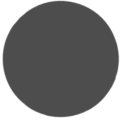
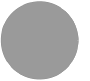



Accés a dades

CFGS.DAM.M06/0.13

Desenvolupament d'aplicacions multiplataforma

Generalitat de Catalunya
Departament d'Educació

ioc
institut obert de catalunya

Aquesta col·lecció ha estat dissenyada i coordinada des de l'Institut Obert de Catalunya.

Coordinació de continguts
Verònica Mascarós Álvarez

Redacció de continguts
Josep Cañellas Bornas
Isidre Guixà Miranda

Imatge de coberta
FellowCreative

Primera edició: Febrer 2013
© Departament d'Ensenyament
Dipòsit legal: B. 29430-2013

Llicenciat Creative Commons BY-NC-SA. (Reconeixement-No comercial-Compartir amb la mateixa llicència 3.0 Espanya).

Podeu veure el text legal complet a

http://creativecommons.org/licenses/by-nc-sa/3.0/es/legalcode.ca

Introducció

En informàtica, a partir de l'objectiu de tractar de forma automatitzada la informació, sorgeix una necessitat: fer que les dades romanguin més enllà de l'execució del procés o l'aplicació que les ha creat. Les aplicacions han de poder guardar i recuperar dades, i per tant, les dades han de ser persistents.

Aquestes dades poden persistir sobre diferents sistemes de gestió i maneig de dades, com ara els fitxers, les bases de dades (ja siguin relacionals, relacionals orientades a objectes o natives) o fins i tot altres sistemes de processament, i no només en dispositius d'emmagatzemament.

Veurem que les aplicacions que tracten amb dades s'enfronten a la dificultat de com desar i accedir de nou a les dades que estan en distints sistemes d'informació. Aquests tipus aplicacions que treballen amb dades que han de perdurar ens creen l'obligació d'utilitzar i conèixer diferents tècniques d'accés per cadascun dels sistemes de gestió que s'utilitzin.

Aquest mòdul té com a finalitat inicial la d'aproximar-nos conceptualment al problema de la persistència de les dades en les aplicacions, analitzant distintes situacions on cal que aquestes siguin guardades i recuperades considerant diversos plantejaments.

En la unitat formativa "Persistència en fitxers" es dóna una primera aproximació a la persistència, basant-nos en el mètode de guardar les dades en un dispositiu no volàtil com pot ser un fitxer. Es tractarà com gestionar el flux de dades, cap als fitxers o des d'aquests, així com les classes associades a les operacions de gestió de fitxers, les formes d'accés i les operacions bàsiques.

En el moment actual, els SGBD són part fonamental en els sistemes informàtics, ja que 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. Segons la tècnica que empren per emmagatzemar la informació, apareixen diversos tipus de bases de dades: jeràrquiques, relacionals, objecte-relacionals, orientades a objectes i XML natives. Per tant, els desenvolupadors de programari han de conèixer com accedir a les dades emmagatzemades en els diversos sistemes existents; aquests conceptes es tractaran al llarg de les unitats formatives "Persistència en BDR-BDOR-BDOO" i "Persistència en BD natives XML".

L'evolució del programari, però, no avança de forma paral·lela a la dels SGBD i sovint es troba un desfase entre els paradigmes usats en desenvolupament de programari i els usats per les eines d'emmagatzematge. En la unitat formativa "Persistència en BDR-BDOR-BDOO" estudiareu la diferència existent entre la Programació Orientada a l'Objecte i els SGBD relacionals. Veureu també l'establiment de connexions i l'evolució dels connectors a bases de dades, sobretot centrant-nos en els connectors usats pel llenguatge JAVA anomenats drivers JDBC.

En la unitat formativa "Persistència en BD natives XML" es tractarà l'accés a les dades emmagatzemades en SGBD XML natives. És a dir, aprendreu com desenvolupar programes que accedeixin a BD XML natives, que són bases de dades que emmagatzemen documents en format XML.

En la unitat formativa "Components d'accés a dades" veureu el concepte de component, com una tècnica de divisió que permet crear petites peces independents que acabin formant part de l'aplicació final. A més, aquest tipus de creació d'aplicacions permet una construcció en paral·lel de múltiples components, per tal que sigui més fàcil automatitzar i rendibilitzar els processos de construcció de les diverses parts, i fa que sigui possible reutilitzar el mateix tipus de peces en diverses aplicacions. Després de veure el concepte de component i analitzar-ne les característiques, la discussió s'encaminarà cap els components de persistència. Es tracta de components específics encarregats de l'emmagatzematge i recuperació de les dades de les aplicacions. Per acabar, es posaran en pràctica els conceptes estudiats fins els moment.

Resultats d'aprenentatge

En finalitzar aquest mòdul l'alumne/a:

Persistència en fitxers

1. Desenvolupa aplicacions que gestionen informació emmagatzemada en fitxers identificant el camp d'aplicació dels mateixos i utilitzant classes específiques.

Persistència en BDR-BDOR-BDOO

1. Desenvolupa aplicacions que gestionen informació emmagatzemada en bases de dades relacionals identificant i utilitzant mecanismes de connexió.
2. Gestiona la persistència de les dades identificant eines de mapatge objecte relacional (ORM) i desenvolupant aplicacions que les utilitzen.
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.

Persistència en BD natives XML

1. Desenvolupa aplicacions que gestionen la informació emmagatzemada en bases de dades natives XML avaluant i utilitzant classes específiques.

Components d'accés a dades

1. Programa components d'accés a dades identificant les característiques que ha de posseir un component i utilitzant eines de desenvolupament.

Continguts

Persistència en fitxers

Unitat 1

Persistència en fitxers

1. Gestió del sistema de fitxers
2. Gestió del contingut dels fitxers
3. Persistència d'objectes en fitxers

Persistència en BDR-BDOR-BDOO

Unitat 2

Persistència en BDR-BDOR-BDOO

1. Gestió de connectors
2. Eines de mapatge d'objecte relacional (ORM)
3. Bases de dades d'objecte relacionals i orientades a objectes

Persistència en BD natives XML

Unitat 3

Persistència en BD natives XML

1. BD-XMLnatives. API Java específica del SGBD
2. API Java estàndards per a BD-XML natives

Components d'accés a dades

Unitat 4

Components d'accés a dades

1. Aproximació al concepte de component de programari
2. Exemples d'implementació de components de persistència

Persistència en fitxers

Josep Cañellas Bornas

Índex

Introducció 5

Resultats d'aprenentatge 9

1 Gestió del sistema de fitxers 11
1.1 La classe File. Generalitats 11
1.2 Funcionalitat de la classe File a partir d'un cas pràctic 13
1.2.1 Obtenció d'informació bàsica 13
1.2.2 Obtenció d'informació detallada 19
1.2.3 Filtratge, ordenació i classificació de fitxers 21
1.2.4 Modificació del sistema de fitxers 23

2 Gestió del contingut dels fitxers 29
2.1 Magatzems i fluxos de dades 29
2.1.1 Seriació i flux de dades 30
2.1.2 Fluxos i tipus de dades 31
2.2 Manipulació dels fluxos 33
2.2.1 Fluxos orientats a Bytes 33
2.2.2 Fluxos orientats a caràcters 39
2.2.3 Implementació d'utilitats 42
2.3 Fluxos eficients: Channels i Buffers 46
2.3.1 Conceptes 47
2.3.2 Instanciació d'un Channel 47
2.3.3 Instanciació d'un Buffer 48
2.3.4 Buffers de bytes 49
2.3.5 Buffers de tipus específics 50
2.3.6 Treball combinat de Buffers i Channels 51
2.3.7 Transferència directa 53
2.4 Estudi pràctic de la funcionalitat d'entrades i sortides de dades bàsiques 54
2.4.1 Còpia i trasllat de fitxers 55
2.4.2 Edició de fitxers de text 57

3 Persistència d'objectes en fitxers 61
3.1 Seriació d'objectes 61
3.1.1 Exemple d'implementació 62
3.2 Fitxers amb formats binaris específics 66
3.3 Fitxers amb formats XML 73
3.4 'Parser' o analitzador XML 76
3.4.1 Analitzadors seqüencials 76
3.4.2 Analitzadors jeràrquics 76
3.4.3 Implementació d'exemple 81
3.5 Binding 86
3.5.1 Configuració amb anotacions 87
3.5.2 Generació automàtica del model de dades 88

3.5.3 Ús de JAXB amb un model de disseny propi 93

Introducció

En aquesta unitat, anomenada "Persistència en fitxers", s'estudia en detall la gestió de fitxers i del seu contingut. La unitat s'estructura en tres grans apartats: la gestió dels sistemes de fitxers, la gestió del contingut dels fitxers i l'estudi de diferents tècniques que permetin la persistència basada en fitxers dels objectes d'una aplicació.

A la primera part es dóna una visió general del que cal entendre com a sistema de fitxers. Es fa palès que cada sistema operatiu pot organitzar els seus fitxers de diferent manera i fer servir diferents sistemes per identificar la seva ubicació. La conseqüència d'això és que no existeix una forma estàndard de referenciar un fitxer, sinó que aquesta depèn de cada sistema operatiu.

Per tal d'apaivagar aquesta dependència Java ha desenvolupat la classe `File`, la qual permet abstraure la referència d'un fitxer amb independència de la notació pròpia que el sistema operatiu faci servir.

Les instàncies de tipus `File` poden representar tant un fitxer com un directori (o carpeta). En qualsevol cas, ens permetran interrogar-les per obtenir informació de l'element representat. Així, per exemple, podem saber si es tracta d'un fitxer o d'un directori, o bé obtenir informació del seu nom en qualsevol de les seves formes (absoluta o relativa), la seva mida, la data de creació, etc.

`File` no disposa de cap utilitat per obtenir una llista ordenada. L'ordenació s'aconseguirà fent servir les utilitats de Java a la classe System, per ordenar col·leccions usant un Comparator.

En l'apartat *Gestió de contingut dels fitxers* s'ofereixen diferents aproximacions al concepte de fitxers. Des d'un punt de vista de la informació, els fitxers són magatzems de dades estructurades, però des del punt de vista de l'aplicació, els fitxers es poden considerar recursos d'intercanvi de dades entre dos sistemes, l'un volàtil (la memòria RAM) i l'altre permanent (els dispositius d'emmagatzematge).

Des del primer punt de vista es fa palès que l'estructura de dades contenint la informació haurà d'adaptar-se a les limitacions dels sistemes d'emmagatzematge, produint-se sovint un desfasament entre l'estructura planificada per suportar la informació i l'estructura adaptada que acabarà utilitzant-se per emmagatzemar-la. Així, distingim entre l'estructura lògica, propera a la representació mental planificada, i l'estructura física o forma com es guarda realment la informació en el dispositiu.

Des del segon punt de vista apareix el concepte de flux de dades, entès com a transferència d'informació des dels dispositius persistents a la memòria de treball de l'aplicació o a l'inrevés.

La direcció de la transferència ens permet classificar els fluxos en fluxos d'entrada (del magatzem a l'aplicació) i fluxos de sortida (de l'aplicació al magatzem).

El concepte de flux és una abstracció que permet tractar totes les dades com a seqüències de Bytes. Ara bé, la complexitat de la representació seriada dels caràcters ens obliga a considerar de forma especial els fluxos destinats a treballar amb caràcters.

Un cop estudiats el fluxos, s'ofereix una àmplia visió del nou paquet d'entrada i sortida de Java anomenat nio.S'estudia en detall la classe `FileChannel` i la classe `Buffer`, veient com es poden fer treballar conjuntament per obtenir una gran eficiència, sense perdre, però, flexibilitat en el tractament de les dades, siguin del tipus que siguin.

En l'apartat *Persistència d'objectes en fitxers* veurem diverses formes d'emmagatzemar objectes. Bàsicament, estudiarem quatre tècniques. En primer lloc, veurem la persistència basada en la seriació per defecte que JAVA fa de qualsevol objecte. Aquesta tècnica aconsegueix emmagatzemar els objectes en una seqüència de bytes que emmagatzemarem en fitxers fent servir fluxos de bytes.

Es tracta d'una tècnica molt poc costosa per al programador, però que presenta moltes limitacions a l'hora de reutilitzar el format en altres llenguatges diferents a JAVA. També presenta limitacions de versionat quan es fan modificacions a les classes persistents i, per tant, malgrat que útil, no és una opció de persistència gaire estesa.

En segon lloc, estudiarem la creació d'un format binari basat en alguna codificació definida específicament en el codi. Es tracta d'un sistema costós per al programador, ja que ha de codificar el procés de persistència gairebé de forma sencera. Aquest sistema presenta també un desavantatge. Les dades binàries no es representen totes igual en les diferents arquitectures d'ordinadors. És a dir, no hi ha garantia que el format aconseguit es pugui llegir en qualsevol plataforma ni tan sols usant el llenguatge JAVA.

Les altres dues tècniques permeten fer persistència basada en formats XML. L'avantatge de fer servir formats XML és que aquests poden ser llegits per qualsevol llenguatge en qualsevol sistema operatiu i no varien en funció de l'arquitectura de l'ordinador. Presenten, això sí, el problema de les múltiples codificacions dels caràcters, però un cop fixada la codificació que farem servir, el format pot exportar-se a qualsevol plataforma i llegir-se usant qualsevol llenguatge de programació sense cap mena de problema.

La primera tècnica XML usada consistirà en la codificació específica d'utilitats que ens facilitin la transformació dels objectes en dades de text estructurades per mitjà d'etiquetes. Igual que la tècnica anterior, és també un sistema costós per al programador. Tot i això, és molt còmoda de fer servir en persistències que requereixin emmagatzemar poques dades o en persistències especials que requereixin formats complexos i difícils d'automatitzar, perquè el programador té tot el control.

La segona tècnica XML s'anomena *binding*. Consisteix en un conjunt d'utilitats que permeten automatitzar la persistència dels objectes en format XML a partir de la vinculació de les classes de l'aplicació a esquemes XML específics. La vinculació s'aconsegueix configurant "mapes" que ajudin a automatitzar la representació XML que cada classe persistent haurà de tenir. La biblioteca usada en aquesta tècnica és JAXB, un paquet estàndard incorporat al J2SE a partir de la versió 5.0.

Resultats d'aprenentatge

En acabar aquesta unitat, l'alumne:

1. Desenvolupa aplicacions que gestionen informació emmagatzemada en fitxers identificant el camp d'aplicació dels mateixos i utilitzant classes específiques.

- Utilitza classes per a la gestió de fitxers i directoris.
- Valora els avantatges i els inconvenients de les diferents formes d'accés
- Utilitza classes per recuperar informació emmagatzemada en un fitxer XML.
- Utilitza classes per emmagatzemar informació en un fitxer XML.
- Utilitza classes per convertir a un altre format informació continguda en un fitxer XML.
- Preveu i gestiona les excepcions.
- Prova i documenta les aplicacions desenvolupades.

1. Gestió del sistema de fitxers

En els sistemes informàtics actuals, en els quals un sol ordinador pot tenir ben bé més d'un milió de fitxers, resulta imprescindible un sistema que permeti una gestió eficaç de localització, de manera que els usuaris puguem moure'ns còmodament entre tants arxius. La majoria de sistemes de fitxers han incorporat contenidors jerarquitzats que actuen a mode de directoris facilitant la classificació, la identificació i localització dels arxius. Els directoris s'han acabat popularitzant sota la versió gràfica de carpetes.

Hem de tenir en compte, a més, que la necessitat desmesurada d'espai d'emmagatzematge ha dut els SO a treballar amb una gran quantitat de dispositius i a permetre l'accés remot a sistemes de fitxers aliens, distribuïts per la xarxa.

Per poder gestionar tanta varietat de sistemes de fitxers, alguns sistemes operatius com Linux o Unixprenen l'estratègia d'unificar tots els sistemes en un de sòl, per tal d'aconseguir una forma d'accés unificada i amb una única jerarquia que faciliti la referència a qualsevol dels seus components, amb independència del sistema de fitxers en què es trobin realment ubicats. En Linux, sigui quin sigui el dispositiu o el sistema remot real on s'emmagatzemarà l'arxiu, la ruta tindrà sempre la mateixa forma.

```
/cami/on/es/troba/el/Fitxer.txt
```

Per contra, l'estratègia d'altres SO com Windows passa per mantenir ben diferenciats cada un dels sistemes i dispositius on tingui accés. Per distingir el sistema al qual es vol fer referència, Windows usa una denominació específica que incorpora a la ruta de l'element a referenciar. Tot i que Microsoft ha apostat clarament per la convenció UNC, l'evolució d'aquest sistema operatiu, que té com a origen l'MS-DOS, ha fet que coexisteixi amb una altra convenció també molt estesa. Ens referim a la identificació dels dispositius i sistemes amb una lletra de l'alfabet seguida de dos punts. A continuació il·lustrem amb un exemple ambdues convencions. Hem marcat en negreta la denominació específica que identifica el sistema de fitxers o dispositiu específic:

```
F:\cami\on\es\troba\el\Fitxer.txt
\\Servidor7\cami\on\es\troba\el\Fitxer.txt
```

1.1 La classe File. Generalitats

En Java, per gestionar el sistema de fitxers s'utilitza bàsicament la classe 'File'. És una classe que s'ha d'entendre com una referència a la ruta o localització de fitxers del sistema. **NO representa el contingut**de cap fitxer, sinó la ruta del sistema on

es localitzen. Com que es tracta d'una ruta, la classe pot representar tant fitxers com carpetes o directoris.

> Els objectes de la classe `File` representen rutes del Sistema de Fitxers.

Si fem servir una classe per representar rutes, s'aconsegueix una total independència respecte de la notació que cada sistema operatiu utilitza per descriure-les. Recordem que Java és un llenguatge multiplataforma i, per tant, pot donar-se el cas que haguem de fer una aplicació desconeixent el SO on acabarà executant-se.

L'estratègia usada per cada SO no afecta la funcionalitat de la classe `File`, ja que aquesta, en col·laboració amb la màquina virtual, adaptarà les crides al SO amfitrió de forma transparent al programador, és a dir, sense necessitat que el programador hagi d'indicar o configurar res.

Anomenem *instàncies* d'una classe els objectes creats durant l'execució d'una aplicació. Per tant, *objecte* i *instància* es poden usar com a sinònims.

Les instàncies de la classe `File` es troben estretament vinculades a la ruta amb la qual s'han creat. Això significa que les instàncies durant tot el seu cicle de vida només representaran una única ruta, la que se'ls va associar en el moment de la creació. La classe `File` **no disposa** de cap mètode ni mecanisme per modificar la ruta associada. En cas de necessitar noves rutes, caldrà sempre crear una nova instància i no serà possible reutilitzar les ja creades vinculant-les a rutes diferents.

Vegeu en la secció "Annexos" l'annex "Parametrització d'aplicacions. Tècniques de configuració", on s'expliquen diverses tècniques de parametrització d'aplicacions.

En implementacions tant properes al SO, els programadors de Java han de fer un esforç per independitzar les aplicacions implementades de les plataformes on s'executaran. Caldrà, doncs, anar amb cura, fent servir tècniques de parametrització que evitin escriure les rutes directament al codi, de manera que en traslladar les aplicacions de plataforma només calgui modificar les rutes en el sistema de configuració.

La classe `File` encapsula pràcticament tota la funcionalitat necessària per gestionar un sistema de fitxers organitzat en arbre de directoris. És una gestió completa que inclou:

1. Funcions de manipulació i consulta de la pròpia estructura jeràrquica (creació, eliminació, obtenció de la ubicació, etc. de fitxers o carpetes)
2. Funcions de manipulació i consulta de les característiques particulars dels elements (noms, mida o capacitat, etc.)
3. Funcions de manipulació i consulta d'atributs específics de cada sistema operatiu i que, per tant, només serà funcional si el sistema operatiu amfitrió suporta també la funcionalitat. Ens referim, per exemple, als permisos d'escriptura, d'execució, atributs d'ocultació, etc.

Anem ara a veure amb més detall la funcionalitat de la classe `File`, així com la sintaxi dels seus mètodes. Però per tal de fer més amena la lectura, i fugir d'explicacions abstractes o descontextualitzades, centrarem el relat en la implementació d'una aplicació real, a mode d'exemple, que posi en pràctica la utilitat de la classe estudiada.

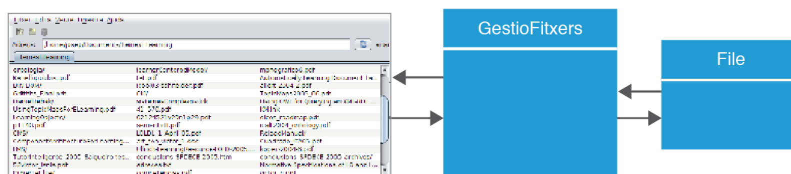
1.2 Funcionalitat de la classe File a partir d'un cas pràctic

L'exemple que ens ajudarà a familiaritzar-nos amb la classe `File` consistirà en implementar un senzill explorador de fitxers semblant al Nautilus de Linux o a l'Explorador de fitxers de Windows.

A l'annex "Biblioteques del mòdul" de la secció "Annexos", trobareu informació de les biblioteques que necessitareu fer servir en aquesta unitat, així com del lloc on podeu obtenir-les.

Cal dir, però, que l'estudi de les interfícies gràfiques necessàries per implementar una aplicació d'aquest estil cau totalment fora dels objectius d'aquest mòdul, per això no implementarem la interfície d'usuari de l'aplicació, sinó que n'usarem una d'existent i ja implementada. Aquesta delega tota la funcionalitat no gràfica a una instància de la interfície `GestioFitxers`.

FIGURA 1.1. Esquema de l'arquitectura de l'exemple que usarem per il·lustrar la funcionalitat de la classe "File"



Només haurem d'implementar `GestioFitxers`codificant una classe que fent ús d'objectes `Files`'adapti als requeriments de la interfície. Com es pot veure a l'arquitectura (figura 1.1), la implementació de `GestorFitxers` pren el paper d'adaptador o intermediari entre la interfície gràfica i les operacions de la classe `File`.

