amb una clau primària interna generada automàticament. Per assegurar que no es puguin emmagatzemar registres coincidents del mateix tipus, la mateixa quantitat i la mateixa unitat de mesura, volem posar una restricció de valors únics.

Quan un paràmetre d'una anotació admeti una col·lecció de valors, s'expressaran tots ells separats per una coma i continguts entre claus.

```
@Entity
@Table(uniqueConstraints={
    @UniqueConstraint(columnNames={"tipus", "quantitat", "unitat_simbol"})
})
public class Envas implements Serializable {
    ...
}
```

A l'exemple, el paràmetre `uniqueConstraints` està format per una col·lecció d'una sola restricció simbolitzada per

`@UniqueConstraint(columnNames={“tipus”, “quantitat”, “unitat_simbol”})`. La restricció es basarà en el valor de les tres columnes indicades en el paràmetre columnNames.

A l'equivalent XML, cada restricció s'inclou en una etiqueta `unique-constraint`, definides a mode de seqüència de `Table`. Els noms de les columnes s'expressen també com a seqüències d'elements `column-name`.

```
<entity-mappings ...>
    ...

    entity class="ioc.dam.m6.exemples.comandes.Envas "
                metadata-complete="true">
        <table>
            <unique-constraint>
                <column-name>tipus</column-name>
                <column-name>quantitat</column-name>
                <column-name>unitat_simbol</column-name>
            </unique-constraint>
        </table>
        ...
    </entity>
    ...

</entity-mappings>
```

Column i JoinColumn

De la mateixa manera que s'indica el nom de les taules, es pot especificar també el nom de les columnes, així com les característiques amb les quals cal generar-les. L'anotació usada és `Column`, indicada com a prefix de qualsevol atribut d'una classe.

`Column`és una anotació molt parametritzable, atès que permet expressar moltes característiques referides a la columna. Entre d'altres, podem expressar el nom (paràmetre `name`), la mida de la columna (paràmetre `length`) quan es vulgui limitar la grandària d'una cadena de caràcters, si s'acceptaran valors nuls (paràmetre `nullable`) o si els valors de la columna hauran de ser únics per a cada registre (paràmetre `unique`).

Si l'atribut en comptes de ser un tipus primitiu fos una entitat (amb una relació de tipus OneToOne o ManyToOne), haurem de substituir l'anotació `Column` per `JoinColumn`, indicant que ens trobarem amb una clau forana. L'especificació de `JoinColumn` és molt similar a Column.

Usarem de nou la classe `Envas` per mostrar un exemple força complet de l'ús de `Column` i `JoinColumn`.

```
@Entity
@Table(name="ENVAS", uniqueConstraints={@UniqueConstraint(name="envasUnic",
        columnNames={"tipus", "quantitat", "unitat_simbol"})
})
public class Envas implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy= GenerationType.TABLE)
    private Long id;
    @Column(length=100, nullable=false)
    private String tipus;
```

```
12      @Column(nullable=false)
13      private double quantitat;
14      @ManyToOne
15      @JoinColumn(name="unitat_simbol", nullable=false)
16      private UnitatDeMesura unitat;
17
18      ...
19  }
```

A l'XML equivalent, els atributs primitius s'identifiquen usant l'element anomenat `basic`.

```
1   <entity-mappings ...>
2
3       ...
4
5       <entity class="ioc.dam.m6.exemples.comandes.Envas"
6                   metadata-complete="true">
7           <table>
8               <unique-constraint>
9                   <column-name>tipus</column-name>
10                  <column-name>quantitat</column-name>
11                  <column-name>unitat_simbol</column-name>
12              </unique-constraint>
13          </table>
14          <attributes>
15              <id name="id">
16                  <generated-value strategy="TABLE"/>
17              </id>
18              <basic name="tipus">
19                  <column length="100" nullable="false" />
20              </basic>
21              <basic name="quantitat">
22                  <column nullable="false" />
23              </basic>
24              <many-to-one name="unitat">
25                  <join-column name="unitat_simbol"
26                          nullable="false" />
27              </many-to-one>
28          </attributes>
29      </entity>
30
31      ...
32
33  </entity-mappings>
```

Càrrega de dades diferida

JPA permet marcar certs atributs de les entitats amb la marca *LAZY,* de manera que en obtenir els objectes emmagatzemats, les dades marcades així no es recuperaran de forma immediata, sinó només quan hi hagi un intent d'accedir a l'atribut. Aquesta característica es coneix com a càrrega *"mandrosa",* tardana o diferida.

És una tècnica molt utilitzada durant la recuperació d'entitats amb un gran volum de dades. La recuperació tardana s'aplica a atributs de tipus imatges, col·leccions, cadenes de text llargues o qualsevol altre tipus que impliqui mobilitzar una gran quantitat de bytes.

Si l'atribut és de tipus primitiu o bé un *Array* d'un tipus *byte* o *char*, la marca LAZY s'indica com a paràmetre de l'anotació `Basic`, però si l'atribut és una rela-

ció amb una altra entitat (*OneToOne*, *ManyToOne*, *OneToMany* o *ManyToMany*), la marca LAZY s'expressa com a paràmetre de la pròpia anotació de la relació.

Imaginem que a la classe `Comercial` guardem el contingut del contracte firmat amb cada comercial on s'especifiquin les comissions i les condicions de venda dels nostres productes. Probablement es tractarà d'un contingut extens que no necessitarem consultar cada cop que recuperem un comercial, sinó només en moments puntuals quan es faci revisió del contracte, quan calgui signar-los, etc. Per això decidim marcar-lo com a LAZY.

Es tracta d'un tipus primitiu (*String*) i, per tant, el qualificarem de càrrega diferida usant la notació `Basic`. En canvi, per fer el mateix amb l'atribut *zona*, el qual representa una entitat, usarem la notació *OneToOne* que el qualifica.

```
@Entity
public class Comercial implements Serializable {
    @Id
    @Column(length=15)
    private String nif;
    private String nom;
    @Basic(fetch= FetchType.LAZY)
    private String contracte;
    @OneToOne(fetch= FetchType.LAZY)
    private Zona zona;
    ...
}
```

Si volem plasmar-ho en format XML:

```
<entity-mappings ...>

    ...

    <entity class="ioc.dam.m6.exemples.comandes.Comercial"
                metadata-complete="true">
        <attributes>
            <id name="nif">
                <column length="15" />
            </id>
            <basic name="contracte" fetch="LAZY" />
            <one-to-one name="zona" fetch="LAZY" />
        </attributes>
    </entity>

    ...

</entity-mappings>
```

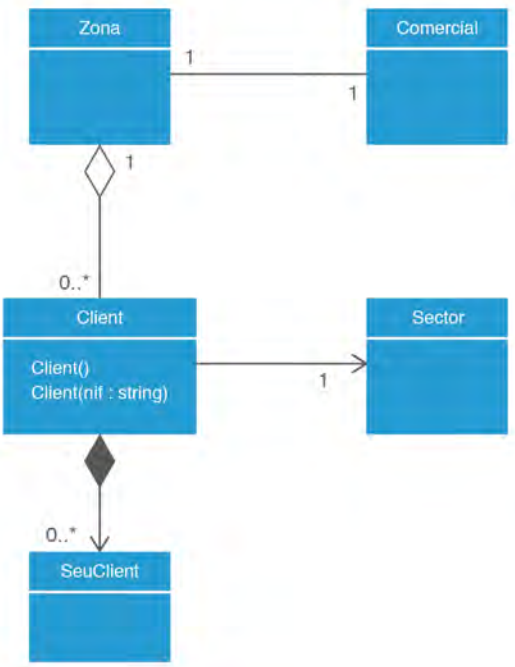
2.5.5 Relacions unidireccionals i bidireccionals

Direm que dos objectes estan relacionats bidireccionalment quan ambdós poden interaccionar entre ells. Per contra, parlem de relació unidireccional quan la manipulació només es pot donar en una sola direcció.

A la figura 2.6 podem veure una part del diagrama de classes de l'aplicació de comandes que estem utilitzant d'exemple. En els diagrames de classe, les

relacions bidireccionals es representen per una línia sense fletxes, mentre que les relacions unidireccionals apunten amb una fletxa a la classe que serà visible des de la classe origen. Com podeu observar, en el diagrama es distingeixen dues relacions unidireccionals (*Client-Sector* i *Client-SeuClient*) i dues bidireccionals (*Zona-Client* i *Zona-Comercial*).

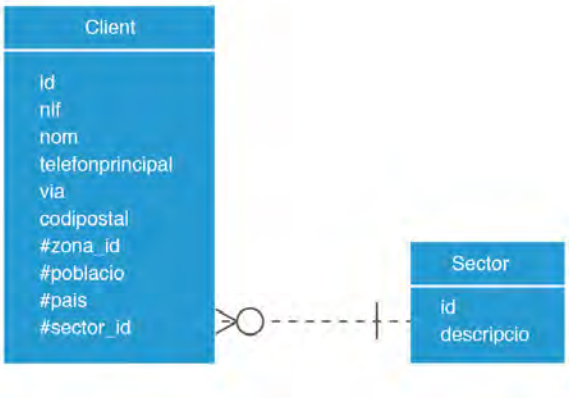
Figura 2.6. Diagrama de classes parcial de l'aplicació exemple



Relació Client-Sector

La relació *Client-Sector* és de tipus *ManyToOne* perquè un client només pot pertànyer a un sector, però cada sector pot tenir diversos clients (figura 2.7).

Figura 2.7. Relació entre les taules Client i Sector



Tot i això, a efectes pràctics, en tractar-se d'una relació unidireccional de client cap a sector, la relació *molts és a un* dels sectors cap als clients és inexistent i, en conseqüència, la implementació no presenta cap diferència amb una relació

unidireccional de tipus *OneToOne*. És per això que aquest tipus de relacions unidireccionals es qualifiquen també com a *OneToOne*. La implementació requerirà que els clients tinguin un atribut de tipus sector, però els sectors no disposaran de la informació per saber quins clients són d'aquell sector.

Des de la perspectiva E-R, a la taula Client s'hi afegirà una clau forana corresponent a la clau primària del sector.

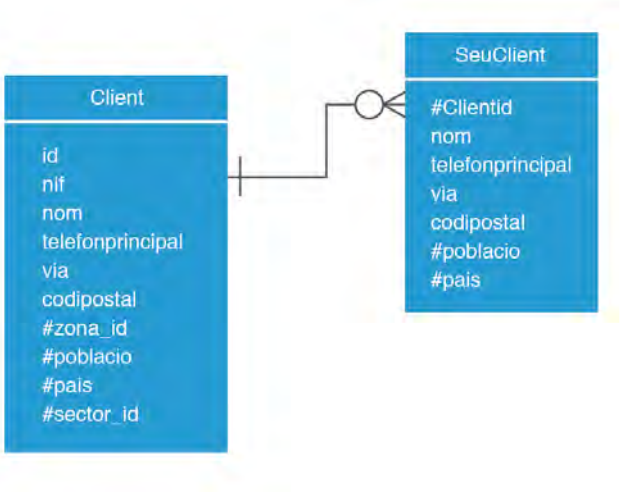
La implementació implicarà un atribut de tipus `Sector` qualificat com a *OneToOne* o *ManyToOne:*

```
1 @OneToOne(fetch= FetchType.LAZY)
2 private Sector sector;
```

Relació Client-SeuClient

Passem ara a analitzar la relació *Client-SeuClient.* És una relació *OneToMany* unidireccional. És a dir, permet associar un únic client amb totes les seves seus. Per contra, les seus no disposen d'informació del client (figura 2.8).

Figura 2.8. Relació entre les taules Client i SeuClient



JPA representa amb certes dificultats aquest tipus de relacions. Es tracta d'una relació *OneToMany* pura. És a dir, des de la perspectiva relacional n'hi ha prou només amb les dues taules (una de cada entitat) per representar-la.

El problema el trobem en el fet que el model E-R imposa que la clau forana se situï a la taula de l'entitat *SeuClient*, just a la banda on no és necessària, ja que és el client qui referenciarà les seus.

En conseqüència, usant JPA, les relacions *OneToMany* són sempre bidireccionals. De fet, seria possible representar una relació *OneToMany* unidireccional però caldria afegir sempre una taula extra i això incrementaria la complexitat del model E-R. Normalment no es fa servir gairebé mai.

Convertirem la relació en bidireccional: una de tipus *OneToMany* des de *Client* a *SeuClient* i una altra de tipus *ManyToOne* des de *SeuClient* a *Client*.

A més d'especificar ambdues relacions, si volem evitar la creació d'una taula extra necessitarem encara indicar-ho fent servir el paràmetre *mapedBy*. Aquest paràmetre modifica la relació *OneToMany*de *Client* a *SeuClient*) i indica a JPA que no cal gestionar la persistència, perquè ja es troba mapada per la seva relació inversa (*de SeuClient* a *Client*). El paràmetre *mapedBy* indicarà quin atribut de *SeuClient* actua de clau forana, de manera que sigui possible recuperar la informació per omplir la col·lecció.

Anem a veure-ho: el paràmetre `mappedBy` indica que l'atribut que referencia les entitats de tipus *SeuClient* que pertanyen a un mateix client s'anomena *client*.

```
@Entity
public class Client extends Empresa {
    @OneToMany(mappedBy="client", fetch= FetchType.LAZY)
    private Set<SeuClient>seus=new HashSet<SeuClient>();
  ...
}

...

@Entity @Access(AccessType.PROPERTY)
public class SeuClient implements Serializable{
    @ManyToOne
    private Client client;
  ...
}
```

La col·lecció de seus s'ha representat com un conjunt (*Set*), però podria representar-se com qualsevol altra col·lecció (llista, vector, etc.).

Queda encara un últim detall per resoldre. Quan ens trobem davant d'una relació bidireccional, cal plantejar-se si ambdues entitats són fortes o bé una d'elles depèn de l'altra. En el cas que ens ocupa, la seus només tenen sentit com a part d'un client, per tant ens trobem davant d'una entitat dèbil (les seus) que forma part d'una entitat forta (els clients).

Per evitar problemes de coherència i consistència de dades, tota la gestió de la persistència recaurà sobre l'entitat forta. JPA ho gestionarà així si afegim un nou paràmetre a la relació *OneToMany* de la classe `Client`indicant-ho. Es tracta del paràmetre anomenat `cascade`.

```
@Entity
public class Client extends Empresa {
    @OneToMany(mappedBy="client",
      cascade={CascadeType.ALL}, fetch= FetchType.LAZY)
    private Set<SeuClient>seus=new HashSet<SeuClient>();
  ...
}
```

L'atribut `cascade` admet el valor `CascadeType.ALL` per indicar que la classe client s'encarregarà de tota la persistència de les seves seus.

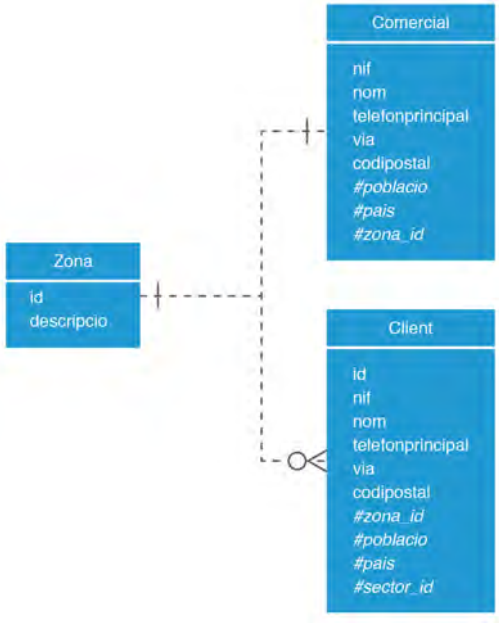
Aquest atribut admet també altres valors com `CascadeType.PERSIST` per indicar que la responsabilitat es limitarà a la inserció de seus. El valor `CascadeType.MERGE` indicarà que la responsabilitat del client s'activa durant els processos de modificació. També és possible delimitar la responsabilitat durant els processos d'eliminació amb `CascadeType.DELETE`, de manera que en eliminar el

client s'eliminin també les seves seus.

Relació Zona-Comercial i Zona-Client

El model entitat relació gestiona la relació bidireccional de tipus *OneToOne* entre `Zona` i `Comercial` afegint una clau forana. Ambdues entitats són igual de fortes, i per tant aprofitem la ubicació de la clau forana per responsabilitzar la classe `Comercial` de l'emmagatzematge de la relació entre `Zona` i `Comercial` (figura 2.9). Aquesta és una decisió més conceptual que tècnica, ja que des d'un punt de vista tècnic i en tractar-se d'una relació *OneToOne* s'hagués pogut optar per una clau forana des de `Zona` a client.

Figura 2.9. Relació entre les taules Zona, Comercial i Client



En conseqüència, a la implementació JPA caldrà indicar que la relació que va des de `Zona` a `Comercial` ja es troba mapada per la relació inversa, utilitzant el paràmetre `mapedBy`.

De forma semblant, la relació *Zona-Client* es trobarà marcada en les dues entitats amb una diferència substancial. De `Zona` a `Client` és *OneToMany* i de `Client` a `Zona` és *ManyToOne*. En aquests casos la clau forana sempre s'ha de situar a l'entitat de la relació *ManyToOne*, és a dir, el `Client` en aquest cas. Aquí, es tracta també de dues entitats fortes.

Aplicant criteris de senzillesa, és més fàcil que cada client emmagatzemi la zona a la qual està assignada que no pas aquesta hagi de modificar la clau forana de tots els clients que tingui assignats. En conseqüència, la responsabilitat de gestionar la persistència de la relació recaurà sobre els clients.

