Persistència en BDR-BDOR-BDOO

Josep Cañellas Bornas

Accés a dades

Índex

Introducció

La present unitat, anomenada “Persistència en BDR-BDOR-BDOO”, ofereix una visió de la manera com les aplicacions orientades a objectes usen els sistemes gestors de bases de dades (SGBD) per aconseguir la persistència de les seves instàncies.

Els SGBD són aplicacions especialitzades en l’emmagatzematge de dades estructurades, amb la capacitat de guardar-les i recuperar-les de forma consistent i amb gran eficiència, amb independència del nombre d’accessos que es realitzin simultàniament.

Cal tenir en compte, però, que la gestió dels SGBD requereix de coneixements tècnics força específics que dificulten als usuaris no especialitzats l’accés a les seves dades emmagatzemades.

Per tal d’aconseguir una simplificació de les tasques d’emmagatzematge, és possible automatitzar-les usant els llenguatges i les eines de consulta i administració propis dels SGBD. Tot i això, aquestes eines no són suficients per aconseguir que un usuari neòfit pugui utilitzar-les sense haver de necessitar un important esforç d’aprenentatge i planificació per traslladar les seves necessitats en una seqüència efectiva de tasques.

La indústria del desenvolupament d’aplicacions utilitza les bases de dades com a eines on derivar les complexes tasques d’emmagatzematge per tal de centrar els esforços en les dificultats pròpies de l’àmbit on es desenvoluparan les aplicacions, així com en l’accés a l’automatització de les tasques d’emmagatzematge requerides de manera que els usuaris puguin reduir la corba d’aprenentatge de les aplicacions que els calgui usar, acostant-se a formes més intuïtives d’acord amb els coneixements de la seva formació específica.

Per aconseguir coordinar els llenguatges de programació amb les potencialitats dels SGBD, la indústria ha desenvolupat un conjunt d’eines específiques per aconseguir connectar amb una determinada base de dades i enviar-li la seqüència de tasques que les aplicacions podran necessitar durant la seva execució.

L’evolució del software però, no ha avançat de forma paral·lela a la dels SGBD, per això es parla de desfasament entre els paradigmes usats en desenvolupament de programari i els usats per les eines d’emmagatzematge.

En l’apartat “Gestió de connectors” estudiarem de forma específica el desfasament que existeix entre la Programació Orientada a l’Objecte i els SGBD relacionals. Veurem també l’evolució dels connectors a bases de dades i acabarem centrant els esforços en els connectors usats pel llenguatge JAVA anomenats *drivers JDBC*.

Els connectors JDBC s'estudien tant des del vessant funcional (de quines classes i quins mètodes disposem per combinar en una aplicació) com des del vessant pràctic que mostra com generar codi genèric, eficient i de qualitat que pugui integrar-se en diverses aplicacions.

El coneixement dels connectors JDBC ens permet introduir a l'apartat "Eines de maptage objecte-relacional", que tracta d'un tipus de sofisticades eines orientades a reduir el desfasament, configurant una forma que permeti automatitzar el traspàs de dades entre el model orientat a objecte i el model relacional. Concretament estudiarem una tecnologia anomenada JPA, integrada a las darreres versions del JDK de JAVA.

En l'apartat "Bases de dades objecte-relacionals i orientades a objectes" s'estudien les iniciatives més punteres amb les que s'aconsegueix reduir el desfasament objecte-relacional des del propi sistema gestor, ja sigui adaptant la definició de les relacions i el llenguatge d'accés utilitzat o redefinint el paradigma de la base de dades i creant estructures que permetin clonar els models orientats a objectes. Ens referim a les bases de dades orientades a l'objecte.

Malgrat que aquesta iniciativa porta temps intentant quallar, el fort arrelament dels Sistemes Gestors de Bases de Dades Relacionals i les dificultats que el propi desenvolupament comporta atenuen la implantació massiva dels sistemes d'emmagatzematge orientats a objectes. Tot i això, aquests sistemes s'estan estenent per aplicacions en les quals el paradigma relacional no és capaç de donar una resposta efectiva: sistemes d'aplicacions en temps real, d'emmagatzematge local en dispositius limitats, etc.

Per realitzar la part pràctica s'ha escollit ObjectDB perquè es tracta d'una base de dades orientada a objectes que proporciona sistemes d'accés (concretament, utilitzarem JPA) que tant els utilitzarem per a accedir a aquesta base de dades orientada a objectes com per a accedir a bases de dades relacionals, però sense sortir-nos del paradigma de l'orientació a objectes. Aquest fet és molt avantatjós pel que fa a l'aprofitament i la portabilitat de programes entre diferents bases de dades.

Així mateix, comentarem la importància d'implementar i provar els exemples que la lectura us anirà presentant, ja que s'han pensat com una eina per anar assolint els conceptes a mida que avanci l'aprenentatge. El codi dels exemples es troba disponible a la secció d'Annexos. Són també importants les activitats proposades de cada apartat, ja que orienten l'estudiant indicant-li si està realitzant una assoliment adequat dels objectius planificats, a la vegada que consolidarà l'aprenentatge dels conceptes i procediments introduïts durant la lectura.

Resultats d'aprenentatge

En acabar aquesta unitat, l'alumne:

1. Desenvolupa aplicacions que gestionen informació emmagatzemada en bases de dades relacionals identificant i utilitzant mecanismes de connexió.

- Valora les avantatges i inconvenients d'utilitzar connectors.
- Utilitza gestors de bases de dades embeguts i independents.
- Utilitza el connector idoni en l'aplicació.
- Estableix la connexió.
- Defineix l'estructura de la base de dades.
- Desenvolupa aplicacions que modifiquen el contingut de la base de dades.
- Defineix els objectes destinats a emmagatzemar el resultat de les consultes.
- Desenvolupa aplicacions que fan consultes.
- Elimina els objectes un cop finalitzada la seva funció.
- Gestiona les transaccions.

2. Gestiona la persistència de les dades identificant eines de mapatge objecte relacional (ORM) i desenvolupant aplicacions que les utilitzen.

- Instal·la l'eina ORM.
- Configura l'eina ORM.
- Defineix els fitxers de mapatge.
- Aplica mecanismes de persistència als objectes.
- Desenvolupa aplicacions que modifiquen i recuperen objectes persistents.
- Desenvolupa aplicacions que realitzen consultes utilitzant el llenguatge SQL.
- Gestiona les transaccions

3. Desenvolupa aplicacions que gestionen la informació emmagatzemada en bases de dades objecte relacionals i orientades a objectes valorant les seves característiques i utilitzant els mecanismes d'accés incorporats.

- Identifica els avantatges i inconvenients de les bases de dades que emmagatzemen objectes.

- Estableix i tanca connexions.
- Gestiona la persistència d'objectes simples.
- Gestiona la persistència d'objectes estructurats.
- Desenvolupa aplicacions que realitzen consultes.
- Modifica els objectes emmagatzemats.
- Gestiona les transaccions.
- Prova i documenta les aplicacions desenvolupades

1. Gestió de connectors

Malgrat que durant els primers anys de la història de la informàtica, la persistència de les dades ha anat passant, de dècada en dècada, per diferents paradigmes de representació i emmagatzematge de dades, la tendència de canvi s'ha anat esvaint amb el creixement del paradigma relacional. Es tracta d'una tecnologia senzilla però molt eficient que ha sabut adaptar-se a la majoria de sistemes de dades que les empreses reclamaven i a un cost prou assequible com per prendre l'hegemonia absoluta del mercat des de l'últim quart del passat segle.

És cert que el model relacional té limitacions importants a l'hora de representar informació poc estructurada, o estructures excessivament dinàmiques i complexes, però malgrat que tot sembli apuntar a un canvi de paradigma imminent, aquest no acaba d'arribar. La principal raó la trobem en la solidesa i maduresa que els sistemes gestors de bases de dades relacionals tenen.

De fet, molts autors apunten cap a una evolució dels sistemes relacionals incorporant eines i tecnologies que els apropin al model orientat a objectes més que no pas cap a la seva desaparició i substitució.

1.1 El desfasament objecte-relacional

Quan necessitem explicar o plasmar una realitat complexa, recorrem sovint a la simplificació construint un model conceptual més senzill que es comporti de forma similar a la realitat. Es tracta de plasmar els aspectes essencials i, a la vegada, alleugerir els detalls insignificants per tal de poder rebaixar la complexitat.

L'ús de models conceptuals durant la implementació d'aplicacions informàtiques és d'una importància extrema per poder portar a bon termini qualsevol projecte d'informatització.

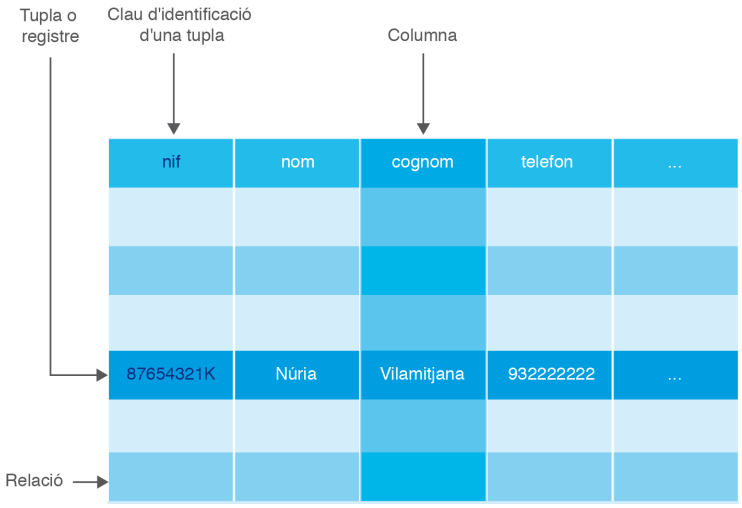
El problema és que els models conceptuals són representacions mentals fruit d'un procés d'abstracció. No hi ha una única forma de plasmar-los o representar-los fora del nostre cervell. Sovint fem servir aproximacions esquemàtiques que poden acostar-se força a la representació mental, però fins i tot les representacions esquemàtiques són difícilment representables en la memòria d'un ordinador.

Els sistemes relacionals plasmen el model en els diagrames entitat relació, els quals serveixen de base per definir l'esquema de taules i relacions necessari per suportar la casuística a representar.

1.1.1 Model relacional

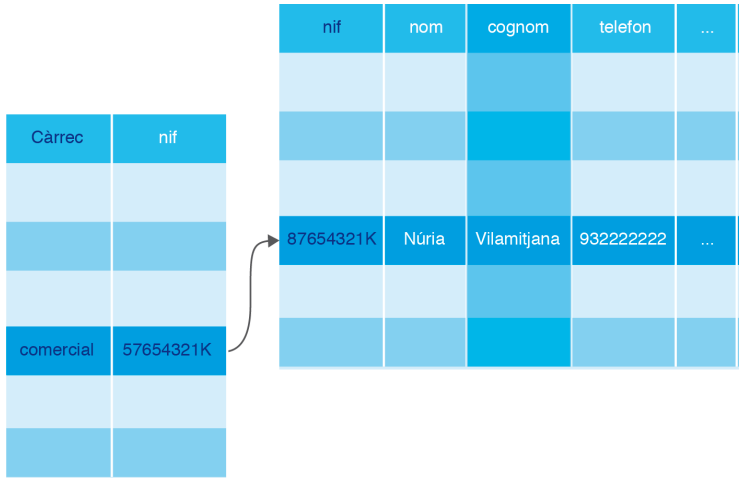
El model relacional és una forma d'organitzar les dades d'una aplicació agrupant-les segons la relació que es pugui establir entre elles d'acord amb el model. Aquesta agrupació es materialitza en forma de taula, on distingim les columnes o conjunt de dades que representen un mateix concepte del model de dades i les tuples o registres que representen entitats diferenciades a l'aplicació.

Figura 1.1. Components d'un sistema relacional



Les taules mantenen ben relacionades les dades corresponents a cada registre, d'acord amb el concepte que representen (NIF, nom, etc.). Els registres s'identifiquen per mitjà del valor d'una columna o d'un conjunt de columnes anomenades *clau principal*. El valor de la clau principal no pot estar mai repetit, de manera que hi hagi una correspondència única entre registres i claus(figura 1.1).

Figura 1.2. Relació forana



En models complexos seran necessàries múltiples taules, els registres de les quals poden relacionar-se també a partir de claus foranes (figura 1.2). Les claus foranes identifiquen registres d'altres taules (a partir de la seva clau primària) de forma

que sigui fàcil identificar totes les dades relacionades amb una determinada entitat malgrat que pertanyin a taules diferents.

El model relacional també permet definir un conjunt de regles i limitacions en els valors de les dades i en les accions a realitzar, amb l'objectiu d'assegurar la consistència de les dades. Així, és possible indicar què cal fer amb els registres d'una taula que es trobin vinculats al registre d'una segona taula en el moment d'eliminar-lo. També permet, per exemple, definir el rang o conjunt de valors possibles que un camp d'una taula podrà prendre, o bé assegurar la no repetició de determinats valors en diferents registres d'una mateixa taula, etc.

1.1.2 Model orientat a l'objecte

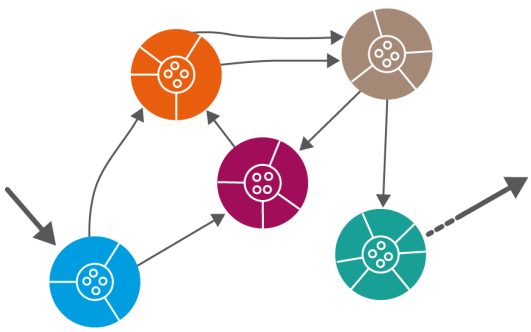
La tecnologia dels objectes ha fet en els darrers temps un esforç molt important per regular i estandarditzar les representacions esquemàtiques dels models conceptuals. El resultat d'aquests esforços ha conduït a un llenguatge esquemàtic però molt expressiu anomenat UML.

Els llenguatges com UML són capaços de descriure amb molta precisió la majoria de models conceptuals sense necessitat de fer gaires adaptacions, perquè disposen d'una gran riquesa semàntica i expressivitat.

Els objectes poden representar qualsevol element del model conceptual, una entitat, una característica, un procés, una acció, una relació... Els objectes són fruit d'un procés d'abstracció centrat en les característiques importants (dades), però també en el comportament o la funcionalitat que tindran en el moment de materialitzar-se durant l'execució de les aplicacions (codi).

La importància de centrar el model en els objectes és, doncs, múltiple. En referència a les dades, els objectes actuen d'estructures jeràrquiques, de manera que la informació queda sempre perfectament contextualitzada dins els objectes.

FIGURA 1.3. Forma de treballar d'una aplicació orientada a l'objecte



Això ho podeu apreciar en la figura 1.3, on les figures circulars representen objectes de diferents tipus segons el color. En el centre dels objectes, les petites figures representen les dades capsulades o l'estat dels objectes, inaccessibles de

forma directa. Les corones circulars exteriors representen els mètodes, els quals a petició d'altres objectes (fletxes) poden consultar o manipular l'estat de l'objecte.

Efectivament, des del paradigma orientat a l'objecte no té sentit referir-nos a una variable isolada. Per exemple, en una aplicació de nòmines l'antiguitat estarà sempre associada (i continguda) a un objecte empleat; igual que el nom, l'edat, la categoria i la resta de dades significatives necessàries per dur a terme el càlcul de les nòmines.

És a dir, les dades es trobaran sempre perfectament relacionades entre elles de forma similar a la relació que s'estableix en una taula en el paradigma relacional. La diferència, però, es troba en el fet que la perspectiva relacional només manté aquesta relació dins la taula, mentre que la perspectiva orientada al'objecte l'estén a tota l'aplicació incorporant-la en el propi codi d'execució.

En referència al comportament o la funcionalitat, els objectes delimiten les accions a realitzar sobre les seves dades i sobre la resta d'objectes, definint les regles del joc del que està permès durant l'execució de les aplicacions.

El conjunt de dades que conformen un objecte s'anomena també **estat**, perquè permet descriure l'evolució de qualsevol objecte durant l'execució d'una aplicació, des del moment de la seva creació fins que siguin eliminats de la memòria.

En aquest paradigma, l'estat dels objectes té un paper fonamental perquè esdevé el fil conductor de l'execució de les aplicacions. Les dades inicials, els càlculs i els resultats es plasmaran sempre, mentre duri l'execució en forma d'*estat dels objectes*, el qual anirà evolucionant de forma coordinada en pro dels objectius de l'aplicació, a mida que es vagin produint interaccions entre el propis objectes.

La interacció es realitza fent crides als mètodes dels objectes (operacions disponibles segons els tipus o classe). És a dir, no es manipulen directament les dades, sinó que es sol·liciten canvis d'estat o consultes als propietaris de la informació.

OO és l'acrònim d'Orientat a Objectes. Així, si parlem de llenguatge OO ens referirem al llenguatge orientat a objectes, i si parlem de paradigma OO haurem de llegir paradigma orientat a objectes.

En el paradigma OO, la interacció entre objectes queda totalment pautada a través del tipus de relacions que establim en el model. Parlarem d'un objecte relacionat amb un altre quan el primer pugui accedir als mètodes del segon. Les relacions definides al model OO indicaran quins tipus objectes podran interactuar i com.

Malauradament, el model conceptual és un model eminentment dinàmic que no contempla, a priori, la persistència dels seus objectes. Això, que a primer cop d'ull pot semblar un defecte del paradigma, és en el fons una oportunitat d'implementar biblioteques i marcs de persistència que treballin de forma totalment transparent al model i al seu funcionament.

Els marcs de persistència s'encarregaran de posar en escena els objectes inicialitzant-los en l'estat en què es trobaven emmagatzemats en el moment de la instanciació, mentre que la lògica del model continguda en els mètodes dels

objectes els farà evolucionar a partir d'aquest estat inicial. Els marcs de persistència aniran emmagatzemant periòdicament l'evolució dels objectes a mida que es vagin produint els canvis, de manera que hi hagi sempre una correspondència entre els objectes en memòria i els seus estats emmagatzemats.

1.1.3 Desfasaments

ER és l'acrònim d'Entitat-Relació, en referència a les relacions que s'estableixen entre les diferents entitats que configuren un model conceptual.

El principal problema que trobem en intentar automatitzar el procés de persistència dels objectes d'una aplicació en un SGBD sota el paradigma relacional, és que es tracta de conceptualitzacions diferents i centrades en aspectes també diferents. Mentre el paradigma ER es troba fortament centrat en les dades i l'estructura que cal donar a aquestes per poder emmagatzemar-les i recuperar-les d'acord al model conceptual, el paradigma OO es troba centrat en els objectes, entesos com a agrupacions de dades i també com a processos de canvi.

Cal observar que el model relacional necessitarà sempre certa quantitat d'informació extra destinada a mantenir les relacions i la coherència de les dades. No és informació pròpia del model, sinó que cal afegir-la per garantir el bon funcionament. Les claus foranes són l'exemple més clar. Es tracta d'informació afegida en alguns registres per tal de vincular-los a uns altres.

La vinculació entre objectes, en canvi, s'aconsegueix de forma estructural. No es necessiten dades extres, sinó que la mateixa estructura de dades defineix la vinculació, la visibilitat, l'accés, etc.

El model relacional, malgrat que pot emmagatzemar també alguns procediments cridats a voluntat del programador (*procedures*) o bé associats a certes accions (*triggers*), fa servir una lògica d'implementació i d'ús força menys evident que la simple codificació de mètodes en les classes del model OO.

Aquestes diferències constitueixen el que en el món de la programació es coneix com a*desfasament objecte-ralacional*. Aquest desfasament ens obliga, quan decidim treballar conjuntament amb ambdós paradigmes, a codificar implementacions extres que funcionin a mode d'adaptadors.

El paradigma ER, per exemple, disposa d'un conjunt de llenguatges (DDL, DCL, SQL, etc.) adequats per explotar al màxim els SGBD tenint en compte les característiques relacionals, mentre que el paradigma OO treballa bàsicament amb llenguatges de programació imperatius dissenyats principalment per agilitzar els processos de càlcul de dades estructurades referenciades per variables. Compatibilitzar-los implicarà crear implementacions que permetin incorporar sentències de llenguatges específicament relacionals dins del llenguatge usat per implementar els objectes.

Un altre exemple de desfasament el trobem també en els resultats recuperats des d'un SGBD. Aquests s'obtenen sempre en un format tabular i, per tant, caldrà

implementar utilitats que transformin les seqüències de dades simples en estats dels objectes de l'aplicació.

1.2 Connectors

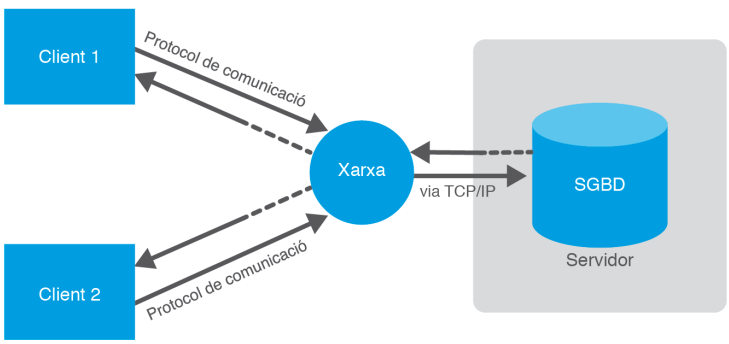
La història de les bases de dades relacionals ha transcorregut íntimament lligada a la història de les aplicacions d'arquitectura distribuïda i en particular, arquitectures client-servidor. Fa poc més d'un quart de segle, la persistència de les aplicacions formava part del desenvolupament d'aquestes, que es concebien com un tot monolític. L'emmagatzematge i la recuperació de les dades es barrejava i difuminava amb l'explotació de les mateixes. Les bases de dades solien disposar de llenguatges de programació propis que incorporaven crides i sentències específiques d'accés a les dades.

En realitat, aquesta situació no representava gaires avantatges ni per a les empreses desenvolupadores dels SGBD ni per a les empreses usuàries. Les primeres es trobaven que mantenir el desenvolupament d'un llenguatge de programació resultava realment costós si no es volia quedar ràpidament desfasat. Les empreses usuàries,d'altra banda, es trobaven lligades a un llenguatge de programació que no sempre els donava resposta a totes les seves exigències. A més, plantejar-se qualsevol canvi de sistema gestor de dades representava una fita impossible, atès que implicava haver de programar de nou totes les aplicacions de l'empresa. Calia desvincular els llenguatges de desenvolupament dels SGBD.

1.2.1 ODBC

A mida que les teories de dades relacionals anaven agafant força i les xarxes guanyaven adeptes gràcies a l'increment de l'eficiència a preus realment competitius, van començar a implementar-se uns sistemes gestors de bases de dades basats en la tecnologia client-servidor.

Figura 1.4. Estructura client-servidor adoptada pels SGBD



La tecnologia client-servidor va permetre aïllar les dades i els programes específics d'accés a les mateixes, del desenvolupament de l'aplicació (figura 1.4). La raó principal d'aquesta divisió va ser segurament possibilitar l'accés remot a les dades de qualsevol ordinador connectat a la xarxa. El cert, però, es que aquest fet va empènyer els sistemes de bases de dades a desenvolupar-se d'una forma aïllada i a crear protocols i llenguatges específics per poder-se comunicar remotament amb les aplicacions que corrien en ordinadors externs.

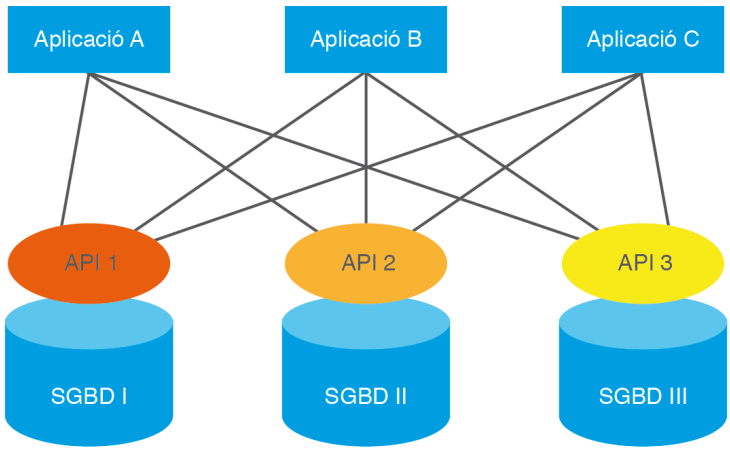
A poc a poc, el programari al voltant de les bases de dades va créixer espectacularment intentant donar resposta a un màxim ventall de demandes a traves de sistemes altament configurables. És el que avui dia es coneix com a *middleware* o capa intermèdia de persistència. És a dir, el conjunt d'aplicacions, utilitats, biblioteques, protocols i llenguatges, situats tant a la part servidor com a la part client, que permeten connectar-se remotament a una base de dades per configurar-la o explotar-ne les seves dades.

L'arribada dels estàndards

Inicialment, cada empresa desenvolupadora d'un SGBD implementava solucions propietàries específiques per al seu sistema, però aviat van adonar-se que col·laborant conjuntament podien treure'n major rendiment i avançar molt més ràpidament.

Sostenint-se en la teoria relacional i en algunes implementacions primerenques de les empreses IBM i Oracle, es va desenvolupar el llenguatge de consulta de dades anomenat SQL. Aquest repte va suposar, sens dubte, un gran pas, però les aplicacions necessitaven API amb funcions que permetessin fer crides des del llenguatge de desenvolupament per enviar les consultes realitzades amb l'SQL (figura 1.5).

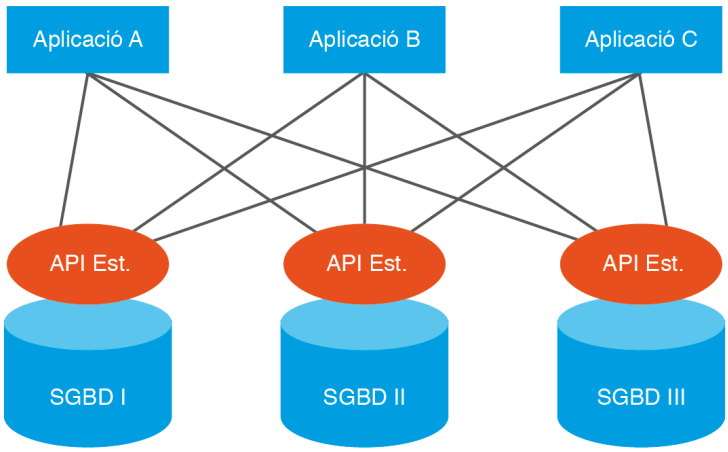
Figura 1.5. Sistema de connexió propietari



Cada SGBD té la seva pròpia connexió i el seu propi API.