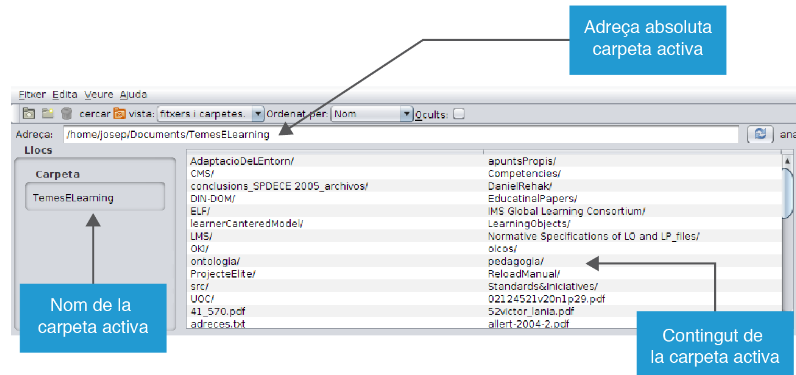
1.2.1 Obtenció d'informació bàsica

La interfície gràfica necessita mostrar la ruta absoluta de la carpeta activa, el nom de la mateixa i el seu contingut (vegeu la figura 1.2). La classe `File`ens ajudarà a recopilar aquesta informació, ja que disposa de tres mètodes per obtenir informació sobre el nom del fitxer o carpeta. El mètode `getName` obté el nom pròpiament dit de l'objecte `File`. És a dir, el nom relatiu que l'element tingui dins la carpeta on es troba contingut.

Els altres dos mètodes (`getAbsolutePath`i `getCanonicalPath`) permeten ambdós obtenir el nom absolut de l'element. Es tracta d'una cadena amb la ruta sencera des de l'arrel d'acord amb la notació emprada pel sistema operatiu amfitrió. Hi ha, però, sistemes operatius que permeten expressar rutes absolutes de manera redundant. Així, per exemple, en Linux la ruta */home/usuari/../../home/usuari/documents/../documents/arxiu.txt* és una forma redundant de

la ruta */home/usuari/documents/arxiu.txt*, ja que apunten al mateix fitxer. Si una instància de File s'ha creat usant una ruta redundant, aquesta es manté al llarg de tot el cicle de vida de la instància. El mètode `getAbsolutePath` retorna el valor absolut de la ruta però sense eliminar les redundàncies, cosa que pot arribar a complicar la comparació literal de dues rutes redundants. Per simplificar aquestes comparacions, la classe `File` disposa de `getCanonicalPath`, un mètode que retorna la ruta absoluta en la versió més simple.

Figura 1.2. Aspecte de la interfície gràfica de l'aplicació en la qual es destaca quina informació del sistema de fitxers es mostra.



Cal destacar que la ruta canònica és una ruta calculada dinàmicament a partir de la ruta original. Per tant, en cas que aquesta tingui errors de notació, no serà possible obtenir el valor canònic, sinó que es produirà una excepció avisant-nos de l'error.

A efectes del nostre exemple, no ens caldrà realitzar cap comparació literal entre rutes absolutes. Escollirem el mètode `getAbsolutePath` en detriment de `getCanonicalPath`, ja que malgrat l'ús d'exempcions és absolutament recomanable per aconseguir aplicacions robustes, la seva presència en el codi dificulta lleugerament la llegibilitat. És per això que hem optat per usar rutes redundants en el nostre gestor.

Quan un objecte de tipus `File` sigui una carpeta, el mètode `list` ens retorna un objecte de tipus Array amb el nom de tots els fitxers i carpetes que contingui. Si precisem informació addicional de cada element contingut, disposem del mètode `listFiles`, que en comptes de retornar els noms dels elements continguts ens retorna in *Array* d'objectes `File` vinculats a cada un dels elements.

Ja podem començar a codificar la classe que implementi `GestorFitxers`, que a partir d'ara anomenarem `GestorFitxersImpl`. Serà necessari mantenir una instància de `File`vinculada a la carpeta activa de la que obtinguem les dades a mostrar en la interfície gràfica i sobre la que recaurà qualsevol operació de fitxers que demandem.

D'altra banda, hem de tenir en compte que les interfícies gràfiques necessiten refrescar les dades força sovint. Per tal de no estar calculant el contingut de les carpetes cada vegada que refresquem, prendrem la decisió de mantenir emmagatzemat el contingut en una matriu d'objectes que caldrà anar sincronitzant només quan es produeixin canvis.

Començarem, doncs, afegint a la classe `GestioFitxersImpl` dos atributs que anomenarem `contingut`i `carpetaDeTreball`.

```
public class GestioFitxersImpl implements GestioFitxers{
    private Object[][] contingut;
    private File carpetaDeTreball=null;
    ...
```

El constructor inicialitzarà l'atribut `carpetaDeTreball` amb l'arrel del primer dels sistemes de fitxers disponibles i actualitzarà el valor de `contingut`. Per a la inicialització de la carpeta de treball, usarem un mètode *static* de `File` anomenat `listRoots` que retorna un vector d'objectes de tipus `File`, cada un d'ells vinculat a l'arrel d'un dels sistemes de fitxers disponible des del SO amfitrió.

Per gestionar l'actualització del contingut crearem un mètode que anomenarem *actualitza*. És preferible modularitzar l'actualització del contingut, ja que a més del constructor, altres operacions provocaran també canvis en el contingut i requeriran d'actualització.

En sistemes operatius com Unix o Linux, `File.listRoots()` retornarà un únic element, l'arrel de sistema de fitxers (/).

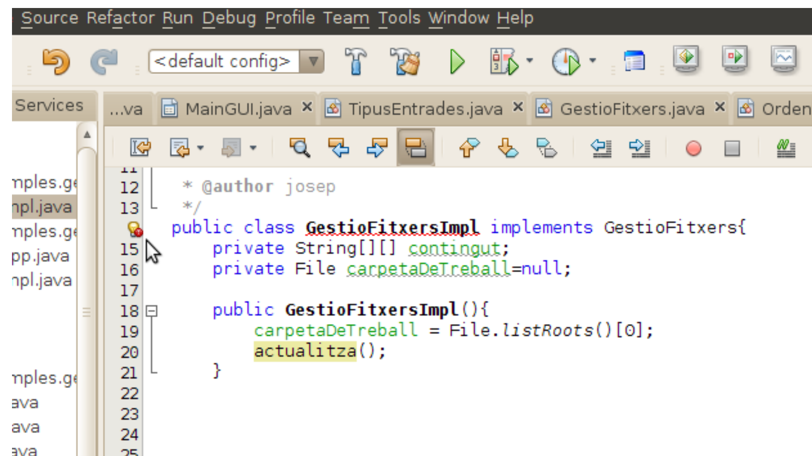
```
public GestioFitxersImpl(){
    carpetaDeTreball = File.listRoots()[0];
    actualitza();
}
```

A continuació, crearem l'esquelet de la classe d'acord amb la interfície `GestioFitxers`que implementa, per tal de poder anar provant els mètodes a mida que els anem codificant sense necessitat d'esperar a tenir-los tots implementats. La creació de l'esquelet consistirà en definir tots el mètodes de la interfície amb una única instrucció que llanci una excepció de tipus `UnsupportedOperationException`. Cada mètode hauria de presentar un aspecte semblant a:

```
@Override
public void amunt() {
    throw new UnsupportedOperationException("Not supported yet.");
}
```

Podem crear manualment l'esquelet, o bé aprofitar la utilitat de propostes de l'IDE que s'activa quan detecta algun error ben definit (vegeu figura 1.3). Immediatament després d'afegir ***implements*** `GestioFitxers` a la declaració de la classe, observarem una bombeta al marge esquerre de l'editor. En clicar damunt la icona, apareixerà la proposta *implements all abstract methods*. Si acceptem la proposta, NetBeans ens crearà l'esquelet de la classe.

Figura 1.3. Figura que il·lustra l'aspecte de l'IDE en detectar un error ben definit



L'IDE mostra la bombeta del marge esquerre i en clicar-la, ens mostrarà una o més propostes que solucionarien el problema detectat

La bombeta que surt al marge dret d'una línia de codi (vegeu la figura 1.3) indica que NetBeans ens vol fer alguna proposta per eliminar algun error.

Si volem realitzar una primera prova, necessitarem almenys la ruta absoluta de la carpeta de treball, el nom curt de la mateixa i el seu contingut en forma de col·lecció de cadena de caràcters amb el nom dels seus elements.

Implementarem el mètode `getAdrecaCarpeta` que llegirà la ruta absoluta de la carpeta de treball i el mètode `getNomCarpeta()` que obtindrà el nom curt.

```
public String getAdrecaCarpeta(){
    return carpetaDeTreball.getAbsolutePath();
}

public String getNomCarpeta(){
    return carpetaDeTreball.getName();
}
```

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: `getColumnes` i `setColumnes` seran els noms dels accessors de l'atribut `columnes`.

Com ja hem comentat, el contingut l'omplirem des del mètode `actualitza`. De moment, només omplirem la matriu `contingut`amb els noms dels elements de la carpeta, però més endavant caldrà modificar el mètode per aconseguir més informació. Per tal que els noms es distribueixin uniformement per l'espai que la interfície gràfica hi destina, els organitzarem en columnes. Per defecte, hi haurà tres columnes que podrem canviar a discreció. La interfície gràfica necessitarà també saber en quantes files i columnes s'ha aconseguit encabir el contingut. Afegirem, doncs, dos atributs, `columnes`i`files`. El primer de lectura i escriptura amb els seus accessors corresponents. El segon només de lectura, perquè el nombre de files dependrà de la quantitat de fitxers existent a la carpeta. Per tant, no implementarem l'accessor d'escriptura.

```
...
private int files=0;
private int columnes=3;

...

@Override
public int getColumnes() {
    return columnes;
}

@Override
public void setColumnes(int columnes) {
```

```
        this.columnes = columnes;
    }

    @Override
    public int getFiles() {
        return files;
    }
    ...

    private void actualitza(){
        String[] fitxers = carpetaDeTreball.list(); //obtenir els noms
        //calcular el nombre de files necessari
        files = fitxers.length / columnes;
        if(files*columnes < fitxers.length){
            files++; //si hi ha residu necessitem un fila més
        }

        //dimensionar la matriu contingut d'acord als resultats

        contingut = new String[files][columnes];
        //Omplir el contingut amb els noms obtinguts
        for(int i=0; i<columnes; i++){
            for(int j=0; j<files; j++){
                int ind = j*columnes+i;
                if(ind<fitxers.length){
                    contingut[j][i]=fitxers[ind];
                }else{
                    contingut[j][i]="";
                }
            }
        }
    }

    @Override
    public Object[][] getContingut() {
        return contingut;
    }
```

Modificarem també l'accessor de lectura a l'atribut `contingut`, que s'anomena `getContingut` per tal que retorni el valor del mateix, adequadament actualitzat.

Si voleu, podeu fer ja una primera prova per veure el resultat del que acabeu d'implementar. Seguiu les indicacions de l'annex "Creació del projecte Gestió de Fitxers i importació de les biblioteques", que trobareu en la secció "Annexos".

Un cop creats els principals mètodes per obtenir informació, començarem a implementar mètodes que ens permetin navegar pel sistema. Amb el mètode `setAdrecaCarpeta` podrem canviar la carpeta de treball a partir de la ruta passada com a paràmetre.

```
@Override
public void setAdrecaCarpeta(String adreca)
                      throws GestioFitxersException{
    File file = new File(adreca);
    //es controla que l'adreça passada existeixi i sigui un directori
    if(!file.isDirectory()){
        throw new GestioFitxersException("Error. S'esperava "
            + "un directori, però "
            + file.getAbsolutePath() + " no és un directori.");
    }
    //es controla que es tinguin permisos per llegir la carpeta
    if(!file.canRead()){
        throw new GestioFitxersException("Alerta. No podeu accedir a "
            + file.getAbsolutePath() + ". No teniu prou permisos");
    }
    //nova assignació de la carpeta de treball
    carpetaDeTreball=file;
    //es requereix actualitzar el contingut
    actualitza();
}
```

Usarem el mètode de la classe `File` anomenat `isDirectory` per controlar que la ruta es correspongui a una carpeta existent, ja que si no existeix el sistema no sabrà si es tracta d'una carpeta o un fitxer i `isDirectory` retornarà fals. Si no és una carpeta, caldrà llançar una excepció informant del problema. També es llançarà una excepció en cas que no hi hagi permisos per poder llegir el contingut de la carpeta. Per comprovar-ho, usarem `canRead`.

La interfície ens indica que cal implementar el mètode `entraA` de forma que serveixi per canviar la *carpeta de treball* a la carpeta passada per paràmetre i expressada com una ruta relativa de la carpeta activa en el moment de l'execució del mètode. En la nostra aplicació servirà per entrar un nivell dins l'arbre de directoris. És a dir, és l'operació que usa la interfície gràfica per mostrar el contingut d'aquella carpeta sobre la qual es faci doble clic. Com en el cas de `setAdrecaCarpeta`, caldrà crear un nou objecte File, però aquest cop relatiu a la carpeta activa. Usarem el constructor, al qual li passarem dos paràmetres: el `File` de la *carpeta de treball,* que farà de ruta base i una cadena amb la ruta relativa a la base.

```
1  @Override
2  public void entraA(String nomCarpeta) throws GestioFitxersException{
3      File file = new File(carpetaDeTreball, nomCarpeta);
4      //es controla que el nom correspongui a una carpeta existent
5      if(!file.isDirectory()){
6          throw new GestioFitxersException("Error. S'ha trobat "
7            + file.getAbsolutePath() + " però s'eperava un directori");
8      }
9      //es controla que es tinguin permisos per llegir la carpeta
10     if(!file.canRead()){
11         throw new GestioFitxersException("Alerta. No podeu accedir a "
12           + file.getAbsolutePath() + ". No teniu prou permisos");
13     }
14     //nova assignació de la carpeta de treball
15     carpetaDeTreball=file;
16     //es requereix actualitzar el contingut
17     actualitza();
18 }
```

Aquí caldrà també comprovar que existeixi el directori i la nova carpeta activa sigui accessible (es pugui llegir el seu contingut).

Per poder pujar de nivell, la `GestioFitxers` preveu el mètode `amunt`. Ho implementarem aprofitant la utilitat de la classe `File` que ens retorna la instància `File` pare. Ens referim a `getParentFile`.

```
1  @Override
2  public void amunt(){
3      if(carpetaDeTreball.getParentFile()!=null){
4          carpetaDeTreball = carpetaDeTreball.getParentFile();
5          actualitza();
6      }
7  }
```

L'execució de `getParentFile` sobre l'arrel del sistema de fitxers ens retorna un valor *null*. Usarem aquesta característica per assegurar de no sobrepassar mai l'arrel i evitar errors amb punters nuls.

1.2.2 Obtenció d'informació detallada

La classe `File` disposa encara de molts mètodes per obtenir informació detallada de cada element. Generalment, les aplicacions com la que estem fent permeten mostrar una finestra amb la informació detallada d'un dels components. La interfície gràfica està preparada per obrir una finestra amb la informació que la instància de `GestioFitxers` li passi en cridar al mètode `getInformacio`. El mètode `getInformacio` rep una cadena amb el nom del fitxer del qual es demana informació. Amb el nom haurem d'instanciar un objecte `File`i organitzar la seva informació dins una cadena de caràcters. Per tal d'anar afegint informació de mica en mica, usarem `StringBuilder`, que optimitza la concatenació successiva de cadenes, millor que la classe `String`.

Des de la interfície gràfica obtindrem la informació addicional seleccionant l'opció *Propietats* des del menú *Fitxer* i un cop s'obri el quadre de diàleg, seleccionant la la pestanya *Informació*.

La informació que extraurem de `File` serà, el nom, usant `getName`; si es tracta d'una carpeta o d'un fitxer; podem fer servir `isDirectory` o be `isFile`indistintament per saber-ho. Recollirem també la ruta absoluta canònica, extreta de `getCannonicalPath` i la data de la darrera modificació o de la creació si no s'ha modificat mai, usant el mètode `lastModified`.

En cas que es tracti d'una carpeta, s'indicarà també el nombre d'entrades que conté interrogant la longitud del vector, carregat amb el contingut i retornat per *list*. Fent servir els mètodes `getFreeSpace`, `getUsableSpace`o `getTotalSpace`obtindrem informació de l'espai lliure, l'espai disponible i l'espai total de la partició on es trobi la carpeta analitzada. L'espai disponible fa referència a l'espai que l'aplicació pot usar en el moment de l'execució i que no ha de coincidir pas amb l'espai lliure.

Si en comptes de tractar-se d'una carpeta fos un fitxer, caldrà mostrar només la seva mida en bytes usant el mètode `length`.

A la biblioteca que se us proporciona trobareu la classe `ByteFormat`, que automatitza la formatació de la mida dels fitxers escollint les unitats pertinents. Vegeu la documentació a l'annex "Documentació API *GestioFitxersBase*" de la secció "Annexos".

En ambdós casos indicarem també si el sistema l'ha marcat com ocult o resta visible a l'usuari. La informació l'extraurem del mètode `isHidden` de `File`. Vegem la implementació:

```
1  @Override
2  public String getInformacio(String nom)
3                  throws GestioFitxersException {
4      ByteFormat byteFormat= new ByteFormat("#,###.0", ByteFormat.BYTE);
5      StringBuilder strBuilder = new StringBuilder();
6      File file = new File(carpetaDeTreball, nom);
7      //Es controla que existeixi l'element a analitzar
8      if(!file.exists()){
9          throw new GestioFitxersException("Error. No es pot"
10            + " obtenir informació " + "de " + nom + ", no existeix.");
11     }
12     //es controla que es tinguin permisos per llegir la carpeta
13     if(!file.canRead()){
14         throw new GestioFitxersException("Alerta. No es pot "
15             + "accedir a " + nom + ". No teniu prou permisos");
16     }
17     //S'escriu el títol
18     strBuilder.append("INFORMACIÓ DEL SISTEMA");
19     strBuilder.append("\n\n");
20     //S'afegeix el nom
21     strBuilder.append("Nom: ");
```

```
22        strBuilder.append(nom);
23        strBuilder.append("\n");
24        //El tipus (carpeta o fitxer)
25        strBuilder.append("Tipus: ");
26        if(file.isFile()){
27            //es fitxer
28            strBuilder.append("fitxer");
29            strBuilder.append("\n");
30            //s'escriu La mida
31            strBuilder.append("Mida: ");
32            strBuilder.append(byteFormat.format(file.length()));
33            strBuilder.append("\n");
34        }else{
35            //es carpeta
36            strBuilder.append("carpeta");
37            strBuilder.append("\n");
38            //S'indica el nombre d'elements continguts
39            strBuilder.append("Contingut: ");
40            strBuilder.append(file.list().length);
41            strBuilder.append(" entrades\n");
42        }
43        //Afegim la ubicació
44        strBuilder.append("Ubicació: ");
45        /* Cal posar el try per exigències del llenguatge, però no
46         * controlarem aquest error doncs sabem que mai es produirà.
47         * Si hem arribat fins aquí és que l'adreça és bona */
48        try {
49            strBuilder.append(file.getCanonicalPath());
50        } catch (IOException ex) {/*Mai es produirà aquest error*/}
51        strBuilder.append("\n");
52        //Afegim la data de la última modificació
53        strBuilder.append("Última modificació: ");
54        Date date = new Date(file.lastModified());
55        strBuilder.append(date.toString());
56        strBuilder.append("\n");
57        //Indiquem si és o no un fitxer ocult
58        strBuilder.append("Ocult: ");
59        strBuilder.append((file.isHidden())?"Si":"No");
60        strBuilder.append("\n");
61
62        if(file.isDirectory()){
63            //Mostrem l'espai lliure
64            strBuilder.append("Espai lliure: ");
65            strBuilder.append(byteFormat.format(file.getFreeSpace()));
66            strBuilder.append("\n");
67            //Mostrem l'espai disponible
68            strBuilder.append("Espai disponible: ");
69            strBuilder.append(byteFormat.format(file.getUsableSpace()));
70            strBuilder.append("\n");
71            //Mostrem l'espai total
72            strBuilder.append("Espai total: ");
73            strBuilder.append(byteFormat.format(file.getTotalSpace()));
74            strBuilder.append("\n");
75        }
76
77        return strBuilder.toString();
78    }
```

Destacarem que en aquest cas, en no saber si es tracta d'un fitxer o una carpeta, la comprovació de l'existència la realitzarem mitjançant l'operació de `File` anomenada `exists`. Un altre fet remarcable és que malgrat sapiguem que `getCanonicalPath`no llançarà mai cap excepció, ja que el nom és correcte, ja que abans s'han obtingut altres dades, la rigidesa de les excepcions ens obliga a embolcallar la crida dins una sentència *try-catch*.

1.2.3 Filtratge, ordenació i classificació de fitxers

Arribats a aquest punt i abans de començar a implementar els mètodes de manipulació del sistema (creació o eliminació d'elements), cal adonar-nos que la visualització del contingut a la interfície gràfica no és gaire bona, ja que no es distingeixen directoris de fitxers, no es visualitzen en cap ordre determinat, els fitxers ocults apareixen sempre i no es mostra informació addicional dels elements.

Començarem per amagar els fitxers ocults. La classe `File`, en el moment de recopilar la informació del contingut d'un directori a través de les operacions `list` o `listFile`, pot acceptar l'ajuda d'un filtre que indiqui quins fitxers cal descartar.

Els filtres han d'implementar la interfície `FilenameFilter`, i poden usar-se indistintament en el mètode `list` i en el mètode `listFile`. Existeix també la interfície `FileFilter` amb un objectiu similar, però malauradament només és acceptada per `listFile`.

Els filtres de tipus `FilenameFilter` implementaran un mètode anomenat `accept` que rebrà dos paràmetres: un `File` indicant el directori on es troba ubicat l'element a avaluar i una cadena amb el nom d'aquest element. El mètode retornarà cert o fals en funció de si es vol descartar o incloure a la llista del contingut.

Davant la multiplicitat de filtres que es poden necessitar, les instàncies solen implementar-se com a classes anònimes o com a classes internes per tal de reduir la quantitat de classes del model, minimitzar l'acoblament i permetre que les instàncies creades tinguin accés total als atributs i mètodes de la classe amfitriona si fos necessari.

En aquesta aplicació optarem per crear una classe interna de tipus `FilenameFilter`, principalment per reduir l'acoblament i mantenir-la aïllada del model. Escollim `FilenameFilter`, ja que necessitarem aplicar el criteri de selecció tant sobre `listFile` com sobre `list`. El mètode accept generarà un `File` amb els paràmetres rebuts i retornarà cert si no és ocult o fals en cas contrari.

```
1 private class FiltreFitxersOcults implements FilenameFilter{
2     @Override
3     public boolean accept(File pfile, String string) {
4         File file = new File(pfile, string);
5         return !file.isHidden();
6     }
7 }
```

El filtre es farà efectiu quan el passem per paràmetre a List:

```
1 String[] fitxers = carpetaDeTreball.list(new FiltreFitxersOcults());
```

En el codi anterior, la variable fitxers contindrà tots els fitxers de la carpeta de treball menys els ocults.