La relació *OneToMany* de la `Zona` s'haurà d'identificar com a ja mapada fent servir el paràmetre `mapedBy`. Anem a veure la implementació sencera:

```java
@Entity
public class Zona implements Serializable {
    @Id
    @Column(length=10)
    private String id;
    @Column(length=50)
    private String descripcio;
    @OneToMany(mappedBy = "zona", fetch=FetchType.LAZY)
    @OrderBy(value="nif")
    private List<Client> clients;
    @OneToOne(mappedBy="zona", fetch= FetchType.LAZY)
    private Comercial comercial;

  ...
}

@Entity
public class Comercial implements Serializable {
    @Id
    @Column(length=15)
    private String nif;
    private String nom;
    @OneToOne(fetch= FetchType.LAZY)
    private Zona zona;
  ...
}

@Entity
public class Client extends Empresa {
    @ManyToOne(fetch= FetchType.LAZY)
    private Zona zona;

    @OneToMany(mappedBy="client", cascade={CascadeType.ALL},
            fetch= FetchType.LAZY)
    private Set<SeuClient>seus=new HashSet<SeuClient>();

    @OneToOne(fetch= FetchType.LAZY)
    private Sector sector=null;

  ...
}

@Entity
public class SeuClient implements Serializable{
    private Seu seu = new Seu();
    @ManyToOne
    private Client client;
  ...
}

@Entity
public class Sector implements Serializable {
    @Id
    @Column(length=50)
    private String id;
  ...
}
```

Fixeu-vos que l'anotació `OrderBy`és molt útil per mantenir sempre les dades d'una llista ordenada sota un criteri determinat. Cal ser conscients, però, que aquesta anotació no servirà per a aquells tipus de col·leccions com els conjunts o els mapes, ja que són col·leccions que imposen un ordre propi als seus elements.

Format XML

Vegem també l'equivalent XML:

```
<entity-mappings ...>

  ...

  <entity class="ioc.dam.m6.exemples.comandes.Zona "
    metadata-complete="true">
    <attributes>
      <id name="id">
        <column length="10" />
      </id>
      <basic name="descripcio">
        <column length="50" />
      </basic>
      <one-to-one name="comercial" mapped-by="zona"
              fetch="LAZY" />
      <one-to-many name="clients" mapped-by="zona"
              fetch="LAZY" >
        <order-by value="nif" />
      </one-to-many>
    </attributes>
  </entity>

  <entity class="ioc.dam.m6.exemples.comandes.Comercial "
    metadata-complete="true">
    <attributes>
      <id name="nif">
        <column length="15" />
      </id>
      <one-to-one name="zona" fetch="LAZY" />
    </attributes>
  </entity>

  <entity class="ioc.dam.m6.exemples.comandes.Client "
                       metadata-complete="true">
    <attributes>
      ...
      <many-to-one name="zona" fetch="LAZY" />
      <one-to-many name="seus" fetch="LAZY">
        <cascade>
          <cascade-all/>
        </cascade>
      </one-to-many>
      <one-to-one name="sector" fetch="LAZY" />
    </attributes>
  </entity>

  <entity class="ioc.dam.m6.exemples.comandes.SeuClient "
              metadata-complete="true">
    <attributes>
      ...
      <many-to-one name="client"/>
      ...
    </attributes>
  </>

  <entity class="ioc.dam.m6.exemples.comandes.Sector "
              metadata-complete="true">
    <attributes>
      <id name="id">
        <column length="50" />
      </id>
      ...
    </attributes>
  </entity>

```

```
  ...

</entity_mappings>
```

Relacions ManyToMany

Des de la perspectiva relacional, les relacions*ManyToMany,* siguin bidireccionals o no,requereixen sempre una taula extra. Per defecte, com ja s'ha comentat, JPA sempre crea una taula extra en les relacions multivalents i en conseqüència, a banda del paràmetre `fetch`(per especificar la càrrega tardana) o del paràmetre `cascade`(per delimitar la responsabilitat de la persistència), no solen especificar-se altres paràmetres per a aquestes relacions.

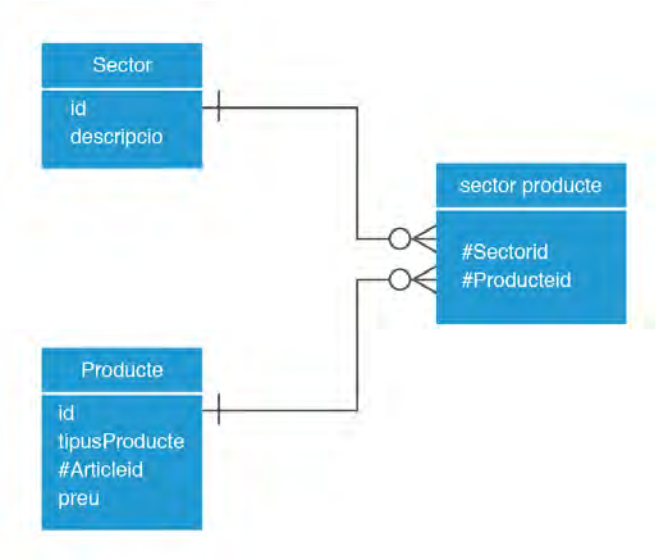
Un exemple de relació *ManyToMany*el podem trobar a la relació entre la classe `Sector` i la classe `Producte`.

```
@Entity
public class Sector implements Serializable {
    @Id
    private String id;
    private String descripcio;
    @ManyToMany(fetch= FetchType.LAZY, cascade= CascadeType.ALL)
    private List<Producte> productes=new ArrayList<Producte>();

  ...
}
```

A l'exemple que ens ocupa, la classe `Sector` es responsabilitza de gestionar tota la persistència de la relació d'acord amb el paràmetre `cascade` i durant la recuperació es farà servir una càrrega diferida.

Com podeu observar a la figura 2.10, els registres de la taula *sector_producte* són només el resultat de combinar un sector i un producte.

Figura 2.10. Esquema E-R de les entitats "Sector" i "Producte"



L'esquema relacional acostuma a mantenir-se, sigui quin sigui el tipus de

col·lecció que finalment implementem en el nostre model. Fins i tot en col·leccions de tipus `Map`,quan la clau formi part de l'entitat emmagatzemada com a valor.

En aquests casos JPA preveu l'anotació `MapKey`per indicar que la clau del `Map` forma part del valor. Prenguem la relació anterior entre `Sector` i `Producte` i modifiquem el tipus de col·lecció. Volem emmagatzemar els productes indexats pel seu propi *Id*.

```
@Entity
public class Sector implements Serializable {
    @Id
    @Column(length=50)
    private String id;
    private String descripcio;
    @ManyToMany(fetch= FetchType.LAZY, cascade= CascadeType.ALL)
    @MapKey(name="id")
    private Map<Long, Producte> productes =
          new HashMap<Long, Producte>();
  ...
}
```

`MapKey(name=‘‘id’’)` indica a JPA que les claus del `Map` productes són els atributs iddels seus valors i, per tant, no cal emmagatzemar cap valor extra a la taula sector_producte.

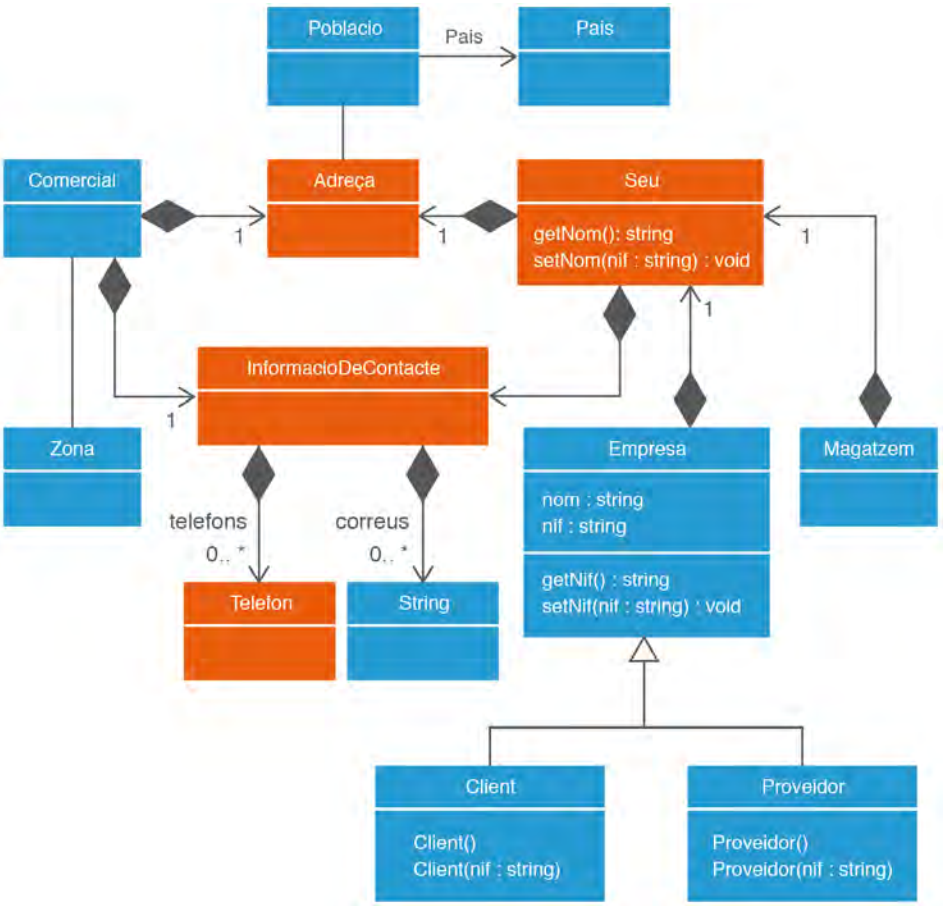
2.5.6 Objectes incrustats (Embedded Objects)

Anomenem objectes incrustats a aquells objectes no marcats com a entitats que estan continguts, directament o indirecta, en una entitat.

Els objectes incrustats tenen una dependència total amb l'entitat que els conté. Per això no tenen identificador. De fet, podríem dir que són una part de l'entitat a la qual pertanyen, però que per raons de disseny les seves dades s'han extret constituint-se en un objecte propi.

En l'exemple que venim arrossegant (aplicació de comandes) podem veure-hi diferents exemples d'objectes incrustats. Aquí analitzarem amb una mica més de detall tres d'aquests objectes: els objectes `Adreca`, els objectes `InformacioDeContacte` i els objectes `Seu` (figura 2.11). Els objectes `Adreca` són una abstracció que permet descriure qualsevol adreça física. D'aquesta manera, podem reutilitzar la classe en molt diverses situacions, els comercials tenen una adreça, però també la tenen els proveïdors, els clients o els nostres magatzems.

Figura 2.11. Diagrama de classes parcial de l'aplicació exemple



Hi podem distingir, de color taronja, les classes de tipus Embeddable.

En la mateixa línia, la classe `InformacioDeContacte` és també una abstracció que permet capsular totes les dades referents als mitjans de contacte, com són els telèfons (mòbils, fixos, fax, etc.) o els correus electrònics. Tal com passa amb la classe `Adreca`, podem extrapolar les dades emmagatzemades en els objectes de tipus `InformacioDeContacte` a comercials, proveïdors, clients o magatzems.

La classe `Seu`, d'altra banda, té la funció d'unificar el conjunt de dades compartides per qualsevol empresa o entitat: un nom, una adreça i les formes de contacte.

Podeu veure al diagrama de classes, en color taronja, aquelles que no són considerades entitats i que, per tant, caldrà tractar-les com objectes incrustats. Destacarem també l'existència de la classe `Empresa` com una generalització de `Client` i `Proveïdor`. Tractarem aquesta herència més tard, i de moment ens centrarem en els objectes incrustats.

La forma com informarem a JPA de l'existència d'objectes incrustats en el nostre model serà amb les marques *Embeddable* i *Embedded*. Qualificarem d'*Embeddable* les classes que generin objectes incrustats. Així, `Adreca`, `InformacioDeContacte` i `Seu` seran *Embeddadble*. Qualificarem *Embedded* els atributs que continguin objectes incrustats (és a dir, qualificats com a*Embeddable*).

Davant d'aquestes especificacions, JPA fusionarà tots els objectes incrustats amb l'entitat que els conté com si es tractés d'una sola classe. És a dir, per cada atribut,

per exemple d'`Adreca`,es generarà una columna addicional en cada una de les entitats que la continguin, per exemple en l'entitat `Comercial`. Les columnes *via*, *codipostal*, *poblacio* i *pais* són en realitat atributs de la classe `Adreca` (:Figure:Figura28:).

FIGURA 2.12. Taula Comercial

Internament, les classes qualificades amb *Embeddable* admeten qualsevol especificació de mapatge en cada un dels seus atributs, com si es tractés d'una entitat. L'avantatge és que, un cop mapada la classe, l'especificació servirà per a totes les entitats que la utilitzin.

Observeu la classe `Adreca`:

```
1  @Embeddable
2  public class Adreca implements Serializable {
3      private String via;
4      @Column(length=10)
5      private String codiPostal;
6      @ManyToOne
7      @JoinColumns({
8          @JoinColumn(name="POBLACIO", referencedColumnName="NOM"),
9          @JoinColumn(name="PAIS", referencedColumnName="NOM_PAIS")
10     })
11     private Poblacio poblacio;
```

La classe s'ha de qualificar d'*Embeddable* usant l'anotació, però els seus atributs es tracten de la mateixa manera que si fossin atributs d'una entitat. En primer lloc es limita la mida de *codipostal* i seguidament s'especifica la relació entre l'entitat que contingui `Adreca` (sigui la que sigui) i l'entitat `Poblacio`. Aquesta entitat necessita dues columnes com a clau forana, el nom de la població i el nom del país. A la taula *POBLACIO*, la columna que referencia el nom de la població s'anomena NOM, i la que referencia el nom del país s'anomena NOM_PAIS. Per evitar problemes hem decidit canviar els noms per uns de més significatius (POBLACIO i PAIS respectivament) fent servir les anotacions `JoinColumns` i `JoinColumn`. Com podeu constatar a la figura 2.12, es crearan quatre columnes extres: `via`, `codipostal`, `poblacio` i `pais`, d'acord amb les especificacions que es mostren a la classe `Adreca`.

La resta de la classe contindrà els mètodes d'accés als seus atributs i a la informació que se'n desprèn.

```
1  public Adreca() {
```

```
  }

  public Adreca(String via, String codiPostal, Poblacio poblacio) {
        this.via = via;
        this.codiPostal = codiPostal;
        this.poblacio = poblacio;
  }

  public String getVia() {
        return via;
  }

  public void setVia(String via) {
        this.via = via;
  }

  public String getCodiPostal() {
        return codiPostal;
  }

  public void setCodiPostal(String codiPostal) {
        this.codiPostal = codiPostal;
  }

  public Poblacio getPoblacio() {
        return poblacio;
  }

  public void setPoblacio(Poblacio poblacio) {
        this.poblacio = poblacio;
  }

  public String getNomPoblacio() {
        return poblacio.getNom();
  }

  public String getNomPais() {
        return poblacio.getPais().getNom();
  }
}
```

Tornem a la classe `Comercial`. Aquí veurem com s'especifiquen els objectes incrustats:

```
@Entity
public class Comercial implements Serializable {
    @Id
    @Column(length=15)
    private String nif;
    private String nom;

    @Embedded
    private Adreca adreca = new Adreca();
    @Embedded
    private InformacioDeContacte informacioDeContacte =
            new InformacioDeContacte();
    @OneToOne(fetch= FetchType.LAZY)
    private Zona zona;
```

Fixeu-vos que els atributs de tipus `Adreca` i `InformacioDeContacte` estan qualificats d'*Embedded*. Es tracta d'atributs fortament dependents i en el model ho hem plasmat definint atributs de només lectura (rang públic). Els accessors d'escriptura s'han declarat *protected* per possibilitar la manipulació de les classes gestores que se n'hagin de fer càrrec.

```
...

    public Adreca getAdreca() {
        return adreca;
    }

    protected void setAdreca(Adreca adreca) {
        this.adreca = adreca;
    }
  ...
```

Donat un objecte `Comercial`, caldria afegir el seu codi postal fent :

```
comercial.getAdreca().setCodiPostal("08573");
```

Les classes de tipus *Embeddable* poden estar contingudes de forma recurrent en altres de tipus també *Embedable*. Aquest és el cas de la classe `Seu`, la qual representa la seu d'una empresa o entitat i està composta d'un objecte `Adreca`i d'un altre de tipus `InformacioDeContacte`.

D'aquesta manera, l'entitat `Empresa` (i en darrer terme les classes hereves `Client` i `Proveidor`) o l'entitat `Magatzem`, contenidores totes elles d'un objecte `Seu`,tindran també una referència a una `Adreca` i a un objecte `InformacioDeContacte`. Això es reflectirà a les seves taules de forma idèntica a com s'ha reflectit a la taula comercial (vegeu la figura **??**).

{:fp:dam:m06:u2:u2_import_Image_28.png?453|}}

La classe `Seu` presenta el següent aspecte:

```
@Embeddable
public class Seu implements Serializable {
    private String nom;
    @Embedded
    private Adreca adreca = new Adreca();
    @Embedded
    private InformacioDeContacte informacioDeContacte=
            new InformacioDeContacte();

    public Adreca getAdreca() {
        return adreca;
    }

    protected void setAdreca(Adreca adreca) {
        this.adreca = adreca;
    }

    public InformacioDeContacte getInformacioDeContacte() {
        return informacioDeContacte;
    }

    protected void setInformacioDeContacte(
      InformacioDeContacte informacioDeContacte) {
        this.informacioDeContacte = informacioDeContacte;
    }

    public String getNom() {
        return nom;
    }

    public void setNom(String nom) {
        this.nom = nom;
    }
```

```
}
```

I l'entitat `Magatzem` només necessitarà un únic objecte incrustat de tipus `Seu`per disposar d'adreça i informació de contacte.

```
@Entity
public class Magatzem implements Serializable {
    @Id
    @Column(length=20)
    private String id;
    @Embedded
    @AttributeOverride(name="nom",
        column = @Column(name="DESCRIPCIO"))
    private Seu seu=new Seu();

  ...
}
```

Cal destacar que `AttributeOverride` permet modificar el nom com es maparà l'atribut d'un objecte incrustat. Fixeu-vos que hem decidit canviar el nom de l'atribut `nom` de la seu pel de `descripcio`.