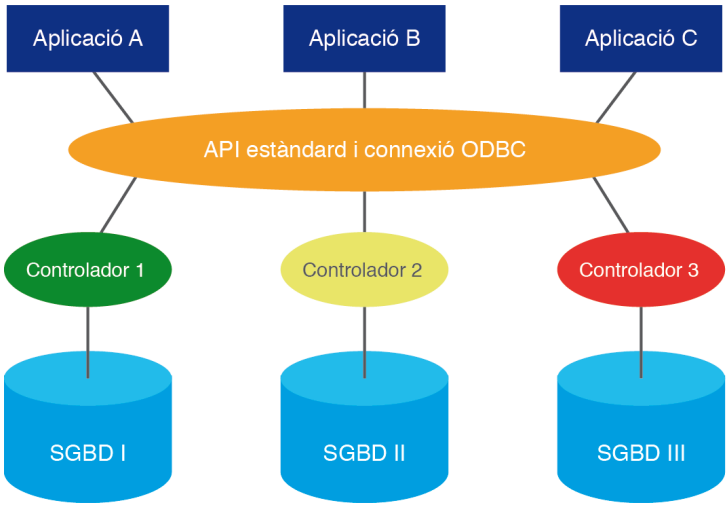
El grup anomenat SQL Access Group, en el qual hi participaven prestigioses empreses del sector com Oracle, Informix, Ingres, DEC, Sun o HP, va definir un API universal amb independència del llenguatge de desenvolupament i la base de dades a connectar (figura 1.6).

FIGURA 1.6. Sistema de connexió propietari amb un API estàndard



El 1992, Microsoft i Simba implementen l'ODBC (Open Data Base Connectivity), un API basat en la definició de l'SQL Acces Group, que s'integra en el sistema operatiu de Windows i que permet afegir múltiples connectors a diverses bases de dades SQL de forma molt senzilla i transparent, ja que els connectors són autoinstal·lables i totalment configurables des de les mateixes eines del sistema operatiu (figura 1.7).

FIGURA 1.7. Sistema de connexió ODBC configurat usant diferents controladors (drivers) i un API estàndard



L'arribada de l'ODBC va representar un avenç sense precedents en el camí cap a la interoperabilitat entre bases de dades i llenguatges de programació. La majoria d'empreses desenvolupadores de sistemes gestors de bases de dades

van incorporar els *drivers* de connectivitat a les utilitats dels seus sistemes i els llenguatges de programació més importants van desenvolupar biblioteques específiques per suportar l'API ODBC.

La situació actual

Actualment, ODBC continua sent una adequada iniciativa de connexió als SGBD relacionals. El seu desenvolupament segueix liderat per Microsoft, però existeixen versions en altres sistemes operatius aliens a la companyia com UNIX/LINUX o MAC. Els llenguatges més populars de desenvolupament mantenen actualitzades les biblioteques de comunicació amb les successives versions que han anat apareixent i la majoria d'SGBD disposen d'un controlador ODBC bàsic.

Actualment, l'ODBC s'estructura en tres nivells. El primer, anomenat *core API*, és el nivell més bàsic corresponent a l'especificació original (basada en l'SQL Access Group). El *Level 1 API* i el *Level 2 API* afegeixen funcionalitats avançades, com les crides a procediments emmagatzemats en el sistema, aspectes de seguretat d'accés, definició de tipus estructurats, etc.

En realitat, l'ODBC és una especificació de baix nivell, és a dir, de funcions bàsiques que possibiliten la connexió, que asseguren l'atomicitat de les peticions, el retorn d'informació, el capsulament del llenguatge de consulta SQL o l'obtenció de dades aconseguides en resposta a un petició.

La funcionalitat de baix nivell fa que es pugui adaptar a un gran nombre d'aplicacions; això sí, a costa d'un considerable nombre de línies de codi necessàries per adaptar-se a la lògica de cada aplicació. És per això que sobre la base de l'ODBC han sorgit altres alternatives de persistència de més alt nivell. Per exemple, Microsoft ha desenvolupat OLE DB o ADO.NET. Aquest darrer abraça ja el paradigma orientat a objectes per a qualsevol tipus d'aplicació basada en la plataforma .NET.

1.2.2 JDBC

Gairebé de forma simultània a ODBC, l'empresa Sun Microsystems, l'any 1997 va treure a la llum JDBC, un API connector de bases de dades, implementat específicament per usar amb el llenguatge Java. Es tracta d'un API força similar a ODBC quant a funcionalitat, però adaptat a les especificitats de Java. És a dir, la funcionalitat es troba capsulada en classes (ja que Java és un llenguatge totalment orientat a objectes) i a més, no depèn de cap plataforma especifica, d'acord amb la característica multiplataforma defensada per Java.

Aquest connector serà l'API que estudiarem en detall en aquestaunitat, ja que Java no disposa de cap biblioteca específica ODBC. Les raons esgrimides per *Sun* són que ODBC no es pot fer servir directament en Java ja que està implementat en C i no és orientat a objectes. En altres paraules, usar ODBC des del llenguatge Java

necessitaria de totes totes una biblioteca que adaptés l'API ODBC als requeriments Java.

Sun Microsystemha optat per una solució que permet fer ambdues coses a la vegada; per un cantó ha implementat un connector específic de Java que pot comunicar-se directament amb qualsevol base de dades fent servir controladors (*drivers*) de manera molt similar a com es fa en ODBC, i per l'altra banda, incorpora de sèrie un *driver* especial que actua d'adaptador entre l'especificació JDBC i l'especificació ODBC. Aquest controlador s'acostuma a anomenar també pont (*bridge* en anglès) JDBC-ODBC. Usant aquest *driver* podrem enllaçar qualsevol aplicació Java amb qualsevol connexió ODBC.

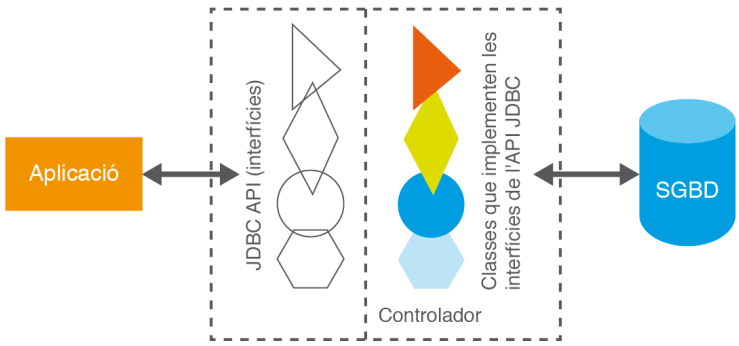
Actualment, la gran majoria d'SGBD disposen de *drivers* JDBC, però en cas d'haver de treballar amb un sistema que no en tingui, si disposa de controlador ODBC, podrem fer servir el pont JDBC-ODBC per aconseguir la connexió des de Java.

Arquitectura JDBC

Igual que ODBC, cada desenvolupador de sistemes gestors de bases de dades implementa els seus propis controladors JDBC. Per tal d'aconseguir la interoperabilitat entre controladors, la biblioteca estàndard JDBC conté un gran nombre d'interfícies sense les classes que les implementen. Cada controlador de cada fabricant incorpora les classes que implementen les interfícies de l'API JDBC. D'aquesta manera, el controlador utilitzat serà totalment transparent a l'aplicació, és a dir, durant el desenvolupament de l'aplicació no caldrà saber quin serà el *driver* que finalment es farà servir per a l'explotació de les dades, ja que l'aplicació declararà exclusivament les interfícies de l'API JDBC (figura 1.8).

D'aquesta manera s'aconsegueix independitzar l'aplicació dels controladors permetent la substitució del controlador original per qualsevol altre compatible JDBC sense pràcticament necessitat d'haver de modificar el codi de l'aplicació.

FIGURA 1.8. Esquema que simbolitza l'arquitectura JDBC



D'una banda torbem les interfícies definides a l'estàndard i incloses al JDK. Es tracta de l'API amb el que l'aplicació treballarà de forma directa. De l'altra banda trobem les classes específiques del controlador (driver) que interaccionen amb el SGBD i que implementen les interfícies de l'estàndard JDBC.

Parlem de biblioteques dinàmiques quan aquestes no estan integrades dins el fitxer executable, sinó que estan ubicades en fitxers externs però poden cridar-se des de qualsevol executable.

És important destacar també que JDBC no exigeix cap instal·lació, ni cap canvi substancial en el codi a l'hora de fer servir un o altre controlador. Aquesta característica se sustenta, en primer lloc, en la utilitat de Java que permet carregar programàticament qualsevol classe a partir del seu nom; en segon lloc, en la funcionalitat de la classe `DriverManager` (de l'API JDBC), que sense necessitat d'indicar-li el *driver* específic que cal fer servir és capaç de trobar-lo i seleccionar-lo d'entre tots els que el sistema tingui carregats en memòria. En darrer lloc, també se sustenta en la manera com s'instancien i obtenen els objectes JDBC responsables de la comunicació amb l'SGBD, ja que partint d'una única classe principal (la classe`Driver`) anirem obtenint totes les instàncies de les interfícies JDBC sense necessitat d'haver de conèixer quines són les classes que les implementen. D'aquesta manera, JDBC només necessitarà conèixer el nom de la classe principal (Driver) per començar a ser operatiu.

Diferències entre biblioteques de llenguatges compilats i biblioteques del llenguatge JAVA.

En llenguatges compilats com C, Pascal o Basic, cal generar un fitxer executable per fer córrer l'aplicació. Si l'executable fa servir biblioteques estàtiques, aquestes estaran contingudes dins el propi executable. Si l'executable fa servir biblioteques dinàmiques, aquestes han d'estar contingudes en algun fitxer del sistema.

El llenguatge Java no genera cap fitxer executable ja que la màquina virtual llegeix, carrega i executa directament els fitxers compilats. Aquesta característica fa que en Java, totes les seves biblioteques siguin sempre dinàmiques.

La màquina virtual reconeix les classes que en cada moment li cal carregar en memòria gràcies a les sentències *import*. Es tracta de la relació de les dependències que una classe té amb d'altres.

Malgrat tot, Java també disposa d'una utilitat per demanar la càrrega de qualsevol classe de manera programàtica indicant només el seu nom. Ens referim a la sentència `Class.forName` ("nom.de.la.Classe").

Tot i això, cal tenir en compte que els controladors els implementa cada fabricant de l'SGBD. Per tal d'adaptar-se a cada un dels fabricants, JDBC accepta diferents tipus de controladors, escrits fins i tot en un llenguatge diferent a Java i que, per tant, podrien requerir de certa instal·lació i configuració específica. A la pràctica, la complexitat o facilitat de la interoperabilitat entre *drivers* es pot veure condicionada pel tipus de *driver* que decidim fer servir.

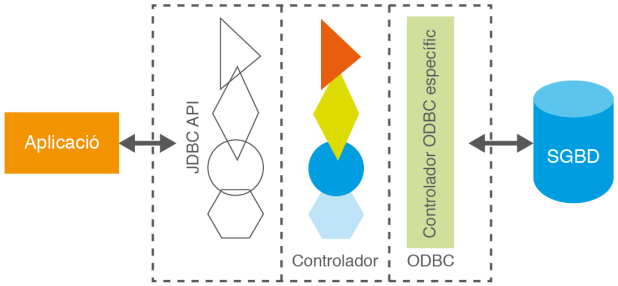
Tipus de controladors

JDBC distingeix quatre tipus de controladors:

1. Tipus I. Controladors **pont** (*bridge driver*) com JDBC-ODBC. Es caracteritzen per fer servir una tecnologia aliena a JDBC i actuar d'adaptador entre les

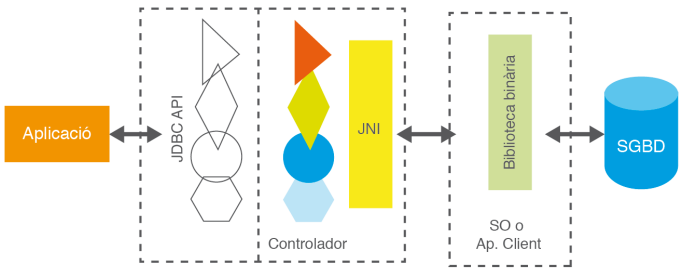
especificacions de l'API JDBC i la tecnologia concreta utilitzada. El més conegut és el controlador pont JDBC-ODBC, però n'hi ha d'altres, com JDBC-OLE DB. La seva principal raó de ser és la de permetre usar una tecnologia molt estesa i assegurar així la connexió amb pràcticament qualsevol font de dades. Per contra, cal dir que és necessari que cada client tingui instal·lada una utilitat de gestió i configuració de fonts de dades ODBC (o de la tecnologia utilitzada) a més del *driver* ODBC específic de l'SGBD que caldrà tenir instal·lat i configurat. El fet d'haver de connectar-se usant un adaptador que fa de pont pot en ocasions donar problemes de rendiment i, per tant, s'aconsella de fer servir només com a darrera alternativa (figura 1.9).

FIGURA 1.9. Esquema JDBC usant un controlador de tipus I



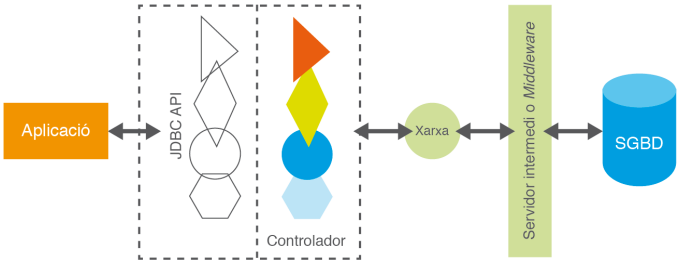
2. Tipus II. Controladors de **Java amb API parcialment nadiu** (*Native-API partly Java driver*). S'anomenen també simplement *nadius*. Com el seu nom indica, estan formats d'una part codificada en Java i una altra part que usa biblioteques binàries instal·lades en el sistema operatiu. Aquest tipus de controladors existeixen perquè alguns sistemes gestors de dades tenen entre les seves utilitats de sèrie connectors propis del sistema gestor. Solen ser connectors propietaris que no segueixen cap estàndard, ja que acostumen a ser anteriors a ODBC o JDBC, però es mantenen degut al fet que solen estar molt optimitzats i són molt eficients. Usant una tecnologia Java anomenada JNI és possible d'implementar classes, els mètodes de les quals invoquen funcions de biblioteques binàries instal·lades en el sistema operatiu. Els controladors de tipus II usen aquesta tecnologia per crear les classes implementadores de l'API JDBC. En alguns casos pot requerir una instal·lació extra de certes utilitats a la part client, exigides pel connector nadiu del sistema gestor (figura 1.10).

FIGURA 1.10. Esquema JDBC usant un controlador de tipus II



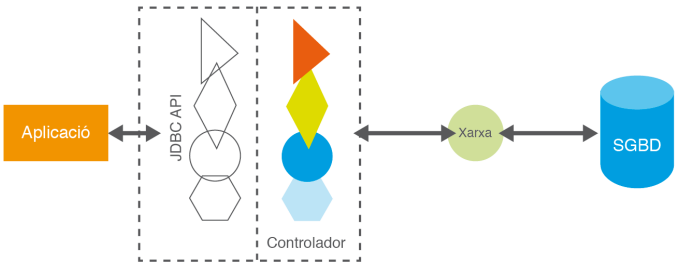
3. Tipus III. Controladors de **Java via protocol de xarxa**. Es tracta d'un controlador escrit totalment en Java que tradueix les crides JDBC a un protocol de xarxa contra un servidor intermedi (anomenat comunament *Middleware*) que pot estar connectat a diversos SGBD. Aquest tipus de *driver* presenta l'avantatge que usa un protocol independent dels SGBD i, per tant, el canvi de font de dades es pot fer de manera totalment transparent als clients. Això el converteix en un sistema molt flexible, malgrat que per contra, es necessitarà instal·lar, en algun lloc accessible de la xarxa, un servidor intermedi connectat a tots els SGBD que calgui. Aquest tipus de controladors són força útils quan hi ha un nombre molt gran de clients, ja que els canvis d'SGBD no requeriran cap canvi en els clients, ni tan sols la incorporació d'una nova biblioteca (figura 1.11).

FIGURA 1.11. Esquema JDBC usant un controlador de tipus III



4. Tipus IV. Controladors de tipus **Java pur**o **Java 100%**. S'anomenen també controladors de *protocol nadiu*. Són controladors escrits totalment en Java. Les crides al sistema gestor es fan sempre a través del protocol de xarxa que usa el propi sistema gestor i, per tant, no es necessita ni codi nadiu en el client ni servidor intermedi per connectar amb la font de dades. Es tracta, doncs, d'un *driver* que no requereix cap tipus d'instal·lació ni requeriment, la qual cosa el fa ser una alternativa força considerada que en els darrers temps ha acabat imposant-se. De fet, la majoria de fabricants han acabat creant un controlador de tipus IV tot i que segueixin mantenint també els dels altres tipus (figura 1.12).

FIGURA 1.12. Esquema JDBC usant un controlador de tipus IV



Requisits previs

Podeu baixar-vos un fitxer autoinstal·lable a l'adreça bit.ly/1gk4Q1E des d'on podreu seleccionar la vostra plataforma.

Abans de començar a desenvolupar aplicacions JDBC cal que assegurem que tenim instal·lat l'SGBD, i a més que hi tenim accés des del lloc on estiguem desenvolupant l'aplicació. Donarem per fet que teniu instal·lada la darrera versió

de Postgresql (PostgreSQL 9.1.2 o PostgreSQL 9.0.6), que serà l'SGBD que usarem per donar-vos les solucions oficials.

Definició i configuració dels SGBD

Malgrat que l'explotació de dades d'un sistema gestor s'ha convertit en un procés molt estàndard gràcies a l'ús de l'SQL, la definició i configuració de cada sistema depèn exclusivament del propi SGBD. És per això que recomanem l'ús de PostgreSQL per resoldre els exercicis, ja que és l'SGBD escollit per a aquest estudis i l'únic del qual oferirem suport tècnic.

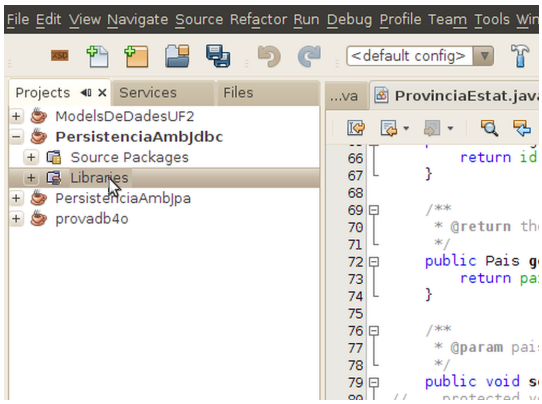
Podeu obtenir el controlador de PostgreSQL cercant entre la col·lecció de diversos controladors que Java posa al nostre abast a l'adreça: bit.ly/2lohwD0. Si ho preferiu, també podeu accedir directament a bit.ly/2l32ouy per obtenir el controlador.

Un cop verificat l'accés al sistema gestor de dades, caldrà obtenir el controlador JDBC del sistema gestor, en el nostre cas PostgreSQL. Generalment, cada fabricant posarà a disposició dels seus usuaris els diferents tipus de controladors que tingui per als seus productes. És habitual que disposi de diferents tipus de controladors, de manera que puguem decidir el que millor s'adapti a les nostres necessitats. Per comoditat, recomanem fer servir *drivers* de tipus IV que en tractar-se de biblioteques 100% Java no requereixen cap mena d'instal·lació. Si fos necessari usar controladors d'un altre tipus, caldrà seguir les indicacions del fabricant.

Aquí suposarem que el controlador és de tipus IV, o bé que ja s'han seguit les indicacions d'instal·lació i configuració del fabricant.

Sigui quin sigui el tipus de controlador que finalment necessiteu, aquest tindrà com a mínim una biblioteca en format .jar amb totes les classes de l'API JDBC. Caldrà afegir el fitxer .jar com a biblioteca de la nostra aplicació. Durant el desenvolupament, si usem NetBeans, caldrà situar el cursor del ratolí a *Libraries*, clicar el botó dret, escollir *Add JAR/Folder...* i navegar pel sistema de fitxers fins seleccionar el JAR del controlador que prèviament haurem descarregat (figura 1.13).

FIGURA 1.13. Captura de pantalla que il·lustra com afegir el fitxer jar com a biblioteca del projecte



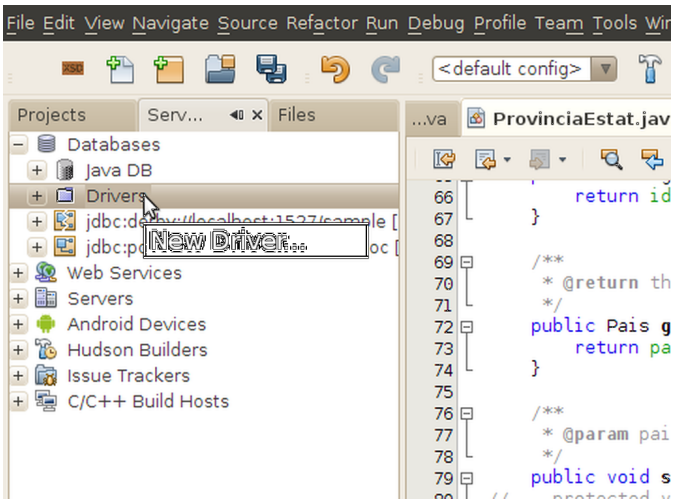
Per fer-ho, caldrà situar el cursor del ratolí a Libraries, clicar el botó dret, escollir add JAR/Folder... i navegar pel sistema de fitxers fins seleccionar el fitxer desitjat.

NetBeans disposa d'una eina força útil quan treballem amb bases de dades que permet definir connexions via controlador JDBC i manipular directament la base de dades. És una utilitat gràfica força completa que ens ajudarà a crear taules, relacions, esquemes, visualitzar i manipular les dades, etc. Recomanem que l'activeu i configureu per connectar-vos a PostgreSQL. No és necessari fer-la servir per desenvolupar aplicacions JDBC, però sol ser de força utilitat durant el desenvolupament.

En primer lloc, caldrà que accediu a la pestanya *Services* de la zona superior esquerra de l'IDE. El primer que caldrà fer serà afegir el *driver* a la utilitat. Despleguem l'ítem *Databases*, si no estigués desplegat, i situem el cursor del ratolí

sobre l'ítem *Drivers*, cliquem amb el botó dret i escollim *New Driver...* Això ens permetrà navegar pel sistema de fitxers fins que trobem el controlador adequat (figura 1.14).

FIGURA 1.14. Afegint el driver a la utilitat de NetBeans de connexió a bases de dades



Un cop afegit el *driver*, crearem una nova connexió. Situarem el cursor del ratolí damunt de l'ítem *Databases*, clicarem de nou el botó dret per fer emergir el menú de context i seleccionarem *New Connection...* Això obrirà un quadre de diàleg. Al camp *Driver*, seleccioneu PostgreSQL i un cop seleccionat, al camp *Driver File(s)* us apareixerà la ruta del controlador que heu afegit abans. Seleccioneu-lo i cliqueu *Next*. Ara cal que ompliu cada una de les opcions d'acord amb les dades del vostre SGBD. Si el teniu instal·lat a la mateixa màquina des d'on esteu desenvolupant, caldrà que escriviu *localhost* al camp *Host* i si no heu canviat el port que PostgresSQL s'assigna per defecte, el seu valor serà 5432. En el camp Database cal que afegiu la base de dades que desitgeu fer servir en les vostres proves. Escrivint el *Nom d'usuari* i el *Password* podeu activar el test de connexió per comprovar que les dades siguin correctes. Si tot funciona correctament, cliqueu *Finish* per crear la connexió i acabar.

1.3 Iniciació a l'API JDBC

Ara verem els elements bàsics de l'API JDBC que permeten a les aplicacions Java comunicar-se amb un SGBD fent servir el llenguatge SQL. Cal que disposeu del connector JDBC de PostgreSQL i que l'afegiu a les biblioteques del vostre projecte. També serà necessari que habiliteu una connexió per consultar la base de dades sense sortir de l'IDE.

Per tal de poder realitzar unes proves inicials que us mostraran el funcionament de l'API JDBC, sense "embrutar" la base de dades, podeu crear des de l'administrador de l'SGBD un contenidor (*Database*) per poder realitzar algunes proves. D'ara endavant suposarem que heu creat una base de dades anomenada ioc_proves propietat de l'usuari *ioc*, la contrasenya del qual és *ioc* i així ens hi referirem.

Per iniciar-nos en el món JDBC, crearem un petit programa que aconsegueixi connectar-se a la base de dades ioc_provesi fer senzilles operacions.

Crearem un projecte nou. Per exemple, anomenat `PersistenciaAmbJdbc`. Un cop creat hi afegirem un paquet on emplaçar-hi l'aplicació. Per exemple: ioc.dam.m6.provesbasiques.Seguidament crearem la classe `ProvesBasiques` amb el mètode main. Abans de començar afegirem el controlador JDBC de PostgesSQL com a biblioteca del projecte i crearem un nou mètode a la classe que anomenarem `provaDeConnexio`.

1.3.1 Càrrega de controladors

Generalment, una aplicació es pot compondre d'un nombre tan elevat de classes que la màquina virtual no pot tenir-les carregades totes en memòria. A mida que va sent necessari, la màquina virtual s'encarregarà de localitzar els fitxers compilats (.class) en el directori des d'on s'està executant, en aquelles carpetes que formin part del *classpath* o en aquells fitxers .jar enumerats també en el *classpath*.

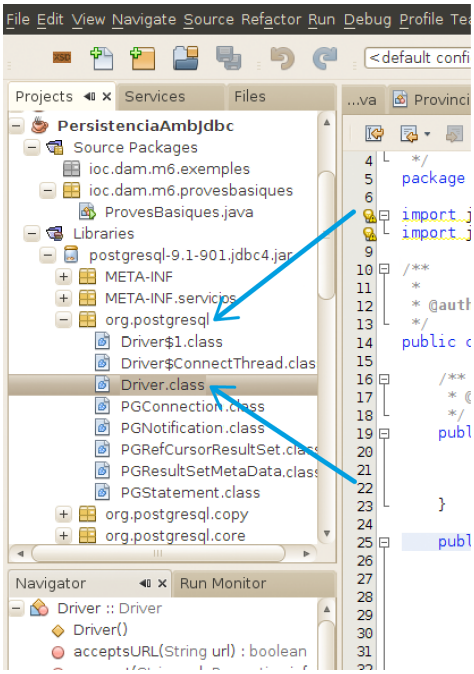
Normalment, la màquina virtual descobreix la localització exacta de la ruta on es troba la classe a carregar analitzant les sentències *import*. En general és una forma força útil i eficient de detectar la ubicació de les classes d'una aplicació. El problema, però,és que per poder-lo usar necessitem conèixer a priori la classe que farem servir. A més, un cop escrita la sentencia import, si algun dia necessitem reanomenar la classe o decidim usar una classe equivalent d'un altre paquet, necessitarem reescriure el codi canviant la sentència `import` i recompilant-lo de nou per fer efectius els canvis.

No és un problema menor, perquè la recompilació implica haver de substituir els arxius binaris en cada màquina client, tornar a fer proves per validar els canvis, etc.

Sortosament, Java disposa d'una altra sentència per localitzar els fitxers compilats en el moment de la càrrega. Es tracta d'un mètode que accepta com a paràmetre una cadena de text amb el nom complet de la classe a carregar (paquet i nom de classe). El mètode `Class.forName`informarà a la màquina virtual de quina classe li cal carregar a partir de la sentència. Malgrat que en aquesta prova inicial escriurem el nom de la classe directament al codi, el fet de poder expressar el nom de la classe com una cadena de caràcters permet que les aplicacions reals parametritzin la dada de manera que siguin fàcilment configurables.