Passem ara a l'ordenació. Usarem la utilitat anomenada `sort`de la classe `Arrays`. Aquest mètode permet ordenar un vector de qualsevol tipus usant una instància de *Comparator*. Aquesta interfície requereix d'un únic mètode amb el nom de *compare*, el qual, rebent dos objectes, podrà analitzar-los i dictaminar quina relació d'ordre s'estableix entre ells. El veredicte de l'anàlisi es retornarà fent servir la següent convenció: si el primer és menor que el segon es retornarà un valor enter negatiu; per contra, si és més gran, es retornarà un valor enter positiu. Es retornarà zero només si són iguals.

La classe `String` disposa ja d'una instància de `Comparator`. Es tracta de l'atribut estàtic anomenat `String.CASE_INSENSITIVE_ORDER`, que compara cadenes en ordre alfanumèric sense tenir en compte les majúscules o minúscules.

Aprofitarem els valors del tipus enumerat `TipusOrdre`, que podeu trobar a la biblioteca, per determinar l'ordenació del contingut. La categoria *DESORDENAT* indicarà que no desitgem cap mena d'ordenació (com fins ara). La categoria *NOM*, en canvi, indicarà que volem el contingut ordenat d'acord amb el nom de cada element. L'enumeració *TipusOrdre* disposa també de les categories *MIDA* i *DATA_MODIFICACIO*, que fareu servir en els exercicis.

De moment crearem dos atributs amb els accessors corresponents, per indicar al gestor si cal o no filtrar els fitxers ocults i quin tipus d'ordre s'espera. Afegirem també una instància privada del filtre `FiltreFitxersOcults` per no haver d'anar creant un objecte nou cada cop que actualitzem el contingut.

```
1  ...
2  private TipusOrdre ordenat;
3  private boolean mostrarOcults;
4  private final FiltreFitxersOcults filtreFitxersOcults=
5      new FiltreFitxersOcults();
6  ...
7  @Override
8  public boolean getMostrarOcults() {
9      return mostrarOcults;
10 }
11
12 @Override
13 public void setMostrarOcults(boolean ocults) {
14     this.mostrarOcults=ocults;
15     actualitza();
16 }
17
18 @Override
19 public TipusOrdre getOrdenat() {
20     return ordenat;
21 }
22
23 @Override
24 public void setOrdenat(TipusOrdre ordenat) {
25     this.ordenat=ordenat;
26     actualitza();
27 }
```

En el mètode `actualitza` usarem els atributs creats per saber si cal ordenar i si cal amagar els fitxers ocults. La modificació del mètode donarà com a resultat:

```
1  private void actualitza(){
2      String[] fitxers ; //obtenir els noms
3      if(mostrarOcults){
```

```
4            fitxers = carpetaDeTreball.list();
5        }else{
6            fitxers = carpetaDeTreball.list(filtreFitxersOcults);
7        }
8        columnes = columnesBase;
9        //calcular el nombre de files necessari
10       files = fitxers.length / columnes;
11       if(files*columnes < fitxers.length){
12           files++; //si hi ha residu necessitem un fila més
13       }
14       //ordenació del contingut
15       if(ordenat==TipusOrdre.NOM){
16           Arrays.sort(fitxers, String.CASE_INSENSITIVE_ORDER);
17       }
18       //dimensionar la matriu contingut d'acord als resultats
19       contingut = new String[files][columnes];
20       /* Omplir el contingut amb els noms dels elements de la
21        * carpeta activa */
22       for(int i=0; i<columnes; i++){
23           for(int j=0; j<files; j++){
24               int ind = j*columnes+i;
25               if(ind<fitxers.length){
26                   contingut[j][i]=fitxers[ind];
27               }else{
28                   contingut[j][i]="";
29               }
30           }
31       }
32   }
```

1.2.4 Modificació del sistema de fitxers

Ara veurem com crear noves carpetes i fitxers, com eliminar-los, com canviar-los el nom i com canviar-los la data de modificació. La creació de noves carpetes es realitza executant `mkdir` o bé `mkdirs`. El primer, crearà el directori vinculat a la instància *File* des de la qual s'executa, però ho farà ubicat a la carpeta pare. És a dir, aquest mètode només té la capacitat de crear un sol nivell, mentre que `mkdirs`crearà totes les carpetes necessàries per tal que la ruta representada per la instància existeixi en finalitzar l'execució.

`mkdir`precisa que la ruta fins el nivell immediatament superior a la instància existeixi. En cas contrari no es crearà la ruta. Aquest mètode retorna cert si aconsegueix la creació i fals en cas contrari. De la mateixa manera, `mkdirs`retorna també cert en cas que la creació tingui èxit o fals si no en té.

`GestiFitxersImpl` no necessita `mkdirs`, perquè totes les creacions es faran des de la carpeta de treball, que serà la carpeta pare de nova creació. Caldrà, això sí, controlar els possibles errors que es puguin produir per manca de permisos d'escriptura a la carpeta de treball o deguts a l'existència prèvia d'una carpeta amb el mateix nom.

```
1    @Override
2    public void creaCarpeta(String nomCarpeta)
3                      throws GestioFitxersException{
4        File file = new File(carpetaDeTreball, nomCarpeta);
5        if(!carpetaDeTreball.canWrite()){
6            throw new GestioFitxersException("Error. No s'ha pogut crear "
```

```
7             + nomCarpeta + ". No teniu suficients permisos");
8     }
9     if(file.exists()){
10        throw new GestioFitxersException("Error. No s'ha pogut crear."
11            + " Ja existeix un fitxer o carpeta amb el nom "
12            + nomCarpeta);
13    }
14    if(!file.mkdir()){
15        throw new GestioFitxersException("Error. No s'ha pogut crear "
16            + nomCarpeta + ".");
17    }
18    actualitza();
19 }
```

La creació de fitxers serà similar, però usant el mètode `createNewFile`. Aquesta operació només crearà el fitxer en cas que no n'existeixi cap altre amb el mateix nom. A més, per aconseguir la creació caldrà que tota la ruta de la carpeta on s'hagi de fer la creació existeixi, sigui vàlida i tingui permisos d'escriptura.

És important adonar-se que les instàncies de `File` d'elements no creats no es poden reconèixer encara ni com a fitxers ni com a carpetes, sinó com a **elements inexistents**.

Aquesta característica té un implicació important. No es pot usar un sola instància per crear la ruta i el fitxer vinculat a un `File` si cap dels dos existeix, ja que l'execució de `mkdirs` interpretaria que el darrer nom de la ruta és també una carpeta i la crearia com a tal. En intentar executar `createNewFile` no es produiria cap efecte perquè el nom ja existiria en forma de carpeta. Es necessiten com a mínim dues instàncies per realitzar la doble creació: una vinculada a la carpeta contenidora i l'altra al fitxer en qüestió.

La implementació a la classe `GestióFitxersImpl` tindrà una forma semblant a:

```
1  @Override
2  public void creaFitxer(String nomFitxer)
3                throws GestioFitxersException {
4      File file = new File(carpetaDeTreball, nomFitxer);
5      if(!carpetaDeTreball.canWrite()){
6          throw new GestioFitxersException("Error. No s'ha pogut crear "
7              + nomFitxer + ". No teniu suficients permisos");
8      }
9      if(file.exists()){
10         throw new GestioFitxersException("Error. No s'ha pogut crear."
11             + " Ja existeix un fitxer o carpeta amb el nom "
12             + nomFitxer);
13     }
14     try {
15         if(!file.createNewFile()){
16             throw new GestioFitxersException("Error. No s'ha pogut "
17                 + "crear " + nomFitxer + ".");
18         }
19     } catch (IOException ex) {
20          throw new GestioFitxersException("S'ha produit un error "
21             + "d'entrada o sortida: '" + ex.getMessage() + "'", ex);
22     }
23     actualitza();
24 }
```

És molt similar a l'operació de creació de carpetes. Aquí destacarem només la possibilitat que el sistema operatiu reporti una excepció d'entrada/sortida (*IOException*) si es troben problemes en escriure al dispositiu. Per evitar el llançament de múltiples tipus d'excepcions, encapsularem els errors que es produeixin dins una excepció de tipus `GestioFitxersException`.

Per a l'eliminació de carpetes o fitxers, usarem el mètode `delete`, que permet eliminar indistintament fitxers o carpetes. Igual que la resta de mètodes de modificació, retorna cert si l'eliminació s'ha pogut dur a terme i fals en cas contrari. Com que l'èxit o el fracàs de l'operació depèn en gran mesura del sistema operatiu, no disposa de controls d'error específics, sinó que caldrà implementar-los a `GestioFitxersImpl`.

```
@Override
public void elimina(String nom) throws GestioFitxersException{
    File file = new File(carpetaDeTreball, nom);
    if(!carpetaDeTreball.canWrite()){
        throw new GestioFitxersException("Error. No s'ha pogut "
            + "eliminar " + nom
            + ". No teniu suficients permisos");
    }
    if(!file.exists()){
        throw new GestioFitxersException("Error. S'intenta "
            + "eliminar " + nom + " però no existeix.");
    }
    if(!file.delete()){
        if(file.isDirectory() && file.list().length>0){
            throw new GestioFitxersException("Error. No s'ha pogut "
                + "eliminar. La carpeta " + nom + "no està buida.");
        }else{
            throw new GestioFitxersException("Error. No s'ha pogut "
                + "eliminar " + nom + ".");
        }
    }
    actualitza();
}
```

La utilitat que reanomena fitxers o carpetes presenta unes característiques semblants a les que ja s'han vist. És a dir, retorna cert si aconsegueix l'objectiu i fals en cas contrari. La diferència principal és que el mètode `rename` rep per paràmetre el nou nom amb el qual es desitja reanomenar la instància `File`.

```
@Override
public void reanomena(String nom, String nomNou)
                        throws GestioFitxersException{
    File file = new File(carpetaDeTreball, nom);
    File fileNou = new File(carpetaDeTreball, nomNou);
    if(!carpetaDeTreball.canWrite()){
        throw new GestioFitxersException("Error. No s'ha pogut "
            + "eliminar " + nom
            + ". No teniu suficients permisos");
    }
    if(!file.exists()){
        throw new GestioFitxersException("Error. No es pot fer el "
            + "canvi de nom, " + nom + " no existeix.");
    }
    if(!file.renameTo(fileNou)){
        throw new GestioFitxersException("Error. No s'ha pogut "
            + "canviar de nom, " + nom + ".");
    }
    actualitza();
}
```

Des de la interfície gràfica, la modificació de la data s'aconsegueix seleccionant l'opció *Propietats* des del menú *Fitxer* i un cop s'obri el quadre de diàleg, seleccionant la pestanya *Modificar*.

L'operació que permet modificar la data de modificació de fitxers o carpetes és `setLastModified`, la qual rep per paràmetre un valor `long` representant la data a assignar, en el mateix format que la retorna `lastModified`. La implementació a l'aplicació que estem construint serà:

```
@Override
public void setUltimaModificacio(String nom, long dataIHora)
                                  throws GestioFitxersException {
    File file = new File(carpetaDeTreball, nom);
    if(!file.exists()){
        throw new GestioFitxersException("Error. No es pot "
            + "obtenir modificar " + nom + ", no existeix.");
    }
    file.setLastModified(dataIHora);
}
```

Finalment farem esmentdels permisos d'accés que el sistema operatiu atorga. Es pot interrogar `File` per saber si un element es pot llegir (`canRead`), es pot escriure (`canWrite`) o be si és executable (`canExecute`). De manera similar a la resta de propietats, és possible modificar aquests permisos amb els corresponents mètodes: `setReadable`, `setWritable` o `setExecutable`. La classe `File` disposa de dues versions per a cada mètode, la versió en què es rep només un paràmetre de tipus booleà i la versió en què se'n reben dos. La darrera versió és útil per a sistemes operatius que distingeixen entre permisos de propietari i permisos d'altres usuaris. El primer valor lògic representa l'estat en què es vol deixar el permís, mentre que el segon valor representa si el canvi afectarà només el propietari (valor cert) o bé afectarà tothom (valor fals). Així, `file.setReadable(false, true)` farà que l'element al qual es trobi vinculat la instància `File` no sigui llegible pel propietari. Els permisos per a la resta d'usuaris no es modifiquen i continuen establerts al seu valor original. En canvi, `file.setReadable(true, false)` farà que`File` sigui llegible tant pel propietari com per la resta d'usuaris. Les versions d'un únic paràmetre són equivalents a les darreres amb el segon valor fixat a true.

Per l'exemple que implementem, usarem només les versions d'un únic paràmetre.

```
@Override
public boolean esPotEscriure(String nom)
                  throws GestioFitxersException {
    File file = new File(carpetaDeTreball, nom);
    if(!file.exists()){
        throw new GestioFitxersException("Error. No es pot obtenir "
            + "informació de " + nom + ", no existeix.");
    }
    return file.canWrite();
}

@Override
public boolean esPotExecutar(String nom)
                  throws GestioFitxersException {
    File file = new File(carpetaDeTreball, nom);
    if(!file.exists()){
        throw new GestioFitxersException("Error. No es pot obtenir "
            + "informació de " + nom + ", no existeix.");
    }
    return file.canExecute();
}

@Override
public void setEsPotLlegir(String nom, boolean permis)
                              throws GestioFitxersException {
    File file = new File(carpetaDeTreball, nom);
    if(!file.exists()){
```

```
            throw new GestioFitxersException("Error. No es pot modificar "
                + nom + ", no existeix.");
        }
        file.setReadable(permis);
    }

    @Override
    public void setEsPotEscriure(String nom, boolean permis)
                                throws GestioFitxersException {
        File file = new File(carpetaDeTreball, nom);
        if(!file.exists()){
            throw new GestioFitxersException("Error. No es pot modificar "
                + nom + ", no existeix.");
        }
        file.setWritable(permis);
    }

    @Override
    public void setEsPotExecutar(String nom, boolean permis)
                                throws GestioFitxersException {
        File file = new File(carpetaDeTreball, nom);
        if(!file.exists()){
            throw new GestioFitxersException("Error. No es pot modificar "
                + nom + ", no existeix.");
        }
        file.setExecutable(permis);
    }
```

Les operacions de copiar o moure fitxers les veurem a l'apartat de "Gestió de Continguts" ja que requereixen traspassar el contingut d'una ubicació a una altra.

Per poder fer servir l'aplicació en plataformes Windows i poder seleccionar entre les diverses unitats d'emmagatzematge, caldrà complementar la implementació amb dos mètodes que ens aportin aquesta informació:

```
@Override
public int numArrels(){
    return File.listRoots().length;
}

@Override
public String nomArrel(int id){
    return File.listRoots()[id].toString();
}
```

Finalment, es podran també implementar les dues últimes operacions contemplades a la interfície `GestioFitxers`, que ens aportaran una visió més estètica de la interfície afegint-hi informació addicional.

```
@Override
public String getEspaiDisponibleCarpetaTreball() {
    ByteFormat format = new ByteFormat("#,##0.00");
    return format.format(carpetaDeTreball.getUsableSpace());
}

@Override
public String getEspaiTotalCarpetaTreball() {
    ByteFormat format = new ByteFormat("#,##0.00");
    return format.format(carpetaDeTreball.getTotalSpace());
}
```

2. Gestió del contingut dels fitxers

En aquest apartat també ens serviremd'un exemple per continuar avançant en l'estudi de les classes de Java que ens permetin gestionar els fitxers. Ara aprofundirem, específicament, en l'estudi de la gestió del contingut de fitxers.

Consulteu l'annex "Biblioteques del mòdul" en la secció "Annexos" per obtenir el codi de partida de l'aplicació, exemple que il·lustra l'ús de les classes Java.

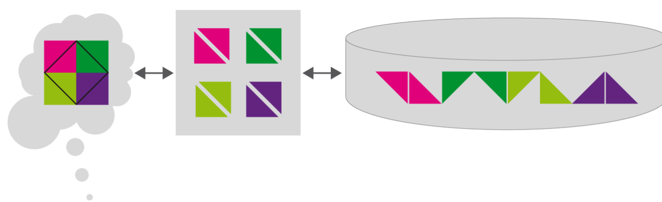
L'exemple que ens il·lustrarà l'aplicació i ús d'aquestes classes enllaça també amb l'anterior aplicació de gestió de fitxers, ampliant algunes de les seves utilitats, com ara la còpia o el trasllat de fitxers en diferents ubicacions o l'edició del seu contingut. Per tant, el punt de partida serà l'aplicació que heu implementat i que podeu aconseguir també als annexos d'aquests materials.

Abans però, caldrà clarificar el concepte de fitxers com a magatzems de contingut, i el concepte de contingut com a flux de dades.

2.1 Magatzems i fluxos de dades

Des del punt de vista de l'usuari, els fitxers són magatzems permanents d'informació guardada en forma de dades estructurades i ben organitzades de manera que la seva identificació i conseqüent interpretació siguin fàcils. Aquesta és una perspectiva centrada en la *representació mental* de la informació i s'anomena *estructura lògica*.

Figura 2.1. Estructura lògica i estructura física de les dades



L'Estructura lògica o representació conceptual de les dadesva canviant a mida que avança pels diferents nivells d'abstracció fins a arribar al dispositiu d'emmagatzematge. La forma final de les dades es coneix també com estructura física.

Ara bé, en el camí de la persistència, la representació mental original passarà per diferents nivells d'abstracció fins arribar a l'emmagatzematge físic (vegeu la figura 2.1). Cada nivell imposarà les seves limitacions a la representació del nivell anterior introduint canvis que alteraran l'estructura original. El primer nivell d'abstracció el trobem en el llenguatge de programació utilitzat, que depenent de les estructures d'informació suportades, els tipus de dades permesos, la representació en memòria d'aquest, etc., alterarà en alguna mesura la representació de partida. El Sistema Operatiu imposarà també les seves restriccions i ho farà també el *driver* o adaptador al sistema d'emmagatzematge, o els mecanismes físics i els suport escollits en últim terme. L'estructura final, com les dades acaben emmagatzemant-se, s'anomena també *estructura física*.

Parlarem d'**estructura lògica de la informació**quan ens referim a estructures de dades properes a la representació mental. En contraposició, parlarem d'**estructura física de les dades** quan ens referim a estructures de dades allunyades d'aquesta.

Generalment, els llenguatges d'alt nivell com Java suporten estructures de dades força properes a la representació mental. D'aquí que se solen classificar com a estructures lògiques. Malgrat tot, són estructures difícilment traslladables al nivell físic, ja que es troben farcides de referències que apunten a d'altres zones de la memòria principal on s'ubiquen part de les dades de l'estructura global. Requeriran, per tant, una adaptació a l'estructura física quan calgui emmagatzemar-les.

2.1.1 Seriació i flux de dades

Les referències a memòria són dades transitòries que varien en cada execució. Si guardéssim de forma persistent les referències a memòria, en recuperar-les obtindríem dades totalment incoherents perquè apuntarien a dades inexistents o ocupades per altra informació.

Si volem un emmagatzematge i una recuperació de les dades eficaç, caldrà assegurar que totes les dades referenciades s'emmagatzemen també al fitxer i es relacionen entre elles de forma que sigui factible localitzar-les i recuperar-les conjuntament.

La forma més senzilla de fer-ho consisteix a compactar les successives dades referenciades agrupant-les, una darrera l'altra, en una única seqüència de dades primitives lliure de referències.

Aquest procés s'anomena **seriació** i cal utilitzar-lo per transformar estructures complexes de la memòria principal en sèries de dades compactes i fàcilment emmagatzemables.

Les dades primitives s'emmagatzemen en els fitxers sense canvis, copiant literalment la seqüència de bits de la memòria al suport físic. Això significa, també, que l'emmagatzematge de la seqüència de dades primitives consistirà simplement en una còpia literal dels bits de la seqüència. És per això que la seqüència de dades primitives s'anomena també seqüència de bits.

El concepte de **seqüència de bits** ens aporta una visió estàtica dels fitxers en el sentit de magatzem de la seqüència.

Tots sabem que l'aigua corrent roman emmagatzemada en pantans i dipòsits abans de rajar de les nostres aixetes, però en el nostre imaginari els pantans i els dipòsits

queden lluny i en parlar d'aigua corrent tendim més aviat a pensar en aixetes i canonades, que són els estris que en últim terme usem per controlar l'aigua corrent.

De forma similar, des del punt de vista de l'aplicació, el que realment cobra importància és la transferència de dades, més que no pas el magatzem. L'estri que ens permet controlar aquestes transferències, de forma similar a les aixetes i canonades, l'anomenem **flux de dades**. És un concepte associat a la transmissió seqüencial d'una sèrie de dades des de l'aplicació al dispositiu d'emmagatzematge o a l'inrevés.

El concepte de **flux** ens dóna un visió dinàmica dels fitxers entenent-los com a processos d'intercanvi seqüencial de dades entre el magatzem i l'aplicació.

El concepte de flux (*stream*) no és pas exclusiu dels fitxers, sinó que es tracta d'una abstracció relacionada amb qualsevol procés de transmissió d'informació entre un contenidor de dades i una zona de la memòria primària controlada per l'aplicació.

Anomenarem **flux d'entrada** aquells processos de transmissió d'informació que traslladin dades des d'un contenidor qualsevol a la zona de memòria primària controlada per l'aplicació. És a dir, que enviïn dades a fi de ser processades durant l'execució.

Anomenarem **flux de sortida** aquells processos de transmissió d'informació que traslladin dades des de la zona de memòria primària controlada per l'aplicació cap a qualsevol altre contenidor de dades.

La transmissió de dades per mitjà de fluxos s'entén sempre de manera seqüencial, és a dir, l'ordre en què surten les dades de l'emissor és el mateix en què arriben al receptor.

2.1.2 Fluxos i tipus de dades

El concepte de flux es contraposa al concepte de tipus de dada en el sentit que en compactar totes les dades, els límits d'aquestes es difuminen fins a desaparèixer, donant com a resultat un flux continu de bytes, ja que la informació compactada no conté cap mena de marca d'on comença o acaba una dada.

Exemples de compactació de dades en un flux

Imaginem un tipus de dada numèric de 16 bits. La representació en binari del número 24941 seria 0110000101101101, i la del número 26979 seria 0110100101100011. En compactar ambdós valors en un flux obtindríem la seqüència 01100001011011010110100101100011.

Imaginem ara una cadena de text que conté la paraula amic en binari; el caràcter a esrepresenta: 01100001, el caràcter i: 01101101, el caràcter m: 01101001 i el caràcter c: 01100011. El flux de dades d'aquest text seria també 01100001011011010110100101100011.

De la mateixa manera, el número 1634560355 representat en 32 bits tindria també la mateixa seqüència 01100001011011010110100101100011.