Abans de passar al següent apartat mostrarem les equivalències XML:

```
<entity-mappings ...>

  ...

  <embeddadble class="ioc.dam.m6.exemples.comandes.Adreca "
    metadata-complete="true">
    <attributes>
      <basic name="codiPostal">
        <column length="10" />
      </basic>
      <many-to-one name="poblacio">
        <join-column name="POBLACIO"
          referenced-column-name="NOM"/>
        <join-column name="PAIS"
          referenced-column-name="NOM_PAIS"/>
      </many-to-one name>
    </attributes>
  </embeddadble>

  <entity class="ioc.dam.m6.exemples.comandes.Comercial "
    metadata-complete="true">
    <attributes>
      <id name="nif">
        <column length="15" />
      </id>
      <embedded name="adreca" />
      <embedded name="informacioDeContacte" />
      <one-to-one name="zona" fetch="LAZY" />
    </attributes>
  </entity>

  <embeddadble class="ioc.dam.m6.exemples.comandes.Seu "
    metadata-complete="true">
    <attributes>
      <embedded name="adreca" />
      <embedded name="informacioDeContacte" />
    </attributes>
  </embeddadble>

  <entity class="ioc.dam.m6.exemples.comandes.Magatzem "
    metadata-complete="true">
    <attributes>
```

```
43        <id name="id">
44          <column length="20" />
45        </id>
46        <embedded name="seu">
47          <attribute_override name="nom">
48            <column name="DESCRIPCIO" />
49          </attribute_override>
50        </embedded>
51      </attributes>
52    </entity>
53    ...
54
55  </entity_mappings>
```

2.5.7 Col·leccions d'objectes bàsics i objectes incrustats

Com ja hem vist, la forma de plasmar una relació multivalent entre dues entitats és implementant en una de les entitats (o en les dues) una col·lecció (vegeu la relació entre `Sector` i `Producte`).

Però en els models orientats a objectes sovint sorgeix la necessitat de definir col·leccions al marge de les relacions entre entitats. Si volem emmagatzemar per exemple tots els correus electrònics d'una persona, necessitarem contenir-los en un objecte col·lecció. Però una adreça electrònica és simplement una cadena de caràcters i sembla excessiu haver de gestionar-los com a entitats separades.

Des de la versió JPA 2.0 és possible mapar col·leccions de tipus bàsics, i fins i tot objectes de tipus *Embeddable* sense necessitat d'haver-los d'emmarcar artificialment en una relació entre entitats.

La principal diferència entre les col·leccions d'entitats (relacions multivaents) i la resta de col·leccions és que a les primeres, les dades de les entitats es guardaran a la seva taula específica, i, per tant, l'emmagatzematge de la col·lecció es limitarà tan sols a guardar la clau forana, ja sigui en una taula extra o a la de la pròpia entitat.

La resta de col·leccions, en canvi, sempre hauran de necessitar una taula extra i, a més de la clau forana a l'entitat, caldrà que incloguin tantes columnes com sigui necessari, d'acord amb el tipus de dades incrustat.

La versió 2.0 incorpora la marca anomenada `ElementCollection` per indicar que la col·lecció marcada no és d'entitats, sinó de tipus bàsics o *Embeddable*. Recuperarem l'exemple de la classe `InformacioDeContacte` compartida per comercials i empreses com els clients o proveïdors.

No hi ha pràcticament diferència entre el tractament de tipus bàsics i el d'objectes *Embeddable*, per això d'ara endavant tot el que afirmem per als tipus bàsics valdrà també per als *Embeddable*.

La classe gestiona entre d'altres un conjunt de correus electrònics que tracta com a tipus bàsics.

```
@ElementCollection(fetch= FetchType.LAZY)
private List<String> correusElectronics = new ArrayList<String>();
```

Fixeu-vos que l'anotació `ElementCollection` admet també alguns dels paràmetres usats en les relacions, com ara `fetch` per condicionar la càrrega diferida de la col·lecció. Òbviament, si eliminem el paràmetre o li donem un valor `FetchType.EAGER`, la càrrega es produirà sempre de forma immediata.

La classe fa servir una llista de cadenes, la posició de les quals serveix també com a criteri de prioritat a l'hora de mostrar tots els correus. A aquest efecte, `InformacioDeContacte`disposa d'un conjunt d'utilitats per gestionar els correus i la seva prioritat.

```
public void afegirCorreuElectronic(String correu){
    correusElectronics.add(correu);
}

public void eliminaCorreuElectronic(String correu){
    correusElectronics.remove(correu);
}

public Iterator<String> getCorreusElectronics(){
    return correusElectronics.iterator();
}

public void incrementaPrioritatCorreuElectronic(int pos){
    if(pos>=1){
        String aux = correusElectronics.get(pos-1);
        correusElectronics.set(pos-1, correusElectronics.get(pos));
        correusElectronics.set(pos, aux);
    }
}

public void decrementaPrioritatCorreuElectronic(int pos){
    if(pos<correusElectronics.size()){
        String aux = correusElectronics.get(pos+1);
        correusElectronics.set(pos+1, correusElectronics.get(pos));
        correusElectronics.set(pos, aux);
    }
}
```

En la majoria de sistemes gestors, però, no existeix garantia de recuperar les dades en el mateix ordre en què s'han introduït, i, per tant, si no fem res més l'ordre es perdrà probablement després de la primera recuperació d'elements.

JPA disposa també d'una marca per corregir-ho. Es tracta de l'anotació `OrderColumn`. Aquesta anotació és específica per a col·leccions de tipus llista quan la posició tingui un paper fonamental durant la recuperació de les dades.

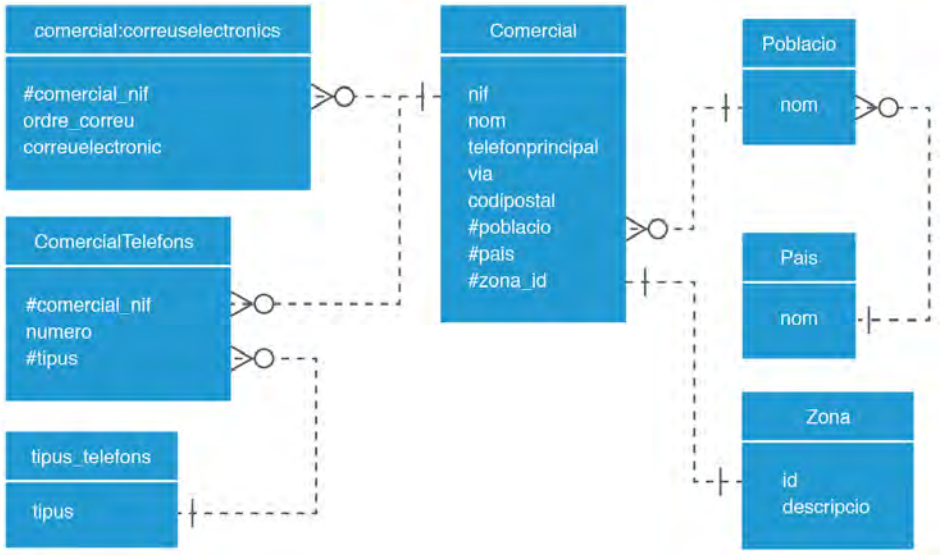
L'anotació `OrderColumn` indica a JPA que a la taula on s'emmagatzemi la col·lecció hi haurà una columna de més on s'emmagatzemarà l'ordre de la col·lecció, amb un valor de tipus numèric.

```
@ElementCollection(fetch= FetchType.LAZY)
@OrderColumn(name="ORDRE_CORREU")
private List<String> correusElectronics = new ArrayList<String>();
```

La taula generada per JPA inclourà la columna *ORDRE_CORREU*. Vegeu l'esquema entitat-relació de `Comercial` (o la de client mostrada també anteriorment).

Recordeu que `InformacióDeContacte`és una classe de tipus *Embeddable* i, per tant, totes les seves anotacions es traslladen íntegrament a les entitats que la contenen. Aquesta és la raó per la qual, al model E-R, els efectes es reflecteixen directament sobre la taula comercial en comptes de generar una taula `InformacioDeContacte`(figura 2.13).

Figura 2.13. Esquema E-R de l'entitat Comercial i totes les seves relacions



Abans de continuar analitzant la següent col·lecció de la classe `InformacioDeContacte`, farem un petit incís per descobrir algunes de les característiques d'un tipus de col·leccions singulars anomenades MAP.

Les col·leccions de tipus `Map` presenten la particularitat de mantenir associats dos objectes. Al primer se l'anomena `clau` i al segon, `valor`. Els objectes `Map` són capaços d'obtenir de forma molt eficient l'objecte valor associat a cada una de les claus contingudes a la col·lecció. D'aquesta manera, la clau agafa les connotacions d'índex de referència del valor associat, com si d'un vector es tractés, però amb la singularitat que l'índex pot ser de qualsevol tipus, no només numèric.

La versió 2.0 de JPA permet gestionar totes les combinacions possibles en cada un dels elements de `Map`. És possible gestionar dues entitats diferents, una actuant com a clau i l'altra com a valor. És possible emmagatzemar una única entitat indexada per algun dels seus valors (habitualment la clau primària). També és possible emmagatzemar claus i valors de tipus bàsic, i fins i tot valors bàsics indexats per entitats.

En els següents exemples il·lustren aquesta afirmació:

- `Map` amb claus i valors de tipus bàsic o *Embeddable*
- `Map` amb la clau de tipus *entitat* i valor de tipus bàsic o *Embeddable*
- `Map` amb valor de tipus entitat

Map amb claus i valors de tipus bàsic o Embeddable

Comencem per la col·lecció de telèfons de la classe `InformacioDeContacte`, que hauria de mantenir associada la informació del número i el tipus de telèfon.

Desitgem emmagatzemar-los en una col·lecció de tipus `Map` per controlar quins números de telèfon ja són dins de la col·lecció. Concretament, el número ens servirà d'índex i els valors seran cadenes indicant el tipus de telèfon.

En tractar-se d'una col·lecció `Map` de tipus bàsics, a més d'especificar que no es tracta d'una relació (usant l'anotació `ElementCollection`) cal indicar el nom de la columna que volem fer servir de clau i el nom de la columna que volem fer servir de valor. Usarem respectivament `MapKeyColumn` i `Column` per especificar-ho:

```
1 @ElementCollection(fetch= FetchType.LAZY)
2 @MapKeyColumn(name="numero")
3 @Column(name="tipus")
4 private Map<String, String> telefons= new HashMap<String, String>();
```

Una alternativa a l'especificació anterior consistiria en fer servir un objecte `Telefon` com a valor. En aquest cas, no caldria especificar el nom de la/es columna/es corresponent al valor, ja que per defecte s'agafarien els noms dels atributs de la classe `Telefon`.

```
1 @ElementCollection(fetch= FetchType.LAZY)
2 @MapKeyColumn(name="numero_id")
3 private Map<String, Telefon> telefons= new HashMap<String, Telefon>();
```

El principal problema que presenta aquesta alternativa és que se'n duplica el número de telèfon (com a clau i com a atribut de l'objecte `Telefon`). De moment, JPA no és capaç d'indicar que la clau usada forma part de l'estat de l'objecte valor, i per tant cal canviar el nom de la columna clau per evitar conflictes. Tot i això, si la duplicitat és mínima, no hauríem pas de considerar-la una solució dolenta.

Map amb la clau de tipus entitat i valor de tipus bàsic o Embeddable

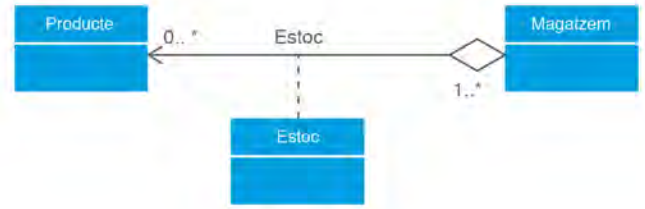
El que determina si cal tractar una col·lecció de tipus `Map` com una relació o com simples elements d'una col·lecció són els valors. És a dir, malgrat que la clau sigui una *entitat*, si els valors no ho són, caldrà considerar el `Map`com una simple col·lecció d'objectes especificant la notació `ElementCollection`.

Tot i això, aquest cas presenta algunes particularitats pel fet de fer servir una entitat com a clau. D'entrada, la clau es maparà sempre fent servir estrictament els seus valors de la clau primària.No cal especificar el nom de la columna on s'emmagatzemarà la clau, ja que per defecte agafarà el nom de l'atribut de l'entitat. Malgrat tot, quan faci falta indicar el nom de la columna, per exemple, si volem sobreescriure el nom de l'atribut degut a problemes de solapament de noms, caldrà especificar-la usant la notació `MapKeyJoinColumn`, ja que es tractarà d'una columna que farà de clau forana de l'entitat corresponent.

Il·lustrarem aquest cas per mitjà de la classe `Magatzem`, que conté una col·lecció de tipus `Map` amb l'estoc de cada producte ubicat en algun lloc del magatzem

propietari (figura 2.14). Esquemàticament, el model es podria representar com una classe associativa entre el `Magatzem` i el `Producte`.

Figura 2.14. Diagrama de classe de l'estoc de productes d'un magatzem



Per implementar-ho en Java proposem d'instanciar una col·lecció de tipus `Map` on el producte faci de clau i l'estoc de valor. El producte és una entitat complexa que analitzarem més endavant, però a l'efecte del que ens interessa cal comentar que conté un atribut identificador de tipus `Long`.

```
@Entity
public class Producte implements Serializable {
    @Id
    @GeneratedValue(strategy= GenerationType.TABLE)
    private Long id;
  ...
}
```

Com a valor ens interessarà guardar la quantitat de producte existent en estoc en el magatzem propietari del `Map` i també el lloc del magatzem on es troba ubicat el producte. Imaginarem que els magatzems es troben dividits en seccions identificades per una cadena de caràcter.

```
@Embeddable
public class Estoc implements Serializable {
    private static final long serialVersionUID = 1L;
    private String ubicacio;
    private Double quantitat=0.0;
  ..
}
```

La classe `Magatzem` ja l'hem analitzada parcialment quan hem vist la funció de la classe `Seu`. Ara acabarem d'analitzar-la amb la col·lecció que estem proposant:

```
@Entity
public class Magatzem implements Serializable {
    @Id
    @Column(length=20)
    private String id;
    @Embedded
    @AttributeOverride(name="nom",
         column=@Column(name="DESCRIPCIO"))
    private Seu seu=new Seu();

    @ElementCollection
    private Map<Producte, Estoc> estoc =
          new HashMap<Producte, Estoc>();
  ...
}
```

Com que l'atribut identificador del producte es diu *id*, decidim canviar-li el nom per un altre de més significatiu fent servir `MapKeyJoinColumn`, que com podeu intuir fa referència a la columna que serà clau forana de l'entitat que sigui clau en el `Map`; per a nosaltres, la classe `Producte`. També volem canviar els noms dels atributs de la classe `Estoc` fent servir `AttributeOverride`.

```
@Entity
public class Magatzem implements Serializable {
    @Id
    @Column(length=20)
    private String id;
    @Embedded
    @AttributeOverride(name="nom",
         column=@Column(name="DESCRIPCIO"))
    private Seu seu=new Seu();

    @ElementCollection
    @MapKeyJoinColumn(name="PRODUCTE_ID")
    @AttributeOverrides({
        @AttributeOverride(name="value.ubicacio",
        column=@Column(name="ZONA_MAGATZEM")),
        @AttributeOverride(name="value.quantitat",
        column=@Column(name="ESTOC"))
    })
    private Map<Producte, Estoc> estoc =
        new HashMap<Producte, Estoc>();
```

Fixeu-vos que per referenciar el nom de l'atribut que cal sobreescriure s'had'anteposar el prefix *value,* perquè fa referència a un atribut dels objectes valor. Si el canvi s'hagués hagut de fer als objectes clau del `Map`, hauríem d'haver anteposat el prefix *key*.

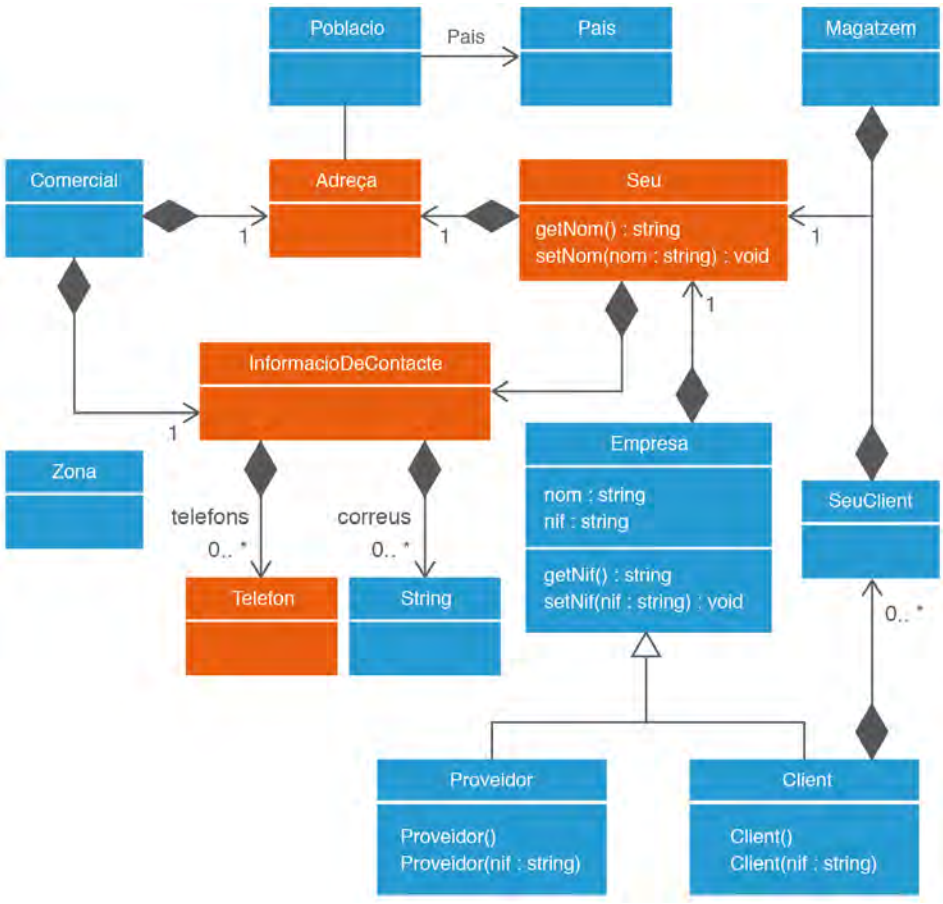
Map amb valor de tipus entitat

Finalment, per acabar de mostrar la flexibilitat de JPA reprendrem l'exemple de la relació entre `Client` i `SeuClient`.