Qualsevol controlador JDBC disposa d'una classe especial anomenada generalment `Driver`, encarregada d'establir la connexió amb el nostre sistema gestor. En realitat el nom i el paquet de la classe depenen de cada fabricant i, per tant, caldrà consultar la documentació per conèixer el nom de la classe.

Figura 1.15. Classe Driver cercada des de NetBeans



Podeu desplegar la biblioteca des de NetBeans intentant cercar una classe anomenada `driver` tal com podeu observar a la figura 1.15.

Per indicar la classe a carregar, escriurem dins el mètode `provaDeConnexio`

```
1 Class.ForName("org.postgresql.Driver");
```

Cal escriure aquesta sentència sempre abans de començar a usar l'API JDBC.

El mètode `forName` de la classe `Classpot` llençar una excepció en cas que no es trobi la classe i convé capturar l'error per saber si s'ha escrit el nom correctament o que ens hem oblidat d'afegir el controlador al *classpath*.

```
1 public void primeraProva(){
2     try {
3         Class.forName("org.postgresql.Driver");
4 
5     } catch (ClassNotFoundException ex) {
6         System.out.println("No s'ha trobat el controlador JDBC");
7     }
8 }
```

1.3.2 Establint connexió

Cal recordar que l'API JDBC, a banda d'algunes classes específiques, majoritàriament està compost d'interfícies que el controlador implementa donant-los la funcionalitat adequada.

Per tal d'assegurar la interoperabilitat, les aplicacions no referenciaran mai les classes concretes de cap controlador sinó les interfícies estàndards de l'API JDBC.

Per aconseguir-ho, l'aplicació mai podrà instanciar directament els objectes JDBC amb una sentencia *new*, sinó que es crearan indirectament cridant algun mètode d'alguna classe o objecte ja existent que l'instanciï internament abans de retornar-lo a l'aplicació.

Així, per exemple,caldrà instanciar la interfície `Connection` a partir del mètode estàtic `genConnection` de la classe `DriverManager`, o bé a partir d'un `Objecte Driver`, el qual caldrà també instanciar a partir de la mateixa classe `DriverManager`, invocant el mètode `getDriver`.

`Connection` representa una connexió a la base de dades, una via de comunicació entre l'aplicació i l'SGBD. Els objectes `Connection` mantindran la capacitat de comunicar-se amb el sistema gestor mentre romanguin oberts. Això és, des que es creen fins que es tanquen fent servir el mètode `close`.

L'objecte `Connection` està totalment vinculat a una font de dades, per això en demanar la connexió cal especificar de quina font es tracta seguint el protocol JDBC i indicant la *url* de les dades, i si s'escau l'usuari i *password*.

La *url* seguirà el protocol JDBC, començarà sempre per la paraula *jdbc* seguida de dos punts. La resta dependrà del tipus de controlador utilitzat, del *host* on s'allotgi l'SGBD, del port que aquest faci servir per escoltar les peticions i del nom de la base de dades o esquema amb el qual volem treballar.

Per exemple, la url que usarem per connectar-nos a la nostra base de dades PostgreSQL anomenada ioc_proves, allotjada en el mateix ordinador des d'on estem treballant (*localhost*) i que escolta el port per defecte en què s'instal·la (*5432*), serà:

```
jdbc:postgresql://localhost:5432/ioc_proves
```

En canvi, si es tractés d'un controlador *Oracle* del tipus IV escoltant el port per defecte (1521), caldria escriure:

```
jdbc:oracle:thin:@localhost:1521:ioc_proves
```

Però si el controlador *Oracle* fos de tipus II:

```
jdbc:oracle:oci:@localhost:1521:ioc_proves
```

Finalment podem veure una *url* pel sistema gestor de base de dades de tipus Java DB. El port per defecte d'aquest gestor és *1527*:

```
jdbc:derby://localhost:1527/ioc_proves
```

Com es pot observar, és important documentar-se adequadament per saber quin serà el format *url* del controlador que haguem d'usar.

La forma més senzilla d'obtenir una connexió és usant el `DriverManager`:

```
String url="jdbc:postgresql://localhost:5432/ioc_proves";
String usuari="ioc";
String password="ioc";
Connection con = DriverManager.getConnection(url, usuari, password);
```

`DriverManager` és una classe de l'API estàndard JDBC molt especial, ja que té la missió d'intercedir entre l'aplicació i el controlador JDBC o controladors (si n'hi hagués més d'un). Cada cop que una classe de tipus `Driver` es carrega en memòria usant per exemple *Class.forName*, es dóna d'alta en el *DriverManager* on, per defecte, romandrà com a controlador actiu fins que finalitzi l'aplicació.

A partir de la *url* de connexió a una font de dades, la classe `DriverManager` és capaç de localitzar d'entre tots els *drivers* que tingui donats d'alta el controlador adequat, que permeti realitzar una connexió usant la *url* especificada. És a dir, gràcies al `DriverManager` no ens caldrà conèixer el nom de la classe principal del controlador (classe que ha d'implementar la interfície *java.sql.Driver* de l'API estàndard JDBC).

El mètode `getConnection` del `DriverManager`, a més de cercar i localitzar el *driver* corresponent, obtindrà una `Connection` del controlador seleccionat i la retornarà activa a l'aplicació.

També és possible obtenir una connexió des d'un objecte *driver*, invocant el mètode `connect`. Ara bé, per aconseguir l'objecte `Driver` adequat, caldrà demanar-lo a `DriverManager`.

```
String url="jdbc:postgresql://localhost:5432/ioc_proves";
 String usuari="ioc";
 String password="ioc";

 Driver driver = DriverManager.getDriver(url);

 Properties properties = new Properties();
 properties.setProperty("user", usuari);
 properties.setProperty("password", password);

 Connection con = driver.connect(url, properties);
```

Podeu observar que `DriverManager` selecciona el *driver* gràcies a la url que passem per paràmetre al mètode estàtic `getDriver`. A partir de l'objecte retornat, obtenim una connexió passant per paràmetre de nou la *url*. A més, en aquest cas s'indicarà l'usuari i la contrasenya passant una llista de propietats (classe `Properties`) amb els atributs *user* i *password* degudament assignats —properties.setProperty(`user`, `nomUsuari`) i properties.setProperty(`password`, `contrasenyaUsuari`)—.

Ambdós són mètodes que poden llançar excepcions i el compilador ens obligarà a protegir el codi incloent-lo dins una sentència `try-catch`.

Finalment cal indicar, abans de començar a treballar amb les sentències SQL, que és important tancar totes les connexions obertes abans d'abandonar l'aplicació, perquè si no es procedeix al tancament, el sistema gestor mantindrà en memòria la connexió malbaratant recursos. Els objectes `Connection` disposen dels mètodes `isClosed` i `close` per gestionar el tancament. Invocant el primer, aconseguirem saber si la connexió resta encara operativa o si ja ha estat tancada. El segon és el mètode que efectua el tancament.

```
public void provaDeConnexio(){
    Connection con=null;
    Driver driver=null;
```

```
4       String url="jdbc:postgresql://localhost:5432/ioc_proves";
5       String usuari="ioc";
6       String password="ioc";
7
8       System.out.println("provaDeConnexio()");
9
10      try {
11          //Carreguem el controlador en memòria
12          Class.forName("org.postgresql.Driver");
13      } catch (ClassNotFoundException ex) {
14          System.out.println("No s'ha trobat el controlador JDBC ("
15                                              + ex.getMessage() +")");
16          //Si no tenim controlador no podem fer res més. Sortim.
17          return;
18      }
19
20      try{
21           //Obtenim una connexió des de DriverManager
22          con = DriverManager.getConnection(url, usuari, password);
23          System.out.println("Connexió realitzada usant"
24                + " DriverManager");
25          con.close();
26
27      } catch (SQLException ex) {
28          System.out.println("Error " + ex.getMessage());
29      }
30
31      try{
32           //Obtenim el Driver del controlador des de DriverManager
33          driver = DriverManager.getDriver(url);
34           //configurem l'usuari i la contrasenya
35          Properties properties = new Properties();
36          properties.setProperty("user", usuari);
37          properties.setProperty("password", password);
38           //Obtenim una connexió des de la instància de Driver
39          con = driver.connect(url, properties);
40          System.out.println("Connexió realitzada usant Driver");
41          con.close();
42      } catch (SQLException ex) {
43          System.out.println("Error " + ex.getMessage());
44      }
45  }
```

1.3.3 Fent peticions SQL bàsiques

Per escriure sentències SQL, JDBC preveu els objectes `Statement`. Es tracta d'objectes instanciats per *Connection*, els quals poden enviar sentències SQL al sistema gestor connectat per tal que siguin executades, en invocar el mètode `executeQuery` o `executeUpdate`.

El mètode `executeQuery`, el veurem més endavant, però serveix per executar sentències de les quals s'espera que retornin dades, és a dir, consultes. En canvi, el mètode `executeUpdate` serveix específicament per a sentències que modifiquen la base de dades connectada però no els cal retornar cap mena de dada.

Sentències que no retornen dades

Malgrat que SQL és en essència un llenguatge de consulta, també inclou unes quantes ordres imperatives que permeten fer peticions per canviar les estructures internes de l'SGBD on s'emmagatzemaran les dades (instruccions conegudes amb les sigles DDL, de l'anglès *Data Definition Language*), atorgar permisos als usuaris existents o crear-ne de nous (subgrup d'instruccions conegudes com a DCL o *Data Control Language*) o modificar les dades emmagatzemades fent servir les instruccions *insert*, *update* i *delete*.

Malgrat que es tracta de sentències molt dispars, des del punt de vista de la comunicació amb l'SGBD es comporten de manera molt semblant, seguint el patró següent:

1. Instanciació a partir d'una connexió activa.
2. Execució d'una sentència SQL passada per paràmetre al mètode `executeUpdate`.
3. Tancament de l'objecte `Statement` instanciat.

Veiem un exemple d'obtenció i execució d'un `Statement` a partir d'un objecte connexió referenciat per la variable `con`.

```
1 ...
2
3     Statement statement = con.createStatement();
4     statement.executeUpdate(sentenciaSQL);
5     statement.close();
```

Assegurar l'alliberament de recursos

Les instàncies de `Connection` i les de `Statement` emmagatzemen, en memòria, molta informació relacionada amb les execucions realitzades. A més, mentre resten actives mantenen en l'SGBD un conjunt important de recursos oberts, destinats a servir de forma eficient les peticions dels clients. El tancament d'aquests objectes permet alliberar recursos tant del client com del servidor.

En realitat, els `Statements` són objectes vinculats íntimament a l'objecte `Connection`que els ha instanciat i si no es tanquen específicament, romandran actius mentre la connexió continuï activa, fins i tot més enllà d'haver desaparegut la variable que els referenciava.

És més, la complexitat d'aquests objectes fa que malgrat s'hagi tancat la connexió, els objectes `Statements` que no s'havien tancat expressament romanguin més temps en memòria que els objectes tancats prèviament, ja que el *garbage collector* de Java haurà de fer més comprovacions per assegurar que ja no disposa de dependències ni internes ni externes i es pot eliminar.

És per això que es recomana procedir sempre a tancar-lo manualment fent servir l'operació `close`. El tancament dels objectes `Statement` assegura l'alliberament immediat dels recursos i l'anul·lació de les dependències.

Si en un mateix mètode hem de tancar un objecte `Statement` i la connexió que l'ha instanciat, caldrà tancar en primer lloc l'`Statement` i després la instància `Connection`. Si ho féssim al revés, quan intentéssim tancar l'`Statement` ens saltaria una excepció de tipus SQLException, ja que el tancament de la connexió l'hauria deixat inaccessible.

Podeu consultar l'annex "Comportament dels flux d'execució durant el llançament d'excepcions" en la secció "Annexos" per veure com es comporta el flux d'execució durant el llançament d'excepcions.

A més de respectar l'orde, caldrà assegurar l'alliberament dels recursos situant les operacions de tancament dins un bloc *finally*. D'aquesta manera, malgrat que es produeixin errors, no es deixaran d'executar les instruccions de tancament.

Cal tenir en compte encara un detall més quan sigui necessari realitzar el tancament de diversos objectes a la vegada. En aquest cas, malgrat que les situéssim una darrera l'altra, totes les instruccions de tancament dins el bloc *finally*, no seria prou garantia per assegurar l'execució de tots els tancaments, ja que, si mentre es produeix el tancament d'un dels objectes es llança una excepció, els objectes invocats en una posició posterior a la del que s'ha produït l'error no es tancaran.

La solució d'aquest problema passa per evitar el llançament de qualsevol excepció durant el procés de tancament. Una possible forma és capsular cada tancament entre sentències *try-catch.*

```
try{
    //sentències que poden llançar una excepció
    ...
} catch (SQLException ex) {
    // captura i tractament de l'excepció
    ...
}finally{
    try {
         stm1.close();
    } catch (SQLException ex) {...}

    try {
         stm2.close();
    } catch (SQLException ex) {...}

    ...

    try {
         con.close();
    } catch (SQLException ex) {...}
}
```

De vegades, l'error en un tancament es produeix perquè l'objecte mai ha arribat a instanciar-se i, per tant, la variable presenta un valor *null*, o perquè ja ha estat tancat amb anterioritat. Ambdós casos són també previsibles fent servir una instrucció condicional que eviti la invocació.

```
...
try{
    // Assegurem que con està instanciada i oberta
    if(con!=null && !con.isClosed()){
  //tanquem la connexió
       con.close();
    }
} catch (SQLException ex) { ... }
```

1.3.4 Exemple senzill

Anem ara a posar en pràctica tot el que s'ha explicat creant una única taula per emmagatzemar recordatoris de tasques pendents. Volem que la taula tingui una clau primària numèrica, una descripció de la tasca pendent, la data d'inici en què està previst començar i la data en què està previst acabar. A més, disposarà d'un camp booleà per indicar si es dóna o no per finalitzada. A la taula 1.1 veureu l'esquema amb algunes dades d'exemple.

Taula 1.1. Taula de tasques pendents que volem crear en la nostra base de dades

id	descripció	data_inici	data_final	finalitzada
1	Comprar pomes	2012-02-15	2012-02-16	false
2	Estudiar examen	2012-05-02	2012-05-18	false
3	Compar bitllet a Sidney	2012-02-15	2012-05-20	false
4	Anar a Austràlia	2012-06-01	2012-08-25	false
5	Revisió del cotxe	2012-04-08	2012-04-16	false
6	Manifestació 1r maig	2012-05-01	2012-05-01	false
7	Pràctica JDBC	2012-04-15	2012-05-16	false
8	FCT	2012-03-01	2012-05-24	false

Creació de la taula

L'SQL que definirà la taula serà:

```
CREATE TABLE tasques(
   id INTEGER NOT NULL,
   descripcio VARCHAR(300) NOT NULL,
   data_inici DATE,
   data_final DATE NOT NULL,
   finalitzada BOOLEAN DEFAULT false,
   CONSTRAINT pk_clauPrimaria PRIMARY KEY (id)
);
```

A la classe `ProvesBasiques` que ja tenim creada hi afegirem un mètode que anomenarem `crearTaula` seguint el mateix patró que ja hem vist.

```
public void crearTaula(){
    Connection con=null;
    Statement statement = null;
    String sentenciaSQL=null;

    try {
        Class.forName("org.postgresql.Driver");

        String url = "jdbc:postgresql://localhost:5432/ioc_proves";
        con = DriverManager.getConnection(url, "ioc", "ioc");


        statement = con.createStatement();

```

```
        sentenciaSQL = "CREATE TABLE tasques(\n"
                + "id INTEGER NOT NULL, \n"
                + "descripcio VARCHAR(300) NOT NULL, \n"
                + "data_inici DATE, \n"
                + "data_final DATE NOT NULL, \n"
                + "finalitzada BOOLEAN DEFAULT false, \n"
                + "CONSTRAINT pk_clauPrimaria PRIMARY KEY (id)"
                + ")";

        statement.executeUpdate(sentenciaSQL);


    } catch (SQLException ex) {
            System.out.println("Error " + ex.getMessage());
    } catch (ClassNotFoundException ex) {
        System.out.println("No s'ha trobat el controlador JDBC ("
                                            + ex.getMessage() +")");
    }finally{
        try {
            if(statement!=null && !statement.isClosed()){
                    statement.close();
            }
        } catch (SQLException ex) {/*llàstima!*/}
        try {
            if(con!=null && !con.isClosed()){
                    con.close();
            }
        } catch (SQLException ex) {/*llàstima!*/}
    }
}
```

Inserció de dades

També volem introduir-hi les dades que es poden veure a la taula anterior. Crearem un `Statetement`que reutilitzarem per anar escrivint totes les sentències `INSERT`.

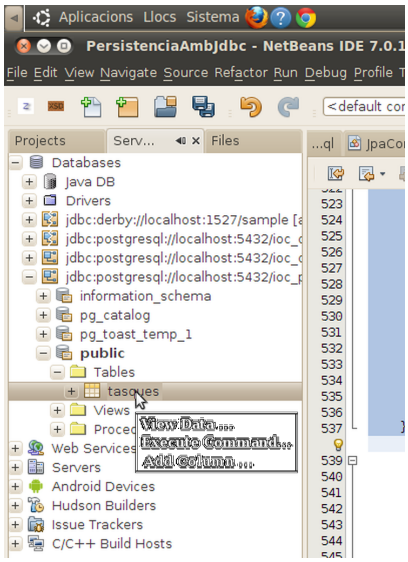
```
public void insereixTasques(){
    Connection con=null;
    Statement statement = null;
    String sentenciaSQL=null;

    try {
        Class.forName("org.postgresql.Driver");

        String url = "jdbc:postgresql://localhost:5432/ioc_proves";
        con = DriverManager.getConnection(url, "ioc", "ioc");


        statement = con.createStatement();

        sentenciaSQL = "INSERT INTO TASQUES VALUES (1, "
                  + "'Comprar pomes', '2012-02-15', '2012-02-16')";
        statement.executeUpdate(sentenciaSQL);
        sentenciaSQL = "INSERT INTO TASQUES VALUES (2, "
                 + "'Estudiar examen', '2012-05-2', '2012-05-18')";
        statement.executeUpdate(sentenciaSQL);
        sentenciaSQL = "INSERT INTO TASQUES VALUES (3, "
          + "'Compar bitllet Sidney', '2012-02-15', '2012-05-20')";
        statement.executeUpdate(sentenciaSQL);
        sentenciaSQL = "INSERT INTO TASQUES VALUES (4, "
               + "'Anar a Australia', '2012-06-01', '2012-08-25')";
        statement.executeUpdate(sentenciaSQL);
        sentenciaSQL = "INSERT INTO TASQUES VALUES (5, "
                   + "'Revisió cotxe', '2012-04-8', '2012-04-16')";
        statement.executeUpdate(sentenciaSQL);
        sentenciaSQL = "INSERT INTO TASQUES VALUES (6, "
```

```
            + "'Manifestació 1r maig', '2012-05-1', '2012-05-1')";
        statement.executeUpdate(sentenciaSQL);
        sentenciaSQL = "INSERT INTO TASQUES VALUES (7, "
                  + "'Pràctica JDBC', '2012-04-15', '2012-05-16')";
        statement.executeUpdate(sentenciaSQL);
        sentenciaSQL = "INSERT INTO TASQUES VALUES (8, "
                       + "'FCT', '2012-03-1', '2012-05-24')";
        statement.executeUpdate(sentenciaSQL);

    } catch (SQLException ex) {
        System.out.println("Error " + ex.getMessage());
    } catch (ClassNotFoundException ex) {
        System.out.println("No s'ha trobat el controlador JDBC ("
                                          + ex.getMessage() +")");
    }finally{
        try {
            if(statement!=null && !statement.isClosed()){
                    statement.close();
            }
        } catch (SQLException ex) {/*llàstima!*/}
        try {
            if(con!=null && !con.isClosed()){
                    con.close();
            }
        } catch (SQLException ex) {/*llàstima!*/}
    }
}
```

Podeu visualitzar el contingut que tindrà la taula després d'executar els mètodes implementats usant la connexió de NetBeans. Aneu a la pestanya *Services*, escolliu la connexió que ja vàreu realitzar o be creeu-ne una de nova per connectar-vos a la base de dades ioc_proves.

Sobre la connexió a la base de dades ioc_proves, cliqueu el botó dret del ratolí i escolliu *Connect...* Amb al connexió feta, heu de poder veure el contingut. Les taules s'emmagatzemen al *schema* public. Obriu-lo amb un doble clic i obriu la carpeta *Tables* també amb un doble clic. Situeu el cursor damunt la taula *Tasques* i amb el botó dret seleccioneu *View Data...*, tal com podeu veure a la figura 1.16.

FIGURA 1.16. Menú de context per visualitzar les dades de la taula



Modificació de dades

Seguidament, modificarem les dades de la taula. Donarem per finalitzades les tasques identificades amb 1, 3 i 5. A més, a la tasca número 7 li ampliarem la data de finalització.

De moment seguirem el mateix patró, tal com hem fet fins ara, però usarem la sentència `UPDATE`.

Crearem el mètode `modificarTasques`:

```
1  public void modificarTasques(){
2      Connection con=null;
3      Statement statement = null;
4      String sentenciaSQL=null;
5
6      try {
7          Class.forName("org.postgresql.Driver");
8
9          String url = "jdbc:postgresql://localhost:5432/ioc_proves";
10         con = DriverManager.getConnection(url, "ioc", "ioc");
11
12
13         statement = con.createStatement();
14
15         sentenciaSQL = "UPDATE TASQUES SET FINALITZADA=true "
16                        + "WHERE ID=1 OR ID=3 OR id=5";
17         statement.executeUpdate(sentenciaSQL);
18         sentenciaSQL = "UPDATE TASQUES SET data_final='2012-06-5' "
19                        + "where id=7";
20         statement.executeUpdate(sentenciaSQL);
21     } catch (SQLException ex) {
22         System.out.println("Error " + ex.getMessage());
23     } catch (ClassNotFoundException ex) {
24         System.out.println("No s'ha trobat el controlador JDBC ("
25                                             + ex.getMessage() +")");
26     }finally{
27         try {
28             if(statement!=null && !statement.isClosed()){
29                     statement.close();
30             }
31         } catch (SQLException ex) {/*llàstima!*/}
32         try {
33             if(con!=null && !con.isClosed()){
34                     con.close();
35             }
36         } catch (SQLException ex) {/*llàstima!*/}
37     }
38 }
```

Consultar les dades de la taula

Les consultes són les instruccions SQL que permeten obtenir, de forma total o parcial, les dades emmagatzemades a la base de dades. En el cas que ens ocupa, recuperem les dades de l'única taula de què disposem.

Amb els objectes `Statements` podem gestionar també consultes SQL, però cal invocar el mètode `execteQuery`, perquè aquest mètode retorna el conjunt de dades corresponents a la consulta realitzada. Les dades es retornen fent servir un objecte `ResultSet`.Per tant, l'execució de les consultes tindrà un forma semblant a la següent:

```
ResultSet rs = statement.executeQuery(sentenciaSQL);
```

L'objecte `ResultSet` conté el resultat de la consulta organitzat per files. Es poden visitar totes les files d'una a una cridant el mètode `next`, ja que a cada invocació de `next` s'avançarà a la següent fila. Immediatament després d'una execució, el `ResultSet` retornat es troba posicionat just abans de la primera fila, per tant per accedir a la primera fila caldrà invocar `next` una vegada. Quan les files s'acabin el mètode `next` retornarà fals.

Des de cada fila es podrà accedir al valor de les seves columnes fent servir la diversitat de mètodes disponibles segons el tipus de dades a retornar i passant per paràmetre el nombre de columna que desitgem obtenir. El nom dels mètodes segueix un patró força senzill: usarem *get* com a prefix i el nom del tipus com a sufix. Així, si volguéssim recuperar la segona columna, sabent que és una dada de tipus String caldria invocar:

```
rs.getString(2);
```

S'ha de tenir en compte que les columnes es comencen a comptar a partir del valor 1 (no pas del zero). La majoria d'SGBD suporten la possibilitat de passar per paràmetre el nom de la columna, però en no poder garantir un funcionament correcte en qualsevol sistema fa que normalment s'opti sempre pel paràmetre numèric.

Vegem de quina manera podem mostrar per pantalla totes les tasques que ja estiguin acabades:

```
public void consultarTasques(){
    Connection con=null;
    Statement statement = null;
    String sentenciaSQL=null;
    ResultSet rs=null;

    try {
        Class.forName("org.postgresql.Driver");

        String url = "jdbc:postgresql://localhost:5432/ioc_proves";
        con = DriverManager.getConnection(url, "ioc", "ioc");


        statement = con.createStatement();

        sentenciaSQL = "SELECT id, descripcio, data_inici,"
                        + " data_final, finalitzada"
                             + " FROM tasques WHERE finalitzada";
        rs = statement.executeQuery(sentenciaSQL);

        //mostrar capçalera per la pantalla
        System.out.print("        id");
        System.out.print("         descripcio       ");
        System.out.print(" data_inici ");
        System.out.print(" data_final ");
        System.out.println("finalitzada");
        System.out.print("————————————");
        System.out.print("————————————————————————————");
        System.out.print("————————————————");
        System.out.print("————————————————");
        System.out.print("————————————");
        System.out.println();
```

```
33          //mostrar files de dades
34          while(rs.next()){
35              System.out.printf("%10d", rs.getInt(1));
36              String desc = rs.getString(2);
37              //fem que la mida de desc sigui sempre 25 caràcters
38              //així no es desquadra la taula.
39              if(desc.length()<=25){
40                 System.out.printf(" %s%s", desc,
41                  "                         ".substring(0,
42                                               25-desc.length()));
43              }else{
44                 System.out.printf(" %25s",
45                  rs.getString(2).substring(0, 25));
46              }
47              //usem Dateformat per formatar les dates
48              //en el nostre cas DD-MM-AAAA
49              DateFormat df = DateFormat.getDateInstance();
50              System.out.print(" " + df.format(rs.getDate(3)) + " ");
51              System.out.print(" " + df.format(rs.getDate(4)) + " ");
52              System.out.println("     " + rs.getBoolean(5));
53          }
54      } catch (SQLException ex) {
55          System.out.println("Error " + ex.getMessage());
56      } catch (ClassNotFoundException ex) {
57          System.out.println("No s'ha trobat el controlador JDBC ("
58                                             + ex.getMessage() +")");
59      }finally{
60          try {
61              if(rs!=null && !rs.isClosed()){
62                      rs.close();
63              }
64          } catch (SQLException ex) {/*llàstima!*/}
65          try {
66              if(statement!=null && !statement.isClosed()){
67                      statement.close();
68              }
69          } catch (SQLException ex) {/*llàstima!*/}
70          try {
71              if(con!=null && !con.isClosed()){
72                      con.close();
73              }
74          } catch (SQLException ex) {/*llàstima!*/}
75      }
76  }
```

Podeu observar com es pot usar un bucle `while` per obtenir el valor de totes les files retornades. També podeu veure els diferents mètodes que retornen les dades de cada columna en funció del tipus. Destacarem que en cas de dates, `ResultSet` retorna un tipus especial d'objecte hereu de la classe `Date`estàndard (la que està situada al paquet *java.util*). La classe en qüestió s'anomena també `Date`, però es tracta d'una classe diferent situada al paquet *java.sql*.

En el codi s'ha fet servir `DateFormat` per poder mostrar les dates en el nostre format i s'ha assegurat una mida constant del camp *descripció* de 25 caràcters.