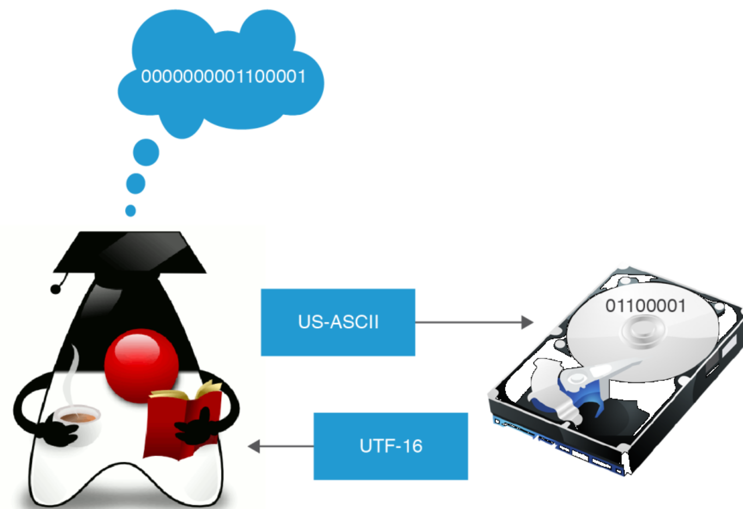
Donada una seqüència de bits, no hi ha manera, a priori, de saber la composició de les seves dades. Només si coneixem la mida i l'ordre en què es van compactar, podrem recuperar els valors originals del flux.

Sortosament, els tipus bàsics de dades tenen una mida prefixada i l'emmagatzematge es fa considerant tots el bits. Així, el valor numèric 1 d'un tipus enter de 16 bits s'empaquetarà com 0000000000000001 i ocuparà el mateix que qualsevol altre número del mateix tipus.

Les classes de Java orientades a fluxos transfereixen les dades de manera transparent al programador. No cal indicar la quantitat de bits que cal transferir, sinó que es dedueix a partir del tipus de dada que la variable representa.

Hi ha, però, una excepció amb el tipus *char*. La multitud d'estàndards de codificació de caràcters existents en l'actualitat i la diversitat de formats utilitzats a l'hora d'implementar les codificacions, usant segons el cas 8, 16, 32 bits o fins i tot una longitud variable en funció del caràcter a representar, fan que sigui molt difícil tractar aquest tipus de dada com una simple seqüència de bytes.

Figura 2.2. Internament, Java treballa amb caràcters de 16 bits per tal de suportar múltiples alfabets a banda de l'occidental



El programa podrà gestionar formats de caràcters de diverses longituds (8, 16 o 32 bits) usant classes de fluxos especialitzades en caràcters que realitzen la conversió de forma automatitzada.

Internament, Java representa el tipus caràcter amb una codificació UNICODE de 16 bits (UTF-16) per tal de suportar múltiples alfabets a banda de l'occidental (figura 2.2). Tot i així, és capaç de gestionar fonts de dades (fitxers entre d'altres) de diverses codificacions (ASCII, ISO-8859, UTF-8, UTF-16...). En funció de la codificació escollida, el nombre de bits usats en l'emmagatzematge variarà. Es fa necessari, doncs, un tractament especial a l'hora de fer la seriació i deseriació d'aquestes dades. Java disposa d'un jerarquia específica de classes orientades a fluxos de caràcters per tal de fer aquests canvis i transformacions totalment transparents al programador.

2.2 Manipulació dels fluxos

A Java trobem diverses jerarquies de classes orientades a fluxos: la jerarquia orientada a bytes, la jerarquia orientada a caràcters i una nova jerarquia que intenta aprofitar les utilitats dels sistemes operatius per obtenir transferències de dades molt eficients. Anem a estudiar-les.

2.2.1 Fluxos orientats a Bytes

Es tracta d'una important jerarquia de classes que intenta donar resposta a la multiplicitat funcional de fluxos de dades que es pugui donar en qualsevol situació. A l'arrel de la jerarquia trobem les classes `InputStream` i`OutputStream`. Són classes abstractes amb l'objectiu de definir el comportament comú de la resta de classes de la jerarquia. Així, `InputStream`, la superclasse d'on hereten la resta de classes orientades a ***fluxos de Bytes d'entrada***, especifica la sintaxi que han de tenir els mètodes de lectura de qualsevol font de dades, o les operacions de consulta que seran necessàries. `OutputStream`, en canvi, especifica els requeriments del comportament i la sintaxi de les classes orientades a ***fluxos de Bytes de sortida***. És a dir, escriptura de les dades a la font de dades.

Com s'especifiquen els mètodes d'InputStream

Els mètodes d'`InputStream` s'especifiquen tal com segueix:

- *int* **read**(). Llegeix el següent byte de dades del flux d'entrada i es retorna com un enter. Si no hi ha cap byte disponible perquè s'ha assolit el final de la seqüència, es retornarà -1. Si no hi ha cap dada disponible en el flux, el mètode es bloquejarà a l'espera d'alguna dada o de la marca que indiqui el final de la seqüència. En cas que s'hagi arribat al final de la seqüència de dades però s'hagi detectat el final, es llançarà una excepció del tipus *IOException*. Es tracta d'un mètode abstracte, que les classes especifiques sobreescriuran adaptant-lo a una font de dades concreta, a un format determinat o una metodologia d'obtenció de dades adequada.
- *int* **read**(*byte[]buffer*). Llegeix un nombre de bytes del flux d'entrada i els emmagatzema dins el paràmetre anomenat *buffer* de tipus vector de *bytes*. El nombre de *bytes*llegits pot dependre de diversos factors externs i no se'n pot garantir cap mínim. Aquests, però, mai sobrepassaran la mida del vector de bytes on escriure'ls. El mètode retorna el nombre de bytes realment llegits com un enter. Si no hi ha cap byte disponible perquè en el flux s'ha arribat al final de la seqüència, es retornarà -1 com a indicador de final de lectura. El mètode romandrà bloquejat en cas que no hi hagi cap dada disponible en el flux ni es trobi el final de la seqüència. Si se li passa un

paràmetre *null* es llançarà l'excepció *NullPointerException;* en canvi, si la longitud del paràmetre fos zero mai es llegiria cap byte però no es produiria cap excepció.

- *int* **read**(*byte[]buffer*, *int offset*, *int len*). S'intenten llegir fins a *len* bytes de dades del flux d'entrada i els copia en el vector de *bytes* anomenat *buffer*. Com en el cas anterior, el nombre de *bytes*llegits pot dependre de diversos factors externs i no se'n pot garantir cap mínim. Aquests, però, mai sobrepassaran la quantitat indicada per *len*. El mètode retorna el nombre de bytes realment llegits com un enter. Si no hi ha bytes a llegir es bloquejarà esperant que arribin dades a través del flux o que arribi el final de la seqüència. Si troba el final de la seqüència retornarà -1, en cas contrari el nombre de Bytes llegits. Cal que *buffer* no sigui *null*. Si *len* fos negatiu o *offset* fos negatiu o *offset+len* fos més gran que la longitud del vector (*buffer*), es llançaria una excepció de tipus `IndexOutOfBoundsException`.
- Els bytes llegits s'emmagatzemen al vector de bytes començant per la posició *ofsset* de vector.
- *int* **available()**. Retorna el nombre de bytes que es troben disponibles per llegir (o saltar) en aquest flux d'entrada en el moment de l'execució del mètode. Aquest mètode és merament informatiu, ja que no hi ha garantia que en el moment que es faci la lectura o el salt les condicions no hagin canviat. Si el flux no es pogués llegir per qualsevol raó, es llançarà una excepció.
- *long* **skip** (*long bytesToSkip*). Salta i descarta *bytesToSkip* bytes de dades d'aquest flux d'entrada. No hi ha garantia que el mètode acabi saltant exactament *bytesToSkip* bytes, atès que pot ser que per diverses raons acabi saltant menys. Es retorna el nombre real de bytes omesos. Si *bytesToSkip* és zero o negatiu, no es produirà cap salt.
- void **close()**. Tanca aquest flux d'entrada i allibera els recursos del sistema associats.

Com s'especifiquen els mètodes d'OutputStream

Els mètodes d'`OutputStream` estan orientats a l'escriptura de la font de dades i s'especifiquen de la següent manera:

- *void* **write**(*int byte*). Escriu un byte contingut al valor passat per paràmetre transferint-lo al flux de sortida. Si no pot escriure'l,es llança una excepció *IOException.*
- *void* **write**(*byte[] data*). Escriu tots els bytes continguts al vector passat per paràmetre. Cal que *data* no sigui *null*, sino es llançarà també una excepció de tipus *NullPointerException*. Si no fos possible l'escriptura es llançaria un excepció.
- *void* **write**(*byte[] data, int offset, int len*). Escriu, si és possible, els *len* bytes ubicats al vector passat per paràmetre a partir de la posició *offset* del

vector, transferint-los a la font de dades connectada per mitjà del flux. Si *len* fos negatiu o *offset* fos negatiu o *offset+len* fos més gran que la longitud del vector, es llançaria una excepció de tipus *IndexOutOfBoundsException*. Cal que *data* no sigui *null*, si no es llançarà també una excepció (*NullPointerException*). Si no fos possible l'escriptura es llançaria una excepció.

- *void* **flush()**. Sovint les fonts de dades poden tenir temps de processament elevats. En aquest casos serà normal l'ús de memòria intermèdia per minimitzar-ne els efectes. Aquest mètode buida la seqüència de sortida forçant a escriure els bytes que quedin encara a la memòria intermèdia.
- *void* **close()**. Tanca aquest flux de sortida i allibera els recursos del sistema associats.

Totes les classes que deriven de les dues superclasses (`InputStream` i `OutputStream`) són també orientades a bytes i, o bé concreten els mètodes abstractes implementant-los específicament per una font de dades determinada (memòria, fitxers, *sockets*, etc.) o bé en milloren la funcionalitat (ús de tipus bàsics en comptes de bytes, ús de doble *buffer* per poder disposar de dos punters dins la font de dades, etc.). Sobre la concreció de la font de dades, farem especial esment a la classe `FileInputStream` i l'homòloga de sortida `FileOutputStrem`, ja que implementen l'accés als fitxers.

Fluxos de dades contra fitxers

Sobre aquestes classes, cal destacar que els constructors admeten un paràmetre de tipus `File` o de tipus cadena de caràcters representant la ruta del fitxer:

- **FileInputStream**(*File fitxer*) o**FileInputStream**(*String ruta*) Obren el fitxer especificat com a paràmetre, en mode lectura i creen una instància amb la qual podrem llegir el contingut usant els mètodes heretats d'`InputStream` (*read*, *skip*, *available*, etc.).
- **FileOutputStream**(*File fitxer*) o **FileOutputStream**(*String ruta*). Creen un fitxer a la ruta especificada com a paràmetre. En cas que ja existeixi el fitxer, n'esborren el contingut. Un cop creat el fitxer, s'obre en mode escriptura. La instància creada amb aquest constructor permetrà escriure en el fitxer usant els mètodes heretats d'`OutputStream` (*write*, *flush* i *close*).

A més, la classe `FileOutputStream` disposa de dos constructors addicionals per poder indicar al sistema que es vol obrir un fitxer existent en mode escriptura sense perdre'n el contingut:

- **FileOutputStream**(*File fitxer*, *boolean append*) o **FileOutputStream**(*String ruta*, *boolean append*). Si el paràmetre *append* és cert, obriran el fitxer existent especificat per la ruta del primer paràmetre, en mode escriptura. Les dades escrites des d'aquesta instància s'afegiran sempre al final del contingut existent. Si `append`té el valor *false* el fitxer es crearà (o

s'esborrarà i crearà de nou si ja existís) igual que si s'hagués instanciat amb algun dels constructors anteriors.

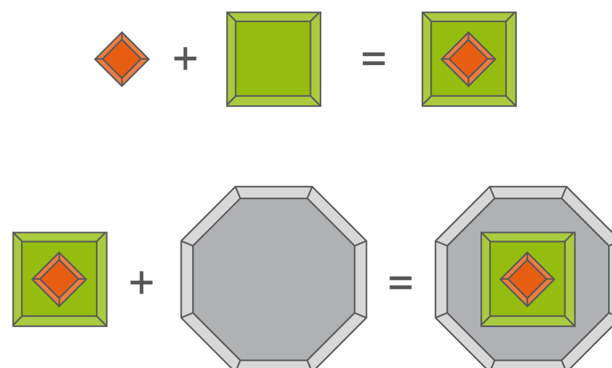
A banda dels constructors, la resta de mètodes coincideixen amb els de les seves superclasses respectives. Quelcom de semblant succeeix amb altres fonts de dades hereves també de les superclasses. Cal veure, però, que els mètodes descrits no són gaire útils a l'hora d'usar-los en una aplicació, ja que rarament es treballa amb bytes i malgrat que la conversió de dades bàsiques a vectors de bytes no és gaire complicada, afegeix una complexitat supèrflua que ens estalviaríem si les classes de fluxos incorporessin mètodes conversors.

Una solució fàcil podria ser incorporar aquestes utilitats a les superclasses (`InputStream`i `OutputStream`) de manera que s'heretessin en tota la jerarquia. Tot i l'aparent senzillesa de la solució, Java va decantar-se per una altra solució força més complexa, que li ha valgut en més d'una ocasió alguna que altra crítica, atès que obliga el programador a conèixer una extensa jerarquia de classes i realitzar diverses instanciacions per acabar obtenint un únic objecte *Stream*.

Fluxos decoradors de fluxos bàsics

La solució escollida es coneix tècnicament amb el nom de *decorator* usant la terminologia anglosaxona, és a dir, *decorador* (figura 2.3). Consisteix a embolcallar dos o més objectes, un dins l'altre, com si es tractessin de nines russes. Cada embolcall aporta una certa funcionalitat extra ("decora" l'objecte intern de forma diferent a l'original). D'aquesta manera és possible crear instàncies "a gust del consumidor", seleccionant només la funcionalitat que sigui necessària. També pot resultar més fàcil crear noves funcionalitats sense problemes de compatibilitat, ja que mai es modifiquen les classes originals, sinó que es crea un nou embolcall.

Figura 2.3. Representació gràfica que simbolitza el concepte "decorador"



Cada embolcall modifica la forma de l'anterior.

Veiem un exemple, suposem que partim d'un flux original de tipus `FileOutputStream` al qual li volem donar la capacitat de disposar de memòria intermèdia per agilitzar el procés de transferència de dades fent servir la classe `BufferedOutputStream`. A més, volem dotar al flux final de l'automatisme

de conversió de dades bàsiques a bytes. Farem servir `DataOutputStream`, i el procés de construcció d'una instància com aquesta seria:

```
...
FileOutputStream fos = new FileOutputStream(ruta);
BufferedOutputStream bos = new BufferedOutputStream(fos);
DataOutputStream streamFinal = new DataOutputStream(bos);
...
```

o bé optimitzant el nombre de variables,

```
...
DataOutputStream streamFinal = new DataOutputStream(
                                   new BufferedOutputStream(
                                       new FileOutputStream(ruta)));
...
```

Tots els decoradors de tipus byte formen part també de la jerarquia d'Streams. És a dir, els decoradors són també, a la vegada, *Streams* d'entrada o de sortida. Així podem definir l'ordre de construcció segons ens interessi.

A la figura 2.4 podeu veure un esquema de les jerarquies de fluxos d'entrada i sortida orientats a bytes. Observeu que la majoria de decoradors (color taronja) hereten de `FilterInputStrem` o `FilterOutputStream`,segons el cas. Probablement l'estructura interna de `ObjecteInputStream`/ `ObjectOutputStream`hagi obligat a descendir directament de les arrels corresponents, malgrat que es tracti també d'un *decorador* semblant als altres.

De forma breu, direm que `PrintStream` és un decorador amb utilitats destinades a la presentació de les dades (precisió en els decimals numèrics, format de les dates, etc.). Podríem dir que és una utilitat encarregada del format extern de les dades en contraposició a DataOutputStream, que s'encarrega de la representació interna de les dades (format intern).

`DataInputStream` i `BufferedInputStream` són el homòlegs de `DataOutputStream` i `BufferedOutputStream`, respectivament. La primera tindrà la funció de convertir els bytes del flux en dades de tipus bàsic de l'aplicació, i la segona suporta un *buffer* (memòria intermèdia) extra per als fluxos d'entrada.

`LineNumberInputStream` afegeix numeració a cada una de les línies arribades des del flux. És a dir, cada vegada que detecta un salt de línia incrementa el recompte i afegeix l'índex a la nova línia.

`PushBackInputStream`és un flux d'entrada que permet retrocedir un byte en el flux a mida que avança la lectura.

Figura 2.4. Jerarquia de les classes de Java que implementen fluxos orientats a bytes

A l'esquema hem pintat de color taronja les classes de tipus "decorador", de color blau les que no ho són i de color verd les classes abstractes que no s'han definit encara. Les classes abstractes s'han escrit en cursiva i es simbolitzen amb línies de punts.

Altres fluxos importants

Recordem que la interfície *serializable* no té declarat cap mètode, sinó que només serveix per marcar quins objectes es permeten seriar i quins no.

`ByteArrayOutpuStream` permet crear un flux de dades cap a la memòria RAM de l'aplicació i definir un *buffer* de memòria de tipus flux. De forma idèntica, `ByteArrayInputStream`defineix també un *buffer* de memòria però de tipus flux d'entrada, en el qual les dades es transfereixen del *buffer* a l'aplicació.

Veurem amb més detall les classes `ObjectInputStream` i `ObjectOutputStream` a la secció "Seriació d'objectes" de l'apartat "Persistència d'objectes en fitxers" d'aquesta mateixa unitat.

Volem destacar també `ObjectInputStream` i `ObjectOutputStream`, les quals permeten automatitzar la seriació de qualsevol objecte que implementi la interfície *java.io.Serializable* des de l'aplicació cap a l'*Stream* que decora (sortida), o bé des de l'*Stream* decorat cap a l'aplicació (entrada).

Finalment, cal indicar que `SequenceInputStream` no és pròpiament un Stream, sinó una utilitat per gestionar i concatenar múltiples Streams. Estrictament no es pot considerar com un decorador, ja que no modifica un únic objecte sinó que gestiona la seqüenciació de diversos objectes. Òbviament no existeix una classe homòloga de sortida, ja que implicaria la divisió d'un flux de dades en múltiples fluxos i la seva implementació seria probablement més complexa que l'ús directe de cada un dels fluxos que es necessitin.

Fluxos d'accés relatiu

Tots els fluxos que acabem de veure treballen de forma seqüencial i en una única direcció. Existeix, però, un flux orientat a bytes que permet l'accés relatiu a qualsevol part del seu contingut. Ens referim a `RandomAccesFile`, un flux especialment dissenyat per permetre accés no seqüencial (anomenat també relatiu o aleatori) dins un fitxer. És un flux bidireccional, és a dir, un flux que permet tant l'escriptura com la lectura, malgrat que es pot configurar com a flux unidireccional. És tracta d'una classe independent de les dues jerarquies estudiades fins ara i per tant no serà possible "*decorar-lo*" amb cap dels "*decoradors*" específics de les jerarquies anteriors. Tot i així, *RandomAccesFile* implementa tots els mètodes de `DataInputStreami` de `DataOutputStream`, de manera que no li cal l'embolcall per treballar amb els tipus de dades primitius. En certa forma, podríem dir que es comporta com un `DataStream` d'entrada i sortida amb utilitats extres adequades per gestionar l'accés relatiu. Així, per exemple, disposa del mètode `seek` per realitzar salts dins el fitxer i aconseguir moure el punter intern de lectura o escriptura, del mètode `getFilePointer` per obtenir la posició del punter en el moment de la petició, o del mètode `length` per determinar la mida total de l'arxiu.

Els constructors de la classe admeten sempre dos paràmetres. El primer indica la ruta del fitxer que es desitja obrir i el segon, el mode en què es desitja fer-ho (només lectura o lectura i escriptura). La ruta pot indicar-se mitjançant un `String` o bé usant una instància de `File`. El mode admetrà qualsevol de les següents combinacions: "r" indicarà que es desitja obrir en el mode de *només lectura* i "rw" indicarà que també serà possible escriure en el fitxer. A banda d'aquestes dues opcions, podem indicar "rwd" per forçar que tot el que escrivim en el flux es buidi en el fitxer de manera immediata.

2.2.2 Fluxos orientats a caràcters

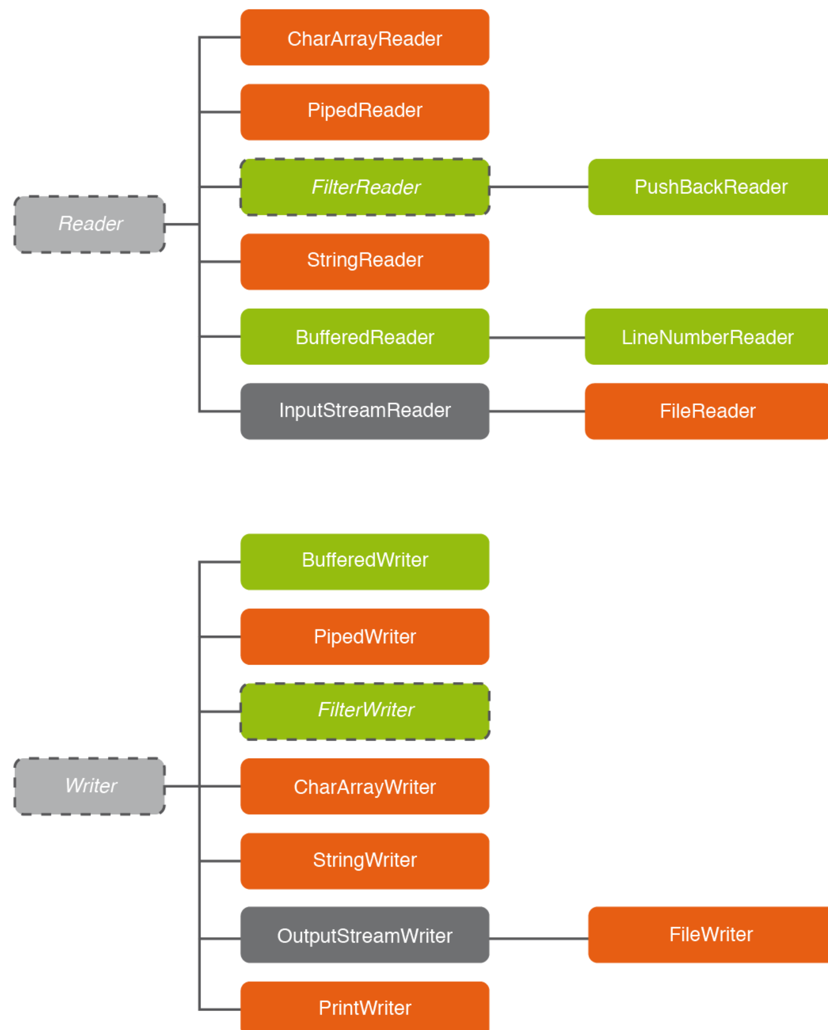
Treballar amb caràcters implica una gran dificultat, a causa sobretot de la diversitat de formats existents. Per poder solucionar-ho, Java disposa de dues jerarquies de classes semblants a les jerarquies de fluxos orientats a bytes.