Malgrat que la implementació de les herències fent servir JPA l'estudiarem més endavant, aquí cal adonar-se que tant `Proveidor` com `Client` són una `Empresa` i com a tal, tots ells tindran una `Seu`. Quelcom de semblant passa amb `Magatzem`, ja que també està vinculat amb `Seu`. Recordem que la classe `Seu` és un tipus *Embeddable* que conté un nom, una adreça i un objecte `InformacioDeContacte`.

Però a més, la classe client permet emmagatzemar un conjunt de seus. S'ha considerat que en la relació comercial d'una empresa és important conèixer les diferents seus que cada client pugui tenir. Per això, a banda de les dades heretades a través de l'empresa i que constituirien el que podem anomenar la seu central del client, el model incorpora també una col·lecció on poder guardar un nombre indeterminat de seus (una sucursal, un magatzem, una botiga, unes oficines, una fàbrica, etc.). Els elements de la col·lecció són del tipus `SeuClient` (figura 2.15). Aquesta classe té rang d'entitat. Es tracta d'una entitat dèbil, ja que depèn totalment de `Client`. Com a entitat contindrà una clau primària i s'emmagatzemarà en una taula específica.

Figura 2.15. Diagrama de classes de l'aplicació comercial



Hi apareix ressaltat l'ús d'objectes incrustats (embedded) en color taronja.

Desitgem poder cercar les seus de cada client segons el nom que li donem a la seu. És a dir, desitgem poder saber l'adreça de la seu de València del client X o potser on es troba la fàbrica del mateix client .

Per aconseguir-ho, decidim modificar la implementació de la col·lecció que ja havíem dissenyat canviant el conjunt de seus per un `Map` de `Seus` indexades per nom. El nom està contingut dins de `SeuClient`. Per tant, ens trobem davant d'un cas en què el valor del `Map` és una entitat i la clau forma part de l'estat dels objectes valor.

EL problema és que la informació que usarem d'índex no és un atribut directe de `SeuClient`, sinó que és un atribut d'un atribut. Per solucionar aquest problema tenim dues solucions. La més senzilla consisteix a especificar que es tracta d'un atribut indirecte usant la sintaxi pròpia de l'orientació a objectes. És a dir, encadenant atributs fent servir un punt com a separador. Vegem-ho:

```
1  @Entity
2  public class Client extends Empresa {
3      @OneToMany(mappedBy="client", cascade={CascadeType.ALL},
4                 fetch=FetchType.LAZY)
5      @MapKey(name="seu.nom")
6      private Map<String, SeuClient>seus=new HashMap<String, SeuClient>();
7
8      @ManyToOne(fetch= FetchType.LAZY)
9      private Zona zona;
10
11     @OneToOne(fetch= FetchType.LAZY)
```

```
    private Sector sector=null;
  ...
}
```

Una altra solució requereix una mica més de feina, però aporta també molta més flexibilitat. Es tracta d'anul·lar l'accés a la informació a través dels atributs i fer servir exclusivament els accessors (*gets* i *sets*) com a via d'obtenció i manipulació de la informació. És el que s'anomena *accés a les propietats*.

Aquesta tècnica permet crear propietats virtuals (sense un atribut que suporti la informació) i també amagar atributs eliminant els seus accessors. A l'apartat d'identificadors compostos, hem escollit aquesta solució per implementar la classe `SeuClient`.

```
@Entity
@Access(AccessType.PROPERTY)
public class SeuClient implements Serializable{
    private Seu seu = new Seu();
    private Client client;
```

L'anotació `Access` indica que en aquesta classe l'accés no es farà per atributs, sinó per *propietats*. Quan s'escull aquest accés, les anotacions cal situar-les en els mètodes *accessors* en comptes dels atributs.

```
public SeuClient() {
    }

    public SeuClient(String nom, Client client) {
        this.seu.setNom(nom);
        this.client = client;
    }

    public SeuClient(String nom) {
        this.seu.setNom(nom);
    }
```

Els constructors de la classe no es veuen afectats pel canvi. Però a partir d'aquí crearem les propietats `nom`, `adreca` i `informacioDeContacte` com a camps calculats. Tots ells referencien la informació corresponent continguda a l'atribut real anomenat `seu`.

```
public String getNom() {
        return seu.getNom();
    }
```

Les propietats acostumen a anotar-se en els accessors de lectura com si es tractés d'un atribut real.

```
@Id
    @ManyToOne
    public Client getClient() {
        return client;
    }
```

De la mateixa manera que haguéssim fet si existís un atribut de tipus `Adreca`, cal indicar que es tracta d'un objecte incrustat.

```
@Embedded
    public Adreca getAdreca() {
        return seu.getAdreca();
    }

    @Embedded
    public InformacioDeContacte getInformacioDeContacte() {
        return seu.getInformacioDeContacte();
    }
```

Acabats els accessors de lectura, escriurem els d'escriptura sense cap anotació. D'altra banda, fixeu-vos també que l'atribut real `seu`quedarà inaccessible, perquè no hem creat cap accessor a la seva informació.

```
protected void setAdreca(Adreca adreca) {
        this.seu.setAdreca(adreca);
    }

    protected void setInformacioDeContacte(InformacioDeContacte
                informacioDeContacte) {
        this.seu.setInformacioDeContacte(informacioDeContacte);
    }

    public void setNom(String nom){
        seu.setNom(nom);
    }

    public void setClient(Client client) {
        this.client = client;
    }
}
```

La representació a les taules del'SGBD serà idèntica en tots els casos, però així tenim més flexibilitat i disposem de processament extra per referenciar les seus a partir del nom que les identifiqui.

Format XML

Anem aquí a posar l'equivalència de totes les anotacions especificades en aquest apartat:

```
<entity-mappings ...>

  ...

     <embeddadble class =
       "ioc.dam.m6.exemples.comandes.InformacioDeContacte "
      metadata-complete = "true">
    <attributes>
      <basic name="telefonPrincipal">
        <column length="20" />
      </basic>
      <element-collection name="telefons" fetch"LAZY">
        <map-key-columns name="numero" />
        <column name="tipus"/>
      </element-collection >
      <element-collection name="correusElectronics"
                fetch"LAZY">
        <order-column name="ordre_correu"/>
      </element-collection >
    </attributes>
  </embeddadble>
```

```
22
23  <entity class="ioc.dam.m6.exemples.comandes.Magatzem "
24    metadata-complete="true">
25    <attributes>
26      <id name="id">
27        <column length="20" />
28      </id>
29      <embedded name="seu">
30        <attribute-override name="nom">
31          <column name="DESCRIPCIO" />
32        </attribute-override>
33      </embedded>
34      <element-collection name="telefons" fetch"LAZY">
35        <map-key-columns name="producte_id" />
36        <attribute-override
37             name="value.ubicacio">
38          <column name="zona_magatzem" />
39        </attribute-override>
40        <attribute-override
41             name="value.quantitat">
42          <column name="estoc" />
43        </attribute-override>
44      </element-collection >
45    </attributes>
46  </entity>
47
48  <entity class="ioc.dam.m6.exemples.comandes.Producte "
49    metadata-complete="true">
50    <attributes>
51      <id name="id">
52        <column length="20" />
53        <table-generator />
54      </id>
55      <embedded name="seu">
56        <attribute-override name="nom">
57          <column name="DESCRIPCIO" />
58        </attribute-override>
59      </embedded>
60    </attributes>
61  </entity>
62
63  <entity class="ioc.dam.m6.exemples.comandes.SeuClient"
64    metadata-complete="true" access="PROPERTY">
65    <attributes>
66      <id name="nom" />
67      <many-to-one name="client" id="true" />
68      <embedded name="adreca" />
69      <embedded name="informacioDeContacte" />
70    </attributes>
71  </entity>
72  <embeddadble class="ioc.dam.m6.exemples.comandes.Estoc " />
73
74       <embeddadble class="ioc.dam.m6.exemples.comandes.Adreca "
75    metadata-complete="true">
76    <attributes>
77      <basic name="codiPostal">
78        <column length="10" />
79      </basic>
80      <many-to-one name="poblacio">
81        <join-column name="POBLACIO"
82          referenced-column-name="NOM"/>
83        <join-column name="PAIS"
84          referenced-column-name="NOM_PAIS"/>
85      </many-to-one name>
86    </attributes>
87       </embeddadble>
88
89  <entity class="ioc.dam.m6.exemples.comandes.Comercial "
90    metadata-complete="true">
91    <attributes>
```

```
92        <id name="nif">
93          <column length="15" />
94        </id>
95        <embedded name="adreca" />
96        <embedded name="informacioDeContacte" />
97        <one-to-one name="zona" fetch="LAZY" />
98      </attributes>
99    </entity>
100
101   <embeddadble class="ioc.dam.m6.exemples.comandes.Seu "
102     metadata-complete="true">
103     <attributes>
104       <embedded name="adreca" />
105       <embedded name="informacioDeContacte" />
106     </attributes>
107   </embeddadble>
108
109   ...
110
111 </entity-mappings>
```

2.5.8 Identificadors compostos

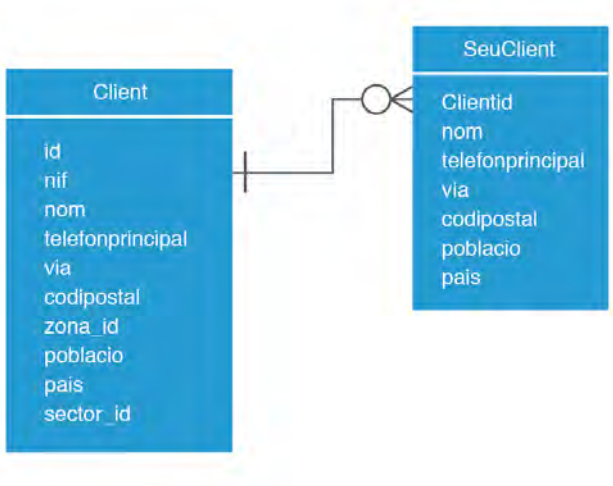
Les claus primàries compostes per diversos camps són força habituals en la majoria d'aplicacions. En aquest apartat veurem les dues respostes que ofereix JPA per tractar aquelles entitats que requereixin identificadors compostos.

Id Class

La forma més bàsica d'implementar identificadors compostos consisteix a implementar una classe auxiliar que convindrem a anomenar *id class*. Aquesta haurà de ser un fidel reflex de la classe `entitat` a la qual auxiliarà en referència als camps o propietats que componguin la clau primària.

Prenguem com a cas l'exemple ja conegut de `SeuClient`, que es tracta d'una entitat dèbil amb una clau primària derivada de client. Això significa que l'atribut client de la classe `SeuClient`, a més de ser clau forana a la taula Client, també formarà part de la clau primària de l'entitat `SeuClient` (vegeu la figura 2.16).

Figura 2.16. Esquema E-R de la taula Client i la taula SeuClient



Ara ja podem desvetllar la raó que ens va fer decidir canalitzar l'accés a la informació de `SeuClient` a través de propietats en comptes de fer servir els seus atributs com havíem fet amb totes les altres.

Com us podeu imaginar, necessitem que el nom de la seu formi part de la clau primària (conjuntament amb el client), però la classe `SeuClient` no disposa de cap atribut directe amb aquesta informació. Una manera d'aconseguir-ho consisteix a crear una propietat, fent servir els accessors, anomenada nom que obtingui la informació de l'atribut `seu`.

La *id class* corresponent haurà de disposar de dues propietats, una anomenada `client` i l'altra anomenada `nom`, per tal de reflectir (*entitat* i *id class)* la mateixa forma d'accedir i manipular la informació específica de la clau composta.

En el cas que ens ocupa, la *id class* s'anomenarà `SeuClientId` i com a mínim caldrà implementar els mètodes `getNom` i `setNom`com a accessors de la propietat nom, i `getClient` i `setClient`com a accessors de la propietat `client`.

Fixeu-vos que en aquest cas no importa que es configurin amb diferents atributs, ja que l'accés es realitzarà via propietats.

```
public class SeuClientId implements Serializable{
    private String nom;
    private Client client;

    public SeuClientId() {
    }

    public SeuClientId(String nom, Client client) {
        this.nom = nom;
        this.client = client;
    }

    public String getNom() {
        return nom;
    }

    public Client getClient() {
        return client;
    }

    protected void setNom(String nom) {
        this.nom = nom;
    }

    protected void setClient(Client client) {
        this.client = client;
    }
}
```

És clar que si l'accés fóra via atributs, ambdues classes (l'*entitat* i la seva *id class*) s'haurien de correspondre amb exactament els mateixos atributs que componguessin la clau primària.

Per tal que l'entitat reconegui quina classe li fa d'*id class* s'haurà d'indicar fent servir l'anotació `IdClass`:

```
@Entity @Access(AccessType.PROPERTY)
@IdClass(SeuClientId.class)
public class SeuClient implements Serializable{
```

```
  ...
}
```

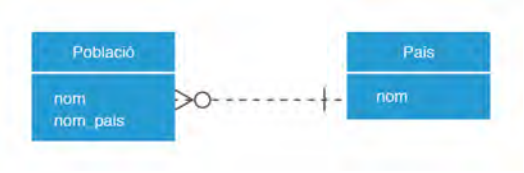
I si fem servir el format XML:

```
...
<entity class="ioc.dam.m6.exemples.comandes.SeuClient"
    metadata_complete="true" access="PROPERTY">
  <id_class class="ioc.dam.m6.exemples.comandes.SeuClientId"/>
  <attributes>
    <id name="nom" />
    <many_to_one name="client" id="true" />
    <embedded name="adreca" />
    <embedded name="informacioDeContacte" />
  </attributes>
</entity>
```

La segona solució consisteix a usar un objecte incrustat com a identificador de l'entitat. Usarem la classe `Població` per exemplificar el seu ús.

La classe `Població`té una clau forana a `País` que, conjuntament amb el nom de la població, defineix la seva clau primària tal com es pot veure a la figura 2.17.

Figura 2.17. Esquema E-R de les taules Població i País



Fent servir la segona tecnologia, la classe `població`, en comptes de tenir dos atributs clau en tindrà només un, però constituït d'un únic objecte incrustat, el qual serà el que disposarà dels dos atributs requerits. Anomenarem aquesta classe `PoblacióId`.

```
@Embeddable
public class PoblacioId implements Serializable{
    @Column(length=100)
    private String nom;

    @ManyToOne
    @JoinColumn(name="NOM_PAIS", referencedColumnName="NOM")
    private Pais pais;


    public PoblacioId() {
    }

    public PoblacioId(Poblacio poblacio){
        this.nom = poblacio.getNom();
        this.pais = poblacio.getPais();
    }

    public PoblacioId(String nom, Pais pais) {
        this.nom = nom;
        this.pais = pais;
    }

    public String getNom() {
        return nom;
    }
```

```
27
28      protected void setNom(String nom) {
29          this.nom = nom;
30      }
31
32      public Pais getPais() {
33          return pais;
34      }
35
36      public void setPais(Pais pais) {
37          this.pais = pais;
38      }
39
40      public String getNomPais() {
41          return getPais().getNom();
42      }
43  }
```

La classe `Pais` disposa d'una clau primària bàsica, el nom del país, i no requereix cap tractament especial a banda de marcar l'identificador.

```
1   @Entity
2   public class Pais implements Serializable {
3       private static final long serialVersionUID = 1L;
4       @Id
5       @Column(length=100)
6       private String nom;
7     ...
8   }
```

Finalment, la classe `Població` contindrà un objecte `PoblacióId` que simplement caldrà identificar com a *EmbeddedId*.

```
1   @Entity
2   public class Poblacio implements Serializable {
3       private static final long serialVersionUID = 1L;
4       @EmbeddedId
5       private PoblacioId id;
6
7       public Poblacio() {
8       }
9
10      public Poblacio(PoblacioId id) {
11          this.id = id;
12      }
13
14      public Poblacio(String nom, Pais pais) {
15          id = new PoblacioId(nom, pais);
16      }
17
18      public String getNom() {
19          return id.getNom();
20      }
21
22      public void setNom(String id) {
23          this.id.setNom(id);
24      }
25
26      public Pais getPais() {
27          return id.getPais();
28      }
29
30      protected void setPais(Pais pais) {
31          this.id.setPais(pais);
32      }
33
34      private String getNomPais() {
```

```
35          return id.getPais().getNom();
36      }
37  }
```

A continuació mostrarem les notacions en format XML:

```
<entity-mappings ...>

  ...

     <embeddadble class="ioc.dam.m6.exemples.comandes.PoblacioId "
    metadata-complete="true">
    <attributes>
      <basic name="nom">
        <column length="100" />
      </basic>
      <many-to-one name="pais">
        <join-column name="NOM_PAIS"
          referenced-column-name="NOM/>
      </many-to-one>
    </attributes>
  </embeddadble>

  <entity class="ioc.dam.m6.exemples.comandes.Poblacio "
    metadata-complete="true">
    <attributes>
      <embedded-id name="id"/>
    </attributes>
  </entity>
  ...

</entity-mappings>
```

Com podeu veure, ambdós mètodes són fàcils d'implementar. Malgrat tot, cal tenir molt en compte que a vegades es donen situacions complexes, com per exemple claus compostes de classes que a la seva vegada tenen objectes incrustats, o d'altres circumstàncies que poden provocar errors implementant una o altra metodologia. És per això que aconsellem que si una us dóna error proveu amb l'altra.

Hi ha encara una alternativa que dóna una mica més de feina però que acostuma a ser força estable en la majoria de situacions. La implementarem també sobre la mateixa classe població.

La tècnica consisteix a separar sempre les claus primàries de les claus foranes. Com podem aconseguir-ho sense haver de construir relacions artificials que podrien complicar el model orientat a l'objecte?