Finalment, cal indicar que els objectes `ResultSet` també s'han de tancar de la mateixa manera com procedim a tancar els `Statements` o les connexions. Cal tenir en compte, però, que els `ResultSet`són els primers que caldrà tancar.

1.4 JDBC Avançat

JDBC disposa d'una altra funcionalitat i estructures que, escollides adequadament, poden ajudar-nos a incrementar la qualitat de les aplicacions que construïm.

Volem construir aplicacions flexibles, robustes i eficients. Necessitarem, doncs, un bon tractament d'errors que traslladi quan faci falta la informació adequada a l'usuari o reconduint el flux de l'execució cap a processos que interpretin i compensin les errades.

L'eficiència és també una característica important de la qualitat. En general, els sistemes gestors disposen de mecanismes automàtics per potenciar l'eficiència de les peticions, com ara l'ús d'emmagatzemament cau d'accés ràpid, la creació d'índexs automàtics, etc. Aquests automatismes responen a determinats patrons a l'hora de fer les peticions. Per això JDBC preveu altres formes, diferents a les estudiades fins ara, per realitzar peticions que millorin el rendiment.

1.4.1 Tractament d'errors en aplicacions JDBC

L'execució de sentències SQL està sotmesa a multitud de factors que poden provocar algun error. Pot passar que la connexió falli, que el controlador no sigui l'adequat, que les sentències tinguin errades, que l'SGBD no suporti la sentència, i un llarg etcètera de possibilitats.

Podeu trobar informació referida als codis de SQLSTATE a la pàgina goo.gl/tkz9w. El codi SQLSTATE està format percinc caràcters. Els dos primers indiquen la tipologia de l'error i elstres darrers el concreten.

Els errors SQL es troben força ben definits a l'especificació estàndard, la qual descriu el valor de la variable anomenada `SQLSATE`, que identifica l'estat d'una sentència SQL immediatament després de la seva execució. Quan JDBC detecta que després d'una execució el valor d'aquesta variable es correspon a un error, dispara una excepció de tipus SQLException la qual, a més de contenir un missatge clarificador, incorpora el valor delSQLSATE. Podem recuperar aquest valor invocant el mètode `getSQLState`.

L'ús de *try-catch* ens permetrà capturar específicament excepcions SQLException o derivades. Un cop capturades, usarem el codi SQLSTATE per decidir com cal actuar.

Imaginem, per exemple, que en intentar connectar amb un SGBD capturem una excepció SQL amb el valor SQLState igual a *28000*. Si consulteu aquest codi a la pàgina que us indiquem al marge dret veureu que el valor *28000* correspon a un error en l'autenticació. En canvi, si el codi rebut hagués estat *08001* significaria que JDBC està trobant problemes de xarxa a l'hora de connectar, ja siguin deguts a una desconnexió física, o simplement a un *host* o adreça IP desconegut.

No cal informar detalladament l'usuari de tots i cada un dels possibles errors, però sí que cal decidir quins errors requeriran un tractament específic i quins no. Segurament no seria mala idea, si detectem un SQLState de valor *08001*,

aconsellar l'usuari que abans de trucar al servei tècnic revisi les connexions de xarxa o s'asseguri que el servei de l'SGBD es troba aixecat i actiu.

D'altra banda, la detecció acurada de l'SQLState ens pot també permetre realitzar accions per reconduir l'error. Imaginem, per exemple, que per raons de seguretat l'administrador de l'SGBD va canviant de contrasenya. L'administrador escull a l'atzar una contrasenya d'entre un conjunt de tres o quatre prefixades. Per tal de no haver d'estar contínuament configurant la nostra aplicació cada vegada que canviï la contrasenya, podem implementar una utilitat que accepti un conjunt de tres o quatre contrasenyes de manera que pugui anar provant d'una en una quan rebi un error d'autenticació.

Anem a veure encara un tercer cas en què ens convindrà fer també un tractament especial de l'error. Ja s'ha vist que quan realitzem el tancament d'objectes de tipus `ResultSet`, `Statement` o `Connection` hem optat per silenciar l'error capturant totes les excepcions que han sigut llançades durant el procés per tal d'assegurar el tancament posterior de la resta d'objectes.

```
try{

    ...

}finally{
    try {
        if(statement!=null && !statement.isClosed()){
                statement.close();
        }
    } catch (SQLException ex) {/*silenci!*/}
    try {
        if(con!=null && !con.isClosed()){
                con.close();
        }
    } catch (SQLException ex) {/*silenci!*/}
}
```

El principal problema d'aquesta tècnica és que si en algun moment acabés produint-se un error durant un tancament, passaria totalment desapercebut. Com que probablement els errors que poguessin succeir aquí serien amb tota seguretat esporàdics, en moltes aplicacions se sol tolerar aquesta incorrecció.

Podeu consultar l'annex Configuració i ús d'enregistradors en JAVA on podreu saber més sobre el sistema enregistrador de successos incorporat al JDK.

Podem fer servir enregistradors per deixar constància dels errors silenciats. Els enregistradors (*loggers*) treballen contra un fitxer que actua com a magatzem de successos.

Vegem un possible exemple on posem en pràctica totes les consideracions que acabem de comentar:

```
public void tractamentErrorConnexio(){
    boolean connectat=false;
    Connection con=null;
    System.out.println("tractamentErrorConnexio()");

    try {
        Class.forName("org.postgresql.Driver");

        String url = "jdbc:postgresql://localhost:5432/ioc_proves";

        for(int i=0; !connectat && i< contrasenyes.length; i++){
           try{
              con = DriverManager.getConnection(url, usuari,
```

```
                                                    contrasenyes[i]);
            connectat=true;
          }catch(SQLException ex){
            if(!ex.getSQLState().equals("28000")){
                //NO és un error d'autenticació
                throw ex;
            }
          }
        }
    } catch (SQLException ex) {
        if(ex.getSQLState().equals("08001")){
            System.out.println("S'ha detectat un problema de"
                  + " connexió. Reviseu els cables de xarxa"
                  + " i assegureu-vos que l'SGBD està"
                  + " operatiu.\n Si continua sense"
                  + " connectar, aviseu el servei tècnic");

        }else{
            System.out.println("S'ha produït un error inesperat."
                    + " Truqueu al servei tècnic indicant el següent codi "
                    + "d'error SQL:" + ex.getSQLState());
        }
    } catch (ClassNotFoundException ex) {
        System.out.println("No s'ha trobat el controlador JDBC ("
                + ex.getMessage() +"). Truqueu al servei tècnic");
    }finally{
        try {
            if(con!=null && !con.isClosed()){
                    con.close();
            }
        } catch (SQLException ex) {
            Logger.getLogger(ProvesBasiques.class.getName()).log(
                                        Level.SEVERE, null, ex);
        }
    }
}
```

Transaccions

Recordeu que en el llenguatge SQL les transaccions permeten gestionar les operacions realitzades en una base de dades. Les transaccions es consideren unitàries. És a dir, les operacions que componen la transacció s'han d'executar totes o cap. Això ajuda a preservarla integritat de les dades i impedeix possibles desfasaments entre clients i servidor.

D'entrada, qualsevol sentència SQL es considera una transacció en si mateixa i si es produeix un error durant la seva execució s'anul·laran totes les operacions simples derivades de l'execució de la sentència.

Imaginem que desitgem eliminar totes les tasques (de la taula TASQUES de la base de dades ioc_proves) finalitzades durant l'any 2011 o anteriors. Per fer-ho, caldria enviar la instrucció SQL següent:

```
DELETE FROM TASQUES
WHERE finalitzada AND data_final < '2012-01-01';
```

Si durant el procés d'eliminació de cada un dels registres es produís un error, es tornarien a reposar totes les tasques prèviament eliminades, ja que les instruccions SQL són per definició unitàries (transaccions) i o bé s'executen senceres o bé no s'executen.

De vegades, però, hi ha dades amb un alt grau de dependència que cal manipular de forma independent perquè es troben en diferents taules, per exemple. El grau de dependència, però, aconsella tractar-les com un tot, ja que contràriament podria provocar pèrdua de dades o fins i tot inconsistència.

Per preservar aquests problemes SQL usa transaccions explícites en combinació amb les instruccions de control `commit` i `rollback`. Les transaccions explícites permeten agrupar un conjunt de sentències SQL tractant-les com una única operació unitària. La sentència `COMMIT` marcarà el final de la transacció i donarà per definitives totes les operacions realitzades. La sentència `rollback`, per contra, revocarà totes les operacions realitzades i deixarà de nou la base de dades en les mateixes condicions com estava abans d'iniciar la transacció.

JDBC trasllada també aquest metodologia al seu API. Per defecte, les connexions JDBC consideren que cada objecte `Statement`és en si mateix una transacció. Abans de cada execució es demana l'inici d'una transacció i al final, si l'execució té èxit, s'envia un `commit` i si no té èxit, un `rollback`. Per això diem que la connexió actua en mode `autocommit`.

Cada execució invocada des de `Statement` pot estar formada per multitud de sentències SQL separades per punt i coma. Malgrat que aquesta seria una possibilitat d'assegurar la revocació d'una acció composta per diverses sentències múltiples, no acostuma a fer-se servir ja que el tractament de l'error no seria gaire detallat.

Hi ha un altre mètode. Els `Statements`poden treballar sense automatitzar el `commit` després de cada execució. Caldrà canviar la connexió de mode. Per canviar-la, invocarem el mètode `setAutoCommit` passant-li per paràmetre el valor `false`.

A partir d'aleshores es consideraran instruccions d'una mateixa transacció totes les sentències executades entre dues invocacions del mètode `commit` (equivalent JDBC a la instrucció `commit` de l'SQL).

El mètode `rollback` actuarà igual que la instrucció SQL equivalent, revocant totes les sentències executades a partir de l'últim `commit`.

A continuació mostrem una implementació genèrica que espera per paràmetre un *array* de sentències SQL pertanyents a una mateixa transacció. La funció implementada executa sentència per sentència havent prèviament desactivat el mode `autocommit`. Si totes les sentències s'executen amb èxit s'invocarà el mètode `commit`; en cas contrari s'invocarà el mètode `rollback`. A la implementació hem suposat que la connexió ja existeix i es troba activa.

```
1  public void executa(String[] sentenciesSql) throws SQLException{
2      boolean autocommit=true;
3      Statement stm = null;
4      try {
5          autocommit = con.getAutoCommit();
6          con.setAutoCommit(false);
7          stm = con.createStatement();
8
9          for(String sent: sentenciesSql){
10             stm.executeUpdate(sent);
11         }
12         con.commit();
```

```
13          con.setAutoCommit(autocommit);
14      } catch (SQLException ex) {
15          con.rollback();
16          throw ex;
17      }finally{
18          try {
19              if(stm!=null && !stm.isClosed()){
20                      stm.close();
21              }
22          } catch (SQLException ex) {
23              Logger.getLogger(ProvesBasiques.class.getName()).log(
24                                                      Level.SEVERE, null, ex);
25          }
26      }
27  }
```

En tractar-se d'una connexió ja existent que s'utilitza en diversos mètodes, desactivarem sempre el mode `autocommit` a l'inici i reposarem de nou el mode que tingués la connexió just abans de sortir, fent servir una variable testimoni.

La majoria d'SGBD permeten usar transaccions explícites amb qualsevol instrucció SQL, fins i tot en sentències DDL (*data definition language*) usades per crear el sistema de taules i objectes de la base de dades de l'aplicació. Les sentències de definició modifiquen directament l'estructura de les dades i, per tant, cal anar molt en compte perquè poden provocar danys importants, pèrdues de dades existents, etc.

En el cas de sentències DDL, és important treballar amb esquemes o bases de dades específiques de cada aplicació. És una manera d'evitar que els possibles errors repercuteixin sobre altres aplicacions i dades alienes. A més, es faran servir transaccions durant els processos de creació i modificació per tal de no arrossegar problemes derivats d'errors produïts durant la seva execució.

Cal tenir en compte que hi ha alguns sistemes gestors com Oracle que no suporten la revocació de sentències DDL i en cas d'invocar `rollback`,obtindrem un error indicant que les sentències DDL no es poden revocar. Per preveure casos com aquest caldrà capturar l'error i avisar l'usuari que no s'ha pogut restaurar la base de dades i que caldria revisar-la manualment.

1.4.2 Millora del rendiment

Un altre aspecte important que mesura la qualitat de les aplicacions és l'eficiència amb la qual s'aconsegueix comunicar amb l'SGBD. Per optimitzar la connexió és important reconèixer quins processos poden actuar de coll d'ampolla i sota quines circumstàncies o quines altres agilitzen les respostes dels SGBD.

En primer lloc, analitzarem la petició de connexió a un SGBD perquè es tracta d'un procés costós però inevitable que cal considerar.

En segon lloc, estudiarem les sentències predefinides, perquè el seu ús facilita la creació de *dades clau* i índexs temporals de manera que sigui possible anticipar-se a la demanda o disposar de les dades de forma molt més ràpida.

Cicle de vida d'una connexió

L'establiment d'una connexió és un procediment força lent, tant a la part client com a la part servidor. A la part client, `DriverManager` ha de descobrir el controlador correcte d'entre tots els que hagi de gestionar. La majoria de vegades les aplicacions treballaran només amb un únic controlador, però cal tenir en compte que `DriverManager` no coneix a priori quina URL de connexió correspon a cada controlador, i per esbrinar-ho envia una petició de connexió a cada controlador que tingui registrat, el controlador que no li retorna error serà el correcte.

A la banda servidor, es crearà un context específic i s'habilitaran un conjunt de recursos per cada client connectat. És a dir, que durant la petició de connexió l'SGBD ha de gastar un temps considerable abans de no deixar operativa la comunicació client-servidor.

Aquesta elevada despesa de temps concentrada en el moment de la petició de connexió ens fa plantejar si podem considerar ineficient obrir i tancar la connexió cada cop que haguem d'executar una sentència SQL, com hem anat fent fins ara. Malauradament no hi ha una única resposta, sinó que depèn de la freqüència d'ús de la connexió i el nombre de connexions contra un mateix SGBD coexistint al mateix temps.

Com en tot, es tracta de trobar el punt d'equilibri entre la quantitat de recursos esmerçats per connexió i la rendibilitat que se'n treu en mantenir-les obertes. Si el nombre de clients, i per tant de connexions, és baix i la freqüència d'ús és alta, serà preferible mantenir les connexions obertes molt de temps. Per contra, si el nombre de connexions és molt alt i l'ús infreqüent, el que serà preferible serà obrir i tancar la connexió cada cop que es necessiti. Mentrestant, hi haurà una multitud de casos en què la solució consistirà a mantenir les connexions obertes però no permanentment. Es pot donar un temps de vida a cada connexió, o bé tancar-les després de restar inactiva una quantitat determinada de temps, o es pot fer servir el criteri de mantenir un nombre màxim de connexions obertes, tancant les més antigues o les més inactives quan se sobrepassi el límit.

D'altra banda, cal tenir en compte també que una mateixa aplicació pot treballar amb diverses connexions simultàniament per tal d'incrementar l'eficiència. Cada connexió obre un fil d'execució independent, de manera que és possible l'enviament simultani de peticions.

Sentències predefinides

JDBC disposa d'un objecte derivat de l'`Statement` que s'anomena `PreparedStatement`. Aquest es diferencia del primer en què les instàncies es generen amb un patró de sentència SQL definit. El patró pot ser una sentència específica semblant a la que passem als `Statements`en el moment de l'execució o bé una sentència que admeti paràmetres en alguna de les seves parts.

Sigui com sigui, `PreparedStatement`presenta avantatges sobre el seu antecessor `Statement` quan haguem de treballar amb sentències que calgui executar diverses

vegades. La raó és que qualsevol sentència SQL, quan s'envia l'SGBD serà compilada abans de ser executada.

Usant un objecte `Statement`, cada cop que fem una execució d'una sentència, ja sigui via `executeUpdate` o bé via `executeQuery`, l'SGBD la compilarà, ja que li arribarà en forma de cadena de caràcters.

En canvi,al `PreparedStament` la sentència mai varia i per tant es pot compilar i emmagatzemar dins el mateix objecte, de manera que les següents vegades que s'executi no caldrà compilar-la. Això reduirà sensiblement el temps d'execució. La parametrització, a més, ajuda a crear sentències molt genèriques que es puguin reutilitzar fàcilment.

En alguns sistemes gestors, a més, fer servir `PreparedStatements`pot arribar a suposar més avantatges, ja que utilitzen la seqüència de bytes de la sentència per detectar si es tracta d'una sentència nova o ja s'ha servit amb anterioritat. D'aquesta manera es propicia que el sistema emmagatzemi les respostes en la memòria cau, de manera que es puguin lliurar de forma més ràpida.

Instanciació i ús d'objectes "PreparedStament"

La principal diferència dels objectes `PreparedStatement` en relació als ja vistos `Statement`, és que els primers s'instancien indicant la sentència SQL predefinida com a paràmetre del constructor. Com que la sentència queda predefinida, ni els mètodes `executeUpdate` ni `executeQuery` requeriran cap paràmetre.

```
try{
    sentenciaSQL = "INSERT INTO PAIS VALUES('Marroc')";
    PreparedStatement stm = con.prepareStatement(sentenciaSql);
    stm.executeUpdate();
}finally{
    try {
        if(stm!=null && !stm.isClosed()){
                stm.close();
        }
    } catch (SQLException ex) {
        Logger.getLogger(ProvesBasiques.class.getName()).log(
                                        Level.SEVERE, null, ex);
    }
}
```

Els paràmetres de la sentència es marcaran amb el símbol d'interrogant (?) i s'identificaran per la posició que ocupin a la sentència, començant a comptar des de l'esquerra i a partir del número 1. El valor dels paràmetres s'assignarà fent servir el mètode específic, d'acord amb el tipus de dades a assignar, anomenat amb el propi nom del tipus antecedit pel prefix *set* (exemples: `setString`, `setInt`, `setLong`, `setBoolean`...). Tots aquest mètodes segueixen la mateixa sintaxi:

```
setXXXX(<posicioALaSentenciaSQL>, <valor>);
```

Veiem un exemple:

```
try {
        autocommit=con.getAutoCommit();
        PreparedStatement stm = con.prepareStatement(
              "INSERT INTO ARTICLE (id, descripcio) VALUES(?, ?)");
```

```
5           stm.setLong(1, article.getId());
6           stm.setString(2, article.getDescripcio());
7           stm.executeUpdate();
8       }finally{
9           try {
10              if(stm!=null && !stm.isClosed()){
11                  stm.close();
12              }
13          } catch (SQLException ex) {
14              Logger.getLogger(ProvesBasiques.class.getName()).log(
15                                                  Level.SEVERE, null, ex);
16          }
17      }
18  }
```

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 potents són 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.

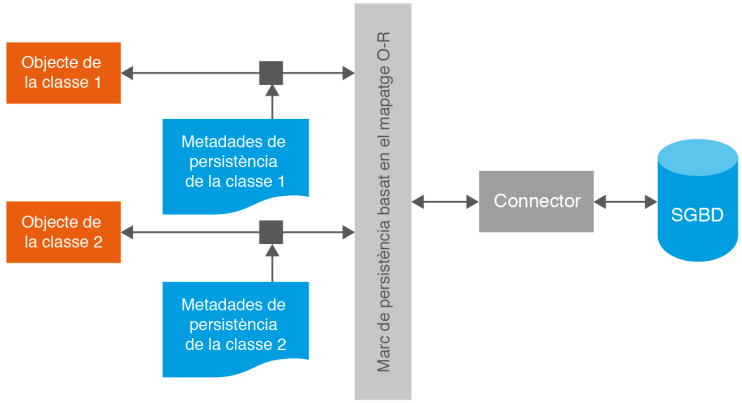
Les eines de mapatge objecte-relacional (ORM) intenten aprofitar la maduresa i l'eficiència de les bases de dades relacionals minimitzant tant com sigui possible el desfasament objecte-relacional. Es tracta de biblioteques i marcs de programació que defineixen un format per expressar múltiples situacions 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++, C#, Visual Basic, 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 de l'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.

Dues eines força utilitzades són Hibernate i EclipseLink. Aquí farem servir EclipseLink, ja que es pot incorporar fàcilment en un projecte de NetBeans.

2.3.1 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`.