Malgrat que l'estructura interna de les jerarquies no és ben bé igual en ambdós tipus de fluxos, les classes finalment implementades tenen característiques equivalents. Així, per exemple, `CharArrayReader/CharArrayWriter` són les equivalències de `ByteArrayInputStream/ ByteArrayOuputStream`, `PipedReader/ PipedWriter` es corresponen amb `PipedInputStream/ PipedOutputStream`, `BufferedReader/ BufferedWriter` les classes equivalents de `BufferedInputStream/ BufferedOutputStreamm`, i d'igual forma `FileReader/ FileWriterho` són de `FileInputStream/ FileOutputStream`, etc.

Com es pot veure a la figura 2.5, on es mostra la jerarquia dels fluxos orientats a caràcters, existeixen també "*decoradors*" de *Readers* i *Writers* que, de forma semblant als "*decoradors"* homòlegs orientats a bytes, aporten diferent funcionalitat

i poden combinar-se entre ells per tal d'obtenir diferents instàncies adaptades a cada necessitat.

Figura 2.5. Jerarquia de les classes de Java que implementen fluxos orientats a caràcters



A l'esquema hem pintat de color taronja les classes de tipus "decorador", de color blau les que no ho són i de color verd les classes abstractes que no s'han definit encara. La classe "InputStreamReader" i "OutputStreamWriter" juguen el paper "d'adaptadors" entre els fluxos orientats a bytes i els fluxos orientats a caràcters. Les classes abstractes s'han escrit en cursiva i es simbolitzen amb línies de punts.

Adaptadors de bytes a caràcters

Adaptadors

En programació orientada a objectes, les classes que aconsegueixen fer compatibles dues jerarquies independents s'anomenen adaptadors. Les classes InputStreamReader i OutputStreamWriter són d'aquest tipus perquè aconsegueixen adaptar qualsevol flux orientat a bytes i transformant-lo en un Reader o Writer segons el cas.

De la jerarquia de *Readers* o *Writers* destacarem la classe `InputStreamReader` i `OutputStreamWriter`. L'objectiu d'aquestes és transformar qualsevol flux orientat a bytes en un flux orientat a caràcters. La construcció d'aquestes instàncies és similar a la dels decoradors, els constructors esperen per paràmetre un InputSream o un OutputStream, segons es tracti d'una classe hereva de*Reader* o de *Writer,* i se'l guarden internament.

Cada cop que s'executi algun mètode d'introducció o extracció de contingut, la instància s'encarregarà de traduir la seqüència de text a una seqüència de bytes i passar-la a l'*Stream* original, o bé recollir la seqüència de bytes des de l'*Stream* original i convertir-la en seqüència de caràcters.

Són classes útils per a aplicacions que treballen amb fluxos de bytes externs, el contingut del qual són caràcters. A més, també permeten manipular fluxos de caràcters codificats en diferents formats.

En el constructor es pot indicar, a més del flux original, el tipus de codificació pertinent. Així, per exemple, si d'una font externa (fitxer, xarxa, etc.) obtinguéssim fluxos de caràcters amb una codificació diferent de la usada per Java (ISO-8859-1, per exemple), caldria crear un Reader o un Writer, passant-li el flux de bytes i indicant que la codificació usada és ISO-8859-1. Vegem-ne un exemple en el qual suposarem que tenim una font externa que disposa d'un mètode anomenat getInputStream i getOutputStream, els quals obtenen fluxos de bytes d'entrada i sortida de la font:

```
1  InputStreamReader reader;
2  OutputStreamWriter writer;
3  InputStream input = fonDadesEnt.getInputStream();
4  OutputStream output = fontDadesSort.getOutputStream();
5
6  reader=new InputStreamReader(input, “ISO”−8859−1);
7  writer=new OutputStreamWriter(output, “ISO”−8859−1);
8
9  ...
```

Les classes d'emmagatzematge intern, com ara `CharArrayReader`, `CharArrayWriter`, `StringReader`, `StringWriter`, `PipedReader`, `PipedWriter` usen sempre la codificació pròpia de Java (unicode de 16 bits), ja que emmagatzemen les dades a la memòria basant-se en els tipus dades de tractament de caràcters de Java (*char* i *String*).

Les classes `FileReader` o `FileWriter`agafen la codificació per defecte del sistema operatiu amfitrió. L'usuari no pot seleccionar diferents sistemes de codificació en crear les instàncies. Així, una màquina virtual Java que corri sobre Windows usarà, per defecte, la codificació ISO-8859-1, però si corre sobre Linux, la codificació serà UTF-8.

Tot i això, és possible llegir o escriure fitxers fent servir codificacions alternatives. Podem crear objectes `InputStreamReader` o `OutputStreamWriter` a partir de fluxos de bytes vinculats a fitxers i indicar la codificació alternativa adequada. Imaginem que tenim un fitxer codificat en ISO-8859-1 i volem llegir-lo des d'un sistema operatiu Linux. Caldrà fer:

```
1  FileInputStream in = null;
2  File fileIn = new File("/home/josep/tmp/fitxerProva.txt");
3  InputStreamReader reader = new InputStreamReader(
4                                      new FileInputStream(fileIn),
5                                      "ISO−8859−1");
6  int charsLlegits=0;
7  char[] buffer = new char[1000];
8  while(charsLlegits!=−1){
9      charsLlegits=reader.read(buffer);
10     ...
11 }
12 ...
```

Recodificació de fonts de caràcters

Amb l'ús d'aquesta tècnica és fàcil fer una utilitat que permeti transformar fluxos d'un format a un altre:

```
public void copiaDeChars(Reader reader, Writer writer)
                                                    throws IOException {
    int charsLlegits=0;
    char[] buffer = new char[1000];
    while(charsLlegits!=-1){
        writer.write(buffer, 0, charsLlegits);
        charsLlegits=reader.read(buffer);
    }
}

public void recodificar(InputStream input,
    String codificacioIn,
    OutputStream output,
    String codificacioOut) throws IOException {

    InputStreamReader reader = new InputStreamReader(input,
        codificacioIn);
    OutputStreamWriter writer = new OutputStreamWriter(output,
        codificacioOut);
    copiaDeChars(reader, writer);

    if (reader != null) {
        reader.close();
    }
    if (writer != null) {
        writer.close();
    }
}
```

A l'exemple, `copiaDeChars` fa una còpia del contingut d'un objecte `Reader` a un objecte Writer. La funció *recodificar* obté per paràmetre el flux origen (*input*) i la codificació usada en aquest magatzem (`codificacioIn`). També per paràmetre se li indicarà el flux de sortida (*output*) on es desitja recodificar el contingut original i la codificació amb la qual es desitja tornar a emmagatzemar (`codificacioOut`). Amb els dos primers paràmetres s'instanciarà un *Reader* per llegir el contingut fent servir la codificació original. Amb els dos darrers paràmetres es podrà instanciar un Writer adaptat a la codificaciódesitjada. Tant el *Reader* com el *Writer* instanciats es passaran a `copiaDeChars` per realitzar la còpia del primer en el segon i aconseguir així la recodificació del contingut copiat.

2.2.3 Implementació d'utilitats

Arribats a aquest punt, ja estem en disposició de crear un conjunt d'utilitats que ens serviran de base per a l'aplicació pràctica que realitzarem més endavant. Aquests mètodes els agruparem a la classe `Utilitats`per tal de tenir-los tots disponibles amb una única instància.

Com ja s'ha vist, per manipular els fluxos de dades és necessari definir un *buffer* d'intercanvi o vector. Normalment treballarem amb vectors de bytes excepte en el cas de treballar amb *readers* o *writers,* atès que aquests ens permeten treballar directament amb vectors de *chars*.

Per tal de poder-nos adaptar a les diferents necessitats segons la mida del fitxer de treball, definirem un atribut que ens indicarà la mida màxima en bytes del *buffer* d'intercanvi. Per defecte el definirem de 1/2 Giga, però implementarem un constructor específic per canviar-ne el valor.

```
public class Utilitats {
    private int capacitatMaximaBufferEnBytes = 524288;

    public Utilitats() {
    }

    public Utilitats(int midaBuffer) {
        capacitatMaximaBufferEnBytes=midaBuffer;
    }
    ...
```

La primera utilitat que implementarem serà la còpia de fluxos, tant de bytes com de caràcters.

```
public void copiaDeBytes(InputStream input, OutputStream output)
    throws IOException{
    int bytesLlegits=0;
    byte[] buffer = new byte[capacitatMaximaBufferEnBytes];
    while(bytesLlegits!=-1){
        output.write(buffer, 0, bytesLlegits);
        bytesLlegits=input.read(buffer);
    }
}

public void copiaDeChars(Reader reader, Writer writer)
                                                throws IOException {
    int charsLlegits=0;
    char[] buffer = new char[midaBytesACaracters(
                                    capacitatMaximaBufferEnBytes
                            )];
    while(charsLlegits!=-1){
        writer.write(buffer, 0, charsLlegits);
        charsLlegits=reader.read(buffer);
    }
}
```

Fixeu-vos que per determinar la mida del vector de caràcters (*buffer* d'intercanvi) sense que sobrepassi la capacitat definida per`capacitatMaximaBufferEnBytes`caldrà dividir per 2 aquesta quantitat. En lloc de fer servir l'operació *divisió*, usarem l'operació de *desplaçament de bits* perquè és una operació més eficient.

```
public long midaBytesACaracters(long bytes){
    return (bytes>>1);
}

public long midaCaractersABytes(long caracters){
    return (caracters<<1);
}
```

Desplaçament de bits

L'operació de desplaçament de bits és una operació que el processador pot realitzar de forma molt més ràpida que la pròpia operació de divisió o multiplicació. Aprofitem la característica que en els tipus de dades int o long un desplaçament de bits a la dreta divideix el número per 2 i un desplaçament a l'esquerra multiplica per 2. Així, 24 >> 1 donarà com a resultat 12. Mentre que 12 << 1 és equivalent a 24.

A continuació, usant les utilitats de còpia que acabem d'implementar, en crearem una de nova per copiar fitxers, tant a nivell de bytes com de caràcters.

```
public void copiaFitxersDeBytes(File origen, File desti)
                                                throws IOException{
    FileInputStream input = null;
    FileOutputStream output = null;
```

```
    try {
        input = new FileInputStream(origen);
        output = new FileOutputStream(desti);
        copiaDeBytes(input, output);
    } finally {
        input.close();
        output.close();
    }
}

public void copiaFitxersDeCaracters(File origen, File desti)
    throws IOException {
    FileReader reader = null;
    FileWriter writer = null;
    try {
        reader = new FileReader(origen);
        writer = new FileWriter(desti);
        copiaDeChars(reader, writer);
    } finally {
        reader.close();
        writer.close();
    }
}
```

Cal anar amb compte de tancar sempre els fitxers en acabar la còpia, per tal de no malbaratar recursos del sistema. Per això és important posar el tancament dins la sentència *finally*. D'aquesta manera, assegurem que malgrat que durant la lectura es produís una excepció, abans d'abandonar el mètode a la cerca d'alguna sentència *catch* que capturi l'error, s'executarà obligatòriament el tancament.

El tancament de fitxers, però, presenta certs problemes que exposarem a continuació, intentant esbossar alguna solució adequada. Quan cal tancar més d'un fitxer, malgrat que les sentències de tancament es trobin dins el *finally*, si una de les sentències falla provocant una excepció (degut al fet que es tracta d'un fitxer inexistent o que el sistema ha esborrat abans del tancament, o qualsevol altre motiu), les sentències de tancament situades immediatament després de la que ha provocat l'error no s'executarien i es malbaratarien recursos del sistema.

La solució ideal és molt pesada d'implementar, ja que ens obligaria a crear sentències *try/finally* imbricades per cada fitxer a tancar de manera que s'asseguri el tancament de tots els fitxers a tancar.

```
...
} finally {
    try{
        f1.close();
    }finally{
        try{
            f2.close();
        }finally{
            f3.close();
            ...
        }
    }
}
```

Hi ha, però, una solució intermèdia, que malgrat que no és ideal simplifica força la implementació i assegura el tancament de tots aquells fitxers que puguin tancar-se. Es tracta de capturar les excepcions degudes als tancaments i deixar-les sense reportar. Per aconseguir-ho, caldrà fer el tancament en un mètode independent que capturi l'excepció.

```
public void intentarTancar(Closeable aTancar){
    try {
        if (aTancar != null) {
            aTancar.close();
        }
    } catch (IOException ex) {}
}
```

D'aquesta manera, aconseguirem que, malgrat que es produeixi una error en el tancament d'algun fitxer, es continuïn executant la resta de tancaments sense complicar el codi.

```
...

} finally {
    intentarTancar(f1);
    intentarTancar(f2);
    intentarTancar(f3);
    ...
    intentarTancar(fn);
}
```

Òbviament, aquest sistema presenta el problema que l'error podria passar inadvertit. Per això caldria afegir algun sistema d'enregistrament dels errors, com ara els fitxers *log*. El que es pretén amb això és que el sistema d'errors sigui consistent, però sense caure en una rigidesa que dificulti la implementació i acabi encarint el producte.

Si no usem cap sistema d'enregistrament *log* podem fer servir la sortida d'error estàndard per informar dels possibles errors en els tancaments de fitxers. Així, finalment, el mètode `copiaFitxersDeBytes`, `copiaFitxersDeCaracters`i`intentarTancar`quedaran com es mostra a continuació:

```
public void copiaFitxersDeBytes(File origen, File desti)
    throws IOException{
    FileInputStream input = null;
    FileOutputStream output = null;
    try {
        input = new FileInputStream(origen);
        output = new FileOutputStream(desti);
        copiaDeBytes(input, output);
    } finally {
        intentarTancar(input);
        intentarTancar(output);
    }
}

public void copiaFitxersDeCaracters(File origen, File desti)
    throws IOException{
    FileReader reader = null;
    FileWriter writer = null;
    try {
        reader = new FileReader(origen);
        writer = new FileWriter(desti);
        copiaDeChars(reader, writer);
    } finally {
        intentarTancar(reader);
        intentarTancar(writer);
    }
}

```

```
29 public void intentarTancar(Closeable aTancar){
30     try {
31         if (aTancar != null) {
32             aTancar.close();
33         }
34     } catch (IOException ex) {
35         ex.printStackTrace(System.err);
36     }
37 }
```

Segur que en força ocasions necessitarem escriure una cadena `String` en un fitxer o a l'inrevés, convertir en una cadena `String` el contingut d'un fitxer de text.

```
1  public void copiarStringAWriter(String text, Writer writer)
2      throws IOException{
3      int length = text.length();
4      int offset = 0;
5      while(offset<length){
6          int charsAEscriure =
7             Math.min(midaBytesACaracters(capacitatMaximaBufferEnBytes),
8             length − offset);
9      writer.write(text, offset, charsAEscriure);
10     offset+=charsAEscriure;
11     }
12 }
13
14 public String copiarReaderAString(Reader reader) throws IOException{
15     StringBuilder stringBuilder = new StringBuilder();
16     int charsLlegits=0;
17     char[] buffer =
18         new char[midaBytesACaracters(capacitatMaximaBufferEnBytes)];
19     while(charsLlegits!=−1){
20         stringBuilder.append(buffer, 0, charsLlegits);
21         charsLlegits=reader.read(buffer);
22     }
23     return stringBuilder.toString();
24 }
```

Del primer mètode (`copiarStringAWriter`) destacarem que cal comprovar si és més gran el text a escriure que no pas el *buffer* d'intercanvi. En cas afirmatiu, caldrà escriure el text en diverses passades.

Del segon mètode (`copiarReaderAString`), cal explicar que `StringBuilder` és una classe especialment concebuda per concatenar cadenes de forma més eficient que l'operació suma. D'altra banda, el mètode `read` intenta llegir el nombre de caràcters que caben en el *buffer*, però no hi ha garantia que ho aconsegueixi; per això, cal assegurar que el fitxer es llegeix íntegrament, moment en el qual el mètode *read* retorni el valor -1.

Abans de continuar amb la implementació presentarem les innovadores classes de fitxers Java que s'han incorporat amb força a les darreres versions.

2.3 Fluxos eficients: Channels i Buffers

L'acrònim *nio* prové de *new input/output*.

Una de les crítiques constants que ha rebut Java en relació amb la biblioteca d'entrada i sortida és la seva baixa eficiència. Des de la versió 1.4 es va introduir

un nova biblioteca i es va reestructurar l'antiga per tal de guanyar velocitat en l'intercanvi de dades en els processos d'entrada i sortida. La biblioteca que dóna major eficiència es troba en el paquet *java.nio* i se sustenta bàsicament sobre dues jerarquies principals, les classes que implementen la interfície `Channel` i la jerarquia que hereta de *Buffer*. La reestructuració de la biblioteca antiga implica només canvis interns: s'han substituït estructures antigues per les classes del paquet *nio* a fi d'incrementar l'eficiència de totes les classes d'entrada i sortida Java.

2.3.1 Conceptes

Channel té el paper de connector a la font de dades, però no en el sentit dels fluxos, sinó més aviat com la porta d'accés al magatzem de dades. La missió d'un *buffer* seria doble. D'una banda, la d'introduir o extreure informació d'un *Channel*, i de l'altra, la de dotar de totes aquelles utilitats necessàries per gestionar còmodament l'intercanvi de dades des del punt de vista del programador (utilitat de conversió dels tipus bàsics, seriació i deseriació d'objectes, suport de diversos formats d'emmagatzematge, gestió del flux, etc.).

La importància del paquet *nio*, però, se sustenta en el fet que tant les implementacions de *Channel* com *Buffer* i els seu derivats utilitzen internament utilitats molt eficients, pròpies de cada sistema operatiu. És a dir, que per exemple la classe `FileChannel` o la classe `ByteBuffer` podrien tenir rendiments diferents en diferents sistemes operatius, ja que cada màquina virtual específica disposa del seu paquet *nio* reescrit i adaptat a les característiques de la plataforma on correrà.

2.3.2 Instanciació d'un Channel

En la versió 1.6 de Java, els Channels no es poden instanciar usant la típica instrucció *new*. Per obtenir un `Channel` és necessari que un altre objecte l'instanciï i ens el retorni en executar algun dels seus mètodes. L'objecte instanciador se sol conèixer amb el nom de *factory* quan la seva funció principal es limita a "fabricar" instàncies.

Es tracta d'una tècnica força usada en programació orientada a objectes, ja sigui per independitzar interfícies o classes abstractes, de les classes finals que les implementin o per restringir i gestionar totes les possibles instàncies generades en una aplicació. En el cas que ens ocupa es compleixen ambdós requisits; desconeixem la classe final, ja que com hem dit en darrer terme depèn del sistema operatiu i, a més, cal evitar, per raons d'eficiència i coherència de dades, que es multipliquin el nombre de canals associats a una mateixa font de dades.

En el context dels fitxers, el paper de *fabricants de canals* és assumit pels objectes de la jerarquia *Stream* Com ja s'ha comentat, la biblioteca *java.io* s'ha reestructurat

de manera que internament es fan servir `Channels` específics, concretament instàncies de la classe `FileChannel`. Des de la versió 1.4, les classes d'entrada i sortida disposen d'un mètode anomenat `getChannel` que retorna el canal usat internament.

Per obtenir un `FileChannel`, doncs, necessitarem sempre la instància de l'*Stream* corresponent, i si no la tinguéssim, hauríem de crear-la. Vegeu tot seguit com obtenir un `FileChannel` a partir d'un `FileInputStream` ja creat:

```
FileInputStream istream;
...
FileChannel channel = istream.getChannel();
...
```

Vegeu ara com obtenir un `FileChannel` a partir d'un `FileInputStream` que no necessitem:

```
FileChannel channel = new FileInputStream(ruta).getChannel();
...
```

Per introduir o treure dades d'un canal ens cal usar un *Buffer*. La introducció es farà executant algun dels mètodes *read*, i l'extracció, usant `write`. A més, `FileChannel` disposa de mètodes per realitzar transferències massives de dades entre canals, de mètodes de posicionament dins el fitxer i de mètodes per gestionar el bloqueig de fitxers.

2.3.3 Instanciació d'un Buffer

ByteBuffer segueix la mateixa tècnica constructora que `FileChannel`, és a dir, no disposa de constructor públic i per tant precisa d'una classe instanciadora que faci el paper de *factory*. En aquest cas, és la pròpia classe `ByteBuffer`la que pren aquest paper per tal de poder servir la classe específica del sistema operatiu amfitrió. Disposa de dos mètodes instanciadors. Són mètodes estàtics anomenats `allocate` i `allocateDirect`.

```
ByteBuffer byteBufffer = ByteBuffer.allocate(1024);
...
ByteBuffer byteBufffer = ByteBuffer.allocateDirect(1024);
```

La principal diferència entre `allocate` i `allocateDirect` és que el segon genera instàncies que estan molt més lligades al sistema operatiu. En funció del sistema amfitrió, la instància aconseguida amb `allocateDirec` pot arribar a ser molt eficient, però sempre gastarà més recursos de memòria i processament que la instància aconseguida amb *allocate*.

2.3.4 Buffers de bytes

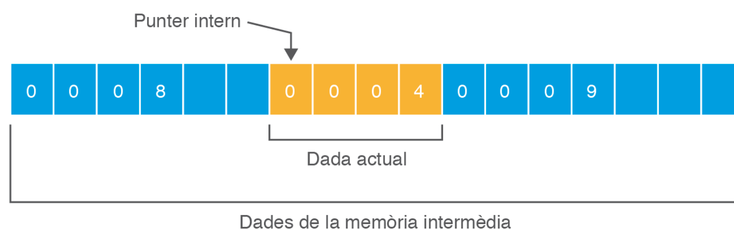
La principal classe de tipus `Buffer` responsable de l'intercanvi és `ByteBuffer`. S'anomena així perquè internament les dades estan representades en bytes. Tot i això, cal dir que aquesta classe ja disposa de mètodes conversors per a tots els tipus bàsics de Java, tant de lectura (`getInt`, `getFloat`, `getDouble` o `getChar`) com d'escriptura (`putInt`, `putFloat`, `putDouble` o `putChar`).