Els manuals de JPA proposen duplicar els atributs que siguin claus foranes i que componguin la clau primària. Els atributs duplicats s'implementaran sempre de tipus primitiu d'acord amb els valors que hagin de representar. Vegem-ho amb l'exemple de la població. Aquesta classe conté una clau forana per enllaçar amb un `Pais`. En últim terme, la clau forana de `Pais`és de tipus String. Doncs caldrà duplicar l'atribut que representi el país, però en un estarà representat per un String (la seva clau primària, que en aquest cas és el nom del país) i en l'altre, per l'objecte de tipus `Pais`.

```
@Entity
@IdClass(value=PoblacioId.class)
```

Cal tenir en compte que aquesta tecnologia només es pot usar amb la metodologia *id class* explicada inicialment.

L'atribut nom (de la població) no correspon a cap clau forana, i per tant no es duplica. Només es marca com a component del'identificador.

```
public class Poblacio implements Serializable {
    @Id
    @Column(length=100)
    private String nom;
    @Id
    @Column(name="NOM_PAIS", length=100)
    private String nomPais;
```

L'atribut `nomPais`s'afegeix per tal de poder referenciar el nom del país com a clau primària a partir d'un tipus primitiu.

```
@ManyToOne
    @JoinColumn(name="NOM_PAIS", referencedColumnName="NOM",
          insertable=false, updatable=false)
    private Pais pais;
```

Finalment, l'atribut país, que és el que relacionarà la població amb el país corresponent (clau forana), s'especificarà com a tal però sense indicar que forma part de la clau primària. Com que en realitat a la taula del model E-R ambdós atributs correspondran al mateix camp, cal indicar-ho fent servir anotacions. Fixeu-vos que a `nomPais` hem indicat que la columna on es maparà s'anomena *NOM_PAIS* i en l'atribut `pais`l'anotació `JoinColumn` indica també que el nom de la columna és coincident (*NOM_PAIS*).

Per evitar el problema que JPA es queixi que dos atributs intenten escriure sobre una mateixa columna de la mateixa taula, cal anul·lar qualsevol intent d'emmagatzematge d'un d'ells. És a dir, deixarem que l'atribut `nomPais`sigui realment qui aporti la informació a la taula durant les insercions i modificacions, mentre que l'atribut país només tindrà vigència a l'hora de recuperar les dades. Per aconseguir-ho especificarem a l'anotació `JoinColumn` que es tracta d'una columna de només lectura posant els paràmetres *insertable* i *updatable* a fals.

Un últim detall: és important que si opteu per aquesta solució afegiu codificació per assegurar que el valor dels dos atributs serà sempre coincident.

```
public Poblacio() {
    }

    public Poblacio(String nom, Pais pais) {
        this.nom = nom;
        this.pais = pais;
        this.nomPais=pais.getNom();
    }

    public String getNom() {
        return nom;
    }

    public void setNom(String id) {
        this.nom = id;
    }

    public Pais getPais() {
```

```
19          return pais;
20      }
21
22      protected void setPais(Pais pais) {
23          this.pais = pais;
24          this.nomPais=pais.getNom();
25      }
26  }
```

La classe `PoblacioId`farà d'*id class,* i d'acord amb el que hem dit, haurà de reflectir exactament tots els atributs que siguin clau primària de població, atès que en aquest cas l'accés es fa via atributs.

```
1   public class PoblacioId implements Serializable{
2       private String nom;
3       private String nomPais;
4
5       public PoblacioId() {
6       }
7
8       public PoblacioId(Poblacio poblacio){
9           this.nom = poblacio.getNom();
10          this.nomPais = poblacio.getPais().getNom();
11      }
12
13      public PoblacioId(String nom, Pais pais) {
14          this.nom = nom;
15          this.nomPais = pais.getNom();
16      }
17
18      public PoblacioId(String nom, String nomPais) {
19          this.nom = nom;
20          this.nomPais = nomPais;
21      }
22  }
```

2.5.9 Herència

És força comú, en les aplicacions orientades a objecte, treballar amb models que apliquin relacions d'herència entre algunes de les seves classes. Aquí estudiarem els diferents tipus de classes que intervenen en una jerarquia, i quines estratègies ofereix JPA per poder plasmar les herències en un conjunt de taules.

Davant d'una jerarquia ens haurem de plantejar per cada classe si es tracta d'una entitat o només d'una classe intermèdia de la jerarquia. Val a dir que la situació de la classe dins la jerarquia no té res a veure amb aquesta classificació. Les entitats poden situar-se en qualsevol punt de la jerarquia: l'arrel, les posicions intermèdies i, per descomptat, les fulles.

Concepte d'entitats en una jerarquia

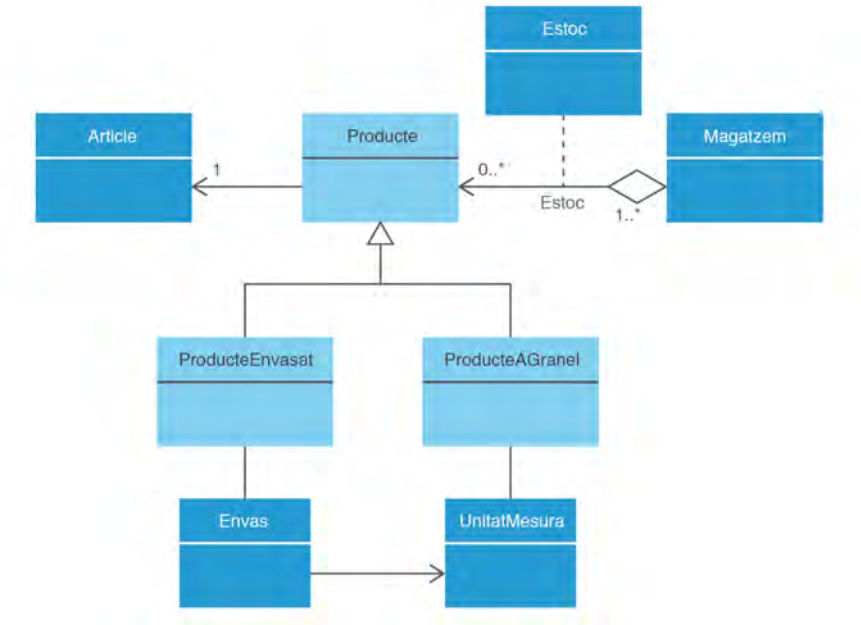
El fet de considerar les classes entitats o no, té més aviat a veure amb l'ús que d'elles se'n faci al model. Voldria insistir que parlem d'ús i no pas d'estructura jeràrquica. És a dir, podria donar-se el cas que una mateixa jerarquia situada en

models diferents considerés de diferent manera les classes d'aquesta jerarquia. El que per a un model podria ser una classe clarament candidata a entitat, per a l'altre podria prendre una consideració diferent.

Anem a veure un exemple intentant clarificar aquests conceptes. A l'aplicació de comandes que ens serveix d'exemple transversal hi trobem dues jerarquies, la d'empreses i la de productes.

La jerarquia de productes permet representar diversos tipus de productes específics amb la idiosincràsia de cada un d'ells (figura 2.18).

Figura 2.18. Diagrama de classes de la jerarquia de productes i classes relacionades



El model interpreta que els productes són articles de consum elaborats i presentats d'alguna forma per a ser venuts. Així, per exemple, un mateix article podria presentar-se envasat de diferents maneres. Cada producte envasat tindria un preu diferent i tot i tractar-se del mateix article, s'haurà de considerar un producte diferent. Per exemple, el paquet d'arròs de mig quilo no pot considerar-se el mateix que el sac d'arròs de 10 quilos. Segur que el sac d'arròs no és 20 vegades més car i, per tant, s'ha de tractar com un article independent. Aquest tipus de productes estarien representats per la classe `ProducteEnvasat`, la qual es troba relacionada am un envàs específic d'una capacitat concreta.

Alguns productes, però, no es venen pas envasats, sinó a granel. La venda de productes a granel precisa conèixer quina unitat de mesura es fa servir a l'hora de mesurar la quantitat de producte. El preu d'aquest producte es fixa d'acord amb una unitat de la mesura establerta per a cada producte concret. Així, per exemple, podem comprar taronges o carn per quilos, però si el que volem comprar és vi a granel caldrà comprar-lo per litres, i si fos cable elèctric el compraríem per metres. La classe que representa aquest tipus de producte és `ProducteAGranel`, la qual es troba associada a la unitat de mesura del producte representat.

Finalment, hi ha productes que es venen per unitats i que l'envàs en què es presenten no fa que s'hagin de considerar un producte diferent. La majoria de productes són així. Quan comprem un ordinador o una escombra o una grapadora no ens fixem en l'envàs, sinó en el producte en si. Aquest tipus de productes es trobaran representats per la classe `Producte`. Es tracta de la classe més genèrica de totes, en el sentit que qualsevol producte envasat podria considerar-se un `Producte` al qual li hem afegit un envàs. De la mateixa manera, un producte a granel seria també un `Producte` associat a una unitat de mesura. És evident que qualsevol producte, sigui del tipus que sigui, es vendrà a un preu unitari determinat i que el preu final que el client haurà de pagar pel producte serà proporcional a la quantitat comprada. Aquestes característiques comunes les gestionarà la classe més genèrica, mentre que les altres dues classes gestionaran les característiques més específiques del tipus representat.

Tal com hem presentat el nostre model resulta molt clar veure que qualsevol de les tres classes de la jerarquia cal considerar-les una entitat. L'aplicació gestionarà productes, productes envasats o a granel, i cada un d'ells constituirà un producte a comprar i vendre, a emmagatzemar en un magatzem, i fins i tot tots ells poden ser sensibles d'obtenir estadístiques de venda, etc.

Canviem ara de jerarquia. En interpretar la jerarquia d'empreses, el nostre model no aplica el mateix criteri. Es tracta també d'una jerarquia de tres classes, `Empresa`, `Client` i `Proveidor`. La primera és la classe genèrica de la qual se'n deriven les altres dues. La diferència entre els clients i els proveïdors és més funcional que conceptual. Ambdós són empreses amb NIF, una seu central, etc. Als clients, però, hi destinem una força comercial que no despleguem per als proveïdors. També els classifiquem per saber què els podem vendre, per fer-hi campanyes específiques, etc. Als proveïdors no. Dels clients ens interessa conèixer totes les seves delegacions o d'altres seus, si en tenen. Per exemple, d'una cadena de supermercats ens pot interessar conèixer la xarxa de botigues si ens cal repartir el producte que venem a cada una d'elles.

Clients i proveïdors comparteixen aspectes estructurals, d'aquí la jerarquia, però no pas funcionals, de manera que ens interessa tractar-los de forma totalment independent. És per això que la classe `Empresa`,en el model que acabem de descriure, no la podem considerar una entitat.

Fixeu-vos que és la interpretació del model la que ens determina si haurem de tractar una classe com a entitat. Si canviem la interpretació també podria canviar la seva consideració. Imagineu que fem la següent interpretació: la nostra aplicació té empreses que a vegades es comporten com a client i a vegades com a proveïdors. Imagineu que en un sentit genèric usem les empreses per obtenir una visió de la nostra tresoreria, independentment de si actuen com a clients o proveïdors. Si féssim aquesta interpretació, aleshores podríem considerar les empreses també com a entitats.

Concepte de MappedSuperClass i estratègia de mapatge

Per tal de veure les diferents opcions que ofereix JPA ens quedarem amb la primera interpretació considerant que en aquesta jerarquia només tindrem dues entitats. JPA permet mapar les dades d'aquelles classes que no siguin entitats, qualificant-les de `MappedSuperClass`.Les dades d'aquelles classes que no s'hagin qualificat no s'emmagatzemaran. No acostuma a ser gaire corrent, però podria donar-se el cas de tenir una classe genèrica que tingués només dades temporals i no calgués emmagatzemar-les.

Un cop s'hagi decidit quines classes de la jerarquia cal mapar, i si cal fer-ho com a entitats o com a superclasses, haurem de determinar quina estratègia seguirem a l'hora de plasmar les dades en taules. JPA suportaqualsevol de les tres estratègies possibles: emmagatzemar tota la jerarquia en una única taula, distribuir les dades de la jerarquia d'acord amb la pertinença a cada entitat concreta o bé emmagatzemar les dades de cada classe en una taula diferent, establint els mecanismes d'especialització pertinents per mantenir l'organització de les dades d'acord amb la jerarquia original de les classes a mapar.

L'elecció de quina estratègia pot ser la millor dependrà de l'ús que n'haguem de fer, i caldrà aplicar criteris d'eficiència durant la recuperació i manipulació de les dades emmagatzemades.

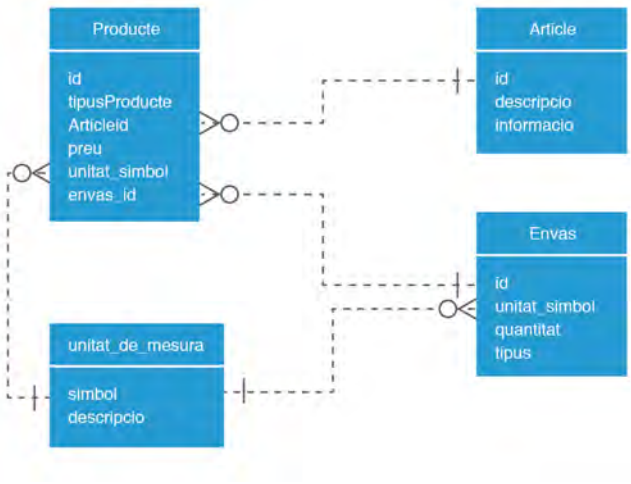
Una taula única per a tota la jerarquia

La primera estratègia es fa adequada en aquells casos en què calgui treballar indiscriminadament amb qualsevol instància de la jerarquia. Aquest seria el cas dels productes, ja que en una mateixa comanda podríem barrejar-hi diferents tipus de productes (envasats, a granel o unitaris), i probablement a l'hora de treure les estadístiques tampoc ens interessarà distingir en quines unitats cal mesurar el producte. Emmagatzemar tots els productes en una mateixa taula implicarà un cert mal ús de l'espai, ja que quedaran sempre camps nuls (sense omplir), però per contra la recuperació indiscriminada de productes serà força més eficient,ja que només caldrà treballar contra una sola taula.

JPA usarà l'anotació `Inheritance` amb el valor del paràmetre `strategy` assignat a SINGLE_TABLE, associat a la classe arrel de la jerarquia per indicar que es desitja fer servir l'estratègia d'emmagatzemar la jerarquia en una única taula.

A més, per saber la classe en què caldrà instanciar cada registre, o bé per saber quins camps corresponen a quins atributs en les diferents classes de la jerarquia, JPA afegirà una camp extra d'algun tipus per poder assignar diferents valors segons la classe que representi el registre de la taula (figura 2.19). El nom del camp i el valor de cada classe es maparà usant les anotacions `DiscriminatorColumn`per indicar el nom i el tipus del camp i `DiscriminatorValue` per indicar el valor de discriminació associat a cada classe. `DiscriminatorColumn`només cal indicar-lo a l'arrel de la jerarquia, juntament amb `Inheritance`. En canvi, `DiscrimantorValue` serà una anotació associada a totes les classes de la jerarquia, cada una amb un valor de discriminació diferent.

Figura 2.19. Taula dels productes seguint l'estratègia d'emmagatzemar la jerarquia en una única taula




```java
@Entity
@Inheritance(strategy= InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="TIPUS_PRODUCTE",
                        discriminatorType= DiscriminatorType.INTEGER)
@DiscriminatorValue(value="0")
public class Producte implements Serializable {
    @Id
    @GeneratedValue(strategy= GenerationType.TABLE)
    private Long id;
    @ManyToOne
    private Article article;
    private double preu;

    public Producte() {
    }

    public Producte(Article article, double preu) {
        this.article = article;
        this.preu = preu;
    }

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    @Override
    public int hashCode() {
        int hash = 0;
        hash += (id != null ? id.hashCode() : 0);
        return hash;
    }

    public Article getArticle() {
        return article;
    }

    public void setArticle(Article article) {
        this.article = article;
    }

    public double getPreu() {
        return preu;
    }
```

```
48
49      public void setPreu(double preu) {
50          this.preu = preu;
51      }
52  }
53
54  @Entity
55  @DiscriminatorValue(value="1")
56  public class ProducteAGranel extends Producte {
57      @ManyToOne
58      private UnitatDeMesura unitat;
59
60      public ProducteAGranel() {
61      }
62
63      public ProducteAGranel(Article article, double preu) {
64          super(article, preu);
65      }
66
67      public ProducteAGranel(Article article, double preu,
68              UnitatDeMesura unitat) {
69          super(article, preu);
70          this.unitat = unitat;
71      }
72
73      public UnitatDeMesura getUnitat() {
74          return unitat;
75      }
76
77      public void setUnitat(UnitatDeMesura unitat) {
78          this.unitat = unitat;
79      }
80  }
81
82  @Entity
83  @DiscriminatorValue(value="2")
84  public class ProducteEnvasat extends Producte {
85      @ManyToOne
86      Envas envas;
87
88      public ProducteEnvasat() {
89      }
90
91      public ProducteEnvasat(Article article, double preu) {
92          super(article, preu);
93      }
94
95      public ProducteEnvasat(Article article, double preu,
96                Envas envas) {
97          super(article, preu);
98          this.envas = envas;
99      }
100 }
```

Fent servir fitxers de configuració XML, caldrà definir:

```
1   <entity-mappings ...>
2
3     ...
4
5     <entity class="ioc.dam.m6.exemples.comandes.Producte "
6       metadata-complete="true">
7       <inheritance strategy="SINGLE TABLE"/>
8       <discriminator-column name="TIPUS_PRODUCTE"
9             discriminator-type="INTEGER" />
10      <discriminator-value> 0 </discriminator-value>
11      ...
12    </entity>
13
```

```
  <entity class="ioc.dam.m6.exemples.comandes.ProducteAGranel "
    metadata_complete="true">
    <discriminator_value> 1 </discriminator_value>
    ...
  </entity>

  <entity class="ioc.dam.m6.exemples.comandes.ProducteEnvasat "
      metadata_complete="true">
    <discriminator_value> 2 </discriminator_value>
    ...
  </entity>

  ...

</entity_mappings>
```

Una taula per a cada entitat

La segona estratègia, la utilitzarem principalment quan haguem de treballar sempre de forma independent amb les classes d'una mateixa jerarquia. Per exemple,la jerarquia d'empreses és bàsicament estructural (vegeu la figura 2.11), ja que conceptualment són entitats diferents i rarament usarem l'entitat empresa de forma genèrica, sinó que treballarem amb clients o amb proveïdors alternativament.