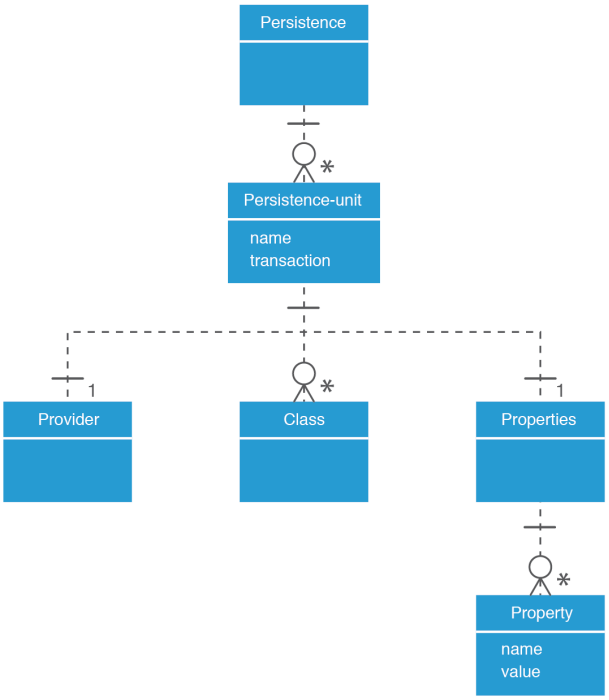
El primer cop que s'instanciï un `EntityManger` es connectarà a l'SGBD i comprovarà si existeixen totes les taules necessàries per mantenir la persistència de les entitats que aquest `EntityManger` controli. En cas que falti alguna, es generaran les sentències de creació adequades d'acord amb les metadades 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 2.2). 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 2.2. 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'EclipseLink, la classe és `org.eclipse.persistence.jpa.PersistenceProvider`. 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 `Properties` podem 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:

```
1 <persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"
2       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3   xsi:schemaLocation=
```

```
  "http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
  <persistence_unit name="UnitatDePersistenciaPersistenciaEmpreses"
  transaction_type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</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.eclipse.persistence.jpa.PersistenceProvider</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 Ús de JPA amb EclipseLink sobre NetBeans i PostgreSQL

Per a crear un projecte de NetBeans que treballi sobre una base de dades de PostgreSQL realitzant mapatge objecte-relacional amb EclipseLink, es poden seguir els següents passos:

- Crear a PostgreSQL la base de dades que utilitzarem. No cal crear-hi cap taula.
- Crear a NetBeans un projecte del tipus *Java Application*.
- Afegir al projecte la biblioteca que té el *driver JDBC* de PostgreSQL. Es fa fent clic amb el botó secundari a la carpeta de biblioteques del projecte,

seleccionant *Add* / *Library*, seleccionant la biblioteca *PostgreSQL JDBC Driver* i fent clic al botó *Add Library*.

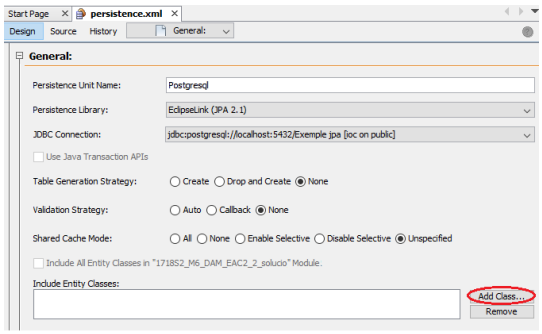
En aquest apartat s'indica com fer-ho.

- Afegir al projecte una unitat de persistència. Un cop afegida, s'incorporen al projecte automàticament les biblioteques pròpies de la implementació JPA seleccionada en afegir-la (EclipseLink al nostre cas).

Consulteu els punts "Metadades per definir la persistència" i "Funcionalitat del gestor d'entitats"

- Completar el projecte amb:
  - Les anotacions de JPA i/o l'especificació XML necessàries per assenyalar les classes persistents, les restriccions i les característiques del mapatge objecte-relacional que sigui necessari.
  - Les crides als mètodes de l'API necessàries per gestionar la persistència dels objectes de l'aplicació.
- Tornar a l'edició de la unitat de persistència (pestanya *Design*) i afegir al quadre *Include Entity Classes* (és al final de la finestra) les classes que volem fer persistents amb el botó *Add Class...*, que apareix encerclat a la figura 2.3. En cas d'afegir una classe no desitjada, pot eliminar-se amb el botó *Remove*, que és a sota del botó anterior.

FIGURA 2.3. Adició de classes a la unitat de persistència



Per a afegir aquesta unitat de persistència, cal seguir els següents passos:

- Fer clic al botó secundari del ratolí a sobre del projecte i seleccionar les opcions *New* / *Persistence Unit...* del menú contextual, tal i com es veu a la figura 2.4. En cas que no aparegui la subopció *Persistence Unit...*, cal triar al seu lloc l'opció *Other...* i, a la finestra que es mostra, seleccionar el tipus de fitxer *Persistence Unit* dins de la categoria / *Persistence*, com es veu a figura 2.5 i figura 2.6.

FIGURA 2.4. Opció New / Persistence Unit...

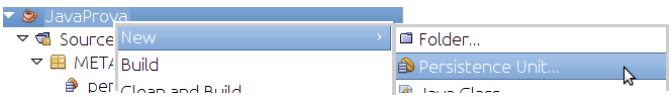


FIGURA 2.5. Opció New / Other...

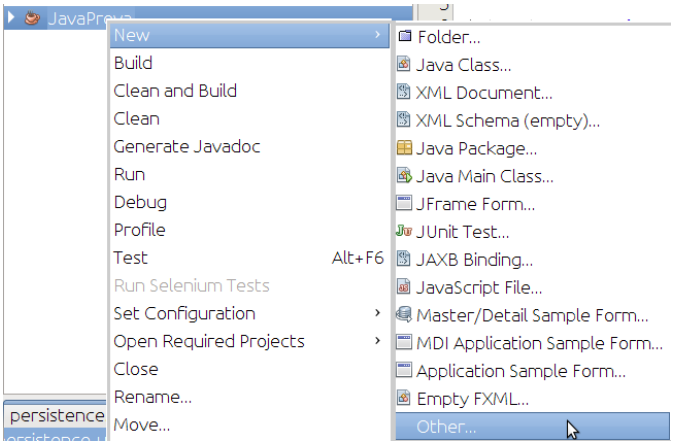
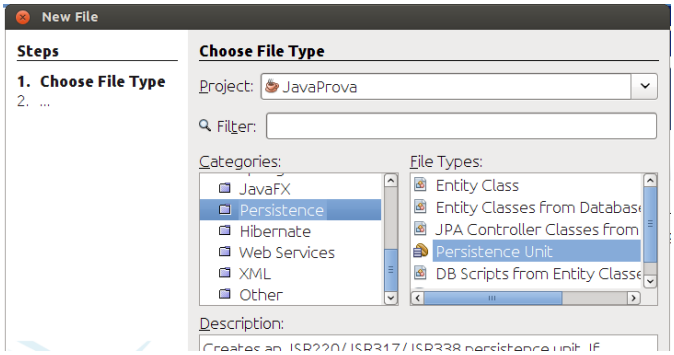
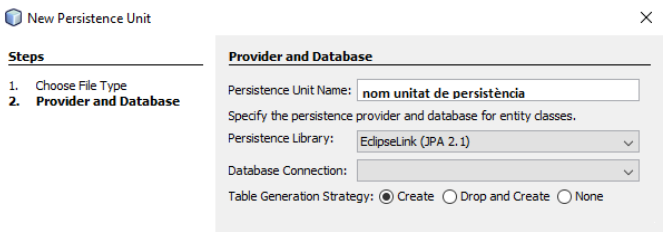


FIGURA 2.6. Selecció del tipus de fitxer Persistence Unit.



- Configurar la unitat de persistència a la finestra que apareix. A la figura 2.7, es veu la configuració de les següents dades: el **nom** de la unitat de persistència (pot ser qualsevol), la **biblioteca de persistència**, que serà *EclipseLink (JPA 2.1.)* i l'**estratègia de generació de taules**; en aquest cas s'ha triat l'estratègia *Create* per aconseguir que, si no existeixen les taules necessàries per a fer persistents els objectes, es creïn, però, si ja existeixen, no es modifiquin. En cas d'haver triat *Drop and Create*, cada cop que s'obrís la base de dades s'esborrarien totes les taules (encara que tinguessin dades) i es tornarien a crear buides. Encara a la mateixa finestra, també cal configurar la **connexió amb la base de dades** que s'ha creat al primer pas. En cas que aquesta connexió s'hagi creat anteriorment, només cal seleccionar-la al desplegable etiquetat *Database Connection*. En cas contrari, caldrà crear-la. Un cop introduides totes les dades de configuració, cal fer clic al botó *Finish*.

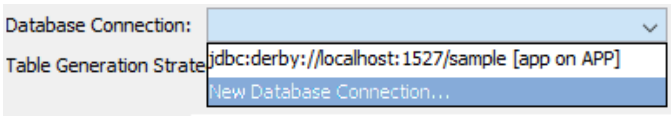
FIGURA 2.7. Finestra de configuració d'una nova unitat de persistència.



Si és necessari crear una nova connexió amb la base de dades, cal seguir els següents passos:

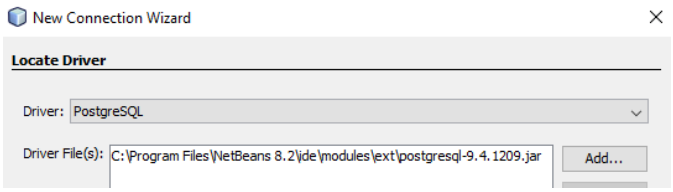
- Triar l'opció *New Database Connection...* al desplegable etiquetat *Database Connection*, com es veu a la figura 2.8

FIGURA 2.8. Nova connexió a una base de dades.



- A la següent pantalla, cal triar *PostreSQL* com a *Driver*, segons es veu a la figura 2.9, i, a continuació, fer clic a *Next*.

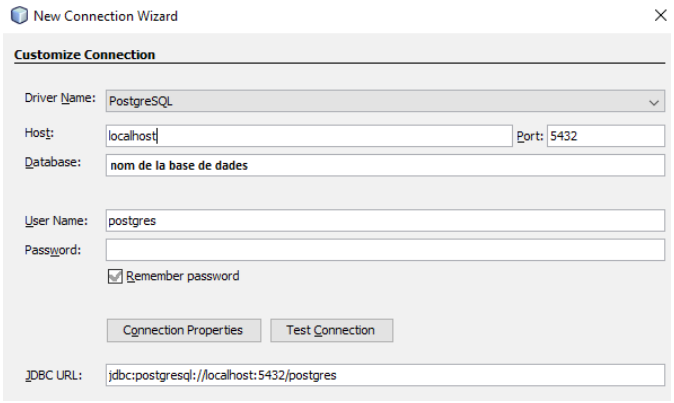
FIGURA 2.9. Selecció del //driver// de la connexió a la base de dades.



- A la nova finestra, que es veu a la figura 2.10, caldrà acabar la configuració de la connexió amb les dades:

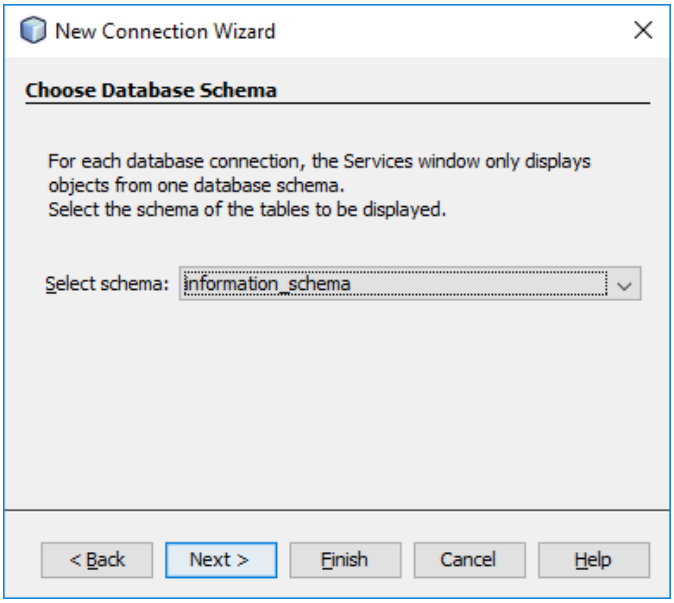
  - ***Host***: si el servidor és a la mateixa màquina on treballem (és habitual quan es fa desenvolupament), es pot deixar el valor que surt per defecte, *localhost*; en cas contrari, caldria posar l'adreça IP o el nom de domini de la màquina on fos el servidor.
  - ***Port***: el port per defecte de PostgreSQL és el que apareix: 5432. Només cal modificar-lo si s'ha canviat a la base de dades.
  - ***Database***: és el nom que heu donat a la base de dades quan l'heu creada.
  - ***User Name***: és el nom que heu donat a l'usuari amb el qual el programa es connectarà a la base de dades.
  - ***Password***: contrasenya de l'usuari amb el qual el programa es connectarà a la base de dades.
  - ***Remember password***: cal activar-lo, tret que vulguem que el programa ens demani la contrasenya a cada execució.

FIGURA 2.10. Configuració de les dades de la connexió.



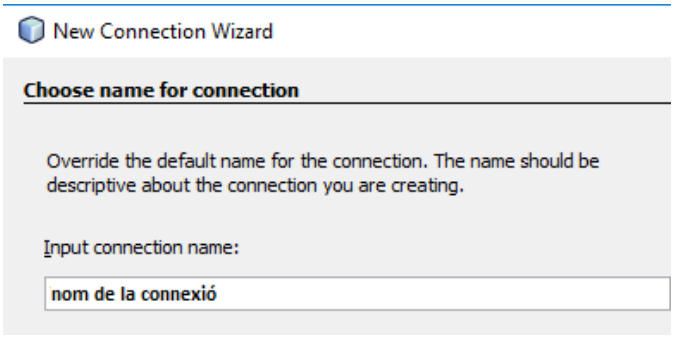
- Al quadre de text etiquetat amb JDBC URL ens apareix la URL de la connexió a partir de les dades que hem entrat als camps anteriors. Abans de passar a la següent pantalla, podem comprovar si la configuració és correcta amb el botó *Test Connection*. Quan ho sigui, caldrà fer clic al botó *Next*. Ens apareixerà la pantalla que es veu a la figura 2.11. En ella cal seleccionar l'equema de la base de dades amb el qual es crearan les taules que s'utilitzaran per a la persistència dels objectes i, a continuació, clicar *Next*.

FIGURA 2.11. Selecció de l'esquema.



- A l'última pantalla, que es mostra a la figura 2.12, caldrà indicar el nom de la connexió i prèmer el botó *Finish* per acabar la creació de la connexió. El nom només serveix per a identificar la connexió les properes vegades que hàgim de crear una unitat de persistència sobre aquesta base de dades.

FIGURA 2.12. Anomenat de la connexió.



2.6 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>
```

2.6.1 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 ...>

  ...
```

```
4
5   <entity class="ioc.dam.m6.exemples.comandes.UnitatDeMesura"
6     metadata_complete="true">
7     <attributes>
8       <id name="simbol"/>
9       <transient name="hashSimbol"/>
10    </attributes>
11  </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.

2.6.2 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 de l'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 de l'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 de l'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.

```java
@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;
    }
}
```

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.

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.

L'equivalent XML seria:

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

...

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

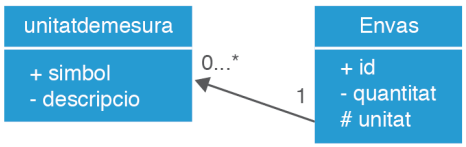
```
        <attributes>
            <id name="id">
                <generated-value strategy="TABLE"/>
            </id>
        </attributes>
    </entity>

...

</entity-mappings>
```

2.6.3 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 2.13. 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 2.13).

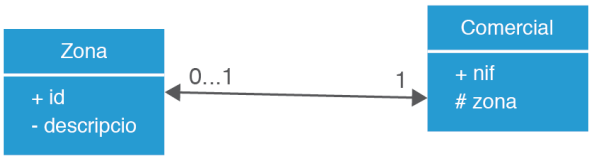
La diferència entre *OneToOne* i *ManyToOne* és més conceptual que pràctica, ja que ambdós (utilitzats per defecte) produeixen la mateixa conversió: una clau forana a la taula on s'emmagatzemarà la relació.

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 2.14). Aquí direm que la relació serà *OneToOne*.

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

```
8       ...
9   }
```

Figura 2.14. Esquema E-R de les taules Comercial i Zona



La versió XML dels dos exemples anteriors és la següent:

```
1   <entity-mappings ...>
2
3       ...
4
5       <entity class="ioc.dam.m6.exemples.comandes.Envas"
6                   metadata-complete="true">
7           <attributes>
8               <id name="id">
9                   <generated-value strategy="TABLE"/>
10              </id>
11              <many-to-one name="unitat"/>
12          </attributes>
13      </entity>
14
15      ...
16
17      <entity class="ioc.dam.m6.exemples.comandes.Comercial"
18                   metadata-complete="true">
19          <attributes>
20              <id name="nif"/>
21              <one-to-one name="zona"/>
22          </attributes>
23      </entity>
24
25  ...
26
27  </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>
```

2.6.4 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 de l'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`.

```java
@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:

```xml
<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.

```java
@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`.

```java
@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;
    @Column(nullable=false)
    private double quantitat;
    @ManyToOne
    @JoinColumn(name="unitat_simbol", nullable=false)
    private UnitatDeMesura unitat;

    ...
}
```

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

```xml
<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>
        <attributes>
            <id name="id">
                <generated-value strategy="TABLE"/>
            </id>
            <basic name="tipus">
                <column length="100" nullable="false" />
            </basic>
            <basic name="quantitat">
                <column nullable="false" />
            </basic>
            <many-to-one name="unitat">
                <join-column name="unitat_simbol"
                        nullable="false" />
            </many-to-one>
        </attributes>
    </entity>

    ...

</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 relació 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:

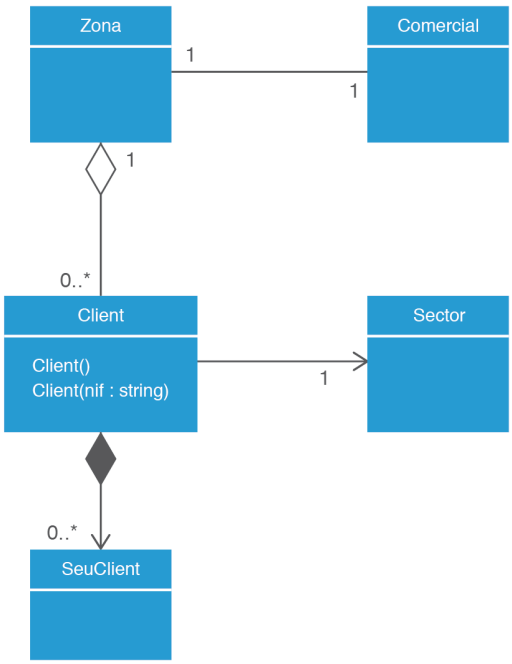
```
<entity-mappings ...>

    ...

    <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>
```

2.6.5 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 2.15 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*).

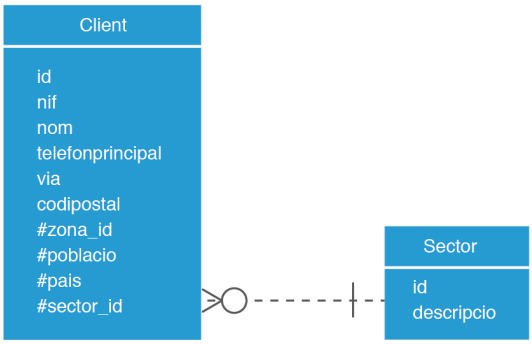
FIGURA 2.15. Diagrama de classes parcial de l'aplicació exemple



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 2.16).

Figura 2.16. 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.

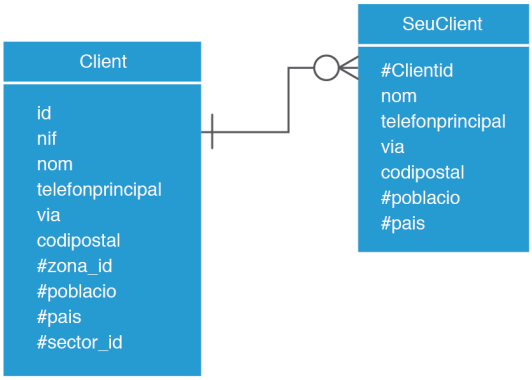
La implementació implicarà un atribut de tipus `Sector` qualificat com a *OneToOne* o *ManyToOne:*

```
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 2.17).

Figura 2.17. 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, les 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>();
  ...
}
```

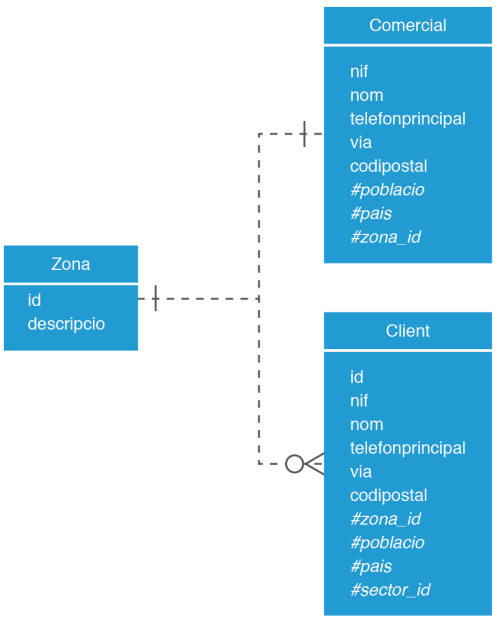
L'atribut `cascade` admet el valor `CascadeType.ALL` per indicar que la classe client s'encarregarà de tota la persistència de les seves seus.

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 2.18). 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 2.18. 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:

```
1  @Entity
2  public class Zona implements Serializable {
3      @Id
4      @Column(length=10)
5      private String id;
6      @Column(length=50)
7      private String descripcio;
8      @OneToMany(mappedBy = "zona", fetch=FetchType.LAZY)
9      @OrderBy(value="nif")
10     private List<Client> clients;
11     @OneToOne(mappedBy="zona", fetch= FetchType.LAZY)
12     private Comercial comercial;
13
14   ...
15 }
16
17 @Entity
18 public class Comercial implements Serializable {
19     @Id
20     @Column(length=15)
21     private String nif;
22     private String nom;
23     @OneToOne(fetch= FetchType.LAZY)
24     private Zona zona;
25   ...
26 }
27
28 @Entity
29 public class Client extends Empresa {
30     @ManyToOne(fetch= FetchType.LAZY)
31     private Zona zona;
32
33     @OneToMany(mappedBy="client", cascade={CascadeType.ALL},
34             fetch= FetchType.LAZY)
35     private Set<SeuClient>seus=new HashSet<SeuClient>();
36
37     @OneToOne(fetch= FetchType.LAZY)
38     private Sector sector=null;
39
40   ...
41 }
42
43 @Entity
44 public class SeuClient implements Serializable{
45     private Seu seu = new Seu();
46     @ManyToOne
```

```
47      private Client client;
48    ...
49  }
50
51  @Entity
52  public class Sector implements Serializable {
53      @Id
54      @Column(length=50)
55      private String id;
56    ...
57  }
```

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:

```
1   <entity-mappings ...>
2
3     ...
4
5     <entity class="ioc.dam.m6.exemples.comandes.Zona "
6       metadata-complete="true">
7       <attributes>
8         <id name="id">
9           <column length="10" />
10        </id>
11        <basic name="descripcio">
12          <column length="50" />
13        </basic>
14        <one-to-one name="comercial" mapped-by="zona"
15                fetch="LAZY" />
16        <one-to-many name="clients" mapped-by="zona"
17                fetch="LAZY" >
18          <order-by value="nif" />
19        </one-to-many>
20      </attributes>
21    </entity>
22
23    <entity class="ioc.dam.m6.exemples.comandes.Comercial "
24      metadata-complete="true">
25      <attributes>
26        <id name="nif">
27          <column length="15" />
28        </id>
29        <one-to-one name="zona" fetch="LAZY" />
30      </attributes>
31    </entity>
32
33    <entity class="ioc.dam.m6.exemples.comandes.Client "
34                        metadata-complete="true">
35      <attributes>
36        ...
37        <many-to-one name="zona" fetch="LAZY" />
38        <one-to-many name="seus" fetch="LAZY">
39          <cascade>
40            <cascade-all/>
41          </cascade>
42        </one-to-many>
43        <one-to-one name="sector" fetch="LAZY" />
44      </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.

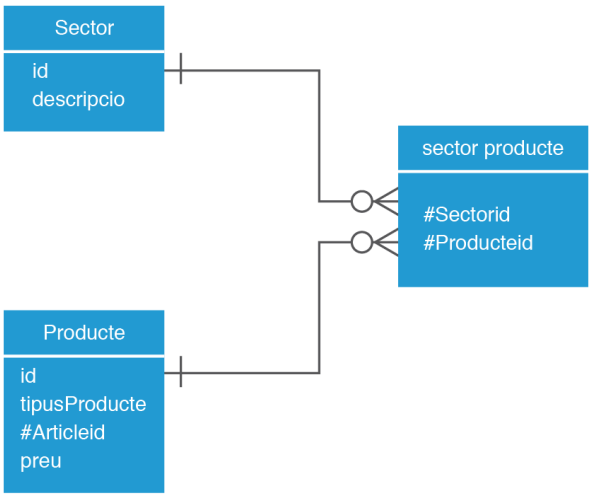
Un exemple de relació *ManyToMany*el podem trobar a la relació entre la classe `Sector` i la classe `Producte`.

```
@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 2.19, els registres de la taula *sector_producte* són només el resultat de combinar un sector i un producte.

FIGURA 2.19. 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*.

```
1  @Entity
2  public class Sector implements Serializable {
3      @Id
4      @Column(length=50)
5      private String id;
6      private String descripcio;
7      @ManyToMany(fetch= FetchType.LAZY, cascade= CascadeType.ALL)
8      @MapKey(name="id")
9      private Map<Long, Producte> productes =
10           new HashMap<Long, Producte>();
11   ...
12 }
```

`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.

2.6.6 Objectes incrustats (Embedded Objects)

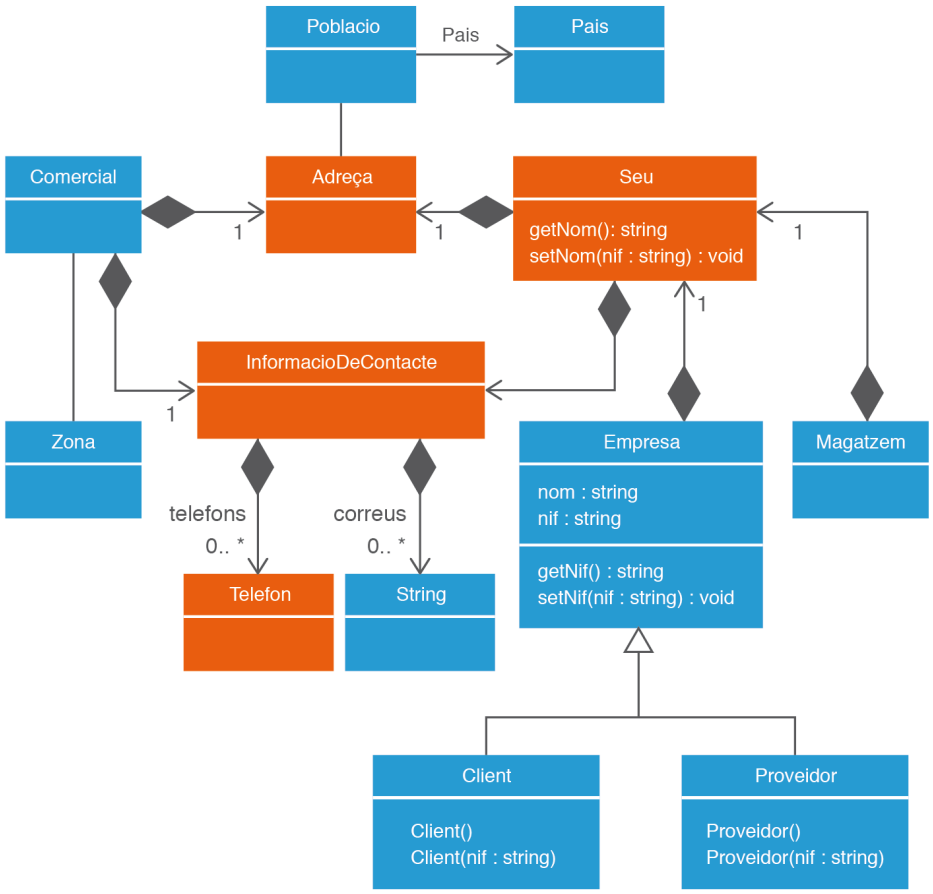
Anomenem objectes incrustats a aquells objectes no marcats com a entitats que estan continguts, directament o indirecta, en una entitat.

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 2.20). 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 2.20. 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 2.21. 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`:

```
@Embeddable
public class Adreca implements Serializable {
    private String via;
    @Column(length=10)
    private String codiPostal;
    @ManyToOne
    @JoinColumns({
        @JoinColumn(name="POBLACIO", referencedColumnName="NOM"),
        @JoinColumn(name="PAIS", referencedColumnName="NOM_PAIS")
    })
    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 2.21, 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.

```
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;
    }
  ...
```

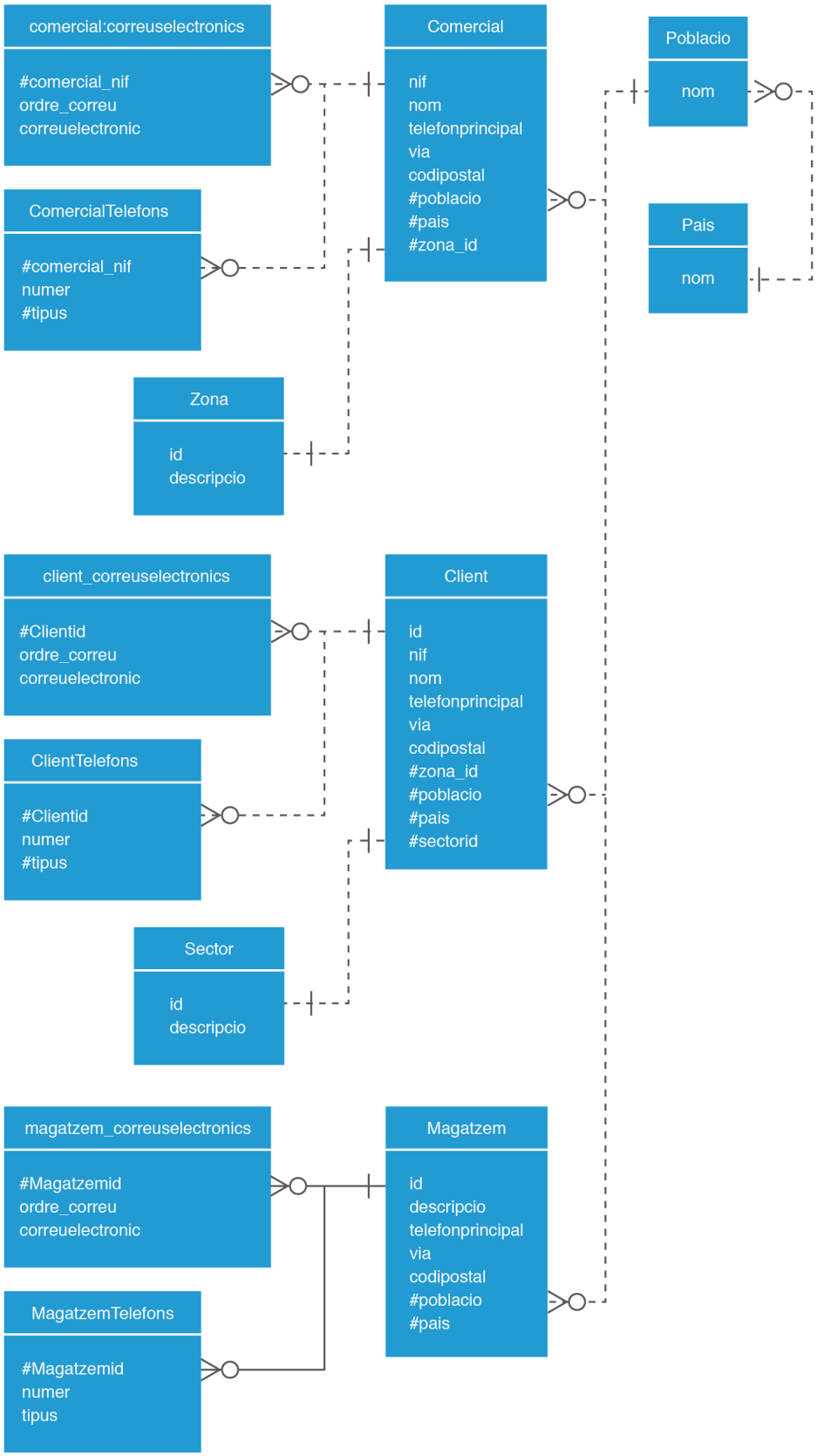
Donat un objecte `Comercial`, caldria afegir el seu codi postal fent :

```
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 `Adrecai` 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 2.22).