Accés absolut versus accés relatiu al buffer

Tant els mètodes d'escriptura com els de lectura admeten un accés relatiu o absolut. Internament, el *buffer* incorpora un punter a les dades que avança a mida que es llegeix o s'escriu (usant el mètode `getXXX` o `putXXX`), però és possible escriure o llegir una posició determinada del *buffer* indicant-la com a paràmetre. En aquest cas parlem de lectura o escriptura absoluta, ja que l'acció es realitza a la posició indicada. A més, cap de les accions absolutes incrementen el punter intern.

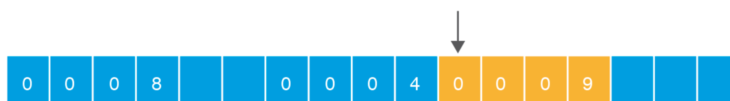
Sigui l'esquema de la figura 2.6, la representació d'un *buffer* amb el punter intern assenyalant un enter (int).

Figura 2.6. Buffer amb el punter intern assenyalant un enter



Si sobre aquest *buffer* executem `getInt()` ens retornarà el valor 4 i mourà el punter intern fins a la següent dada (figura 2.7).

Figura 2.7. Resultat d'executar getInt()



En canvi, si executem `getInt(0)`, ens retornarà el valor 8, però el punter intern no es mourà. De manera que, si seguidament tornem a executar `getInt()`, obtindrem el valor 9.

Quelcom de semblant passa amb els mètodes *put* l'execució de `putDouble(3.5)`: escriuria 8 bytes a partir de la posició del punter intern i el desplaçaria al byte situat a la posició ubicada just després del darrer byte escrit. Per contra, `putDouble(4, 9.5)` escriuria 8 bytes començant en el 5è byte del *buffer*, però no mouria el punter intern.

Vegeu a continuació dos exemples equivalents d'escriptura relativa i absoluta respectivament:

```
1  ...
2  ByteBuffer byteBuffer = ByteBuffer.allocate(MIDA_BYTES);
3
```

```
...
for (int i = 0; i< (MIDA_BYTES/BYTES_PER_INT); i++){
    byteBuffer.putInt(i);
}

...
for (int i = 0; i< MIDA_BYTES; i+=BYTES_PER_INT){
    byteBuffer.putInt(i, i/BYTES_PER_INT);
}
...
```

2.3.5 Buffers de tipus específics

En cas de treballar amb un únic tipus de dada primitiu, és possible obtenir, a partir d'una instància de `ByteBuffer`, diversos *buffers* específics per a un tipus de dada concret. És a dir, `ByteBuffer` actuaria de `Factory` generant instàncies específiques per treballar amb dades de tipus `char`, `long`, `double`, etc.

```
ByteBuffer bufferOriginal = ByteBuffer.allocate(MIDA_MAXIMA);

...
LongBuffer bufferDeLongs = bufferOriginal.asLongBuffer();

...
CharBuffer bufferDeChars = bufferOriginal.asCharBuffer();

...
DoubleBuffer bufferDeDoubles = bufferOriginal.asDoubleBuffer();

...
```

El principal avantatge de treballar amb *buffers* específics consisteix en el fet que els mètodes d'assignació i obtenció de dades genèriques (*get* i *put*) són específics per al tipus de dada concreta de treball del *buffer*. Així, el mètode `get` d'un `LongBuffer` retornarà *longs*, mentre que el d'un `DoubleBuffer`retornarà *doubles*. De la mateixa manera, el mètode *put* d'un *ShortBuffer* acceptarà només *shorts*, i el d'un `FloatBuffer`, per contra, acceptarà només *floats*.

Malgrat que seria possible crear directament un *buffer* especific usant el mètode instanciador anomenat *allocate*, el més comú és obtenir les instàncies a partir d'un ByteBuffer. La raó és ben senzilla: totes les instàncies generades per un mateix *ByteBuffer* comparteixen les mateixes dades en la mateixa ubicació de memòria, fet que significa que qualsevol canvi en les dades d'algun d'ells repercuteix també en les dades de tots els altres.

Feu la prova. Instancieu tres *buffers*: un de tipus *ByteBuffer*, un segon de tipus *ShortBuffer* i el darrer de tipus *CharBuffer*. Implementeu un algoritme que escrigui, usant el *buffer* de tipus *short,*tots els valors compresos entre *32* i *127*. Seguidament, feu que l'algorisme recuperi els valor assignats, però fent servir el *buffer* de tipus *char*. Observareu que, malgrat que s'hagin entrat valors numèrics, recuperareu tots els caràcters compresos entre el codi *32* i el codi *127*.

Cal destacar que els *buffers* comparteixen només les dades, no pas els seus estats interns. És a dir, cada *buffer* manté de forma independent la seva posició, les seves marques internes, etc. Vegeu-ho en el següent exemple, una variant de la prova que us hem proposat on es van recuperant de forma independent les dades de cada *buffer*:

```
ByteBuffer buffer = ByteBuffer.allocate(500);
CharBuffer charBuffer = buffer.asCharBuffer();
ShortBuffer shortBuffer = buffer.asShortBuffer();

//clear, neteja el buffer de qualsevol dada que pogués haver−hi
shortBuffer.clear();

for(short value=32;value<128; value++){
    shortBuffer.put(value); //Assignem cada valor usant put
}

/* flip, marca la posició actual com el limit de dades entrades
 * i endarrereix la posició actual del buffer a la posició
 * inicial. En el nostre cas marcarà la posició 94 com a límit
 * de les dades i es situarà a la posició zero*/
shortBuffer.flip();

/* Com que es tracta de buffers independents cal indicar al
 * buffer de caràcters que el seu limit és el mateix que el del
 * shortBuffer. EL mètode limit, retorna la posició límit si no
 * se li passa cap paràmetre, però assigna com a nou límit, el
 * valor passat per paràmetre. Així, el valor retornat per
 * limit() del shortBuffer es assignat com a limit en el
 * charBuffer. */
charBuffer.limit(shortBuffer.limit());

System.out.println("Cars:");
int i=0;

/* Volem escriure matriu (de 6x32) de correspondència entre cada
 * caràcter i el seu codi, de manera que a les línies senars
 * aparegui el valor del codi i la les línies parells el
 * caràcter corresponent aliniat correctament.*/
while(i<charBuffer.limit()){
    for(int j=0; j<32; j++){
        //shortBuffer avança 32 posicions, però no pas charBuffer
        System.out.printf("%4d", shortBuffer.get());
    }
    System.out.println();
    for(int j=0; j<32; j++){
        //charBuffer avança 32 posicions, però no pas shortBuffer
        System.out.printf("%4c", charBuffer.get());
        i++;
    }
    System.out.println();
}
System.out.println();
```

2.3.6 Treball combinat de Buffers i Channels

Quan es fan treballar conjuntament *buffers* i *channels*, cal tenir en compte que cada un d'ells disposa d'un cursor o punter de posicionament propi i independent. El procés de lectura o escriptura intenta arribar, sempre que sigui possible, fins al final del *buffer*. Per exemple, mireu la figura 2.8 si la taula més gran (color

groc) representés un fitxer amb el seu punter posicionat al byte 11, i la taula més petita (color blau) representés un *buffer* amb el seu punter posicionat al byte 9, el procés de lectura del `Channel`, mètode `read`, aconseguiria escriure, en absència de bloquejos, la paraula "socials" en el *buffer* i omplir-lo totalment des de la posició 9 fins al final. La posició del `FileChannel` avançarà també fins al byte 19.

FIGURA 2.8. Treball combinat de Buffers i Channels

Cal tenir en compte que els bloquejos dels fitxers poden alterar aquest comportament. De fet, tant el procés de lectura com el d'escriptura s'interrompran en intentar operar en alguna zona bloquejada del fitxer. Per assegurar l'escriptura sencera del *buffer* caldria controlar que el procés arribi al final.

```
1  public void escriu(ByteBuffer buffer, FileChannel channel)
2                                                  throws IOException{
3      //Mentre no s'hagi escrit tot el contingut del buffer...
4      while(buffer.position()<buffer.limit()){
5          channel.write(buffer);
6      }
7  }
```

De forma semblant caldrà controlar també la lectura, però en aquest cas cal tenir en compte que la lectura parcial pot ser deguda al fet que s'ha arribat al final del fitxer. Aprofitarem el fet que el procés de lectura retorna el valor -1 quan s'intenta llegir més enllà del límit del fitxer.

```
1  public void llegeix(int mida, ByteBuffer buffer, FileChannel channel)
2      throws IOException{
3      int llegit = 0;
4      int totalLLegit=0;
5      //Mentre no s'hagi llegit la quantitat desitjada...
6      while(llegit!=-1 && totalLLegit<mida){
7          llegit = channel.read(buffer);
8          totalLLegit+=llegit;
9      }
10 }
```

La utilitat anterior llegirà el contingut del fitxer des de la posició del cursor del `Channel` fins que s'hagin llegit el nombre de bytes indicats pel paràmetre mida, o bé fins que s'hagi arribat al final del fitxer, cas en què la variable llegida prendrà el valor -1.

Cal tenir en compte que `FileChannel` facilita també una versió absoluta del procés de lectura i del d'escriptura. És a dir, una versió on s'ignora la posició del

cursor del Channel, ja que es començarà a operar a partir de la posició indicada per paràmetre.

```
public void escriu(long offsetFitxer, ByteBuffer buffer,
                        FileChannel channel) throws IOException{
    int escrit = 0;
    //Mentre no s'hagi escrit tot el contingut del buffer...
    while(escrit<buffer.limit()){
        escrit += channel.write(buffer, offsetFitxer+escrit);
    }
}

public void llegeix(long offsetFitxer, int mida, ByteBuffer buffer,
    FileChannel channel) throws IOException{
    int llegit = 0;
    int totalLLegit=0;
    //Mentre no s'hagi llegit la quantitat desitjada...
    while(llegit!=-1 && totalLLegit<mida){
        llegit = channel.read(buffer, offsetFitxer+totalLLegit);
        totalLLegit+=llegit;
    }
}
```

La diferència entre aquests dos mètodes i els dos anteriors és que, en el cas d'*escriu,* aquesta implementació emmagatzema el contingut del *buffer* a la posició indicada per *offsetFitxer* sense canviar la posició interna del cursor del `FileChannel`. En canvi, l'anterior emmagatzemava el contingut a la posició del cursor del `FileChannel` i en acabar, el cursor avançava la quantitat de bytes emmagatzemada.

Quelcom de semblant passa amb *llegeix*. La lectura en la darrera implementació s'inicia a la posició indicada per *offsetFitxer*, i, com és de suposar, el cursor del `FileChannel` no es mou de lloc.

2.3.7 Transferència directa

Encara hi ha una operació més del paquet *nio* que val la pena esmentar. Es tracta de la transferència directa de dades entre *Channels*. Són dues operacions equivalents anomenades `transferFromo` `transferTo`, que aprofiten al màxim la capacitat de còpia massiva dels sistemes operatius. Són operacions en què es realitza una connexió directa a través dels recursos específics del sistema operatiu. Aquests mètodes no fan servir *buffers* gestionats des de Java, sinó que es delega directament sobre les eines externes. Aquestes eines són ideals per realitzar còpies massives entre fitxers quan la màquina virtual corri en sistemes potents. En cas contrari, no se'n treu rendiment.

Acabarem de completar la classe `Utilitats` que hem anat implementant afegint un mètode de còpia massiva. Tant el mètode `transferTocom` `transfetFrom` són sensibles als fitxers bloquejats i poden interrompre la còpia sense haver acabat si es troben un bloqueig durant l'execució. Caldrà assegurar la còpia sencera del fitxer usant un bucle adequat. En el codi podeu veure aquesta instrucció ressaltada.

```java
public void copiaFitxersDeBytesNio(File origen, File desti)
    throws IOException {
    long size;
    long count = 0;
    FileChannel input = null;
    FileChannel output = null;
    try {
        //Obrim el canal origen
        input = new FileInputStream(origen).getChannel();
        //Obrim el canal destí
        output = new FileOutputStream(desti).getChannel();
        //Calculem la mida
        size = input.size();
        //Mentre no s'hagi copiat tot el fitxer...
        while (count < size) {
            count += output.transferFrom(input, 0, size - count);
        }
    } finally {
        //Tancar els canals
        intentarTancar(input);
        intentarTancar(output);
    }
}
```

2.4 Estudi pràctic de la funcionalitat d'entrades i sortides de dades bàsiques

En aquest apartat continuarem la implementació del gestor de fitxers. Ara afegirem al gestor la capacitat de moure o copiar fitxers d'una ubicació a una altra, així com la capacitat d'editar el contingut de fitxers de text. La sintaxi de la nova funcionalitat es troba descrita a la interfície `GestioFitxersPlus` de la biblioteca que se us proporciona amb el material del mòdul. Es tracta d'un complement de la interfície `GestioFitxers`. És a dir, que hereta d'ella, i per tant la classe que la implementi haurà d'implementar tots els mètodes de `GestioFitxersPlus`, a més dels de `GestioFitxers`.

Podem reutilitzar la classe `GestorFitxersImple`, que ja hem implementat. La millor forma d'aconseguir-ho és heretant, de manera que només hem de codificar els nous mètodes declarats a `GestioFitxersPlus`.

Començarem per la declaració i els atributs. Com que usarem algunes de les utilitats implementades a la classe `Utilitats`, hi declararem un atribut d'aquest tipus.

```java
public class GestioFitxersPlusImpl extends GestioFitxersImpl
    implements GestioFitxersPlus {
    private Utilitats utilitats=new Utilitats();
    ...
```

De forma semblant a la vegada anterior, deixeu que NetBeans generi l'esquelet de la classe clicant la bombeta del marge esquerre.

2.4.1 Còpia i trasllat de fitxers

Anem a implementar ara el mètode de trasllat de fitxers anomenat *moure*. La implementació consistirà en realitzar una còpia del fitxer origen a la ubicació de destí i a continuació eliminarem el fitxer origen. Per fer la còpia usarem la funció que hem implementat a la classe `Utilitats`.

Abans d'iniciar la còpia caldrà comprovar que el fitxer destí no existeixi (de forma que mai es pugui perdre el contingut de cap fitxer existent) i que el fitxer origen sigui accessible. Per tal de simplificar el codi del mètode crearem dues funcions privades que realitzaran aquesta feina. El mètode `TestejarExistencia` rep dos paràmetres: el fitxer a testejar i el *test* a realitzar. Un *test* amb valor *cert* significarà que és necessari que el fitxer existeixi per poder continuar, mentre que un *test* amb valor *fals* el que indicarà és que no hi hauria d'haver cap fitxer amb aquell nom.

El mètode `TestejarAccés`, de forma semblant, comprova que el fitxer passat per paràmetre existeix i es troba accessible per ser llegit.

Els controls es fan llançant una excepció en cas que no es passi el test. Com que les crides als controls es posen a l'inici del mètode, abans de començar cap acció específica, el llançament de l'excepció impedirà l'execució del codi restant: còpia i eliminació i actualització del nou contingut per tal de reflectir els canvis a la pantalla.

```
1  @Override
2  public void moure(File origen, File desti)
3                                          throws GestioFitxersException{
4      TestejarExistencia(desti, false);
5      TestejarAcces(origen);
6      try {
7          utilitats.copiaFitxersDeBytesNio(origen, desti);
8          origen.delete();
9      } catch (IOException ex) {
10         throw new GestioFitxersException("S'ha produit un error "
11             + "d'entrada o sortida: '" + ex.getMessage() + "'", ex);
12     }
13     actualitza();
14 }
15
16 private void TestejarExistencia(File file, boolean test)
17     throws GestioFitxersException{
18     //es controla l'existència del fitxer d'acord amb el test
19     if(file.exists()!=test){
20         if(test){
21             throw new GestioFitxersException("Error. el fitxer "
22                 + file.getAbsolutePath()+ " no existeix");
23         }else{
24             throw new GestioFitxersException("Error. el fitxer "
25                 + file.getAbsolutePath()+ " ja existeix");
26         }
27     }
28 }
29
30 private void TestejarAcces(File file) throws GestioFitxersException{
31     //es comprova que existeixi el fitxer
32     TestejarExistencia(file, true);
33     //es controla que es tinguin permisos per llegir la carpeta
34     if(!file.canRead()){
35         throw new GestioFitxersException("Alerta. No es pot llegir "
```

```
                + file.getAbsolutePath() + ". No teniu prou permisos");
    }
}
```

Potser us preguntareu per què cal fer una còpia i una eliminació si reanomenant el fitxer podem aconseguir un efecte semblant. El problema és que aquesta afirmació només és certa si el trasllat es fa dins d'un mateix dispositiu, però si el trasllat implica diferents dispositius cal fer una còpia física del fitxer i una posterior eliminació de l'origen.

El mètode usat per copiar el contingut d'un fitxer (que hem anomenat *còpia*), té força semblança amb l'anterior (que havíem anomenat *moure*). Es diferencien en que la còpia no realitza el testeig per comprovar si el fitxer destí existeixi, sinó que es crea sempre. Per descomptat a la còpia no elimina mai el fitxer original.

```
@Override
public void copiar(File origen, File desti)
                                      throws GestioFitxersException{
    TestejarAcces(origen);
    crearSiCal(desti);
    try {
        utilitats.copiaFitxersDeBytesNio(origen, desti);
    } catch (IOException ex) {
        throw new GestioFitxersException("S'ha produit un error "
            + "d'entrada o sortida: '" + ex.getMessage() + "'", ex);
    }
    actualitza();
}

private void crearSiCal(File file) throws GestioFitxersException{
    if(!file.exists()){
        File pare = file.getParentFile();
        if(!pare.exists()){
            pare.mkdirs();
        }
        try {
            if(!file.createNewFile()){
               throw new GestioFitxersException("Error. No s'ha pogut"
                    + " crear " + file.getAbsolutePath() + ".");
            }
        } catch (IOException ex) {
            throw new GestioFitxersException("S'ha produit un error "
              + "d'entrada o sortida: '" + ex.getMessage() + "'", ex);
        }
    }
}
```

La utilitat `creaSiCal` assegura l'existència del fitxer, bé perquè ja hi era, bé perquè es crea en el moment de l'execució. Per tal que la creació es pugui dur a terme sense problemes,es comprova l'existència de la ruta, la qual també es crearia si fos necessari. Si, malgrat tot, la creació del fitxer no acabés amb èxit, es passaria a llançar una excepció per impedir que es continuï executant l'acció de còpia.

Seguint amb els requeriments de la interfície `GestioFitxersPlus`, implementarem dues utilitats més. La primera permet obtenir un objecte File corresponent al nom passat per paràmetre (el nom serà relatiu a la carpeta de treball del gestor). La segona utilitat permetrà obtenir un nom de fitxer, assegurant que es tracta d'un nom inexistent encara en el sistema. El nou nom tindrà com a prefix la cadena passada per paràmetre concatenada a un sufix numèric abans de l'extensió.

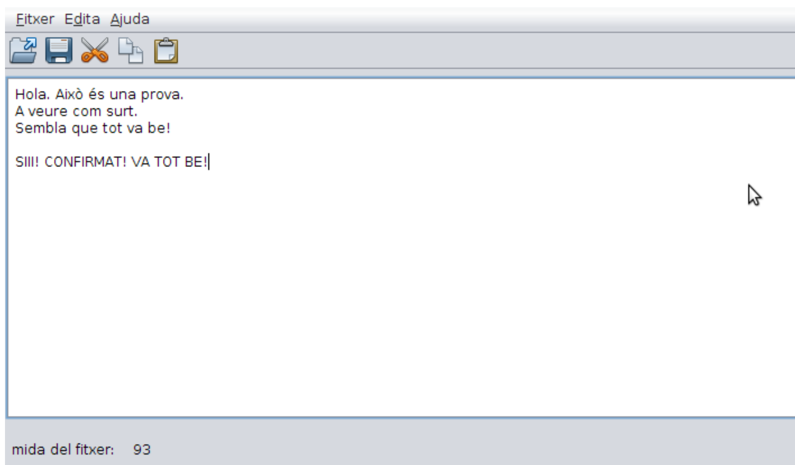
```
@Override
public File getFile(String relativeName) {
    File ret = new File(carpetaDeTreball, relativeName);
    return ret;
}

@Override
public String getNouNom(String relativeName) {
    int cont=1;
    File file = new File(carpetaDeTreball, relativeName);
    String[] nom = new String[2];
    int indPunt = relativeName.lastIndexOf('.');
    nom[0] = relativeName.substring(0, indPunt);
    nom[1] = relativeName.substring(indPunt+1);
    while(file.exists()){
        if(nom[1].length()>0){
           file = new File(carpetaDeTreball, nom[0] + "("+ cont +")"
                + "." + nom[1]);
        }else{
           file = new File(carpetaDeTreball, nom[0] + "("+ cont +")");
        }
        cont++;
    }
    return file.getName();
}
```

2.4.2 Edició de fitxers de text

Podeu trobar la documentació de les interfícies i classes usades en aquest apartat a l'annex "Documentació API *GestioFitxersBase*. Paquet `ioc.dam.m6.exemples.gestiofitxers.contingut` de la secció "Annexos".

Per tal d'aconseguir editar els fitxers de text des del gestor de fitxers, caldrà disposar d'una interfície gràfica que contingui una àrea de text editable (figura 2.9). En aquest cas, també se us proporciona la interfície gràfica a la biblioteca del mòdul. Aquesta està basada en l'editor de text que l'empresa Sun Microsystems incorpora en els exemples de la versió estàndard del llenguatge Java. La interfície gràfica original ha estat modificada per adaptar-la a la nostra aplicació.

Figura 2.9. Interfície gràfica de l'editor de text que usarem en aquesta aplicació



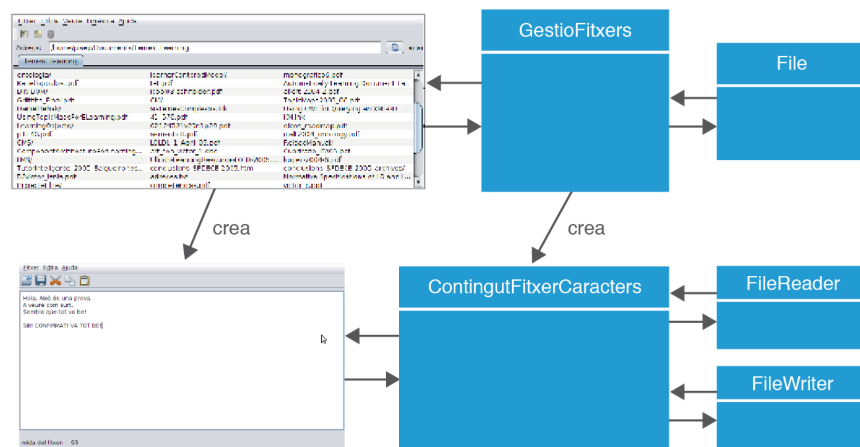
L'editor de text està preparat per treballar amb instàncies de la interfície anomenada `ContingutFitxerCaracters`, l'objectiu de la qual és fer d'intermediari entre el contingut real del fitxer i el text visible de la interfície gràfica. És a dir, fa d'*adaptador*.