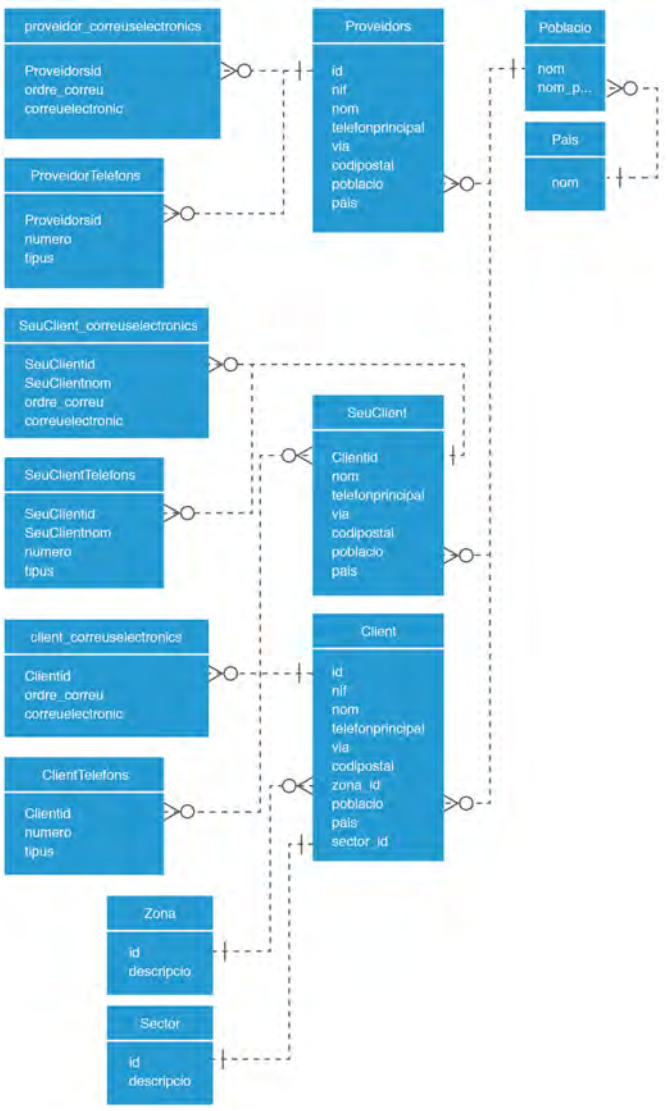
Amb aquesta estratègia, cada entitat acabarà mapada en una taula diferent. Les classes tipificades com a `MappedSuperclass` afegiran camps extres a cada taula per tal de suportar l'estructura de dades corresponent. Així, la jerarquia d'empreses de l'aplicació de comandes generarà dues taules: una per emmagatzemar els clients i l'altra per emmagatzemar els proveïdors (figura 2.20). Ambdues taules disposaran també dels camps mapats a partir de la classe `Empresa`, la qual no es constituirà en taula, ja que està qualificada com a `MappedSuperClass`.

La selecció d'aquesta estratègia passa per associar a la classe arrel de la jerarquia l'anotació `Inheritance` i assignar al paràmetre `strategy` el valor *TABLE_PER_CLASS*.

```
@MappedSuperclass
@Inheritance(strategy= InheritanceType.TABLE_PER_CLASS)
public abstract class Empresa implements Serializable {
    @Id
    @GeneratedValue(strategy= GenerationType.TABLE)
    private int id;
    @Column(length=15, unique=true, nullable=false)
    private String nif;
    @Embedded
    private Seu seuEmpresa=new Seu();

    protected Empresa() {
    }

    public Empresa(String nif) {
        this.nif = nif;
    }

    public Empresa(int id, String nif) {
        this.id = id;
        this.nif = nif;
    }

    public String getNif() {
```

```
25          return nif;
26      }
27
28      public void setNif(String nif) {
29          this.nif = nif;
30      }
31
32      public Seu getSeuEmpresa() {
33          return seuEmpresa;
34
35      }
36
37      public int getId() {
38          return id;
39      }
40
41      protected void setId(int id) {
42          this.id = id;
43      }
44  }
```

Figura 2.20. Taules de clients i proveïdors separades seguint l'estratègia d'una classe per entitat



Com que cada entitat s'emmagatzemarà en taules diferents, no és necessari cap camp discriminador. Per això, usant aquesta estratègia no és necessari fer cap més

canvi a la resta de classes de la jerarquia.

La versió XML quedaria:

```
<entity-mappings ...>

  ...

  <entity class="ioc.dam.m6.exemples.comandes.Empresa "
    metadata-complete="true">
    <inheritance strategy="TABLE PER CLASS"/>
    ...
  </entity>
  ...

</entity-mappings>
```

Mapatge de la jerarquia

La darrera estratègia seria útil en cas que la jerarquia estigués formada per una classe principal que calgués recuperar sovint més un conjunt de classes derivades que especialitzessin la classe principal i només calgués recuperar en moments puntuals. Imaginem que necessitem una jerarquia en la qual calguessindos tipus de clients: client de la zona euro i client d'altres països. Imaginem que la distinció es justifiqui a causa del fet que als clients de la zona euro se'ls fa un contracte de manteniment diferent que els d'altres països. El model disposa de dues classes `Contracte`: `ContracteEuro` i `ContracteNoEuro`. Ambdues implementen la mateixa interfície, però permeten gestionar de diferent manera cada tipus de contracte. El treball amb contractes no es realitza de forma quotidiana, sinó només cada cop que cal revisar la seva vigència. A banda de la gestió dels contractes, la resta d'aspectes serien idèntics per a ambdós clients.

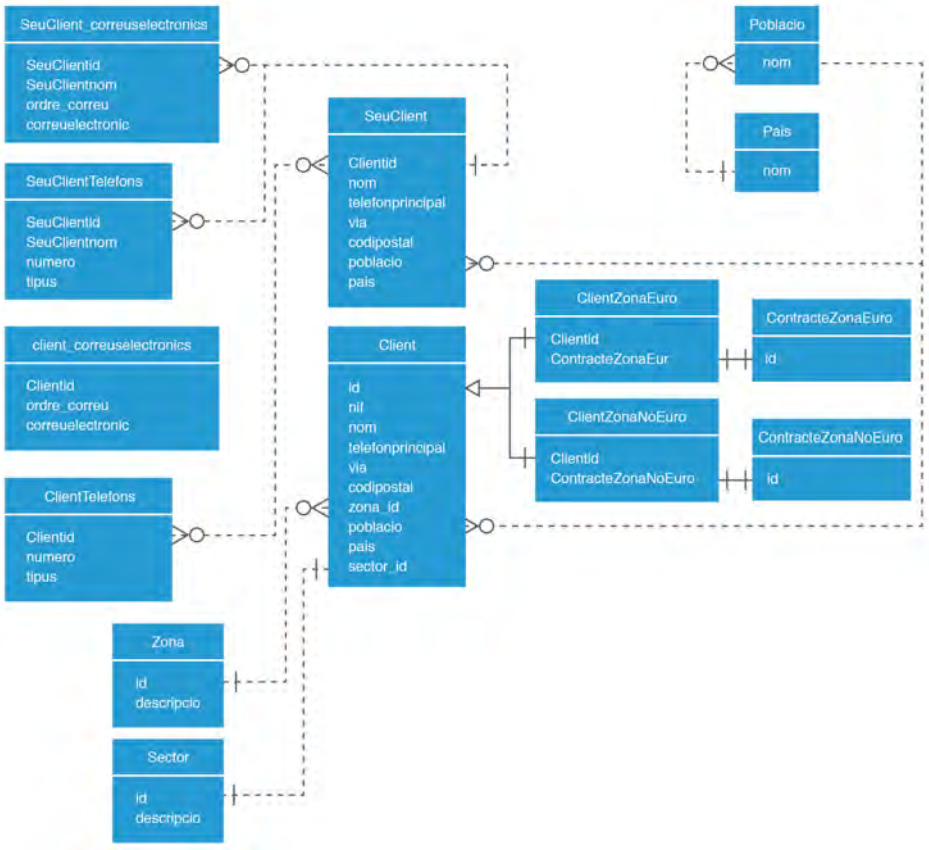
En aquest cas podria ser oportuna l'estratègia que discutim aquí, atès que normalment treballaríem amb la classe genèrica (sense contracte) per a les accions habituals, però per a les accions específiques (revisió o impressió de contracte) podríem treballar amb les instàncies de client especialitzades (figura 2.21).

Per especificar aquesta estratègia, en el nostre model caldrà anotar a la classe arrel que es tracta d'una jerarquia de tipus `JOINED`. A més, en aquesta estratègia també necessitarem indicar un camp de discriminació amb els valors de discriminació associats a cada classe.

```
@Entity
@Inheritance(strategy= InheritanceType.JOINED)
@DiscriminatorColumn(name="tipus_client",
    discriminatorType=DiscriminatorType.INTEGER)
public abstract class Client implements Serializable {
  ...
}


@Entity
@DiscriminatorValue("0")
public class ClientZonaEuro extends Client {
  ...
}

```

```java
@Entity
@DiscriminatorValue("1")
public class ClientZonaNoEuro extends Client {
  ...
}
```

FIGURA 2.21. Esquema entitat-relació



en l'esquema es pot veure l'estructura de les taules que suportarien l'estratègia de mapatge d'una jerarquia en taules d'especialització enllaçades amb una genèrica.

Veiem ara la versió XML:

```xml
<entity-mappings ...>

  ...

    <entity class="ioc.dam.m6.exemples.comandes.Client "
     metadata-complete="true">
    <inheritance strategy="JOINED"/>
    <discriminator-column name="tipus_client"
          discriminator-type="INTEGER" />
    ...
     </entity>

     <entity class="ioc.dam.m6.exemples.comandes.ClientZonaEuro "
      metadata-complete="true">
    <discriminator-value> 0 </discriminator-value>
    ...
     </entity>

     <entity class="ioc.dam.m6.exemples.comandes.ClientZonaNoEuro "
      metadata-complete="true">
    <discriminator-value> 1 </discriminator-value>
    ...
  </entity>

```

```
25    ...
26
27 </entity-mappings>
```

2.6 Funcionalitat del gestor d'entitats

En aquest apartat descobrirem com configurar una unitat de persistència que ens ajudi a adaptar el gestor d'entitats als projectes específics que ens calgui implementar. També descobrirem com obtenir una instància del gestor i discutirem sobre la funcionalitat bàsica del gestor: emmagatzemar entitats a la font de dades, recuperar entitats a partir del seu identificador o clau primària, actualització del'SGBD per tal de sincronitzar els canvis que les entitats ja emmagatzemades vagin tenint, eliminació d'entitats concretes a la font de dades on es trobin emmagatzemades, etc.

2.6.1 Creació d'una unitat de persistència per configurar la connexió al'SGBD

La unitat de persistència de JPA és un fitxer XML que contindrà els paràmetres d'inicialització de la persistència de cada aplicació. Bàsicament els paràmetres de connexió al'SGBD i les entitats que caldrà sincronitzar fent servir aquella unitat de persistència.

Com ja s'ha comentat, una aplicació pot tenir diverses unitats de persistència quan sigui necessari emmagatzemar diferents entitats en SGBD diferents. És per això que es fa necessari detallar la llista d'entitats que cada unitat de persistència haurà de gestionar. Tot i això, el més comú és tenir una única unitat de persistència en cada aplicació.

Malgrat que és possible crear el fitxer de forma manual, NetBeans ens ofereix una eina amb una interfície gràfica que ens facilita la introducció de dades. Per activar-la, cal seleccionar el projecte on volem fer la creació i clicar la icona *New file* o bé fer servir el menú *File* seleccionant l'opció *New File* (també es pot clicar la combinació de tecles *CTRL+N*).

Aquesta opció obrirà un quadre de diàleg com el de la figura. Escolliu la categoria *Persistence* i el tipus de fitxer, cal que sigui *Persistence Unit* (figura 2.22). Feu clic al botó *Next*.

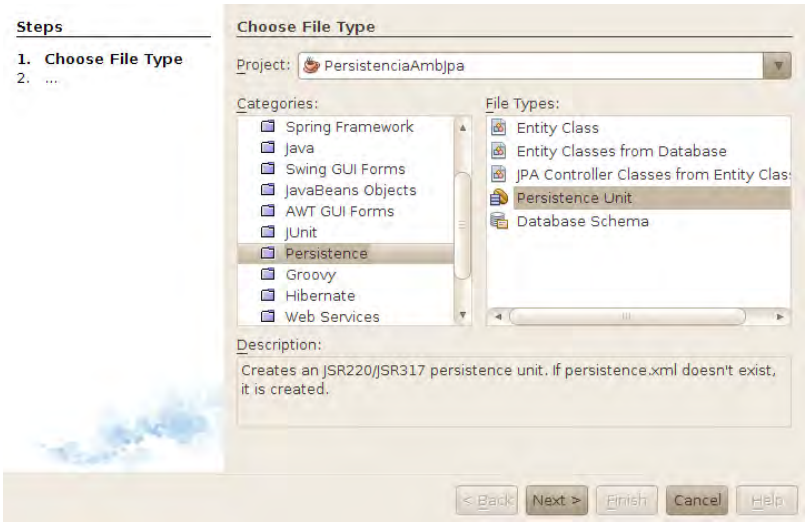
S'obrirà un segon quadre de diàleg que ens permetrà escollir un nom amb el qual reconèixer la unitat de persistència. Qualsevol nom és adequat, i com que només tindrem una sola unitat de persistència no hi trobarem conflictes.

Consulteu el punt "Instal·lació de l'estàndard JPA implementat per Hibernate".

El mateix quadre ens permetrà també seleccionar la biblioteca que implementarà JPA. Com ja hem comentat anteriorment usarem la biblioteca d'Hibernate, que

dóna suport a la versió 2.0 de JPA.

Figura 2.22. Quadre de diàleg per crear un fitxer de tipus Persistence Unit



Obriu el desplegable i assegureu-vos que surt l'opció *Hibernate (JPA 2.0)* (figura 2.23). Vigileu, perquè NetBeans porta per defecte la biblioteca d'*Hibernate amb suport a JPA 1.0,* si escollíssiu aquesta opció no tindríeu suport per a moltes de les opcions explicades en aquest mòdul. Heu d'assegurar-vos que es tracta de la biblioteca que vosaltres heu definit en els apartats inicials. Si no apareix encara aquesta opció perquè encara no s'ha fet servir mai, caldrà escollir l'opció *ManageLibraries*. D'aquesta manera podrem navegar per totes les biblioteques definides de NetBeans i seleccionar la correcta (*Hibernate (JPA 2.0)*). A partir d'aquest moment, la biblioteca estarà ja disponible com a opció per defecte durant la creació d'una nova unitat de persistència, tant en aquest com en d'altres projectes.

Figura 2.23. Quadre de diàleg per escollir el nom, la biblioteca que implementarà JPA, la connexió a la base de dades i les opcions de generació automàtica

Consulteu el punt "Requisits Previs" de l'apartat "JDBC" de *Gestió de Connectors*.

Després de seleccionar la biblioteca caldrà escollir una connexió a un SGBD. A l'opció *JDBC connection* apareixen totes les connexions que tenim donades d'alta en el NetBeans. Seria adequat que tinguéssiu creada al'SGBD de PostgresSQL una base de dades específica per fer les proves de JPA. Creeu-la si no la teniu encara.

Si la connexió es troba definida a NetBeans, seleccioneu-la. Si encara no està definida, escolliu *New Database Connection* o bé doneu d'alta la connexió a la qual pertany*Services* clicant el botó esquerre damunt de *Databases* i seleccionant l'opció *New Connection*.

Un cop definida dins la llista de connexions de NetBeans, escolliu-la com a valor del camp *JDBC Connection* del quadre de diàleg de configuració de la unitat de persistència.

Finalment, el quadre de diàleg permet configurar si desitgem que JPA s'encarregui de crear les taules de forma automàtica en cas que no existeixin. Podeu escollir *Create* per optar per la generació automàtica o *None* per optar per altres maneres de crear-les. L'opció *Drop and Create* és una opció que només és vàlida per realitzar proves. No l'escolliu,ja que si ho feu cada cop que abandoneu el programa s'eliminaran les taules existents per tal de poder-les tornar a crear de nou en la següent execució.

Aconseguireu generar el fitxer XML clicant el botó *Next*. La generació del fitxer també implica que afegirà els arxius JAR de la biblioteca JPA corresponent a *Libraries*. Podeu comprovar-ho desplegant l'opció *Libraries* de l'arbre de recursos de l'àrea esquerra de NetBaens.

Finalment, per deixar la unitat de persistència efectiva caldrà afegir el *driver JDBC* de Pstgres a *Libraries*.

2.6.2 Obtenció d'un EntityManager

Les aplicacions que tinguin un rang d'acció local (no les distribuïdes) faran servir el gestor d'entitats o `EntityManager` com a pivot a partir del qual girarà tota la persistència de l'aplicació. És a dir, no disposarem de cap altra estructura superior que controli els gestors d'entitats de l'aplicació.

De vegades es farà necessari treballar amb diferents instàncies de gestors d'entitats en una mateixa aplicació, però si es tracta d'aplicacions locals els diferents gestors no es coordinaran ni sincronitzaran entre ells. En aplicacions distribuïdes, en canvi, serà possible treballar amb gestors d'entitats que treballin de forma coordinada, però la seva implementació s'escapa de l'estudi d'aquest mòdul.

La gran complexitat de situacions en les quals podem fer servir JPA obliga a obtenir l'`EntityManager` a partir d'un objecte de tipus EntityManagerFactory, de manera que en cada situació concreta l'`EntityManagerFactory` corresponent pugui instanciar un `EntityManager` adequat.