FIGURA 2.22. Esquema E-R de les taules mapades a partir de les entitats "Comercial", "Client" i els seus respectius



La classe Seu presenta el següent aspecte:

```
1 @Embeddable
2 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>
      <id name="id">
        <column length="20" />
      </id>
      <embedded name="seu">
        <attribute-override name="nom">
          <column name="DESCRIPCIO" />
        </attribute-override>
      </embedded>
    </attributes>
  </entity>
  ...

</entity-mappings>
```

2.6.7 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);
```

```
24          correusElectronics.set(pos+1, correusElectronics.get(pos));
25          correusElectronics.set(pos, aux);
26      }
27  }
```

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.

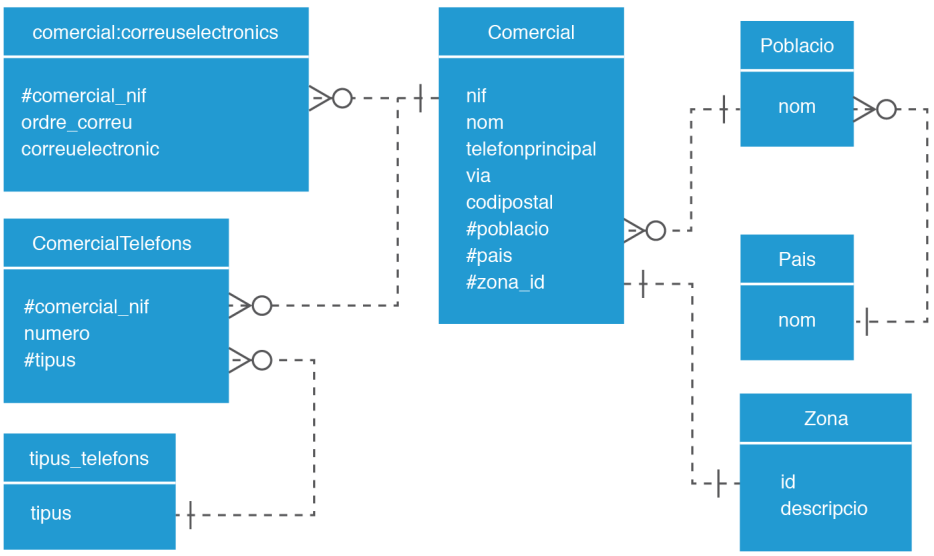
L'anotació `OrderColumn` indica a JPA que a la taula on s'emmagatzemi la col·lecció hi haurà una columna de més on s'emmagatzemarà l'ordre de la col·lecció, amb un valor de tipus numèric.

```
1  @ElementCollection(fetch= FetchType.LAZY)
2  @OrderColumn(name="ORDRE_CORREU")
3  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 2.23).

Figura 2.23. 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.

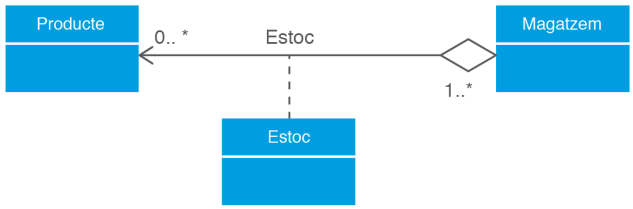
Map amb la clau de tipus entitat i valor de tipus bàsic o Embeddable

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 `Mapcom` 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 2.24). Esquemàticament, el model es podria representar com una classe associativa entre el `Magatzem` i el `Producte`.

Figura 2.24. 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`.

```
1 @Entity
2 public class Producte implements Serializable {
3     @Id
4     @GeneratedValue(strategy= GenerationType.TABLE)
5     private Long id;
6   ...
7 }
```

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*.

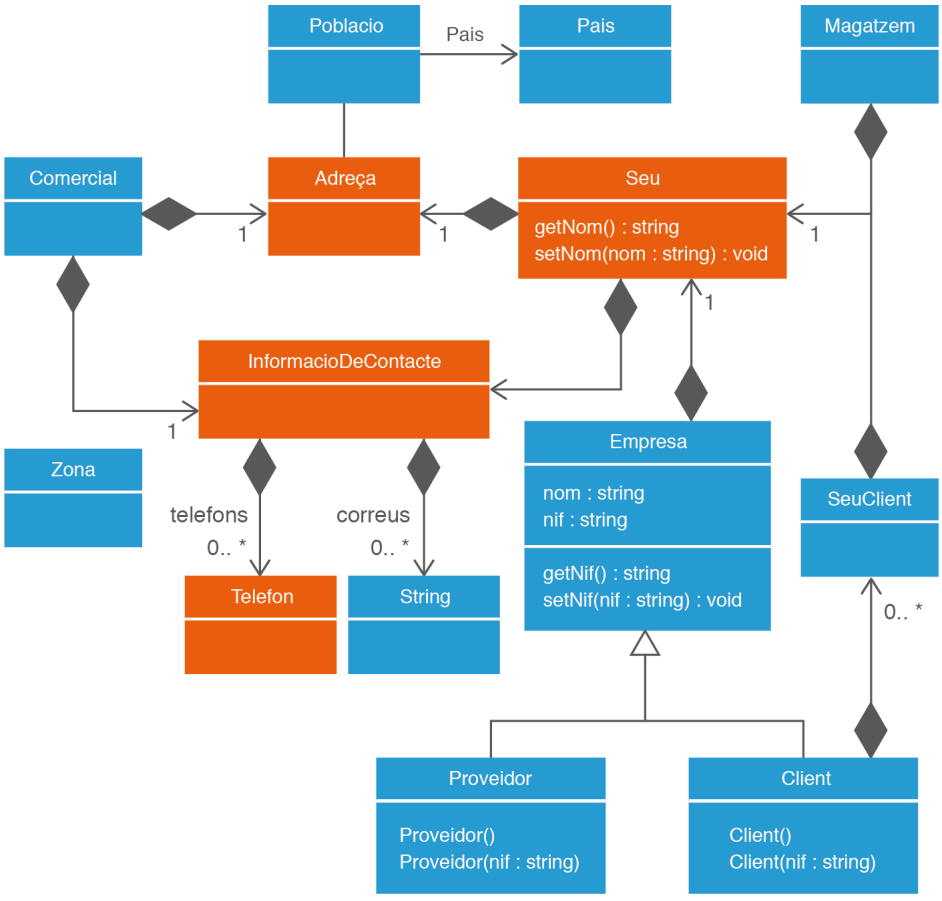
Map amb valor de tipus entitat

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 2.25). 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 2.25. 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:

```
@Entity
public class Client extends Empresa {
    @OneToMany(mappedBy="client", cascade={CascadeType.ALL},
               fetch=FetchType.LAZY)
    @MapKey(name="seu.nom")
    private Map<String, SeuClient>seus=new HashMap<String, SeuClient>();

    @ManyToOne(fetch= FetchType.LAZY)
    private Zona zona;

    @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;
    }
```

```
8
9       public SeuClient(String nom) {
10          this.seu.setNom(nom);
11      }
```

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`.

```
1   public String getNom() {
2           return seu.getNom();
3       }
```

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

```
1   @Id
2       @ManyToOne
3       public Client getClient() {
4           return client;
5       }
```

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

```
1   @Embedded
2       public Adreca getAdreca() {
3           return seu.getAdreca();
4       }
5
6       @Embedded
7       public InformacioDeContacte getInformacioDeContacte() {
8           return seu.getInformacioDeContacte();
9       }
```

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.

```
1   protected void setAdreca(Adreca adreca) {
2           this.seu.setAdreca(adreca);
3       }
4
5       protected void setInformacioDeContacte(InformacioDeContacte
6                   informacioDeContacte) {
7           this.seu.setInformacioDeContacte(informacioDeContacte);
8       }
9
10      public void setNom(String nom){
11          seu.setNom(nom);
12      }
13
14      public void setClient(Client client) {
15          this.client = client;
16      }
17  }
```

La representació a les taules de l'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>

  <entity class="ioc.dam.m6.exemples.comandes.Magatzem "
    metadata–complete="true">
    <attributes>
      <id name="id">
        <column length="20" />
      </id>
      <embedded name="seu">
        <attribute–override name="nom">
          <column name="DESCRIPCIO" />
        </attribute–override>
      </embedded>
      <element–collection name="telefons" fetch"LAZY">
        <map–key–columns name="producte_id" />
        <attribute–override
             name="value.ubicacio">
          <column name="zona_magatzem" />
        </attribute–override>
        <attribute–override
             name="value.quantitat">
          <column name="estoc" />
        </attribute–override>
      </element–collection >
    </attributes>
  </entity>

  <entity class="ioc.dam.m6.exemples.comandes.Producte "
    metadata–complete="true">
    <attributes>
      <id name="id">
        <column length="20" />
        <table–generator />
      </id>
      <embedded name="seu">
        <attribute–override name="nom">
          <column name="DESCRIPCIO" />
        </attribute–override>
      </embedded>
    </attributes>
  </entity>

  <entity class="ioc.dam.m6.exemples.comandes.SeuClient"
    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>
```

2.6.8 Identificadors compostos

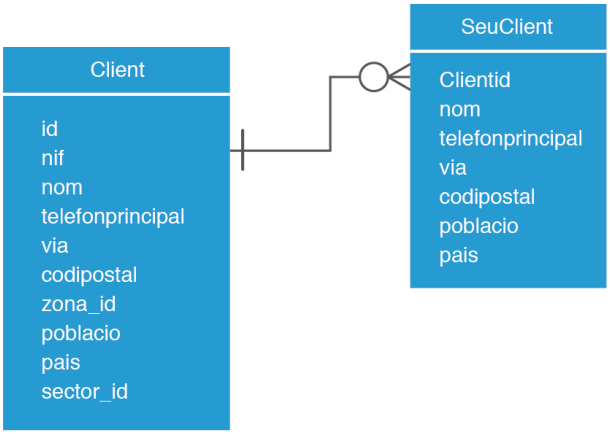
Les claus primàries compostes per diversos camps són força habituals en la majoria d'aplicacions. En aquest apartat veurem les dues respostes que ofereix JPA per tractar aquelles entitats que requereixin identificadors compostos.

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 2.26).

Figura 2.26. 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.

```
1  public class SeuClientId implements Serializable{
2      private String nom;
3      private Client client;
4
5      public SeuClientId() {
6      }
7
8      public SeuClientId(String nom, Client client) {
9          this.nom = nom;
10         this.client = client;
11     }
12
13     public String getNom() {
```

```
14          return nom;
15      }
16
17      public Client getClient() {
18          return client;
19      }
20
21      protected void setNom(String nom) {
22          this.nom = nom;
23      }
24
25      protected void setClient(Client client) {
26          this.client = client;
27      }
28  }
```

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`:

```
1  @Entity @Access(AccessType.PROPERTY)
2  @IdClass(SeuClientId.class)
3  public class SeuClient implements Serializable{
4    ...
5  }
```

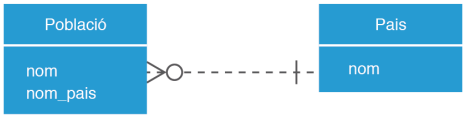
I si fem servir el format XML:

```
1  ...
2  <entity class="ioc.dam.m6.exemples.comandes.SeuClient"
3      metadata-complete="true" access="PROPERTY">
4    <id-class class="ioc.dam.m6.exemples.comandes.SeuClientId"/>
5    <attributes>
6      <id name="nom" />
7      <many-to-one name="client" id="true" />
8      <embedded name="adreca" />
9      <embedded name="informacioDeContacte" />
10   </attributes>
11 </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 `Pais` que, conjuntament amb el nom de la població, defineix la seva clau primària tal com es pot veure a la figura 2.27.

Figura 2.27. 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;
    }

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

    public Pais getPais() {
        return pais;
    }

    public void setPais(Pais pais) {
        this.pais = pais;
    }

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

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.

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

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

```
@Entity
public class Poblacio implements Serializable {
    private static final long serialVersionUID = 1L;
    @EmbeddedId
```

```
    private PoblacioId id;

    public Poblacio() {
    }

    public Poblacio(PoblacioId id) {
        this.id = id;
    }

    public Poblacio(String nom, Pais pais) {
        id = new PoblacioId(nom, pais);
    }

    public String getNom() {
        return id.getNom();
    }

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

    public Pais getPais() {
        return id.getPais();
    }

    protected void setPais(Pais pais) {
        this.id.setPais(pais);
    }

    private String getNomPais() {
        return id.getPais().getNom();
    }
}
```

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 de l'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() {
        return pais;
    }

    protected void setPais(Pais pais) {
        this.pais = pais;
        this.nomPais=pais.getNom();
    }
}
```

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.

```
public class PoblacioId implements Serializable{
    private String nom;
    private String nomPais;

    public PoblacioId() {
    }

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

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

    public PoblacioId(String nom, String nomPais) {
        this.nom = nom;
        this.nomPais = nomPais;
    }
}
```

2.6.9 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.

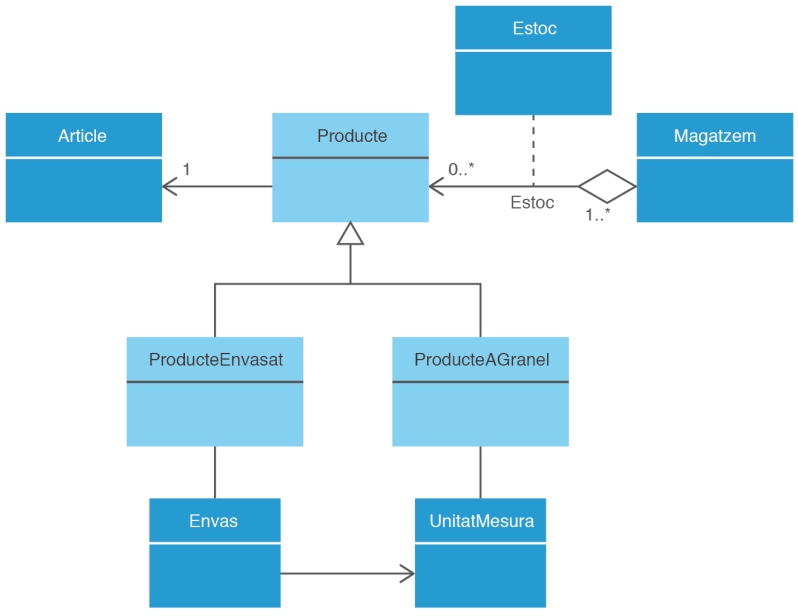
Concepte d'entitats en una jerarquia

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 2.28).

Figura 2.28. 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.

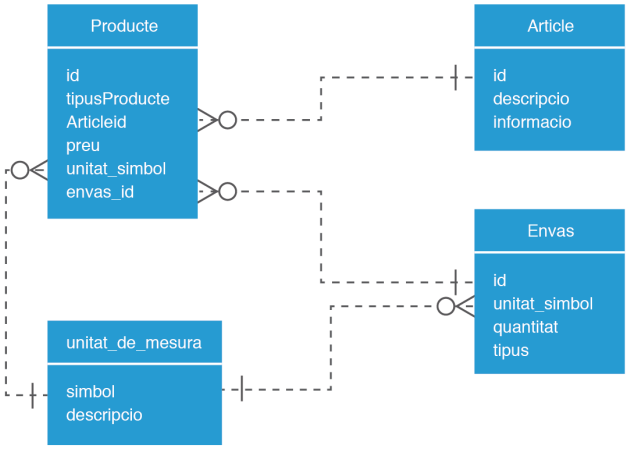
Una taula única per a tota la jerarquia

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 2.29). 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 2.29. Taula dels productes seguint l'estratègia d'emmagatzemar la jerarquia en una única taula




```
1  @Entity
2  @Inheritance(strategy= InheritanceType.SINGLE_TABLE)
3  @DiscriminatorColumn(name="TIPUS_PRODUCTE",
4                       discriminatorType= DiscriminatorType.INTEGER)
5  @DiscriminatorValue(value="0")
6  public class Producte implements Serializable {
7      @Id
8      @GeneratedValue(strategy= GenerationType.TABLE)
9      private Long id;
10     @ManyToOne
11     private Article article;
12     private double preu;
13
14     public Producte() {
15     }
16
17     public Producte(Article article, double preu) {
18         this.article = article;
19         this.preu = preu;
20     }
21
22     public Long getId() {
23         return id;
24     }
25
26     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;
    }

    public void setPreu(double preu) {
        this.preu = preu;
    }
}

@Entity
@DiscriminatorValue(value="1")
public class ProducteAGranel extends Producte {
    @ManyToOne
    private UnitatDeMesura unitat;

    public ProducteAGranel() {
    }

    public ProducteAGranel(Article article, double preu) {
        super(article, preu);
    }

    public ProducteAGranel(Article article, double preu,
            UnitatDeMesura unitat) {
        super(article, preu);
        this.unitat = unitat;
    }

    public UnitatDeMesura getUnitat() {
        return unitat;
    }

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

@Entity
@DiscriminatorValue(value="2")
public class ProducteEnvasat extends Producte {
    @ManyToOne
    Envas envas;

    public ProducteEnvasat() {
    }

    public ProducteEnvasat(Article article, double preu) {
        super(article, preu);
    }

    public ProducteEnvasat(Article article, double preu,
              Envas envas) {
```

```
97          super(article, preu);
98          this.envas = envas;
99      }
100 }
```

Fent servir fitxers de configuració XML, caldrà definir:

```
<entity-mappings ...>

  ...

  <entity class="ioc.dam.m6.exemples.comandes.Producte "
    metadata-complete="true">
    <inheritance strategy="SINGLE TABLE"/>
    <discriminator-column name="TIPUS_PRODUCTE"
          discriminator-type="INTEGER" />
    <discriminator-value> 0 </discriminator-value>
    ...
  </entity>

  <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>
```

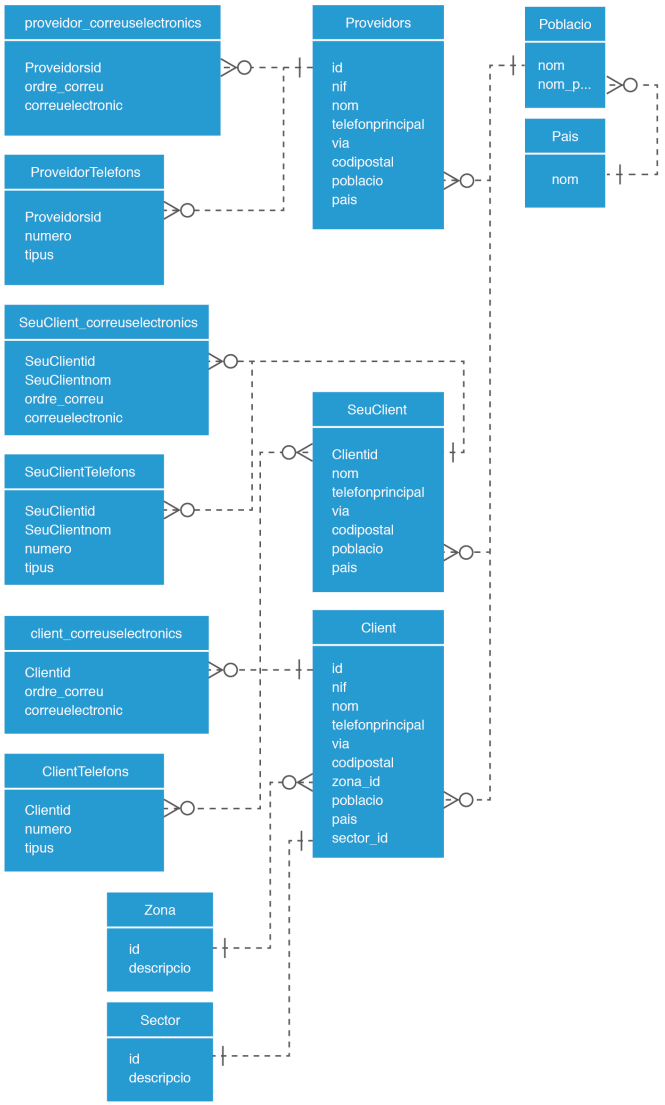
Una taula per a cada entitat

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 2.20), 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 2.30). 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*.

Figura 2.30. Taules de clients i proveïdors separades seguint l'estratègia d'una classe per entitat




```java
@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() {
        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  }
```

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:

```
1   <entity-mappings ...>
2
3     ...
4
5     <entity class="ioc.dam.m6.exemples.comandes.Empresa "
6       metadata-complete="true">
7       <inheritance strategy="TABLE PER CLASS"/>
8       ...
9     </entity>
10    ...
11
12  </entity-mappings>
```

Mapatge de la jerarquia

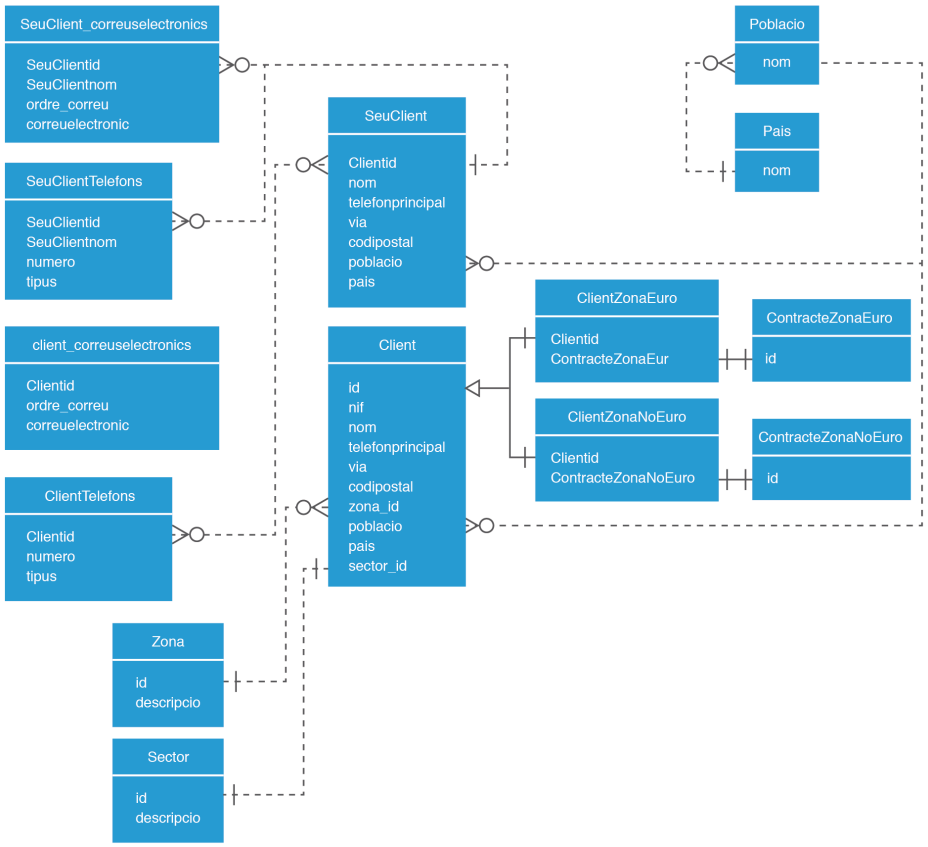
La darrera estratègia seria útil en cas que la jerarquia estigués formada per una classe principal que calgués recuperar sovint més un conjunt de classes derivades que especialitzessin la classe principal i només calgués recuperar en moments puntuals. Imaginem que necessitem una jerarquia en la qual 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 2.31).

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 {
  ...
}

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

Figura 2.31. 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:

```
<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>

  ...

</entity-mappings>
```

2.7 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 de l'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.

2.7.1 Creació d'una unitat de persistència per configurar la connexió a l'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 a l'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.

Consulteu el punt "Ús de JPA amb EclipseLink sobre NetBeans i PostgreSQL".

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.

2.7.2 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.

```
1 EntityManagerFactory emf =
2     Persistence.createEntityManagerFactory(
3         "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`.

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

2.7.3 Gestió d'entitats a l'EntityManager

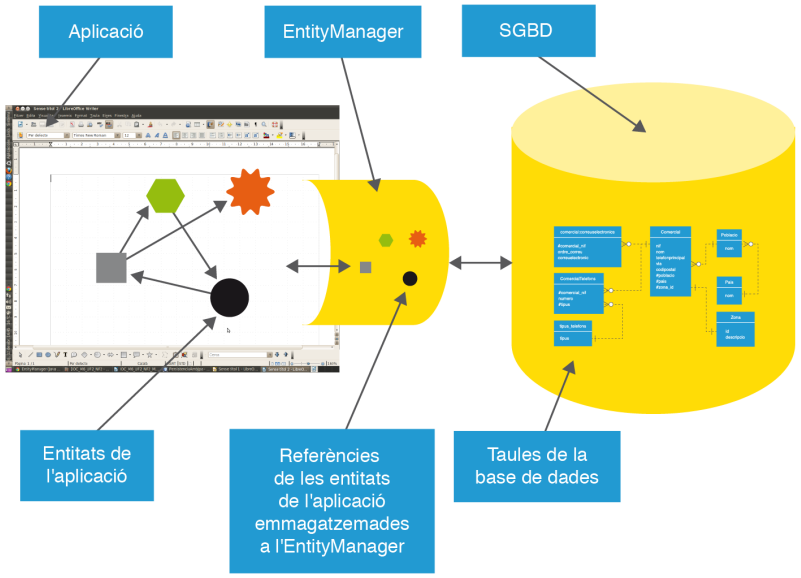
Per encarar amb èxit una aplicació gestionada per JPA caldrà donar a l'`EntityManager` un paper central en el tractament de les entitats. De fet, podem considerar-lo com el representant o interlocutor local de l'SGBD a la nostra aplicació.

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 2.32).

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 de l'SGBD per aconseguir emmagatzemar les dades de l'entitat.

FIGURA 2.32. 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 de l'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:

```
1 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:

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

o bé,

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

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

```
1 EntityManager em = emf.createEntityManager();
2
3 ...
4
5 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 a l'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();
```

```
17 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:

```
1  em = emf.createEntityManager();
2  ...
3  Query qry = em.createQuery("Select z from Zona z");
4  List<Zona> listZona = qry.getResultList();
5  ...
6  em.getTransaction().begin();
7  for(Zona v: listZona){
8      em.remove(v);
9  }
10 em.getTransaction().commit();
11 ...
12 em.getTransaction().begin();
13 em.flush();
14 em.getTransaction().commit();
15 em.close();
```

2.8 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*',

```
SELECT z
FROM Zona z
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:

```
SELECT z.id, z.comercial
FROM Zona z
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:

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

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

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

```
SELECT p
FROM Producte p
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*.

2.8.1 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();
```

2.8.2 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>
```

2.8.3 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 instanciï `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 a l'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:

```
client.nom = 'Bon Menjar S.L.'
producte.preu < 12.50
comanda.data = {d '2012-04-01'}
comanda.hora < {t '20-00-00'}
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*.

. . . . . .

TAULA 2.2 (continuació)

Operació	Descripció	Exemple
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*.
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:

```
SELECT s.id, COUNT(p)
FROM Sector s LEFT JOIN s.productes p
GROUP BY s.id
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
```

3. Bases de dades objecte-relacionals i orientades a objectes

En aquest apartat veurem encara dues alternatives més que intenten minimitzar el desfasament objecte-relacional. La primera de les alternatives està protagonitzada pels mateixos SGBD relacionals. El fort arrelament del paradigma orientat a objectes en el disseny i la implementació de les aplicacions actuals està obligant els SGBD relacionals a incorporar característiques orientades a objectes en els elements de disseny i explotació dels sistemes gestors. És el que s'ha batejat com a Sistemes Gestors de Bases de Dades Objecte-Relacionals.

La segona alternativa consisteix a usar directament un sistema gestor de bases de dades orientat a objectes. Per descomptat, es tracta de l'única alternativa que no presenta desfasament Objecte-Relacional, ja que les dades es tracten sempre, fins i tot durant l'emmagatzematge, com si fossin objectes i no taules.

3.1 Definició de dades en sistemes Objecte-Relacionals

Les diferents versions del llenguatge SQL han anat incorporant diverses característiques que els donen cada cop més flexibilitat per treballar directament amb objectes. La principal revisió es va produir el 1999 donant lloc a l'SQL99.