El gestor de fitxers aconseguirà les instàncies de la interfície a partir del `GestioFitxersPlus`. Aquest serà l'encarregat de crear un objecte `ContingutFitxerCaracter` que gestioni el contingut del fitxer seleccionat. És a dir, farà de *Factory*.

La instància recent creada s'assignarà a la interfície gràfica d'edició i se li passarà el control (focus).

La interfície `ContingutFitxerCaracters` gestionarà el contingut dels fitxers com un tot (figura 2.10). És a dir, carregarà tot el contingut del fitxer en memòria o emmagatzemarà tot el contingut de la memòria al fitxer. Així doncs, la manipulació del text es farà sempre en memòria, i només el manipularà el fitxer en la càrrega o la gravació.

Figura 2.10. Relacions entre el gestor de fitxers i l'editor de fitxers de text



Com es pot veure, la finestra d'edició, la crearà la mateixa interfície gràfica de gestió de fitxers, mentre que GestioFitxers s'encarrega de crear les instàncies de ContingutFitxerCaracters.

Si consultem els mètodes de la interfície `ContingutFitxerCaracters` trobarem els següents:

- `void setFitxer (File fitxer) throws ContingutFitxerException`: associa al gestor de contingut el fitxer de text passat per paràmetre.
- `File getFitxer()`: obté el fitxer associat a aquest gestor de contingut.
- `void setText (String text) throws ContingutFitxerException`: assigna el text passat per paràmetre com a nou contingut del fitxer. El contingut romandrà en memòria fins que no s'executi el mètode salvar().
- `String getText() throws ContingutFitxerException`: obté el contingut existent en memòria corresponent al contingut del fitxer. Si el contingut no s'ha salvat podrien existir desfasaments entre ambdós.
- `void salvar() throws ContingutFitxerException`: guarda el contingut de la memòria al fitxer associat.

- `void carregar() throws ContingutFitxerException`: carrega en memòria el contingut del fitxer.
- `boolean estaSalvat()`: indica si el contingut de la memòria ja ha estat salvat i, per tant, es correspon amb el del fitxer.
- `boolean estaCarregat()`: indica si el fitxer ja ha estat carregat o si encara no s'ha executat aquesta acció.
- `long midaContingut() throws ContingutFitxerException`: obté la mida del contingut de la memòria.

Serà necessari un atribut per mantenir el fitxer associat, un per emmagatzemar el text en memòria, i dos per controlar les accions de carregar i salvar. A més, afegirem un instància de la classe `Utilitats` com a eina de suport.

```java
public class ContingutFitxerCaractersImpl
                                  implements ContingutFitxerCaracters{
    private String text="";
    private boolean salvat = true;
    private boolean carregat = true;
    private File fitxer=null;
    private Utilitats utilitats = new Utilitats();
    ...
```

Implementem els constructors i els accessors...

```java
public ContingutFitxerCaractersImpl() {
}

public ContingutFitxerCaractersImpl(File file) {
    setFitxer(file);
}

@Override
public void setText(String text) {
    this.text = text;
    //quan modifiquem al text perdem l'estatus de salvat.
    salvat=false;
}

@Override
public String getText() {
    return text;
}

@Override
public boolean estaSalvat() {
    return salvat;
}

@Override
public boolean estaCarregat() {
    return carregat;
}

@Override
public File getFitxer() {
    return fitxer;
}
```

Finalment, podem fer la implementació de l'operació de *carregar* i *salvar* usant la funcionalitat de la classe `Utilitats`. Usarem un `FileWrite` per salvar i un

FileReader per carregar. D'aquesta manera, usarem la codificació per defecte del sistema. Així, ens podem despreocupar de la codificació. Només caldrà tenir cura del llançament d'excepcions.

```
@Override
public void salvar() throws ContingutFitxerException {
    FileWriter writer = null;
    try {
        writer = new FileWriter(getFitxer());
        utilitats.copiarStringAWriter(text, writer);
    } catch (IOException ex) {
        throw new ContingutFitxerException(ex);
    } finally {
        utilitats.intentarTancar(writer);
    }
    salvat=true; //acabem de salvar i actualitzem l'estat
}

@Override
public void carregar() throws ContingutFitxerException {
    if(getFitxer().exists()){
        FileReader reader = null;
        try {
            reader = new FileReader(getFitxer());
            text = utilitats.copiarReaderAString(reader);
        } catch (IOException ex) {
            throw new ContingutFitxerException(ex);
        } finally {
            utilitats.intentarTancar(reader);
        }
    }else{
        text="";
    }
    carregat=true; //acabem de carregar i actualitzem l'estat
}
```

Si voleu provar l'aplicació cal implementar el mètode de GestioFItxersPlusImpl anomenat instanciarContingutFitxerCaracters per tal que, rebent un File, retorni una instància deContingutFitxerCaracters.

```
@Override
public ContingutFitxerCaracters
                  instanciarContingutFitxerCaracters(File fitxer)
                                      throws GestioFitxersException{
    return new ContingutFitxerCaractersImpl(fitxer);
}
```

3. Persistència d'objectes en fitxers

A banda dels fitxers de text, sovint ens pot interessar emmagatzemar objectes sencers per poder-los recuperar en qualsevol moment. Bàsicament, disposem de tres maneres diferents d'aconseguir fer-los persistents. Podem seriar-los usant les eines específiques que Java té per convertir qualsevol objecte en una sèrie de bytes. Podem implementar algun format binari específic de conversió d'objectes a dades primitives que permeti l'emmagatzematge i la recuperació de les dades bàsiques. Finalment, podem convertir els objectes persistents en una jerarquia XML capaç d'emmagatzemar les dades bàsiques i les relacions establertes entre objectes.

3.1 Seriació d'objectes

La tècnica de la seriació és segurament la més senzilla de totes, però també a la vegada la més problemàtica. Java disposa d'un sistema genèric de seriació de qualsevol objecte, un sistema recursiu que es repeteix per cada objecte contingut a la instància que s'està seriant. Aquest procés s'atura en arribar als tipus primitius, els quals s'emmagatzemen com una sèrie de bytes. A banda dels tipus primitius, Java serialitza també força informació addicional o metadades específiques de cada classe (el nom de les classe, els noms dels atributs i molta més informació addicional). Gràcies a les metadades es fa possible automatitzar la seriació de forma genèrica amb garanties de recuperar un objecte tal com es va emmagatzemar.

Malauradament, aquest és un procediment específic de Java. És dir, no és possible recuperar els objectes seriats des de Java usant un altre llenguatge. D'altra banda, el fet d'emmagatzemar metadades pot arribar a comportar també problemes, tot i que usem sempre el llenguatge Java. La modificació d'una classe pot fer variar les seves metadades. Aquestes variacions poden donar problemes de recuperació d'instàncies que hagin estat guardades amb algunes versions anteriors a la modificació, impedint que l'objecte pugui ser recuperat.

Aquestes consideracions desestimen aquesta tècnica per emmagatzemar objectes de forma més o menys permanent. En canvi, la seva senzillesa la fa una perfecta candidata per a l'emmagatzematge temporal, per exemple dins la mateixa sessió.

Per tal que un objecte pugui ser seriat cal que la seva classe i tot el seu contingut implementin la interfície `Serializable`. Es tracta d'una interfície sense mètodes, perquè l'únic objectiu de la interfície és actuar de marcador per indicar a la màquina virtual quines classes es poden seriar i quines no.

Totes les classes equivalents als tipus bàsics ja implementen `Serializable`. També implementen aquesta interfície la classe `String` i tots els contenidors i

els objectes Array. Tot i això, la seriació de col·leccions depèn en darrer terme dels elements continguts. Si aquest són seriables, la col·lecció també ho serà.

En cas que la classe de l'objecte que s'intenti seriar, o les d'algun dels objectes que contingui, no implementessin la interfície `Serializable`, es llançaria una excepció de tipus `NotSerializableException`, impedint l'emmagatzematge.

L'Stream `ObjectInputStream` i `ObjectOutputStream` són decoradors que afegeixen a qualsevol altre Stream la capacitat de seriar qualsevol objecte `Serializable`. L'*stream* de sortida disposarà del mètode `writeObject`. L'*stream* d'entrada, en canvi, obtindrà l'objecte invocant el mètode de lectura `readObject`.

El mètode `readObject` només permet recuperar instàncies que siguin de la mateixa classe que la que es va emmagatzemar. En cas contrari, es llançaria una excepció de tipus `ClassCastExeception`. A més, cal que l'aplicació disposi del codi compilat de la classe; si no fos així, l'excepció llançada seria `ClassNotFoundException`.

3.1.1 Exemple d'implementació

Consulteu els annexos “Biblioteques de la unitat” i “Documentació API GestioFitxersBase”. Hi trobareu el codi i la documentació del paquet `ioc.dam.m6.exemples.calculadora` en la secció “Annexos”.

Vegem primer un exemple senzill. Suposem que desitgem guardar temporalment l'estat d'una calculadora representada per la classe `EstatCalculadora`. Aquesta classe tindrà un atribut enter representant la darrera operació demanada (*oprea-ciActiva*), un atribut de tipus *double* amb un valor del darrer número introduït (*valor*) i un altre, també *double,* on s'emmagatzemarà el resultat de les darreres operacions calculades (*resultat*).

```
1  public class EstatCalculadora implements Serializable, Cloneable{
2      double valor=0;
3      int operacioActiva=Constants.IGUAL;
4      double resultat=0;
5      ...
```

Podem implementar mètodes per emmagatzemar o recuperar una instància d'`EstatCalculadora` fent:

```
1  public void guardar(EstatCalculadora estat, String nomFitxer )
2                                  throws EstatCalculadoraException{
3      try {
4          ObjectOutputStream out = new ObjectOutputStream(
5              new FileOutputStream(nomFitxer));
6          out.writeObject(estat);
7      } catch (IOException ex) {
8          throw new EstatCalculadoraException(ex);
9      }finally{
10         utilitats.intentarTancar(out);
11     }
12 }
13
14 public EstatCalculadora obtenir(String nomFitxer)
15                             throws EstatCalculadoraException{
16     EstatCalculadora estat = null;
17     try {
```

```
18            ObjectInputStream in = new ObjectInputStream(
19                new FileInputStream(nomFitxer));
20            estat = (EstatCalculadora) in.readObject();
21        } catch (ClassNotFoundException ex) {
22            throw new EstatCalculadoraException(ex);
23        } catch (IOException ex) {
24            throw new EstatCalculadoraException(ex);
25        }finally{
26            utilitats.intentarTancar(in);
27        }
28        return estat;
29    }
```

on `EstatCalculadoraException` seria una excepció especifica de l'aplicació.

El mètode `obtenir` recuperaria els diferents estats emmagatzemats en el mateix ordre en què els vàrem guardar. Imaginem, però, que volem implementar un sistema que permeti desfer les accions realitzades a la calculadora en l'ordre invers a com s'han realitzat. Normalment, això es fa amb una pila en memòria, però si preveiem que la pila pot créixer molt, ens pot interessar un cert suport persistent.

Per implementar aquest sistema en un fitxer caldrà emmagatzemar els diferents estats per on vagi passant la calculadora en un fitxer, un darrere l'altre, de forma semblant a com s'ha fet en el mètode *guardar*. Per recuperar-lo, caldrà anar a la darrera entrada del fitxer, recuperar l'objecte guardat i truncar el fitxer per tal de fer desaparèixer l'entrada recuperada.

La recuperació, però, ens planteja diversos problemes. En primer lloc, cal adonar-nos que no podem usar instàncies de `FileInputStream` per fer la lectura del darrer objecte, ja que necessitem avançar pel fitxer des del final cap a l'inici. És a dir, necessitem un `RandomAccesFile`. El segon problema que se'ns planteja és saber on comença el darrer objecte, o el que ve a ser el mateix, saber quants bytes ocupa l'objecte emmagatzemat. ObjectOutputStream no ens dóna aquesta informació, i necessitem algun mecanisme de càlcul. La solució és una mica enrevessada, però efectiva. Cal convertir un objecte a un vector de bytes i comptabilitzar la mida del vector.

La conversió la realitzarem escrivint l'objecte sobre un `ByteArrayOutputStrem` en lloc de fer-ho directament sobre el fitxer. Les instàncies de `ByteArrayOutputStrem` disposen d'un mètode anomenat `toByteArray` que retorna el seu *buffer* de bytes intern (un vector de bytes de la mida exacta que ocupen les dades escrites).

Els objectes `EstatCalculadora`seran tots de la mateixa mida, perquè només contenen atributs de tipus de dades primitives. No sempre passa això. Sovint els objectes contenen estructures dinàmiques que creixen en funció de les dades contingudes i fan que la seva mida no sigui fixa.

Sembla, doncs, un bona pràctica acompanyar la seriació dels objectes amb la seva pròpia mida per tal de poder navegar pel fitxer, d'objecte en objecte, quan faci falta. És el que farem a la nostra pila d'estats, emmagatzemarem la mida dels objectes al final per tal de saber quants bytes caldrà retrocedir per anar a l'inici de les dades de l'objecte.

Com que les instàncies de RandomAccesFilepoden ser *Streams* d'entrada i sortida a la vegada, només crearem una única instància, que ens permetrà tant escriure com llegir en el fitxer.

Per gestionar el fitxer implementarem una classe que anomenarem PilaEstatsPersistent. Per defecte, aquesta classe usarà un fitxer anomenat *estats.stk*, però si fos necessari es podria passar una instància de File al constructor per usar un fitxer alternatiu.

Per assegurar que el fitxer es trobi obert cada cop que escrivim o llegim, obrirem el fitxer en el constructor i el mantindrem obert fins a la destrucció de l'objecte. Per aconseguir-ho, sobreescriurem el mètode finalize, des d'on farem el tancament. El mètode finalize es crida automàticament just abans que el *garbagecollector* destrueixi la instància de la memòria RAM.

Abans d'obrir el fitxer, assegurarem que aquest s'eliminarà en acabar l'execució gràcies a la invocació del mètode deleteOnExit. D'aquesta manera, cada execució treballarà sempre amb un fitxer nou.

```
1  public class PilaEstatsPersistent {
2      public static final String NOM_FITXER_PER_DEFECTE="estats.stk";
3      File file;
4      RandomAccessFile fin;
5      Utilitats utilitats = new Utilitats();
6
7      @Override
8      public void finalize() throws Throwable{
9          utilitats.intentarTancar(fin);
10         super.finalize();
11     }
12
13     public PilaEstatsPersistent() throws EstatCalculadoraException {
14         setFile(new File(NOM_FITXER_PER_DEFECTE));
15     }
16
17     public PilaEstatsPersistent(File file)
18                                throws EstatCalculadoraException {
19         setFile(file);
20     }
21
22     private void setFile(File file) throws EstatCalculadoraException{
23         this.file=file;
24         try {
25             //assegurem que el fitxer s'eliminarà en acabar l'execució
26             //de l'aplicació on utilitzem aquesta pila.
27             this.file.deleteOnExit();
28             //Assegurem que abans d'obrir el fitxer existeixi
29             this.file.createNewFile();
30             //obrim el fitxer en mode lectura i escriptura
31             fin = new RandomAccessFile(file, "rw");
32         } catch (IOException ex) {
33             throw new EstatCalculadoraException(ex.getMessage(), ex);
34         }
35     }
36 ...
```

Per emmagatzemar cada estat al final del fitxer junt amb la mida implementarem el mètode empilar. Aquest escriu l'objecte en un ByteArrayOutputStream (via ObjectOutputStream), del qual obtindrem el vector de bytes que ens servirà per detectar la mida de l'objecte i per escriure els bytes al RandomAccesFile. Un cop emmagatzemat l'objecte, a continuació hi emmagatzemarem la mida.

```
public void empilar(EstatCalculadora estat)
                                    throws EstatCalculadoraException{
    try {
        ByteArrayOutputStream bOut=new ByteArrayOutputStream();
        ObjectOutputStream oOut = new ObjectOutputStream(bOut);
        oOut.writeObject(estat);
        byte[] bArray = bOut.toByteArray();
        //saltem al final del fitxer
        fin.seek(fin.length());
        //escrivim l'objecte a mode de vector de bytes.
        fin.write(bArray);
        //escrivim la mida de l'objecte just després d'aquest.
        fin.writeInt(bArray.length);
    } catch (IOException ex) {
        throw new EstatCalculadoraException(ex.getMessage(), ex);
    }
}
```

La lectura del darrer objecte emmagatzemat la realitzarem en el mètode desempilar, el qual, a més de llegir-lo, l'eliminarà del fitxer.

```
public EstatCalculadora desempilar() throws EstatCalculadoraException{
    EstatCalculadora estat = null;
    try {
        /* Saltem 4 bytes enrere del final del fitxer per poder llegir
         * la mida de l'objecte ja que un enter ocupa 4 bytes*/
        fin.seek(fin.length()- 4);
        //Llegim la mida de l'objecte.
        int length = fin.readInt();
        //Ens desplacem fins el començament de l'objecte a llegir
        fin.seek(file.length()- 4 -length);
        //Llegim l'objecte com una col·lecció de bytes.
        byte[] bArray = new byte[length];
        fin.readFully(bArray);
        /* Amb el vector de bytes, construïm un ByteArrayInputStream
         * encapsulat dins un ObjectInputStream.*/
        ObjectInputStream oIn = new ObjectInputStream(
                                  new ByteArrayInputStream(bArray));
        //Instanciem l'objecte llegint-lo de l'Stream recent creat.
        estat = (EstatCalculadora) oIn.readObject();
        //Trunquem el fitxer per tal de fer desaparèixer l'objecte.
        fin.setLength(file.length()- 4 -length);
    } catch (IOException ex) {
        throw new EstatCalculadoraException(ex.getMessage(), ex);
    } catch (ClassNotFoundException ex) {
        throw new EstatCalculadoraException(ex.getMessage(), ex);
    }
    return estat;
}
```

Implementarem també el mètode `esBuit` per saber si el fitxer d'estats té algun estat per llegir o és buit. Per esbrinar-ho, farem servir el mètode `length` de la classe `RandomAccesFile`.

```
boolean esBuit() throws EstatCalculadoraException{
    try {
        return fin.length()==0;
    } catch (IOException ex) {
        throw new EstatCalculadoraException(ex.getMessage(), ex);
    }
}
```

Podeu comprovar el funcionament de `PilaEstatsPerssitent` amb el codi següent:

```java
public class ProvaPilaEstatPersistent {
    public static void main(String[] args) {
        try {
            PilaEstatsPersistent pila = new PilaEstatsPersistent();

            pila.empilar(new EstatCalculadora());
            pila.empilar(new EstatCalculadora(100, 1, 0));
            pila.empilar(new EstatCalculadora(10, 1, 100));
            pila.empilar(new EstatCalculadora(100, 1, 110));
            pila.empilar(new EstatCalculadora(0, 0, 210));

            while(!pila.esBuit()){
                EstatCalculadora estat = pila.desempilar();
                System.out.print("Valor: "
                                    + estat.getValorAcumulat());
                System.out.print(" Operacio: "
                                    + estat.getOperacioActiva());
                System.out.println(" Resultat:"
                                    + estat.getResultat());
            }
        } catch (EstatCalculadoraException ex) {
            ex.printStackTrace(System.err);
        }
    }
}
```

3.2 Fitxers amb formats binaris específics

Podeu trobar el model de classes necessàries per a aquesta implementació a la biblioteca `GestioFitxersBase`. Consulteu l'annex "Biblioteques del mòdul" i l'annex "Documentació API GestioFitxersBase", on trobareu informació sobre el paquet `ioc.dam.m6.exemples.dibuix`.

Si volem treballar amb fitxers que es puguin llegir des de diferents llenguatges de programació caldrà definir un format binari independent. La majoria de formats estàndards que coneixeu són independents del llenguatge i es poden llegir des d'aplicacions escrites en Java, C, Pascal, etc. Els fitxers JPG, PNG, PDF, VSD, AVI, MP3 i MPG són exemples de formats independents. Qualsevol aplicació escrita amb qualsevol llenguatge podrà llegir el format amb independència de l'aplicació que haguem utilitzat per escriure'l.

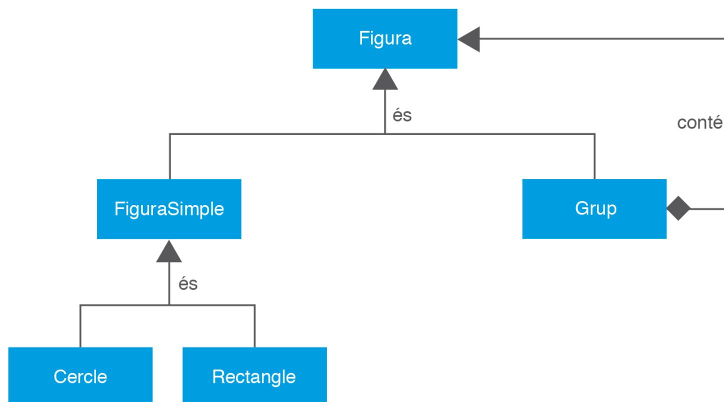
Es tracta d'una tècnica més laboriosa que l'anterior, perquè cal implementar mètodes específics que converteixin l'objecte a bytes agrupant-los d'acord amb el format, o a l'inrevés, mètodes que llegint els bytes en el format adequat, els converteixin en les respectives instàncies d'objectes que els han originat.

Imaginem que decidim inventar un format independent per aconseguir emmagatzemar dibuixos compostos de figures geomètriques. Per simplificar, només tindrem en compte dues figures geomètriques, el cercle i el rectangle. A més, les figures es poden agrupar creant composicions de diverses figures simples o agrupacions ja existents amb anterioritat, tal com podem observar a l'*esquema de la jerarquia de classes de les figures geomètriques* (figura 3.1). Totes les figures, les simples o les compostes, tenen un antecessor comú, la `Figura`. El `Cerclei` el `Rectangle`, a diferència del grup, són figures simples i deriven de la mateixa branca.

La classe `Figura` conté tres atributs: la posició, de tipus `Point` representant les coordenades x i y on s'haurà de dibuixar la figura, l'escala indicant l'ampliació o reducció que caldrà aplicar a la mida de la figura, i la rotació, mesurada en radians,

que és el gir que caldrà donar a la figura des de la seva posició original.