Concretament, per a aplicacions de rang local usarem l'`EntityManagerFactory` per defecte, a partir del qual podrem obtenir instàncies d'`EntityManager` adequades per a aquest tipus d'aplicació.

La creació d'EntityManagerFactory es realitzarà invocant

`createEntityManagerFactory`, de la classe `Persistence`. Cada invocació rebrà per paràmetre el nom de la unitat de persistència amb la qual s'inicialitzarà.

```
EntityManagerFactory emf =
    Persistence.createEntityManagerFactory(
        "UnitatDePersistenciaPersistenciaAmbJpa");
```

És important crear només un `EntityManagerFactory` per a cada unitat de persistència. Per això, generalment, les aplicacions creen una instància al començament de l'execució que romandrà activa fins que l'aplicació es tanqui.

Contràriament, les aplicacions utilitzen sovint moltes instàncies d'`EntityManager` que aniran obrint i tancant segons les necessitats. De fet, cada `EntityManager` actiu representa una connexió *JDBC* oberta i podem fer servir criteris similars a l'hora de mantenir-los oberts o de tancar-los.

De la seva creació se n'encarrega l'`EntityManagerFactory` invocant el mètode `createEntityManager`.

```
EntityManager em = emf.createEntityManager();
```

2.6.3 Gestió d'entitats a l'EntityManager

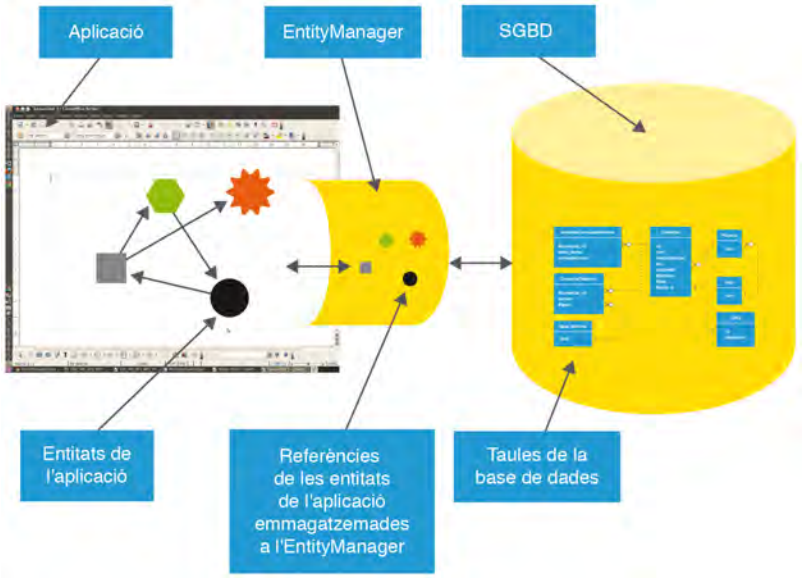
Per encarar amb èxit una aplicació gestionada per JPA caldrà donar a l'`EntityManager` un paper central en el tractament de les entitats. De fet, podem considerar-lo com el representant o interlocutor local del'SGBD a la nostra aplicació.

Per tal de poder gestionar la sincronització entre les entitats del model orientat a objectes i l'SGBD, l'`EntityManager` necessita mantenir sempre en memòria una referència a les instàncies que l'aplicació vagi generant durant l'execució de la mateixa.

El gestor d'entitats pot obtenir les referències a gestionar de diverses maneres. Si es tracta d'una entitat emmagatzemada, cada cop que l'`EntityManager` obtingui una o diverses entitats usant el mètode `find` o a partir de l'execució d'una consulta (fent servir el llenguatge OQL de JPA) en el moment de fer la instanciació de cada entitat, enregistrarà també una referència a la mateixa per tal de controlar a partir d'aquell moment la sincronització amb la base de dades (figura 2.24).

Si es tracta d'una entitat nova, encara no emmagatzemada a la base de dades, serà possible passar la referència a l'`EntityManager`invocant el mètode `persist`. Aquest mètode, a més de passar-li la referència al gestor, inserirà tots aquells registres que calgui a les taules del'SGBD per aconseguir emmagatzemar les dades de l'entitat.

Figura 2.24. Esquema de com actua l'EntityManager



Es pot considerar l'intermediari entre l'SGBD i l'aplicació. L'EntityManager precisa emmagatzemar en memòria una referència de totes les entitats actives de l'aplicació. Mitjançant la configuració extreta del mapatge del model, interactua amb l'SGBD per mantenir la sincronització.

Si es tracta d'una entitat instanciada a l'aplicació, ja emmagatzemada l'SGBD, però encara no referenciada dins l'`EntityManager`, caldrà fer servir el mètode `merge`, que enregistrarà la referència dins l'`EntityManager` de forma semblant a com ho faria `persist`, però en comptes d'intentar donar d'alta l'entitat en l'SGBD, si cal, optarà per una actualització dels registres del'SGBD corresponents a l'entitat referenciada.

El mètode `merge`, en realitat, és un mètode molt flexible. L'objectiu principal d'aquest és sincronitzar la instància passada per paràmetre amb l'SGBD, de manera que en funció de si l'entitat ja està donada d'alta o nol'SGBD optarà per inserir o només actualitzar l'entitat, respectivament.

Mentre el gestor d'entitats resti actiu no és necessari haver d'invocar el mètode `merge` cada cop que canvia l'estat d'alguna entitat, ja que les referències del gestor també mantenen els mateixos canvis per a cada instància.

És possible forçar una sincronització real amb l'SGBD invocant el mètode `flush`. L'execució d'aquest mètode implica sincronització de totes les entitats enregistrades pel gestor que quedessin pendents d'actualitzar.

Usant el mètode `flush` aconseguirem minimitzar el trànsit de xarxa entre l'aplicació i l'SGBD, ja que concentrarem el diàleg de sincronització als punts en què invoquem el mètode. Fent servir aquesta estratègia caldrà invocar sempre el mètode abans de realitzar el tancament per si un cas quedés alguna actualització pendent. L'ús de `flush`, en detriment de `merge`, s'anomena també *persistència passiva,* ja que el programador es despreocupa d'haver de realitzar una a una cada actualització.

Transaccions

Tant la invocació del mètode `persist` com la del mètode `merge` o la del mètode `flush`necessitaran que hi hagi una transacció activa. Aconseguirem iniciar una transacció fent la crida següent:

```
em.getTransaction().begin();
```

On `em` és una instància activa d'un `EntityManager`.

La transacció s'acabarà quan fem una invocació d'acceptació (mètode `commit`) o de revocació (mètode `rollback`). Les sentències senceres serien:

```
em.getTransaction().commit();
```

o bé,

```
em.getTransaction().rollback();
```

Vegem-ne uns exemples. En primer lloc, veurem com crear una instància de `Zona`i inserir-la al'SGBD. Posteriorment modificarem la descripció de la instància que actualitzarem invocant l'operació `flush`abans de tancar el gestor.

```
EntityManager em = emf.createEntityManager();

...

Zona obj = new Zona("Europa", "Euro");
em.getTransaction().begin();
em.persist(obj);
em.getTransaction().commit();

...

obj.setDescripcio("Tots els països de l'Europa occidental");

...

em.getTransaction().begin();
em.flush();
em.getTransaction().commit();
em.close();
```

Podríem realitzar les mateixes operacions invocant el mètode `merge` en comptes de `persist` i `flush`.

```
EntityManager em = emf.createEntityManager();

...

Zona obj = new Zona("Europa", "Euro");
em.getTransaction().begin();
em.merge(obj);
em.getTransaction().commit();

...

obj.setDescripcio("Tots els països de l'Europa occidental");
em.getTransaction().begin();
em.merge();
```

```
em.getTransaction().commit();

...

em.close();
```

La diferència entre ambdós algorismes és que el primer controla si ja existeix la clau primària i llença una excepció en cas que així sigui. El segon, en canvi, no realitza aquest control, sinó que en cas que la clau primària ja existeixi, es modificaran els valors de l'entitat emmagatzemada.

A més, el primer algoritme usaria una tècnica de persistència passiva, a l'inrevés del segon, que invocaria el *merge* per assegurar la sincronització quan fos necessari.

Obtenció d'entitats existents en l'SGBD

L'obtenció d'entitats emmagatzemades en l'SGBD es pot fer invocant el mètode `find`, el qual rebrà per paràmetre la classe de la que volem obtenir una instància i el valor de la clau primària.

També és possible obtenir una o diverses instàncies executant una consulta amb el llenguatge OQL de JPA. Es tracta d'un llenguatge complex que avançarem amb dues consultes en què obtindrem una Zona i un conjunt de zones, respectivament.

El codi per obtenir una instància invocant `find` seria el següent:

```
EntityManager em = emf.createEntityManager();
...
Zona zona = em.find(Zona.class, "BCN");
```

On BCN seria el valor d'una clau primària existent. En cas que no existís la clau, es llançaria una excepció.

Per crear una consulta cal invocar `createQuery` amb la consulta a realitzar passada com a paràmetre. Si sabem que la consulta només retornarà una única instància, podem fer servir el mètode `getSingleResult`.

```
...
Query qry = em.createQuery(
      "Select z from Zona z where z.descripcio='Maresme i Montnegre'");
zona = (Zona) qry.getSingleResult();
```

En canvi, per a consultes on es retornin diverses entitats usarem el mètode `getResultList`.

```
...
qry = em.createQuery("Select z from Zona z");
List<Zona> list = qry.getResultList();
for(Zona z: list){
    System.out.println("Zona:" + z.toString());
}
...
```

Com haureu observat, els mètodes de consulta i obtenció de dades no requereixen definir cap transacció.

Eliminació d'una entitat

Fent servir el gestor d'entitats també podrem eliminar de forma persistent aquelles instàncies que ja no hagin de formar part de les emmagatzemades l'SGBD. Per fer-ho, cal que el gestor tingui enregistrada un referència de l'entitat que volem eliminar i, per tant, caldrà obtenir-la per alguna de les vies indicades més amunt.

Així, per exemple, per eliminar una zona que tinguem emmagatzemada al'SGBD caldrà recuperar-la a partir del seu identificador i usar la instaància obtinguda per invocar el mètode `remove` del gestor.

```
em = emf.createEntityManager();
...
Zona zona = em.find(Zona.class, "Europa");
...
em.getTransaction().begin();
em.remove(zona);
em.getTransaction().commit();
if(em.find(Zona.class, "Europa")==null){
    System.out.println("Eliminació correcta");
}else{
    System.out.println("No s'ha eliminat la zona");
}
...
em.getTransaction().begin();
em.flush();
em.getTransaction().commit();
em.close();
```

Tot i que aquesta és una manera força habitual de cercar l'entitat que desitgem eliminar, també podem fer servir qualsevol altre dels mètodes vistos. Per exemple, si volguéssim eliminar totes les zones de la seva taula podríem fer una consulta per recuperar la llista de totes les zones existents i després, amb un bucle, passaríem a la seva eliminació:

```
em = emf.createEntityManager();
...
Query qry = em.createQuery("Select z from Zona z");
List<Zona> listZona = qry.getResultList();
...
em.getTransaction().begin();
for(Zona v: listZona){
    em.remove(v);
}
em.getTransaction().commit();
...
em.getTransaction().begin();
em.flush();
em.getTransaction().commit();
em.close();
```

2.7 El llenguatge de consulta JPQL

El llenguatge de consulta que JPA fa servir s'anomena JPQL (Java Persistence Query Language). Es tracta d'un llenguatge de consulta molt similar a OQL, l'estàndard definit per l'Object Data Management Group. Tot i això, JPQL estén algunes funcionalitats d'OQL per adaptar-se a un altre estàndard, l'Entreprise Java Bean.

Tots ells fan servir una sintaxi molt similar a SQL amb adaptacions específiques per aconseguir treballar amb objectes en comptes de taules.

De forma semblant a SQL, les sentències OQL es divideixen també en tres parts: la clàusula de declaració de les dades que es vol obtenir, encapçalada per la paraula clau SELECT; la clàusula de declaració del tipus d'objectes que farem servir en la sentència, encapçalada per la paraula FROM, i la clàusula de condicions que hauran de complir les entitats recuperades, encapçalada per la paraula WHERE.

La diferència entre SQL i OQL, però, és que les referències no es fan contra les taules d'una base de dades sinó sobre un hipotètic univers format per totes les instàncies persistents de l'aplicació. L'univers d'entitats s'organitza en classes o tipus d'objectes per facilitar la consulta. És a dir, en comptes de fer una cerca entre totes les entitats de l'univers limitarem la cerca a les entitats que siguin de les classes especificades en la clàusula FROM.

La resta d'adaptacions sintàctiques segueixen els prototipus orientats a l'objecte. És a dir, es fa referència als atributs d'un objecte separant el nom de l'atribut del de l'objecte amb un punt. A més, en cas de tractar-se d'atributs de dades de tipus no primitiu, serà possible concatenar crides separades sempre per un punt, tal com es preveu a la sintaxi dels llenguatges orientats a l'objecte.

Així, per exemple, si analitzem la sentència que ja hem fet servir anteriorment, per recuperar la zona que tingui una descripció igual a '*Maresme i Montnegre*',

```
1 SELECT z
2 FROM Zona z
3 WHERE z.descripcio='Maresme i Montnegre'
```

Veurem que caldrà limitar la cerca a les entitats de tipus `Zona` (*FROM Zona z*). La paraula *z* actua de símbol d'una entitat qualsevol de tipus `Zona` dins la sentència OQL. Continuant amb l'anàlisi, veurem que, concretament, de totes les entitats `Zona` volem seleccionar només les entitats que tinguin com a valor de l'atribut `descripció`, la cadena '*Maresme i Montnegre*' (*WHERE z.descripcio='Maresme i Montnegre'*). A més, l'obtenció de les dades es correspondrà amb instàncies de `Zona` (*SELECT z*).

Tot i que generalment la recuperació de dades acostuma a coincidir amb instàncies d'entitats, és possible especificar el retorn dels valors de només certs atributs. Per exemple:

```
1 SELECT z.id, z.comercial
```

```
2 FROM Zona z
3 WHERE z.descripcio='Maresme i Montnegre'
```

Permetria recuperar només l'identificador de les zones coincidents amb les condicions especificades a la sentència i el comercial associat a la zona.

Quan executem sentències JPQL que retornen una combinació d'atributs, obtindrem per cada instància analitzada un *array* d'objectes de tantes posicions com atributs calgui retornar. Si només hi ha una instància que respongui a la condició de la sentència, el retorn serà d'un únic *array* d'objectes:

```
1 Object[] retorn;
2 Query qry = em.createQuery(
3     "SELECT z.id, z.comercial FROM Zona z WHERE z.descripcio='Maresme i
          Montnegre'");
4 retorn = (Object[]) qry.getSingleResult();
```

Però si el nombre d'instàncies que compleixen la condició fos superior, el retorn seria una llista d'*arrays*:

```
1 List<Object[]> retorn;
2 Query qry = em.createQuery(
3     "SELECT z.id, z.comercial FROM Zona z");
4 retorn = qry.getResultList();
```

En ambdós casos, cada *array* estaria format per dues posicions. A la posició zero trobaríem els identificadors de les zones coincidents, i a la posició u, el comercial.

JPQL disposa d'un important conjunt d'operacions (molt similars al llenguatge SQL) que poden definir-se a la clàusula WHERE i en fan un llenguatge molt flexible i potent. A banda de les operacions de comparació bàsiques (=, >, <, >=, <=, <>), JPQL també permet l'ús d'operacions de comparació avançades com LIKE, IN, BETWEEN, IS EMPTY, IS NULL, IS MEMBER, etc.

Cada comparació pot enllaçar-se en una única sentència condicional fent servir les típiques operacions lògiques, AND, OR i NOT.

Les sentències, a més, admeten valors literals i també operacions matemàtiques com sumes, restes, multiplicacions o divisions, etc.

Així, per exemple, la següent sentència:

```
1 SELECT p
2 FROM Producte p
3 WHERE p.article.descripcio = 'Llet' AND p.preu < 50 + 50
```

obtindrà tots aquells productes que tinguin un preu inferior a 100 € i en la descripció de l'article sigui *Llet*.

2.7.1 Parametrització de sentències

JPQL suporta un alt grau de parametrització de les sentències de consulta, que incrementa la seva flexibilitat i potència. JPQL admet dos tipus de sintaxis a l'hora d'expressar els paràmetres. Una sintaxi posicional en què els paràmetres s'identifiquen segons l'ordre en què seran introduïts i una sintaxi nominal que identifica els paràmetres amb un nom.

Amb la primera sintaxi els paràmetres es plasmen anteposant el caràcter "?" a un número que representarà un identificador numèric del paràmetre.

```
SELECT p
FROM Producte p
WHERE p.article.descripcio = ?1 AND p.preu < ?2
```

A la sentència d'exemple anterior, el primer paràmetre serà el valor de la descripció i el segon, el límit superior del preu desitjat.

A la segona sintaxi, els paràmetres s'identifiquen perquè van precedits del símbol ":" seguit d'un nom. La introducció dels paràmetres es realitzarà referenciant el nom (sense els dos punts) i el valor a assignar.

```
SELECT p
FROM Producte p
WHERE p.article.descripcio = :descripcio AND p.preu < :preuMaxim
```

La introducció dels valors dels paràmetres es fa invocant el mètode `setParametre` d'un objecte `Query`. En primer lloc, passarem l'identificador del paràmetre (un número si fem servir la sintaxi posicional o el nom si fem servir la qualitativa) seguit del valor que desitgem assignar.