Es tracta d'una revisió profunda que afegeix un nombre considerable de tipus poc convencionals, a més de permetre també la definició de tipus compostos o estructurats de dades. Els principals tipus que incorpora són:

- **Boolean**. Fins aleshores, els valors lògics se solien indicar usant el tipus BIT.
- **Grans objectes**. SQL99 defineix dos tipus d'objectes grans, l'anomenat BLOB (Binary Large Object), adequat per emmagatzemar dades binàries com ara imatges, vídeos, música, certificats digitals, etc. L'altre tipus s'anomena CLOB (Character Large Object), ideal per a dades extensives de tipus text com ara informes, pàgines web, articles, etc.
- **Col·leccions**. Permet emmagatzemar de forma directa col·leccions senceres de dades tant de tipus bàsic com de tipus estructurat.
- **Tipus compostos o estructurats**. Gràcies a la incorporació d'aquests tipus de dades és possible crear tipus de dades definits per l'usuari.
- **Referències a tipus estructurats**. Es tracta d'un tipus especial que actua com a apuntador de tipus compostos. Són útils perquè permeten fer una abstracció del lloc (taula) on realment s'emmagatzemaran aquests tipus. El sistema està preparat per realitzar un emmagatzematge per defecte, de

manera que aquests tipus són l'única manera de referenciar les dades emmagatzemades.

La incorporació de tots aquests tipus de dades aporta molta més flexibilitat a l'hora de mapar les classes del model fent servir directament el llenguatge de definició DDL. A més, SQL amplia la sintaxi del llenguatge de consulta per poder usar directament els nous tipus, de forma semblant a la manipulació dels atributs dels objectes.

Malgrat que no es tracta d'una revisió recent, la profunditat del canvi necessari per incorporar aquests nous tipus fa que a dia d'avui encara hi ha SGBD que no acompleixen tota l'especificació de l'any 1999. Per exemple, PostgreSQL no suporta encara l'ús de referències a tipus estructurats. A més, cal remarcar també que la sintaxi usada pels diferents gestors O-R és menys estandarditzada que les revisions anteriors. Són elements a tenir en compte a l'hora d'escollir aquesta solució, ja que ens podem trobar amb aplicacions difícilment portables.

3.1.1 Suport de PostgreSQL als tipus definits per SQL99

PostgreSQL suporta el tipus *BOOLEAN* declarat com indica l'estàndard:

```
CREATE TABLE enquesta_client(
  id INTEGER PRIMARY KEY,
  recomanaria BOOLEAN,
  ...
);
```

En canvi, malgrat que dóna suport als tipus *BLOB* i *CLOB*, la seva sintaxi varia de l'estàndard. Aquest darrer especifica que s'indicarà la mida màxima que es destinarà a l'emmagatzematge de l'atribut:

```
CREATE TABLE Client (
  ...
  contracte CLOB(50K),
  signatura_electronica BLOB(5K),
  ...
);
```

El tipus de dada que PostgreSQL destina per emmagatzemar dades binàries grans és BYTEA, i el substitut de CLOB és TEXT. Ambdós tipus s'adapten de forma dinàmica a la quantitat de dades que s'emmagatzemi en cada moment. No admet la indicació de la mida. Exemple:

```
CREATE TABLE Client (
  ...
  contracte TEXT,
  signatura_electronica BYTEA,
  ...
);
```

Els tipus que especifiquen col·leccions presenten també algunes diferències sintàctiques. Mentre que l'estàndard admet formes com LIST, SET, MULTISET o

ARRAY, PostgreSQL només suporta el tipus ARRAY, el qual pot expressar-se seguint la forma estàndard:

```
CREATE TABLE Client (
  ...
  correus_electronics VARCHAR(100) ARRAY,
  ...
);
```

O bé usant la forma específica de PostgreSQL:

```
CREATE TABLE Client (
  ...
  correus_electronics VARCHAR(100)[],
  ...
);
```

Fixeu-vos que especificant l'atribut `correus_electronics` dels clients com un tipus ARRAY ens evitarem haver de crear una taula on emmagatzemar-los.

Quelcom de semblant succeeix amb els tipus compostos. L'estàndard admet dues formes: l'ús de la paraula clau ROW amb la composició de camps tancada entre parèntesis o bé la declaració prèvia de tipus a usar en una taula. PostgreSQL suporta només el darrer. Exemple:

```
CREATE TYPE t_adreca AS(
  via VARCHAR(255),
  codipostal VARCHAR(10),
  poblacio VARCHAR(100),
  pais VARCHAR(100)
);

CREATE TYPE t_telefon AS(
  tipus VARCHAR(25),
  numero VARCHAR(20)
);

CREATE TABLE Client (
  ...
  adreca t_adreca,
  correus_electronics VARCHAR(100) ARRAY,
  telefons t_telefon ARRAY
  ...
);
```

Els tipus compostos es declaren com si fossin taules però no s'indiquen restriccions, només s'accepten els noms i els tipus de cada camp. Fixeu-vos que fins i tot podem fer servir un tipus compost en una col·lecció.

És important aclarir que les taules creen, per defecte, un tipus compost amb el mateix nom de la taula. Cal anar en compte de no posar als tipus que haguem de definir el nom d'una taula existent, perquè ens donaria un error de duplicitat de nom.

Extrapolant aquesta consideració, el fet que les taules creïn tipus per defecte ens permet usar el nom de qualsevol taula com a tipus definit. Així, tenint definides les taules PAIS i POBLACIO podem usar els seus noms en la definició d'altres tipus de dades. Exemple:

```
CREATE TABLE pais(
  nom VARCHAR(100) NOT NULL,
  CONSTRAINT pais_pkey PRIMARY KEY (nom)
);

CREATE TABLE poblacio(
  nom VARCHAR(100) NOT NULL,
  nom_pais VARCHAR(100) NOT NULL,
  CONSTRAINT poblacio_pkey PRIMARY KEY (nom, nom_pais),
  CONSTRAINT fk32ab30ac759422e0 FOREIGN KEY (nom_pais)
      REFERENCES pais (nom) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION
);

CREATE TYPE t_adreca AS(
  via VARCHAR(255),
  codipostal VARCHAR(10),
  poblacio poblacio
);
```

És important aclarir també que els tipus estructurats no permeten definir referències a claus foranes en les sentències DDL. Es tracta d'un repte que els SGBD hauran de plantejar-se si volen incorporar les tècniques d'orientació a objectes.

3.2 Manipulació de dades en sistemes de gestió Objecte-Relacionals

JDBC va haver de fer canvis també per adaptar-se al nou SQL. La versió 2.0 va incorporar un conjunt de classes que facilitaven el mapatge d'objectes a partir de les millores introduïdes des de l'SQL.

Aquestes millores són les següents:

- Tipus especials de dades
- Ús d'*arrays* en sentències DML
- Ús de tipus estructurats en sentències DML

3.2.1 Tipus especials de dades

La incorporació dels tipus de dades BLOB, CLOB o ARRAYS va obligar a redissenyar les classes d'obtenció de dades, ja que poden representar objectes de grans dimensions i pot arribar a no ser aconsellable traslladar tota la informació des de l'origen de dades a la màquina client.

La manipulació d'aquests tipus de dades no es realitza bolcant-les a tipus de dades estàndards, sinó que s'utilitzen unes classes especials (java.sql.Blob, java.sql.Clob o java.sql.Array) que actuen com a punters lògics a la informació desada a la base de dades.

Si fem servir objectes `Blob` o `Clob` disposarem del mètode `getBinaryStream` o `getAsciiStream`,respectivament, per anar obtenint la informació parcialment, a mida que la necessitem. Els mètodes retornen un objecte `InputStream` que tractarem de forma habitual. Generalment, disposarem d'una eina de manipulació de la informació (processador de textos, editors d'imatges, vídeos, gestor de fitxers, etc.) que treballi amb 'Streams'.

És important tenir en compte que la connexió haurà de romandre oberta mentre duri la lectura dels *Streams* obtinguts, ja que en cas contrari el punter perdria la font de dades i obtindríem un error.

Si fem servir objectes *array*, disposem del mètode `getArray` per obtenir un vector de Java. Heu de saber però, que les dades no es troben a la memòria local, sinó a l'SGBD. Aquest vector actua amb caràcter general com si es tractés d'un vector local, de manera que a l'hora de codificar no notarem la diferència.

```
ResultSet rs = stmt.executeQuery(
    "SELECT correus_electronics FROM client WHERE id=" + client.getId());
while (rs.next()) {
    Array arraySqlCorreus = rs.getArray(1);
    String[] correus = (String[])arraySqlCorreus.getArray();
    for (int i = 0; i < correus.length; i++) {
        ...
    }
}
```

Observeu la diferència entre la crida al mètode `getArray()` del ResultSet que retorna un objecte de tipus `java.sql.Array`(`arraySqlCorreus`)i la crida al mètode `getArray` de la variable `arraySqlCorreus`, la qual retorna un vector Java (`correus`).

3.2.2 Ús d'arrays en sentències DML

Com ja s'ha comentat, els canvis introduïts des de la revisió SQL99 no només afecten les sentències DDL, sinó que es va modificar la sintaxi de les sentències DML per poder treure el màxim de profit d'aquests canvis.

La introducció de dades permet afegir col·leccions senceres d'elements tractant-los com si fossin una única columna. Per exemple, és possible definir una sentència d'inserció d'un client que li afegeixi diversos correus electrònics usant el constructor de tipus ARRAY:

```
INSERT INTO CLIENT (id, correus_electronics)
    values(1, ARRAY['aaa@a.cat', 'bbb@bb.es', 'ccc@m.es']);
```

Ara bé, normalment obtindrem les dades des d'algun mètode de la classe client que ens retorni la col·lecció de correus del client concret. Per això JDBC disposa d'un constructor propi per poder automatitzar aquesta conversió:

```
PreparedStatement comStm = null;
        ...
```

```
StringBuilder comStr = new StringBuilder();
       comStr.append("INSERT INTO client (id, correus_electronics) ");
       comStr.append("VALUES(?, ?)");
  ...
comStm = getConnexio().prepareStatement(comStr.toString());

comStm.setInteger(1, entitat.getId());
Array correu = getConnexio().createArrayOf("varchar",
    entitat.getInformacioDeContacte().getArrayCorreusElectronics());
comStm.setArray(2, correu);

comStm.executeUpdate();
...
```

El mètode `createArrayOf` de la classe `Connection` ens permet crear un objecte `java.sql.Array` a partir d'un vector Java. La instància `java.sql.Array` es podrà passar com a paràmetre d'un `PreparedStatment`, el qual realitzarà la conversió a l'estàndard SQL per poder fer la inserció correctament.

En sentències d'actualització, els tipus ARRAY poden ser actualitzats de forma global o parcialment, element per element. A l'exemple següent realitzem una actualització global de tota la col·lecció sencera.

```
UPDATE CLIENT SET correus_electronics= ARRAY['aaa@a.cat', 'bbb@bb.es', 'ccc@m.
    es'] WHERE id=10;
```

Mentre que a continuació només es modificarà la posició 1 de la col·lecció:

```
UPDATE CLIENT SET correus_electronics[1] = 'aaa@a.cat' WHERE id=10;
```

És important que tingueu en compte que la indexació dels ARRAY SQL inicien la seva numeració per 1 en comptes d'iniciar-la per 0, com és habitual en Java.

En cas d'actualitzar un element, el tipus a utilitzar no serà un `java.sql.Array`, sinó un element del tipus adequat.

```
PreparedStatement comStm = null;
        ...
StringBuilder comStr = new StringBuilder();
       comStr.append("UPDATE CLIENT SET correus_electronics[1] = ? ");
       comStr.append("WHERE id=?");
  ...
comStm = getConnexio().prepareStatement(comStr.toString());

comStm.seString(camp, entitat.getInformacioDeContacte()
                                      .getArrayCorreusElectronics()[0]);
comStm.setInteger(1, entitat.getId());

comStm.executeUpdate();
...
```

Òbviament, és possible actualitzar també tota la col·lecció des de JDBC:

```
PreparedStatement comStm = null;
        ...
StringBuilder comStr = new StringBuilder();
       comStr.append("UPDATE CLIENT SET correus_electronics = ? ");
       comStr.append("WHERE id=?");
  ...
comStm = getConnexio().prepareStatement(comStr.toString());

```

```
Array correu = getConnexio().createArrayOf("varchar",
    entitat.getInformacioDeContacte().getArrayCorreusElectronics());
comStm.setArray(1, correu);
comStm.setInteger(1, entitat.getId());

comStm.executeUpdate();
...
```

De forma semblant, les consultes admeten una sintaxi similar, tant en format SQL com en l'adaptació JDBC. Per exemple, la sentència següent obtindria aquells clients que tinguessin la cadena *iii@m.es* com a valor d'alguna de les seves adreces electròniques.

```
SELECT c1.id
FROM CLIENT c1
WHERE 'iii@m.es' = ANY(c1.correus_electronics);
```

3.2.3 Ús de tipus estructurats en sentències DML

Com en el cas del tipus *ARRAY*, SQL preveu una sintaxi específica per poder manipular dades estructurades. La notació és semblant a la sintaxi dels llenguatges orientats a objectes, és a dir, separant amb punts el camí cap a la dada que es vulgui referir.

Aquí, però, apareixen també problemes de compatibilitat d'alguns SGBD amb l'estàndard SQL. Per exemple, PostgreSQL utilitza diferent notació per referenciar una dada estructurada en una consulta que en la resta de sentències DML. A les consultes cal tancar entre parèntesis els successius atributs que referenciïn la dada, mentre que a la resta de sentències no cal.

Inserció de registres que continguin dades estructurades

Les dades estructurades durant les sentencies d'inserció cal tancar-les entre parèntesis i separar cada valor per comes. En cas de tractar-se d'estructures imbricades, el valor que representi una dada composta anirà també tancat entre parèntesis. El gestor deduirà l'estructura que representi la dada. En cas que el gestor no pugui fer la deducció de forma automàtica, és possible indicar expressament el tipus de dada representat fent servir la funció CAST. Vegem un exemple d'inserció d'un registre a la taula COMERCIAL, suposant que aquesta estigui definida amb els tipus `t_adreca` i `t_info_contacte`, tipus que detallem també aquí i que com podeu observar també es troben constituïts d'altres tipus compostos o *arrays*.

```
CREATE TYPE t_telefon AS(
  tipus VARCHAR(25),
  numero VARCHAR(20)
);

CREATE TYPE t_info_contacte AS (
  telefonprincipal VARCHAR(15),
  correus_electronics VARCHAR(100) ARRAY,
  telefons t_telefon ARRAY
```

```
);

CREATE TYPE t_adreca AS(
  via VARCHAR(255),
  codipostal VARCHAR(10),
  poblacio poblacio
);

CREATE TABLE comercial(
  nif VARCHAR(15) NOT NULL,
  nom VARCHAR(255),
  adreca t_adreca,
  info_contacte t_info_contacte,
  zona_id VARCHAR(10),
  CONSTRAINT comercial_pkey PRIMARY KEY (nif),
  CONSTRAINT fk400731df1cf26051 FOREIGN KEY (zona_id)
      REFERENCES zona (id) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION
);
```

Un exemple d'inserció podria ser el següent:

```
INSERT INTO COMERCIAL (nif, nom, adreca, info_contacte, zona_id)
  values('12365478F', 'Comercial 1',
         ('carrer fum', '08009', ('Barcelona', 'Espanya')),
         ('93333333', ARRAY['aaa@a.cat', 'bbb@bb.es', 'ccc@m.es'],
         CAST(ARRAY[('fix', '93333333'), ('mbl','666666666')]
                AS t_telefon ARRAY))
         , 'BCN');
```

És possible definir NULL en qualsevol valor d'una dada estructurada. Aquesta seria la manera d'inserir parcialment les dades d'un registre.

```
INSERT INTO COMERCIAL (nif, nom, adreca, info_contacte, zona_id)
  values('12365478F', 'Comercial 1',
         ('carrer fum', '08009', ('Barcelona', 'Espanya')),
         ('93333333', ARRAY['aaa@a.cat', 'bbb@bb.es', 'ccc@m.es'],
         NULL, 'BCN');
```

Actualització de registres que continguin dades estructurades

En les sentències d'actualització de registres ens referirem a una dada que formi part d'una estructura indicant el camí fent servir punts de separació.

```
UPDATE CLIENT
SET seu.info_contacte.telefons = CAST(ARRAY[('fix', '93333333'),
          ('mbl','666666666')] AS t_telefon[])
WHERE id=1;
```

En aquest exemple hem de suposar que la taula CLIENT s'ha definit amb un tipus que representa la seu, que conté un tipus `t_info_contacte` de manera semblant a com s'ha estructurat la classe descrita en l'anterior unitat.

Sentències de consultes amb referències a dades estructurades

Com ja hem explicat, PostgreSQL fa servir una sintaxi lleugerament diferent en les seves sentències SQL per tal de diferenciar el que són columnes d'una taula

del que són atributs d'un tipus de dades. Quan es tracta d'atributs en comptes de columnes caldrà tancar el nom de l'atribut entre parèntesis.

```
1 SELECT cl.id
2 FROM CLIENT cl
3 WHERE 'bbb@bb.es' = ANY(((cl.seu).info_contacte).correus_electronics);
```

Fixeu-vos que *seu*, en tractar-se d'una columna, no va tancada entre parèntesis. En canvi, *cl.seu* sí que s'hi tanca perquè fa referència a una dada estructurada, igual que `( (cl.seu).info_contacte)`.

3.2.4 Tractament d'objecte en aplicacions que usin JDBC 2.0 o superior

La notació SQL pot representar un cert avantatge a l'hora de construir les peticions SQL, però segueix requerint la creació de sentències específiques d'inserció i actualització de cada entitat persistent sense permetre aprofitar realment els mecanismes del paradigma orientat a objectes.

És per això que, a mida que JDBC va evolucionant, incorpora cada cop més abstracció respecte al codi SQL, per tal que sigui més fàcil la reutilització del codi emprat en el mapatge d'una classe.

La tècnica consisteix a estructurar de la mateixa manera el model orientat a objectes i les taules i tipus compostos a l'SGBD. Per exemple, si la classe `Client` conté una classe de tipus `Seu`, la qual està formada per un nom, un atribut de tipus `Adreca` i un atribut de tipus `InformacioContacte`, al'SGBD caldrà crear un tipus `t_adreca`, `t_info_contacte` i `t_seu` que reflecteixin la mateixa estructura que la que es desprèn del model.

Totes les entitats i les seves classes contingudes implementaran la interfície SQLData de l'API JDBC. Es tracta d'una interfície que declara dos mètodes *readSQL* i *writeSQL*. Cada un d'ells servirà per codificar com s'ha d'intercanviar la informació de les dades obtingudes en una consulta amb els atributs de l'objecte o com traspassar la informació de l'objecte a una sentència SQL, respectivament.

El mètode `readSQL` rep una instància d'un `InputStream` específic anomenat `SQLInput`, més una cadena indicant el nom del tipus de dada definit a l'SGBD d'on provenen les dades contingudes a l'*Stream*. La cadena es pot fer servir per validar la compatibilitat de les dades o, en cas d'existir diversos tipus compatibles, per assignar a l'objecte lector el tipus originari.

El mètode `writeSQL` rep només un `OutputStream` específic anomenat `SQLOutput`, el qual farà de receptor de les dades que l'objecte haurà de traspassar a l'SGBD durant una inserció o actualització.

Aquests són mètodes que cal implementar, però que l'usuari no haurà de cridar mai. De la invocació se n'encarreguen els mètodes `getObject` del `ResultSet` (obtingut com a resultat d'una consulta) o `setObject` del `PreparedStatement`, utilitzat per enviar l'*stream* a una sentència parametritzada SQL.

Si codifiquem adequadament el nostre model fent que totes les entitats i classes contingudes implementin `SQLData`, la implementació d'algorismes d'intercanvi de dades se simplificarà moltíssim, perquè aconseguirem una reutilització del codi molt eficaç.

Vegem un exemple amb la classe `Comercial` que ens ha servit d'exemple. Per tal de deixar intacte el model farem servir `Classe` derivades. D'aquesta manera, quan es decideixi implementar un altre sistema de persistència, no caldrà modificar el model. Les classes derivades heretaran de les classes del model i afegiran la implementació adequada dels mètodes de la interfície SQLData.

Així, per exemple, codificarem la classe `ComercialSQLData` de la següent manera:

```
1  public class ComercialSQLData extends Comercial implements SQLData{
2      private String sqlTypeName;
3
4      // Com que per defecte es crea una instància d'Adreca i d'
5      //InformacioDeContacte, aquí cal assegurar que les instàncies seran
6      // respectivament de tipus AdrecaSQLData i InformacioContacteSQLData
7      public ComercialSQLData() {
8          setAdreca(new AdrecaSQLData());
9          setInformacioDeContacte(new InformacioContacteSQLData());
10     }
11
12     // Com que per defecte es crea una instància d'Adreca i d'
13     //InformacioDeContacte, aquí cal assegurar que les instàncies seran
14     // respectivament de tipus AdrecaSQLData i InformacioContacteSQLData
15     public ComercialSQLData(String nif) {
16         super(nif);
17         setAdreca(new AdrecaSQLData());
18         setInformacioDeContacte(new InformacioContacteSQLData());
19     }
20
21     public String getSQLTypeName() throws SQLException {
22         return sqlTypeName;
23     }
24
25     public void readSQL(SQLInput sqli, String typeName)
26                                                 throws SQLException {
27         sqlTypeName=typeName;
28         setNif(sqli.readString());
29         setNom(sqli.readString());
30         setAdreca((Adreca) sqli.readObject());
31         setInformacioDeContacte(
32         (InformacioDeContacte) sqli.readObject());
33         setZona((Zona) sqli.readObject());
34     }
35
36     public void writeSQL(SQLOutput sqlo) throws SQLException {
37         sqlo.writeString(getNif());
38         sqlo.writeString(getNom());
39         sqlo.writeObject((AdrecaSQLData) getAdreca());
40         sqlo.writeObject(
41                   (InformacioContacteSQLData) getInformacioDeContacte());
42         sqlo.writeObject((ZonaSQLData) getZona());
43     }
44 }
```

La invocació del mètode `readObject` de `SQLInput` crearà una instància de l'objecte esperat i farà una crida al mètode `readSQL` per tal que pugui omplir les seves dades.

De forma semblant, la invocació de `writeObject` farà una crida a writeSQL per tal que la instància passada per paràmetre pugui bolcar les seves dades al'*Stream*.

Això significa que també caldrà codificar les classes `AdrecaSQL`, `InformacioContacteSQL` o `ZonaSQL`. La implementació d'aquestes classes presentarà un format força similar. Per exemple, `AdrecaSQLData` s'implementaria fent:

```
public class AdrecaSQLData extends Adreca implements SQLData{
    private String sqlTypeName;

    public AdrecaSQLData() {
    }

    //En aquest cas assegurem també que Poblacio és PoblacioSQLData
    public AdrecaSQLData(String via, String codiPostal,
                                          PoblacioSQLData poblacio) {
        super(via, codiPostal, poblacio);
    }

    public String getSQLTypeName() throws SQLException {
        return sqlTypeName;
    }

    public void readSQL(SQLInput sqli, String typeName)
                                                  throws SQLException {
        sqlTypeName=typeName;
        setVia(sqli.readString());
        setCodiPostal(sqli.readString());
        setPoblacio((PoblacioSQLData) sqli.readObject());
    }

    public void writeSQL(SQLOutput sqlo) throws SQLException {
        sqlo.writeString(getVia());
        sqlo.writeString(getCodiPostal());
        sqlo.writeObject((PoblacioSQLData) getPoblacio());
    }
}
```

Com que `Adreca`conté un objecte `Població`, també serà necessària la implementació de `PoblacioSQLData`, i així successivament, de manera que les successives crides recursives vagin transferint la informació des dels objectes al JDBC o a l'inrevés.

Des de l'aplicació caldrà assegurar que sempre es treballa amb les classes derivades en comptes de les del model original. És a dir, que si cal instanciar un comercial caldrà assegurar que es tracta d'una instància de `ComercialSQLData`.

En aquest mateix sentit, quan a JDBC li calgui instanciar objectes, necessitarà saber quina classe haurà d'instanciar en cada moment. És possible configurar les connexions JDBC per tal d'associar els noms de tipus i taules residents a l'SGBD amb la seva classe corresponent. L'associació es fa mitjançant un objecte de tipus `Map` i es pot configurar passant per paràmetre la instància del `Map` durant la invocació de `SetTypeMap`.

Imaginem el codi necessari per configurar la connexió per treballar amb l'exemple de comercials que acabem de veure.

```
Map map = new HashMap();
map.put("pais", PaisSQLData.class);
```

```
map.put("poblacio", PoblacioSQLData.class);
map.put("t_adreca", AdrecaSQLData.class);
map.put("t_telefon", TelefonSQLData.class);
map.put("t_info_contacte", InformacioContacteSQLData.class);
map.put("zona", ZonaSQLData.class);
...
getConnexio().setTypeMap(map);
```

Usant aquest sistema, la inserció d'un comercial es podria reduir a:

```
PreparedStatement comStm = null;
        ...
StringBuilder comStr = new StringBuilder();
       comStr.append("INSERT INTO Comercial (nif, adreca ");
       comStr.append(", info_contacte, zona) ");
       comStr.append("VALUES(?, ?, ?, ?)");
  ...
comStm = getConnexio().prepareStatement(comStr.toString());

comStm.setString(1, entitat.getNif());
comStm.setObject(2, entitat.getAdreca());
comStm.setObject(3, entitat.getInformacioContacte());
comStm.setObject(4, entitat.getZona());
comStm.executeUpdate();
...
```

Malauradament, aquest sistema no està implementat per tots els *drivers* JDBC. Per exemple, els *drivers* de PostgreSQL encara no han incorporat aquesta utilitat, i si intenteu implementar l'exemple anterior amb aquest *driver* obtindreu un error.

3.3 Bases de Dades Orientades a Objecte

Podeu trobar informació sobre JDO a l'apartat *Eines de mapatge objecte-relacional (ORM)*

La darrera alternativa és fer servir directament una base de dades orientada a objectes. Aquestes bases de dades fan persistents els objectes de memòria, sense que calgui cap transformació. Per tant, s'emmagatzemen com objectes. Posteriorment també seran recuperats com objectes, també sense fer cap transformació.

Tot i que sense fer una anàlisi gaire profunda pot semblar que es tracta de la millor solució, aquesta modalitat de bases de dades presenta certes peculiaritats que cal tenir molt en compte.

Actualment trobem al mercat dos tipus de bases de dades orientades a objecte. Les que compleixen l'estàndard proposat per l'ODMG (Objecte Database Management Group) a finals de l'any 2000 i les anomenades bases de dades de tipus NoSQL, iniciatives tecnològiques posteriors. El llenguatge de consulta i manipulació de les primeres té clares influències d'SQL. El de les segones és proper a alguns llenguatges de programació orientats a objectes (com Java, C#, C++ o Smalltalk).

L'ODMG es va desfer l'any 2001, després de culminar l'estàndard anomenat ODMG 3.0. Tot i el temps passat, es tracta d'un estàndard molt infrautilitzat. Els llenguatges que especifica (ODL i OQL), tot i tenir una potència expressiva gran i permetre una adaptació real a la sintaxi habitualment utilitzada per la majoria

de llenguatges orientats a objectes, presenta una enorme influència del paradigma relacional.

De fet, ha representat una important influència en l'especificació d'altres estàndards, com JDO o JPA, que es postulen com a pont entre sistemes gestors de bases de dades objecte-relacionals i les aplicacions orientades a l'objecte.

Us podeu descarregar ObjectDB des de la seva pàgina web: www.objectdb.com

ObjectDB és una base de dades orientada a objectes. És programari propietari, però el fabricant, *ObjectDB Software*, ofereix una versió gratuïta limitada que és suficient per al seu estudi. Ofereix dues interfícies al programador: JPA (des de 2010) i JDO (des de la seva primera versió, el 2003). Des del 2010 està previst també oferir suport per a .NET, però en el moment d'escriure aquest text (maig del 2019), continua essent un projecte. L'avantatge d'utilitzar aquestes interfícies és que es pot traslladar molt fàcilment programes realitzats amb mapatge objecte-relacionats a ObjectDB i a l'inrevés. A més, ObjectDB disposa també d'un entorn gràfic per a consultar i actualitzar les dades. Aquest entorn gràfic admet la realització de querys amb JPQL i JDOQL, que són els DML (*Data Manipulation Language*) respectius de JPA i JDO.

Per treballar des del programa amb ObjectDB cal afegir la biblioteca *objectdb.jar* al nostre projecte. És la primera cosa que hem de fer després de descarrgar i descomprimir el fitxer que conté ObjectDB. Aquest fitxer *.jar* és a la carpeta *bin*.

El programa pot treballar amb ObjectDB de dues maneres:

- Com una base de dades encastada. En aquest cas, no cal iniciar cap servidor. És el propi programa que s'executem qui gestiona directament el fitxer del sistema operatiu on s'emmagatzemen els objectes. Ho fa a través de les crides als mètodes estàndards de JPA o JDO, implementats per la biblioteca que hem afegit al projecte. Perquè la biblioteca treballi d'aquesta manera, només cal indicar que la connexió amb la base de dades és un fitxer local. A JPA això es fa al fitxer *persistence.xml*.

- Amb un model client servidor. En aquest cas, cal posar en marxa el servidor. Quan el programa realitzi crides als mètodes de JPA o JDO, la implementació de la biblioteca farà que aquests facin peticions al servidor a través dels sòcols de xarxa (igual que passa, per exemple, amb PostgreSQL). El servidor, en rebre les peticions, realitzarà la gestió demanada sobre el fitxer que conté els objectes i retornarà els resultats al programa. Perquè la biblioteca treballi d'aquesta manera, només cal indicar correctament la cadena de connexió amb la base de dades, a més de l'usuari i la contrasenya. Novament, a JPA tot això s'indica al fitxer *persistence.xml*. Si el servidor s'estés executant en el nostre ordinador i la base de dades s'anomenés *odb*, la cadena de connexió que caldria especificar seria: objectdb://127.0.0.1:6136/odb. *6136* és el port per defecte. Si el servidor s'executés en un altre port, caldria canviar-lo per posar-hi el número correcte; igualment i si cal, poden canviar-se *127.0.0.1* per una altra adreça IP o nom de domini i *odb* per un altre nom, de manera que tots ells facin referència al servidor i la base de dades amb els quals volem treballar.

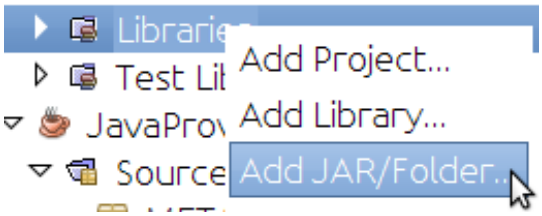
A tots dos casos, el sistema és transparent al programador. Aquest només ha d'ocupar-se de fer les crides adients als mètodes de la biblioteca, posar les anotacions necessàries i, al cas de JPA, definir correctament la unitat de persistència al fitxer *persistence.xml*. La resta de feina la fan els mètodes de la biblioteca.

3.3.1 Ús d'ObjectDB amb NetBeans i JPA

Per utilitzar ObjectDB amb NetBeans cal seguir els següents passos:

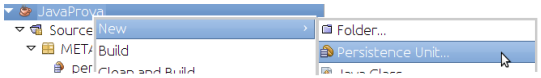
- Descomprimir el fitxer *.zip* que ens hem descarregat del web d'ObjectDB.
- Crear un projecte de Java del tipus *Java Application*.
- Afegir al projecte la biblioteca *objectdb.jar* de la següent manera: fent clic amb el botó secundari a la carpeta de biblioteques, seleccionant *Add / Jar Folder* i triant el fitxer *.jar* que conté aquesta biblioteica. El fitxer *objectdb.jar* es troba a la subcarpeta *bin* de la carpeta on s'hagi descomprimit el fitxer que conté ObjectDB. A la figura 3.1 es veu la seqüència d'opcions per afegir biblioteques al nostre projecte.

FIGURA 3.1. Afegit d'una biblioteca .jar al nostre projecte.



- Afegir al projecte una unitat de persistència. Això es fa fent clic al botó secundari del ratolí a sobre del projecte, seleccionant *New / Persistence Unit*, com es veu a la figura 3.2

FIGURA 3.2. Opció New / Persistence Unit.



- Si al menú del pas anterior no hagués aparescut *Persistence Unit* al menú, caldria seleccionar l'opció *Other...*, com es veu a la figura 3.3. A la finestra que es mostra a continuació, caldria seleccionar la categoria *Persistence* i, dins d'aquesta, el tipus de fitxer *Persistence Unit*, com es veu a la figura 3.4

FIGURA 3.3. Opció Other....

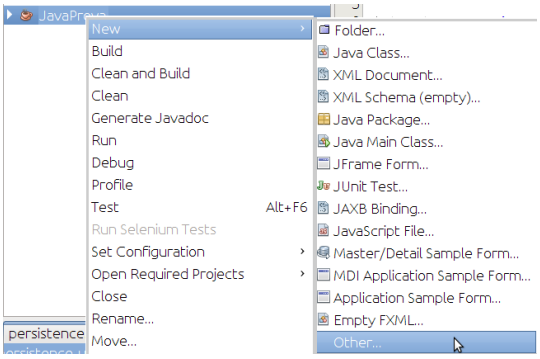
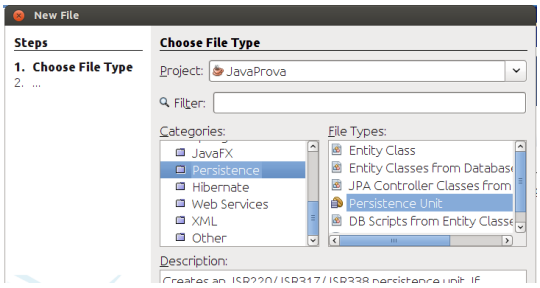


FIGURA 3.4. Selecció del tipus de fitxer Persistence Unit



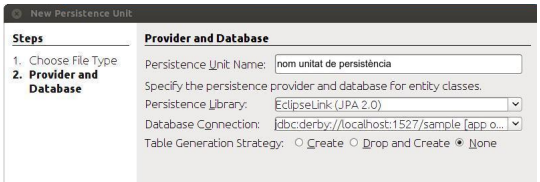
- Configurar la unitat de persistència amb els valors que s'especifiquen a continuació. A la figura :figura 3.5 es veu un possible resultat.

  – *Persistence Unit Name:*: s'hi pot posar qualsevol nom.

  – *Persistence Library:* i *Database Connection*: s'hi poden deixar els valors per defecte o qualssevols altres, ja que NetBeans no preveu la utilització d'ObjectDB. Després caldrà canviar aquests dos valors directament sobre la configuració textual de la unitat de persistència.

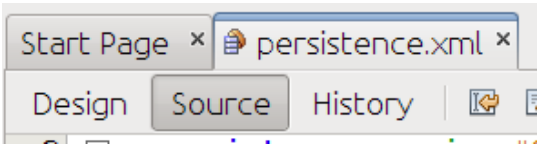
  – *Table Generation Strategy*: es pot seleccionar *None*, ja que no s'ha de crear cap taula.

FIGURA 3.5. Creació d'una unitat de persistència.



- Seleccionar la pestanya *Source* (figura 3.6) per poder modificar directament el text amb els paràmetres de la unitat de persistència que no poden modificar-se des del formulari de configuració.

FIGURA 3.6. Pestanya //Source//



- Canviar-hi

```
<provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
```

per

```
<provider>com.objectdb.jpa.Provider</provider>
```

- Posar els valors correctes a les següents línies:
  - Substituir a

```
<property name="javax.persistence.jdbc.url" value="..."/>
```

    els tres punts per:

    * Si treballem amb servidor: el nom del fitxer on seran els objectes.
    * Si treballem amb el mode encastat (és a dir, sense servidor): la cadena de connexió `objectdb://<adreçaServidor>:<port>/<baseDades>`, on, evidentment, caldrà també canviar-hi `<adreçaServidor>`, `<port>` i `<baseDades>` pels valors adients.
  - Substituir a

```
<property name="javax.persistence.jdbc.user" value="..."/>
```

    els tres punts pel nom de l'usuari.
  - Substituir a

```
<property name="javax.persistence.jdbc.password" value="..."/>
```

    els tres punts per la contrasenya.

Podeu trobar la llista sencera de les anotacions que indiquen com realitzar el mapatge objecte-relacional i que, per tant, són ignorades per ObjectDB a bit.ly/2n5DStl.

- Afegir al projecte les anotacions al codi de JPA i/o l'especificació XML necessàries per assenyalar les classes persistents i les restriccions.

- Afegir al codi font les crides als mètodes de l'API necessàries per a gestionar la persistència dels objectes de l'aplicació.

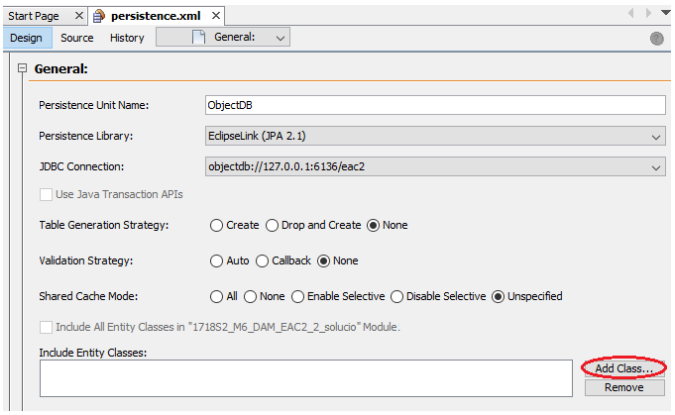
ObjectDB, en ser una base de dades orientada a objectes, no necessita les anotacions de JPA que indiquen com realitzar el mapatge d'objecte a relacional. Si en troba alguna, no dona error per compatibilitat amb l'especificació, però **la ignora**.
Algunes d'aquestes anotacions són:

- Les relacionades amb les característiques (nom, longitud...) de les taules o les columnes d'aquestes com: `@Column`, `@Lob` o `@Table`.
- Les relacionades amb la **implementació** de l'herència o les relacions entre les classes com: `@DiscriminatorColumn`, `@DiscriminatorType`, `@DiscriminatorValue`, `@InheritanceType`, `@JoinColum`, `@JoinColumns` o `@JoinTable`.

El que es diu per les anotacions és igualment vàlid per a les especificacions XML de JPA equivalents.

- Tornar a l'edició de la unitat de persistència (pestanya *Design*) i afegir al quadre *Include Entity Classes* (és al final de la finestra) les classes que volem fer persistents amb el botó *Add Class...* (apareix encerclat a la figura 3.7). En cas d'afegir-hi una classe no desitjada, pot eliminar-se amb el botó *Remove*, que és a sota del botó anterior.

Figura 3.7. Addició de classes a la unitat de persistència

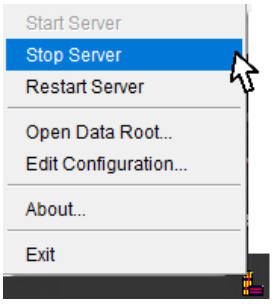


A bit.ly/2mDW7Wn trobareu la documentació oficial sobre totes les eines i utilitats de la base de dades ObjectDB.

3.3.2 Ús del servidor i el client d'ObjectDB

Per posar en marxa el **servidor** només cal executar amb la JVM el fitxer *objectdb.jar*. El fitxer *objectdb.jar* és a la carpeta *bin*. Per aturar el servidor, només cal seleccionar l'opció *Stop* a la icona que representa el procés, com es veu a la figura 3.8.

FIGURA 3.8. Opció Stop del servidor.



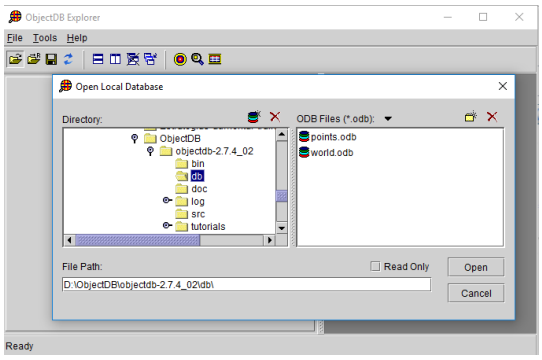
El servidor ObjectDB requereix un fitxer de configuració. Per defecte, aquest es troba a la carpeta de la instal·lació, és a dir, just a l'arrel del resultat de descomprimir el fitxer que cal descarregar-se d'Internet. En cas que no n'hi hagués cap, la pròpia biblioteca *.jar* (que no deixa de ser un fitxer comprimit) en porta un. Podeu veure'l obrint el fitxer *.jar* amb una utilitat que treballi amb fitxers comprimits. Hi trobareu, entre altres, els següents valors:

- Port:6136
- Carpeta amb les dades: subcarpeta *db* del resultat de descomprimir el fitxer que conté Objectdb.

A la carpeta *bin*, a més del fitxer *objectdb.jar*, trobareu el fitxer *explorer.jar*. En executar-lo amb la JVM, s'obre una aplicació client d'ObjectDB que permet realitzar les operacions bàsiques sobre la base de dades. En concret:

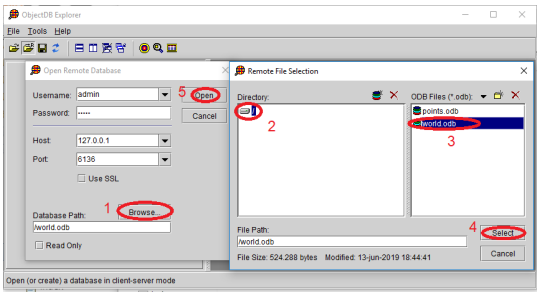
- Obrir una base de dades directament indicant la seva ubicació. Es fa clicant la icona *Open Embedded*, seleccionant-la a la finestra que s'obre. A la figura 3.9 podeu veure l'operació: la icona *Open Embedded* apareix polsada i la finestra de selecció del fitxer oberta. Al quadre gran de l'esquerra es pot seleccionar la carpeta que conté el fitxer. Al quadre de la dreta apareixen els fitxers *.odb* de la carpeta seleccionada (*.odb* és l'extensió dels fitxers que contenen les bases de dades d'Objectdb). A la part inferior hi ha un quadre de text amb la selecció actual. Un cop s'ha seleccionat el fitxer, cal clicar el botó *Open* per obrir la base de dades seleccionada.

FIGURA 3.9. Obrir un fitxer .odb.



- Obrir una base de dades a través d'una connexió amb el servidor. Lògicament, abans d'utilitzar aquesta opció hem hagut d'arrencar el servidor. En aquest cas, per obrir la base de dades es clica la icona *Open C/S Connection*, a la finestra que s'obre s'escriu l'usuari, la contrasenya (per defecte, *admin* tots dos), el host (adreça *IP* o nom *DNS*), el port (recordeu: per defecte 6136) i el nom i camí de la base de dades, agafant com a arrel la carpeta amb les dades (a la màquina on s'executa el servidor). Igual que al cas anterior, un cop introduïda aquesta informació, cal clicar el botó *Open* per obrir la base de dades seleccionada. El nom i el camí de la base de dades poden introduir-se directament al quadre de text *Database Path* o en una finestra de selecció que apareix si fem clic al botó *Browse...*. Un cop seleccionada, cal fer clic al botó *Select*. A la figura 3.10 podeu veure l'operació amb la seqüència de passos numerada. Es pot veure també que la icona *Open C/S Connection* i el botó *Browse...* apareixen polsats; la finestra de selecció que es mostra s'ha obert en fer clic al botó *Browse...*. En aquesta finestra es pot seleccionar la carpeta -al quadre gran de l'esquerra-, el fitxer -al quadre gran de la dreta- i clicar el botó *Select* per confirmar la selecció. En aquest cas el servidor s'està executant a la màquina local, però el mecanisme seria el mateix si el servidor s'executés en una altra màquina. En aquest segon cas, el servidor seria l'única via per obrir les bases de dades que conté.

FIGURA 3.10. Obrir una connexió amb el servidor.



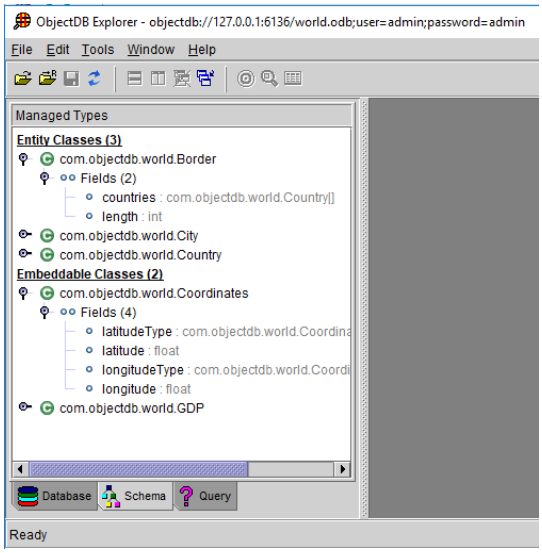
- Tancar una connexió. Pot fer-se bé amb l'opció *File / Close connection* o bé tancant el programa.

- Obrir una connexió recent o una base de dades local recent. Es fa, respectivament, amb les opcions *File / Recent Connections* o *File / Local Databases*.

Les dades de les imatges que en tenen corresponen a la base de dades d'exemple *world* que és inclosa a ObjectDB.

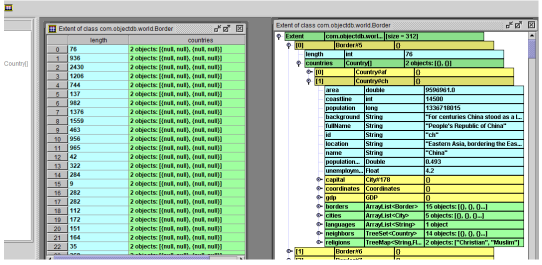
- Visualitzar l'estructura de les dades. Un cop oberta una base de dades directament des del programa o a través del servidor, la columna esquerra queda amb la pestanya *Schema* activada i s'omple amb la informació *Managed Types*, que mostra l'estructura de les dades. Concretamnet i com es veu a la figura 3.11, es poden consultar: les classes els objectes de les quals es fan persistents (*Entity Classes*), els seus membres i, si n'hi ha, les classes encastades (*Embeddable Classes*), és a dir, que han de formar part d'altres classes i, també, els membres d'aquestes.

FIGURA 3.11. Estructura de les dades.



- Visualitzar les dades. Novament a la pestanya *Schema*, es pot activar el botó secundari del ratolí a sobre d'una de les *Entity classes* i seleccionar l'opció *Open Tree Window* o *Open Table Window*. La primera obre una finestra amb els objectes de la classe seleccionada i els presenta en forma d'arbre. La segona obre una finestra on es veuen els objectes en forma de taula. Pot haver més d'una finestra oberta a la vegada. A la :figura 3.12 podeu veure els objectes de la classe *Border* a través d'aquests dos tipus de finestra. En la finestra que mostra els objectes en forma de taula, a cada fila es mostren directament els membres de cada objecte. Si un d'ells és un objecte i volem veure'l en detall, només cal fer doble clic a sobre del requadre que el representa i s'obrirà una nova finestra en forma de taula amb els membres d'aquest objecte. En la finestra que mostra els objectes en forma d'arbre, cada fila representa un objecte. Per veure els seus membre, només cal fer doble clic a sobre de la línia que el respresenta i apareixerà un nou nivell de detall amb una línia per a cada membre; novament, si un d'aquests membres fa referència a un objecte, un doble clic a sobre ens mostrarà els seus membres.

FIGURA 3.12. Visualització dels objectes de la classe Border.



La finestra de l'esquerra mostra els objectes en forma tabular mentre que la de la dreta els mostra en forma d'arbre.

- Modificar dades primitives d'un objecte. Un cop visualitzem la dada que es vol modificar, només cal fer-hi doble clic a sobre i el programa ens permetrà editar-la directament. S'acaba l'edició de la dada picant la tecla de retorn.

- Modificar la referència a un objecte. Un cop visualitzem la referència que es vol modificar, cal obrir el menú contextual a sobre del requadre que el respresenta i triar l'opció *Set Reference...*. S'obrirà un nou menú, com mostra la figura figura 3.13, amb tres opcions:
  - *Set to Null*: posa la referència a *null*.
  - *Set to Existing Entity...*: permet fer referència a un altre objecte de la base de dades; en seleccionar aquesta opció, s'obre una finestra on hem d'identificar l'objecte al qual volem referenciar. Aquesta finestra, com es mostra a la figura 3.14, només té un quadre de text amb el nom de la classe adient a la referència seguida del signe #. A continuació, hem de posar la posició que ocupa l'objecte a referenciar a la base de dades. Aquesta posició es pot veure obrint una finestra que mostri tots els objectes d'aquesta classe que hi ha a la base de dades. En aquest quadre de text podem, a més de posar la posició de l'objecte, modificar el contingut que surt per defecte, encara que no sol ser necessari.
  - *Set to New Object...*: s'obre una finestra amb la classe de l'objecte que volem crear; en acceptar, es crea un nou objecte al qual apuntarà la referència que estem modificant. El nou objecte creat té totes les dades amb el valor per defecte que assigna Java a cada tipus. Igual que passava amb l'opció anterior, el quadre de text pot ser modificat abans d'acceptar, encara que normalment no cal fer-ho; una excepció és quan l'objecte és una llista: en aquest cas la classe per defecte és `java.util.List`; com és una interfície, cal substitiur-la per una classe que la implementi com, per exemple, `java.util.ArrayList<>`.

FIGURA 3.13. Opcions per modificar una referència.

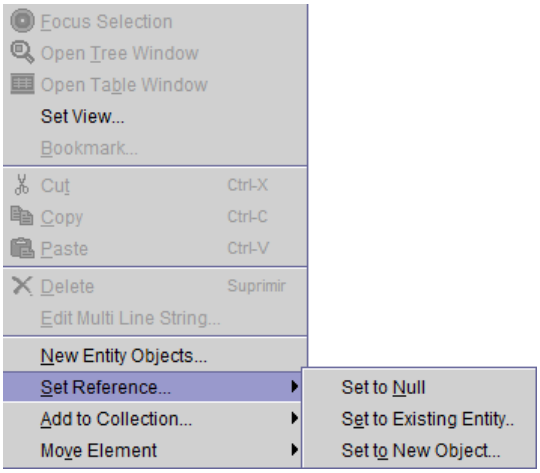
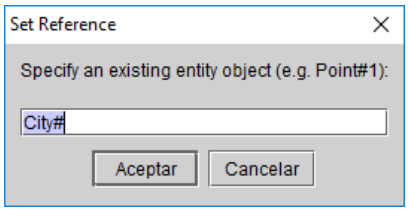


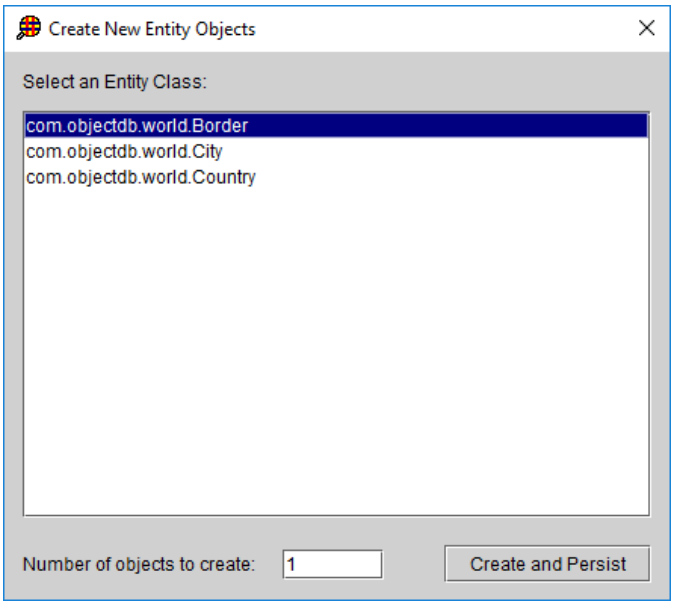
FIGURA 3.14. Referenciar una entitat de la base de dades.



- Afegir elements a una llista. Un cop visualitzem la llista a la qual volem afegir elements, cal obrir el menú contextual a sobre del requadre que el respresenta i triar l'opció *Add to Collection...*. S'obrirà un nou menú amb les opcions: *Add Null*, *Add Existing Entity...* i *Add New Object...*. Aquestes opcions afegeixen al final de la llista, respectivament, un null, un objecte ja existent a la base de dades o un nou objecte. El funcionament és similar, respectivament, al de les opcions *Set to Null*, *Set to Existing Entity...* i *Set to New Object...* de l'opció *Set Reference...*.

- Afegir un nou objecte. Cal triar l'opció de menú *Edit* / *New Entity Objects..* i, a la finestra que ens surt (figura 3.15), triar la classe de l'objecte que es vol crear i el nombre d'objectes d'aquesta classe que es volen crear.

FIGURA 3.15. Elecció de la classe de l'objecte de nova creació.



- Esborrar un objecte o eliminar-lo d'una llista. Per fer-ho, cal visualitzar l'objecte que volem esborrar, obrir el menú contextual a sobre d'ell i seleccionar l'opció *Delete*. Si estem situats en un element d'una llista, aquest s'elimina de la llista, però no s'esborra de la base de dades; si estem situats en un objecte de la base de dades, l'objecte s'esborra d'aquesta.

- Transsaccions: quan s'obre una base de dades i es fa la primera actualització dels seus objectes, s'inicia una transacció; aquesta transacció pot acabar amb una validació (*commit*) o un retrocés (*rollback*), segons executem, respectivament, les opcions del menú *File* / *Save Changes* o *File* / *Discard Changes*. A més, cada cop que tanquem una base de dades o el programa, aquest demana si volem desar els canvis (és a dir, fer un *commit*) o no (és a dir, fer un *rollback*).

- Executar una ordre JPQL sobre la base de dades. Cal seleccionar la pestanya *Query*. Es mostrarà la secció *Query Execution*. Per executar la instrucció només cal escriure-la al quadre *Query Text (JPQL or JDOQL):* i clicar el botó *Execute*. Com es veu a la figura 3.16, al quadre gran de la dreta apareix

el resultat i al quadre *Log* de l'esquerra apareix informació sobre l'execució de la instrucció.

FIGURA 3.16. Consulta del nom de les ciutats amb més de 10 milions d'habitants.