En primer lloc, un exemple de sintaxi posicional.

```
Query qry;
List<Producte> productes;
EntityManager em = emf.createEntityManager();

...

qry= em.createQuery("SELECT p "
        + "FROM ProducteAGranel p "
        + "WHERE p.unitat.simbol = ?1 AND p.preu < ?2 ");
qry.setParameter(1, "kg.");
qry.setParameter(2, 100.0);

productes=qry.getResultList();

for(Producte p: productes){
    System.out.println(p);
}

em.getTransaction().begin();
em.flush();
em.getTransaction().commit();
em.close();
```

I també nominal.

```
EntityManager em = emf.createEntityManager();
Query qry;
List<Producte> productes;

...

qry= em.createQuery("SELECT p "
     + "FROM Producte p "
     + "WHERE p.article.descripcio = :descripcio "
     + "AND p.preu < :preuMaxim");
qry.setParameter("descripcio", "Escombra");
qry.setParameter("preuMaxim", 100.0);


productes = qry.getResultList();
for(Producte p: productes){
    System.out.println(p);
}

em.getTransaction().begin();
em.flush();
em.getTransaction().commit();
em.close();
```

2.7.2 Consultes predefinides o Named Queries

Invocar el mètode `createQuery` tot passant-li per paràmetre la sentència de consulta a executar, és una manera molt flexible a l'hora de crear consultes, ja que permet generar sentències de forma dinàmica durant l'execució de l'aplicació.

Malgrat tot, un percentatge molt gran de les consultes que es necessiten executar en una aplicació solen estar perfectament definides i amb l'ús de paràmetres tot just hi ha necessitat de generar les sentències dinàmicament.

JPA disposa d'una eina que permet crear sentències de consultes predefinides, el que en l'argot de JPA es coneix com a *Named Queries*. L'ús d'aquesta utilitat permet incrementar en gran mesura l'eficiència de les consultes.

Les *Named Queries* es defineixen en una anotació o en el mateix fitxer XML de definició. Un cop definides s'emmagatzemen en memòria i, per tant, són sensiblement més ràpides de crear. Si usem la concatenació per suma a l'hora d'estructurar la cadena de la sentència de manera que en facilitem la lectura, aquesta només es farà efectiva durant la càrrega de l'execució, ja que després romandrà concatenada a memòria i, per tant, la seva obtenció serà immediata.

No és possible modificar la cadena que constitueix la sentència predefinida. Per tant, l'única manera de generalitzar la consulta és fer servir paràmetres. Es pot fer servir qualsevol de les sintaxis que hem vist.

Les *Named Queries* es creen associades a un nom i a la classe on es trobin definides. Ambdues serviran de referència per recuperar la sentència en el moment de la creació. L'anotació que possibilitarà la definició és `NamedQuery`. Normalment, les consultes predefinides solen associar-se a cada entitat, de manera

que cada una d'elles disposa de totes les consultes predefinides que necessita durant l'excussió. Tot i això, les consultes predefinides poden especificar-se en qualsevol altra classe. No és necessari definir-les en cada entitat.

Vegem un exemple de definició d'una `NamedQuery`.

```
@NamedQuery(name="Client.clientsDUnSector",
            query= "SELECT c "
                 + "FROM Client c "
                 + "WHERE c.sector.id=:sector")
```

Per poder recuperar la consulta predefinida i executar-la caldrà invocar el mètode `createNamedQuery` de l'`EntityManager` i actuar com si d'una consulta qualsevol es tractés.

```
Query qry;
List<Client> list;
EntityManager em = emf.createEntityManager();

...

qry= em.createNamedQuery("Client.clientsDUnSector", Client.class);
qry.setParameter("sector", "Electrònica");


list = qry.getResultList();
for(Client v: list){
    System.out.println(v);
}

...

em.getTransaction().begin();
em.flush();
em.getTransaction().commit();
em.close();
```

Podem també definir vàries `Named Queries` en una sola classe usant l'anotació `NamedQueries`, la qual rebrà per paràmetre una col·lecció de consultes predefinides.

```
@Entity
@NamedQueries({
        @NamedQuery(name="Client.clientsDUnSector",
        query= "SELECT c "
        + "FROM Client c "
        + "WHERE c.sector.id=:sector"),
        @NamedQuery(name="Client.clientPerNif",
        query= "SELECT c "
        + "FROM Client c "
        + "WHERE c.nif=:nif"),
        @NamedQuery(name="Client.clientPerNom",
        query= "SELECT c "
        + "FROM Client c "
        + "WHERE c.seuEmpresa.nom=:nom")
})
public class Client extends Empresa {
  ...
}
```

És possible també afegir *Named Queries* fent servir un document XML. El format del document seria el següent:

```
<entity-mappings ...>

  ...

  <entity class="ioc.dam.m6.exemples.comandes.Client "
    metadata-complete="true">
    <inheritance strategy="JOINED"/>
    <discriminator-column name="tipus_client"
          discriminator-type="INTEGER" />
    ...
    <named-query name="Client.clientsDUnSector">
      <query>
        SELECT c
        FROM Client c
        WHERE c.sector.id=:sector
      </query>
    </named-query>
    <named-query name="Client.clientPerNif">
      <query>
        SELECT c
        FROM Client c
        WHERE c.nif=:nif
      </query>
    </named-query>
    <named-query name="Client.clientPerNom">
      <query>
        SELECT c
        FROM Client c
        WHERE c.seuEmpresa.nom=:nom
      </query>
    </named-query>
  </entity>
  ...

</entity-mappings>
```

2.7.3 JPQL en detall

En aquest apartat estudiarem JPQL amb detall. Per fer-ho, analitzarem la capacitat expressiva de cada una de les clàusules. Començarem per la clàusula SELECT.

Clàusula SELECT

Com ja hem avançat, la clàusula SELECT de les sentències JPQL indica les dades que obtindrem en executar la consulta. En aquesta clàusula es pot especificar si volem recollir el conjunt d'entitats que compleixin les condicions expressades a la sentència. És la forma més senzilla i s'indica amb un símbol que representarà les entitats retornades:

```
SELECT c
FROM Client c
```

És possible que de vegades estiguem només interessats en algun dels atributs de l'entitat seleccionada. En aquest cas, especificarem, separats per comes, quins atributs volem obtenir de forma combinada.

```
SELECT c.nif, c.seuEmpresa.nom
FROM Client c
```

Recordeu que a l'hora de recuperar les dades, el valor de cada expressió es retornarà en una posició d'un *array*. Així, la consulta de més amunt retornaria *arrays* de dues posicions.

JPQL també permet especificar les dades a retornar amb la forma de constructor d'instàncies d'alguna classe específica de la nostra aplicació. La classe ha d'estar codificada i ha de tenir un constructor que admeti el nombre de paràmetres descrit a la sentència. Per exemple, imaginem que per evitar la recuperació de tota l'entitat de tipus `Client` implementem una classe anomenada `DadesBasiquesClient`, la qual tindrà només tres atributs: el codi del client, el seu NIF i el seu nom. La classe disposarà d'un constructor que permeti inicialitzar el valor dels tres atributs:

```
public class DadesBasiquesClient {
    private int id;
    private String nif;
    private String nom;

    public DadesBasiquesClient(int id, String nif, String nom) {
        this.id = id;
        this.nif = nif;
        this.nom = nom;
    }

    public int getId() {
        return id;
    }

    public String getNif() {
        return nif;
    }

    public String getNom() {
        return nom;
    }
}
```

És possible especificar una clàusula SELECT que instancïi `DadesBasiquesClient` indicant l'expressió del constructor que usarem.

```
SELECT NEW ioc.dam.m6.exemples.comandes.DadesBasiquesClient(
        c.id, c.nif, c.seuEmpresa.nom)
FROM Client c
```

El resultat d'aquesta consulta serien instàncies de `DadesBasiquesClient` amb la id, el NIF i el nom de cada client de la nostra aplicació.

Finalment, les clàusules SELECT també permeten especificar funcions d'agrupació (AVG, COUNT, MAX, MIN i SUM) que es retornaran com a valors numèrics. Per exemple, la sentència

```
SELECT COUNT(c)
FROM Client c
```

retornarà la quantitat de clients que tenim emmagatzemats al'SGBD.

Clàusula FROM

La clàusula FROM de les consultes JPQL declara les variables usades en qualsevol de les expressions de la sentència. La declaració de les variables segueix la mateixa sintaxi usada a Java. És a dir, el tipus precedeix el símbol que representarà la variable. Cada variable anirà separada per una coma.

```
SELECT s
FROM Client c, Sector s
WHERE c.sector=s
```

Quan intervenen diverses variables, la clàusula FROM podrà establir la relació existent entre elles fent servir l'operador JOIN o LEFT JOIN. Per definir la relació s'especifica l'atribut de l'entitat implicat.

```
SELECT s
FROM Client c JOIN c.sector s
```

La sentència anterior obtindria tots els sectors que pertanyen a algun client de l'aplicació.

L'operador JOIN també pot expressar-se sempre com una condició de la clàusula WHERE. De fet, si us hi fixeu, les dues darreres sentències són equivalents.

És possible encadenar diversos operadors JOIN. Per exemple, per expressar una sentència que obtingui tots els clients assignats a un comercial fent servir la relació existent entre `Comercial`, `Zona` i `Client`, escriuríem:

```
SELECT cl
FROM Comercial c JOIN c.zona z JOIN z.clients cl
WHERE c.nif=:nif
```

Observeu l'encadenament: *z* representa la zona assignada a un comercial (*c.zona*) i *cl* els clients que hi ha en una zona *z* (*z.clients*).

Usant múltiples operadors JOIN és possible obtenir dades repetides en relacions de tipus molts-és-a-un o molts-és-a-molts. Per evitar-ho es pot fer servir DISTINCT acompanyant la paraula clau SELECT.

```
SELECT DISTINCT cl
FROM Comercial c JOIN c.zona z JOIN z.clients cl
WHERE c.nif=:nif
```

Clàusula WHERE

Finalment, analitzarem la clàusula WHERE, que com deveu suposar permet especificar les condicions que hauran de complir les entitats obtingudes després d'executar la consulta. La clàusula WHERE especifica una expressió lògica que actuarà de filtre per reconèixer les entitats a ser retornades.

L'expressió lògica de la clàusula estarà formada per múltiples operacions lògiques o de comparació operades per AND o OR, de manera que el resultat final obtingut sigui un valor lògic.

Les expressions de comparació permeten comparar expressions de diferents tipus, l'avaluació de les quals representarà també un valor lògic. Les operacions de comparació són similars a les operacions d'SQL (=, >, <, >=, <=, <>, LIKE, IN, BETWEEN, IS EMPTY, IS NULL, IS MEMBER, etc.).

Els operands de les operacions de comparació poden ser valors literals, variables, resultats de subconsultes o expressions formades per diferents tipus d'operacions(+, -, *, /) o funcions pròpies de JPQL (vegeu taula 2.1).

Taula 2.1. Funcions pròpies que suporta el llenguatge JPQL

Funció	Resultat
ABS (nombre)	Calcula el valor absolut del nombre passat per paràmetre.
MOD (dividend, divisor)	Calcula el residu de dividir el nombre passat com a dividend entre el nombre passat com a divisor.
SQRT (nombre)	Calcula l'arrel quadrada del nombre passat per paràmetre.
CURRENT_DATE	Obté la data actual del sistema.
CURRENT_TIME	Obté l'hora actual del sistema.
CURRENT_TIMESTAMP	Obté la data i l'hora del sistema.
CONCAT(cadena1, cadena2)	Obté una cadena formada per la concatenació de la `cadena2`després de la `cadena1`.
LENGTH (cadena)	Obté la longitud (en caràcters) de la cadena passada per paràmetre.
LOWER (cadena)	Obté la cadena passada per paràmetre en minúscula.
UPPER (cadena)	Obté la cadena passada per paràmetre en majúscula.

Les variables es declararan sempre a la clàusula FROM i han de coincidir amb els tipus d'entitats controlades per l'`EntityManager` que gestioni la consulta.

Els valors literals poden ser de tipus numèric, lògic, data, mesura de temps o cadena de caràcters.

Si els valors literals són numèrics o lògics (TRUE o FALSE) s'escriuen sense cometes. Si són cadenes de caràcters o dates, es marcarà l'inici i final del literal amb una cometa simple en el cas de les cadenes o es tancaran entre claus en cas que siguin dates o mesura temps. Les dates es distingiran perquè es marcaran amb la lletra *d* seguida d'una cadena de caràcters indicant la data segons els format següent: aaaa-mm-dd (on *a* simbolitzen els dígits de l'any, *m* els del mes i *d* els del dia). Les mesures de temps aniran precedides de la lletra "t" i seguiran el format hh-mm-ss, on*h* són els dígits de l'hora, *m* els dels minuts i *s* els dels segons. És possible expressar una combinació de data i temps anteposant el símbol "ts" abans d'expressar la data i l'hora en el format ja descrit. Mireu els exemples següents:

```
1 client.nom = 'Bon Menjar S.L.'
2 producte.preu < 12.50
3 comanda.data = {d '2012-04-01'}
4 comanda.hora < {t '20-00-00'}
5 comanda.dataIHora > {ts '2012-01-01 12-00-00'}
```

La taula 2.2 us donarà una idea de les principals operacions suportades per les

expressions WHERE de JPQL.

Taula 2.2. Operacions suportades a les expressions de la clàusula WHERE.

Operació	Descripció	Exemple
=, >, <, >=, <=, <>	Comparen dues expressions situades a banda i banda de l'operació d'acord amb el símbol matemàtic que representin.	`p.preu *0.20 >= c.descompte` Compara si el 20% del preu de l'entitat *p* és major o igual al descompte de l'entitat *c*.
BETWEEN	Compara si una expressió numèrica es troba dins d'un rang definit.	`p.preu BETWEEN 10 AND 100` Compara si el preu de l'entitat *p* es troba entre 10 i 100 (ambdós inclosos).
LIKE	Compara si una expressió de caràcters segueix un determinat patró. El patró es defineix a partir de seqüències de caràcters en combinació amb caràcters especials anomenats comodins. Hi ha dos caràcters comodí: _ i %. El primer representa qualsevol caràcter i el segon qualsevol expressió de caràcters de mida indefinida.	`LOWER(a.nom) LIKE '%poma%'` Compara si el nom de l'entitat *a* conté la seqüència `poma`. `c.adreca.poblacio.pais LIKE 'E%'` Compara si el país de la població de l'adreça de l'entitat *c* comença per la lletra *E*. `c.nif LIKE '_12345678'` Compara si la numeració del NIF de l'entitat *c* es correspon a *12345678* amb independència de la lletra inicial.
IN	Compara si una expressió es correspon amb algun dels valors continguts entre parèntesis. El conjunt de valor pot ser una seqüència de literals o bé els valors retornats per una subconsulta.	`c.adreca.poblacio.nom IN ('Barcelona', 'Sabadell', 'Badalona')` Compara si el nom de la població de l'adreça de l'entitat *c* es correspon amb *Barcelona*, *Sabadell* o *Badalona*. `c.zona IN (SELECT cl.zona FROM Client cl WHERE cl.sector.id=:sector)` Compara si la zona de l'entitat *c* es correspon amb la zona d'aquells clients que tenen per sector el valor passat pel paràmetre que respon al nom de *sector*.
IS EMPTY IS NOT EMPTY	Compara si un valor és *null* o no.	`z.descripcio IS NOT EMPTY` Compara si la descripció de l'entitat *z* no presenta un valor *null*.
ANY	Compara si una expressió es CERTA per algun dels valors tancats entre parèntesis.	`p.preu < ANY (SELECT p.preu FROM Sector s JOIN s.productes p) WHERE s.id=:sector)` Compara si el preu de l'entitat *p* és inferior al preu d'algun dels productes que es venen a clients del sector amb una*id* corresponent al valor del paràmetre que respon al nom de *sector*.
ALL	Compara si una expressió es CERTA per a tots i cada un dels valors tancats entre parèntesis.	`p.preu > ALL (SELECT p.preu FROM Sector s JOIN s.productes p) WHERE s.id=:sector)` Compara si el preu de l'entitat *p* és superior a tots i cada un dels preus dels productes que es venen a clients del sector amb una*id* corresponent al valor del paràmetre que respon al nom de*sector*.

Taula 2.2 (continuació)

Operació	Descripció	Exemple
EXIST	Compara si la subconsulta tancada entre parèntesis retorna algun valor.	`EXIST (SELECT 1 FROM Comercial c WHERE c.zona.id=:zona)` Compara si hi ha algun comercial assignat a la zona amb una*id* corresponent al valor del paràmetre que respon al nom de *zona*.

Clàusula ORDER BY

Aquesta clàusula permet indicar l'ordre en què desitgem obtenir les dades de la consulta, i hi podem especificar una llista de variables el valor de les quals, amb una prioritat d'esquerra a dreta, el farem servir per ordenar les dades de la consulta. Cada una de les variables podrà acompanyar-se de la paraula clau DESC per indicar que l'ordre de la variable precedent s'establirà de forma descendent. En cas de no escriure la paraula DESC, l'ordre contemplat serà sempre ascendent.

En el següent exemple els clients es retornaran ordenats per població i dins la mateixa població s'ordenaran per nom.

```
1 SELECT c
2 FROM Client c
3 ORDER BY c.seuEmpresa.adreca.poblacio.nom, c.nom
```

En canvi, en la següent, es retornen els productes ordenats per preu de forma descendent. És a dir, primer obtindrem els més cars. En cas que hi hagi productes amb el mateix preu s'ordenaran per descompte en forma ascendent:

```
1 SELECT p
2 FROM Producte p
3 ORDER BY p.preu DESC, p.descompte
```

Clàusula GROUP BY

La clàusula GROUP BY permet agrupar els resultats segons algun criteri definit aquí. S'utilitza sobretot de forma combinada amb les funcions d'agrupació que ja hem vist a la clàusula SELECT, de manera que en comptes d'obtenir càlculs globals, obtinguem els càlculs específics de cada grup.

Per exemple, si volem saber quants productes tenim a cada sector, podem fer la següent sentència:

```
1 SELECT s.id, COUNT(p)
2 FROM Sector s LEFT JOIN s.productes p
3 GROUP BY s.id
4 ORDER BY s.id
```

Clàusula HAVING

Usarem la clàusula HAVING quan vulguem establir un filtre sobre les dades agrupades. És a dir, sobre un dels resultats (calculat o no) o sobre alguna de les expressions especificades a la clàusula GROUP BY.

Si volguéssim limitar els resultats de la sentència anterior excloent els sectors que tinguessin pocs productes (posem menys de 5), hauríem d'usar la clàusula HAVING.

```
SELECT s.id, COUNT(p)
FROM Sector s LEFT JOIN s.productes p
GROUP BY s.id
HAVING COUNT(p)>=5
ORDER BY s.id
```