Figura 3.1. Jerarquia de classes per representar figures geomètriques d'un dibuix



La *FiguraSimple* afegeix dos atributs més, el color que tindrà la línia o contorn de la figura i el color interior de la mateixa. El `Cercle`, a més, afegeix un radi i el *Rectangle* una alçada i una amplada. El Grup conté una vector de figures. Al grup no se li pot definir cap color específic per a les seves figures, sinó que cada `FigureSimple`de la composició tindrà definit el seu propi color. En canvi, la posició, l'escala i la rotació sí que afecten totes les figures del grup, per això hereta directament de figura.

Totes les figures, a més, disposen d'un atribut de només lectura anomenat *tipusFigura* que indicarà el tipus que l'objecte representa (cercle, rectangle o grup). Es tracta d'un conjunt de valors enumerats amb quatre valors possibles (INDETERMINAT, CERCLE, RECTANGLE o GRUP). Tots els atributs de les classes disposen dels seus accessors de lectura i escriptura (get... i set...), excepte aquest darrer, que serà només un atribut de lectura.

```
public abstract class Figura {
    private TipusFigura tipusFigura=TipusFigura.INDETERMINADA;
    private Point posicio=null;
    private int escala=1;
    private float rotacio=0f;
    ...
}

public abstract class FiguraSimple extends Figura {
    private Color colorLinia;
    private Color colorFigura;
    ...
}

public class Cercle extends FiguraSimple {
    private int radi=0;
    ...
}

public class Rectangle extends FiguraSimple{
    private int amplada;
    private int alcada;
    ...
}

public class Grup extends Figura{
```

```
    ArrayList<Figura> elements = new ArrayList<Figura>();
    ...
}
```

A continuació implementarem un sistema de conversió de dades per poder seriar les instàncies de les figures en dades primitives. Crearem la classe `CtrlPersistenciaFigura`, on definirem un conjunt de mètodes estàtics que ens ajudin a fer-ho. Cada classe de la jerarquia disposarà d'un mètode de seriació independent. D'aquesta manera, podrem anar ampliant la jerarquia sense gaires modificacions.

```
public class CtrlPersistenciaFigura {

...

}
```

Començarem per la classe `Figura`. Els mètodes `escriureFigura` i `llegirFigura` seqüencien només els atributs específics d'aquesta classe. Cal destacar que quan ens trobem amb atributs de tipus no primitius s'han de localitzar les equivalències adequades per poder fer la transformació en ambdós sentits. És el que passa amb l'atribut *posició*, que cal convertir-lo en una seqüència de dos enters (la coordenada x i la coordenada y del punt), o amb l'atribut *tipusFigura*, que en ser un tipus enumerat es pot convertir en un identificador enter (usant l'operació *ordinal*). Per realitzar la recuperació caldrà llegir exactament en el mateix ordre com es va emmagatzemar.

```
protected static void escriureFigura(Figura figura,
                                     DataOutputStream out)
                          throws PersistenciaDibuixException{
    try {
        out.writeInt(figura.getTipusFigura().ordinal());
        out.writeInt(figura.getPosicio().x);
        out.writeInt(figura.getPosicio().y);
        out.writeInt(figura.getEscala());
        out.writeFloat(figura.getRotacio());
    } catch (IOException ex) {
        throw new PersistenciaDibuixException(ex);
    }
}

protected static void llegirFigura(DataInputStream in, Figura figura)
                                  throws PersistenciaDibuixException{
    int x,y;
    try {
        x = in.readInt();
        y = in.readInt();
        figura.setPosicio(x, y);
        figura.setEscala(in.readInt());
        figura.setRotacio(in.readFloat());
    } catch (IOException ex) {
        throw new PersistenciaDibuixException(ex);
    }
}
```

Realitzarem les mateixes operacions amb els atributs de les instàncies que derivin de *FiguraSimple* fent el següent:

```
protected static void escriureFiguraSimple(FiguraSimple figura,
                                           DataOutputStream out)
```

```
                                throws PersistenciaDibuixException{
    try {
        //escriureFigura(figura, out);//
        out.writeInt(figura.getColorLinia().getRGB());
        out.writeInt(figura.getColorFigura().getRGB());
    } catch (IOException ex) {
        throw new PersistenciaDibuixException(ex);
    }
}

protected static void llegirFiguraSimple(DataInputStream in,
                                        FiguraSimple figura)
                              throws PersistenciaDibuixException{
    try {
        //llegirFigura(in, figura);//
        figura.setColorLinia(new Color(in.readInt()));
        figura.setColorFigura(new Color(in.readInt()));
    } catch (IOException ex) {
        throw new PersistenciaDibuixException(ex);
    }
}
```

Observeu la crida al mètode `escriureFigura/llegirFigura`. Fent aquestes crides assegurarem que qualsevol instància derivada de `FiguraSimple` seqüenciï, abans dels atributs propis d'aquesta classe, els atributs propis de `Figura`.

Observeu també la seqüenciació dels atributs `Colorv` usant l'operació `getRGB`, que converteix qualsevol color en enter. Aquest valor servirà, durant la recuperació de la instància, per construir un color amb el mateix valor.

De forma semblant, anem implementant la seriació dels cercles i dels rectangles:

```
public static void escriure(Cercle figura, DataOutputStream out)
                             throws PersistenciaDibuixException{
    try {
        escriureFiguraSimple(figura, out);
        out.writeInt(figura.getRadi());
    } catch (IOException ex) {
        throw new PersistenciaDibuixException(ex);
    }
}

public static Cercle llegirCercle(DataInputStream in)
                        throws PersistenciaDibuixException{
    Cercle res=new Cercle();
    try {
        llegirFiguraSimple(in, res);
        res.setRadi(in.readInt());
    } catch (IOException ex) {
        throw new PersistenciaDibuixException(ex);
    }
    return res;
}

public static void escriure(Rectangle figura, DataOutputStream out)
                                throws PersistenciaDibuixException{
    try {
        escriureFiguraSimple(figura, out);
        out.writeInt(figura.getAmplada());
        out.writeInt(figura.getAlcada());
    } catch (IOException ex) {
        throw new PersistenciaDibuixException(ex);
    }
}

public static Rectangle llegirRectangle(DataInputStream in)
```

```
35                              throws PersistenciaDibuixException{
36     Rectangle res=new Rectangle();
37     try {
38         //llegirFiguraSimple(in, res);//
39         res.setAmplada(in.readInt());
40         res.setAlcada(in.readInt());
41     } catch (IOException ex) {
42         throw new PersistenciaDibuixException(ex);
43     }
44     return res;
45 }
```

També seriarem la classe `Grup`, de la qual destacarem que, per aconseguir una recuperació correcta, cal emmagatzemar tantes figures com componguin el grup. La seriació/deseriació de cada figura s'aconseguirà cridant la seva pròpia operació d'escriptura o lectura.

```
1  public static void escriure(Grup figura, DataOutputStream out)
2                                  throws PersistenciaDibuixException{
3      try {
4          int size= figura.size();
5          escriureFigura(figura, out);
6          out.writeInt(size); //quantitat de figures de l'agrupació
7          for(int i=0; i<size; i++){
8              escriure(figura.get(i), out);
9          }
10     } catch (IOException ex) {
11         throw new PersistenciaDibuixException(ex);
12     }
13 }
14
15 public static Grup llegirGrup(DataInputStream in)
16                         throws PersistenciaDibuixException{
17     Grup res=new Grup();
18     try {
19         llegirFigura(in, res);
20         int size = in.readInt(); //Numero de figures a recuperar.
21         for(int i=0; i<size; i++){
22             res.add(llegir(in));
23         }
24     } catch (IOException ex) {
25         throw new PersistenciaDibuixException(ex);
26     }
27     return res;
28 }
```

Finalment, dos mètodes genèrics ajuden a reconduir i trobar l'operació adequada per a cada figura:

```
1  public static Figura llegir(DataInputStream in)
2                  throws PersistenciaDibuixException{
3      Figura res = null;
4      try {
5          int tipusFigura = in.readInt();
6          if(tipusFigura == TipusFigura.CERCLE.ordinal()){
7              res = llegirCercle(in);
8          }else if(tipusFigura == TipusFigura.RECTANGLE.ordinal()){
9              res = llegirRectangle(in);
10         }else if(tipusFigura == TipusFigura.GRUP.ordinal()){
11             res = llegirGrup(in);
12         }
13     } catch (IOException ex) {
14         throw new PersistenciaDibuixException(ex);
15     }
16     return res;
17 }
```

```
public static void escriure(Figura figura, DataOutputStream out)
                              throws PersistenciaDibuixException{
    TipusFigura tipusFigura = figura.getTipusFigura();
    if(tipusFigura == TipusFigura.CERCLE){
        escriure((Cercle) figura, out);
    }else if(tipusFigura == TipusFigura.RECTANGLE){
        escriure((Rectangle) figura, out);
    }else if(tipusFigura == TipusFigura.GRUP){
        escriure((Grup) figura, out);
    }
}
```

Abans de continuar, cal comentar que hauríem pogut crear els diferents mètodes d'escriptura i lectura en cada una de les classes pròpies del model. És a dir, hauríem pogut implementar un mètode per escriure a la classe `Figura`, un altre a la classe `FiguraSimple`, etc. El principal problema d'aquesta solució seria l'excessiva dependència entre el model (la jerarquia de figures) i el sistema d'emmagatzematge. Un model ha de ser independent de la forma com es decideixi emmagatzemar les seves instàncies. Per això, si es pot, és preferible independitzar ambdós sistemes creant una o diverses classes encarregades específicament de l'emmagatzematge.

Continuant amb l'exemple, per tal de poder acabar la implementació, necessitarem que el nostre model encapsuli les diferents figures en un sola instància que representi un dibuix. La classe `dibuix` contindrà la col·lecció de figures i permetrà realitzar un seguit d'accions sobre aquestes, com ara seleccionar figures, canviar-ne l'ordre de visualització, afegir noves figures al dibuix, etc.

Simplificarem aquestes accions fent que la nostra classe dibuix hereti d'ArrayList. Així:

```
public class Dibuix extends ArrayList<Figura>{
    ...
}
```

Ara ja podem crear una classe que tingui com a funció associar un dibuix a un fitxer i sigui capaç d'emmagatzemar les figures que contingui o bé recuperar-lo i omplir el dibuix amb les figures emmagatzemades.

```
public class CtrlPersistenciaDibuix {
    File file=null;
    private Dibuix dibuix=null;
    Utilitats utilitats=new Utilitats();

    public CtrlPersistenciaDibuix() {
        dibuix = new Dibuix();
    }

    public CtrlPersistenciaDibuix(Dibuix dibuix){
        this.dibuix = dibuix;
    }

    public CtrlPersistenciaDibuix(Dibuix dibuix, File file){
        this.file=file;
        this.dibuix = dibuix;
    }

    public File getFile(){
        return file;
```

```
    }

    public void setFile(File file) {
        this.file=file;
    }

    public Dibuix getDibuix() {
        return dibuix;
    }

    public void emmagatzemar() throws PersistenciaDibuixException{
        DataOutputStream fout=null;
        try {
            fout=new DataOutputStream(new FileOutputStream(file));
            for(Figura f: dibuix){
                CtrlPersistenciaFigura.escriure(f, fout);
            }
        } catch (FileNotFoundException ex) {
            throw new PersistenciaDibuixException(ex);
        }finally{
            utilitats.intentarTancar(fout);
        }
    }

    public void recuperar() throws PersistenciaDibuixException{
        DataInputStream fin=null;
        dibuix.clear();
        try {
            fin=new DataInputStream(new FileInputStream(file));
            while(fin.available()>0){
                dibuix.add(CtrlPersistenciaFigura.llegir(fin));
            }
        } catch (IOException ex) {
            throw new PersistenciaDibuixException(ex);
        }finally{
            utilitats.intentarTancar(fin);
        }
    }
}
```

Farem la verificació de la implementació amb una classe de prova:

```
public class ProvaFormatBinari {

    public static void main(String[] args) {
        File file = new File("prova");
        file.deleteOnExit();

        if(escriure(file).equals(llegir(file))){
            System.out.println("OK");
        }else{
            System.out.println("KO");
        }
    }

    private static Dibuix escriure(File file){
        Dibuix dibuix = new Dibuix();
        try {
            CtrlPersistenciaDibuix ctrlPersistenciaDibuix =
                new CtrlPersistenciaDibuix(dibuix, file);
            dibuix.add(new Rectangle(0, 5, 10, 10));
            dibuix.add(new Cercle(50, 32, 24));
            Grup grup = new Grup();
            grup.add(new Rectangle(1,1,100, 100));
            grup.add(new Rectangle(100, 100, 10, 10));
            grup.add(new Cercle(50, 55, 30));
            dibuix.add(grup);

            ctrlPersistenciaDibuix.emmagatzemar();
```

```
        } catch (PersistenciaDibuixException ex) {
            ex.printStackTrace(System.err);
        }
        return dibuix;
    }

    private static Dibuix llegir (File file){
        Dibuix dibuix = new Dibuix();
        try {
            CtrlPersistenciaDibuix ctrlPersistenciaDibuix =
                new CtrlPersistenciaDibuix(dibuix, file);
            ctrlPersistenciaDibuix.recuperar();
        } catch (PersistenciaDibuixException ex) {
            ex.printStackTrace(System.err);
        }
        return dibuix;
    }
}
```

3.3 Fitxers amb formats XML

L'ús de formats binaris independents sobrepassa la barrera del llenguatge Java, permetent que altres llenguatges puguin llegir els fitxers que segueixin el format especificat. Tot i això, els formats binaris presenten encara certa incompatibilitat si els sistemes operatius sobre els quals es treballa fan servir diferents sistemes de representació de dades binàries. Com ja sabeu, no tots els sistemes representen les dades de la mateixa manera. N'hi ha que usen BCD per representar números, altres complement 2. Quan una dada ocupa més d'un byte, aquesta es pot llegir començant pel Byte menor (Little Endian) o bé pel major (Big Endian), etc.

És a dir, els formats binaris no garanteixen tampoc la compatibilitat de dades entre sistemes. Per això, quan es volen emmagatzemar dades que hagin de ser llegides per aplicacions executades en múltiples plataformes serà necessari recórrer a formats més estandarditzats, com ara els llenguatges de marques.

Els documents XML aconsegueixen estructurar la informació intercalant un seguit de marques anomenades etiquetes. En XML, les marques o etiquetes tenen certa similitud amb un contenidor d'informació. Així, una etiqueta pot contenir altres etiquetes o bé informació textual. D'aquesta manera, aconseguirem subdividir la informació estructurant-la de forma que pugui ser fàcilment interpretada.

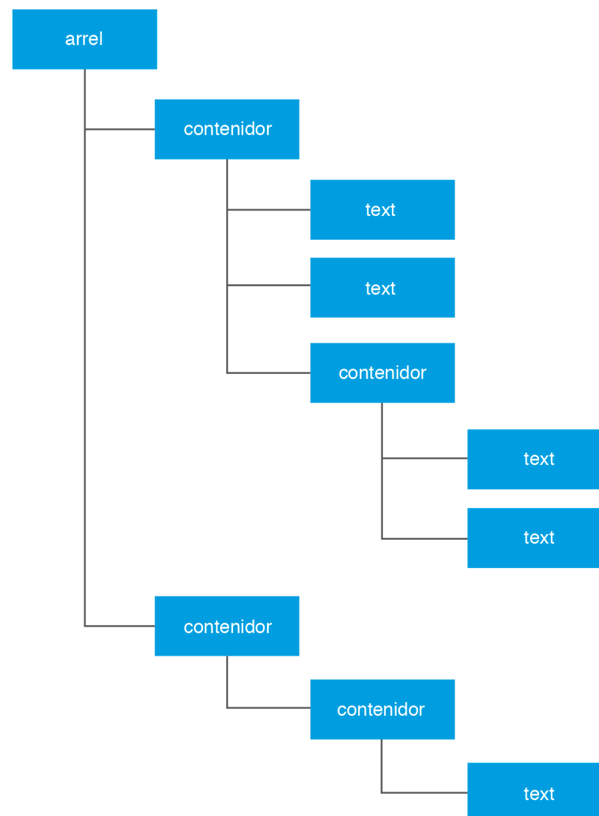
Com que tota la informació és textual, no existeix el problema de representar les dades de diferent manera. Qualsevol dada, ja sigui numèrica o booleana, caldrà transcriure-la en mode text, de manera que sigui quin sigui el sistema de representació de dades serà possible llegir i interpretar correctament la informació continguda en un fitxer XML.

És cert que els caràcters es poden escriure usant també diferents sistemes de codificació, però XML ofereix diverses tècniques per evitar que això sigui un problema. Per exemple, és possible incloure a la capçalera del fitxer quina codificació s'ha fet servir durant l'emmagatzematge, o també es poden escriure els caràcters de codi ASCII superior a 127, fent servir *entities de caràcter*, una

forma universal de codificar qualsevol símbol.

XML aconsegueix estructurar qualsevol tipus d'informació jeràrquica. Es pot establir certa similitud amb la forma com la informació s'emmagatzema en els objectes d'una aplicació i la forma com s'emmagatzemaria en un document XML. La informació, en les aplicacions orientades a objecte, s'estructura, agrupa i jerarquitza en classes, i en els documents XML s'estructura, organitza i jerarquitza en etiquetes contingudes unes dins les altres i atributs de les etiquetes (figura 3.2).

Figura 3.2. Les estructures arbòries són la forma més adient de representar informació jerarquitzada



Imaginem que volem representar les dades d'un dibuix com el de l'aparat anterior usant un format XML. No existeix una única solució, però cal que totes respectin la jerarquia del model. Sabem que totes les figures conformen un dibuix i que aquestes poden ser de tipus rectangles, cercles o grups de figures, i que cada una d'aquestes figures conté informació específica. Un possible format podria ser el següent:

```
1  <dibuix>
2      <rectangle x='5' y='10' esc='1' rot='0.0' alc='100'
3                amp='50' colli='0' colfig='325678'/>
4      <grup x='10' y='25' esc='1' rot='0,0'>
5          <cercle x='7' y='0' esc='1' ret='0.0' colli='0'
6                  colfig='325678' radi='50'/>
7          <cercle x='50' y='50' esc='1' ret='0.0' colli='0'
8                  colfig='325678' radi='20'/>
9      </grup>
10 </dibuix>
```

O també:

```xml
<dibuix>
    <rectangle>
        <figurasimple>
            <figura>
                <posicio>
                    <x>5</x>
                    <y>10</y>
                </posicio>
                <escala>1</escala>
                <rotacio>0.0</rotacio>
            </figura>
            <colorlinia>0</colorlinia>
            <colorfigura>325678</colorfigura>
        </figurasimple>
        <alcada>100</alcada>
        <amplada>50</amplada>
    </rectangle>
    <grup>
        <figura>
            <posicio>
                <x>10</x>
                <y>25</y>
            </posicio>
            <escala>1</escala>
            <rotacio>0.0</rotacio>
        </figura>
        <colleccio>
            <cercle>
                <figurasimple>
                    <figura>
                        <posicio>
                            <x>7</x>
                            <y>0</y>
                        </posicio>
                        <escala>1</escala>
                        <rotacio>0.0</rotacio>
                    </figura>
                    <colorlinia>0</colorlinia>
                    <colorfigura>325678</colorfigura>
                </figurasimple>
                <radi>50</radi>
            </cercle>
            <cercle>
                <figurasimple>
                    <figura>
                        <posicio>
                            <x>50</x>
                            <y>50</y>
                        </posicio>
                        <escala>1</escala>
                        <rotacio>0.0</rotacio>
                    </figura>
                    <colorlinia>0</colorlinia>
                    <colorfigura>325678</colorfigura>
                </figurasimple>
                <radi>20</radi>
            </cercle>
        </colleccio>
    </grup>
</dibuix>
```

3.4 'Parser' o analitzador XML

Un *Parser* XML és una classe que té per objectiu analitzar i classificar el contingut d'un arxiu XML extraient la informació continguda en cada una de les etiquetes i relacionant-la d'acord amb la seva posició dins la jerarquia.

3.4.1 Analitzadors seqüencials

Els analitzadors seqüencials, que permeten extreure el contingut a mida que es van descobrint les etiquetes d'obertura i tancament, s'anomenen analitzadors sintàctics. Són analitzadors molt ràpids, però presenten el problema que cada cop que es necessita accedir a una part del contingut cal rellegir tot el document de dalt a baix.

En Java, l'analitzador sintàctic més popular s'anomena SAX, que és l'acrònim de Simple API for XML. És una analitzador molt usat en diverses biblioteques de tractament de dades XML, però no sol usar-se en aplicacions finals.

Els **analitzadors sintàctics** són capaços d'aïllar les dades XML en una sola lectura seqüencial detectant les etiquetes d'obertura i tancament. Són molt ràpids, però han de llegir tot el document a cada consulta.

3.4.2 Analitzadors jeràrquics

Generalment, les aplicacions finals que els cal treballar amb dades XML solen usar analitzadors jeràrquics, perquè a més de realitzar una anàlisi seqüencial que els permet classificar el contingut, s'emmagatzemen a la memòria RAM seguint l'estructura jeràrquica detectada en el document. Això facilita molt les consultes que calgui repetir diverses vegades, atès que les estructures jeràrquiques de la memòria RAM tenen un rendiment d'accés parcial a les dades molt eficient.

Els **analitzadors jeràrquics** guarden totes les dades del'XML en memòria dins una estructura jeràrquica. Són ideals per a aplicacions que requereixin una consulta contínua de les dades.

El format de l'estructura on s'emmagatzema la informació a la memòria RAM ha estat especificat per l'organisme internacional W3C (World Wide Web Consortium) i s'acostuma a conèixer com a DOM (Document Object Model). És una estructura que HTML i javascript han popularitzat molt i es tracta d'una especificació que Java materialitza en forma d'interfícies. La principal s'anomena

Document i representa tot un document XML. En tractar-se d'una interfície, pot ser implementada per diverses classes.

Vegeu l'apartat "Fluxos eficients: Channels i Buffers", on es parla de les classes "factory".

L'estàndard W3C defineix l'especificació de la classe `DocumentBuilder` amb el propòsit de poder instanciar estructures DOM a partir d'un XML. La classe `DocumentBuilder` és una classe abstracta, i per tal que es pugui adaptar a les diferents plataformes, pot necessitar fonts de dades o requeriments diversos. Recordeu que les classes abstractes no es poden instanciar de forma directa. Per aquest motiu, el consorci W3 especifica també la classe `DocumentBuilderFactory`.

Les instruccions necessàries per llegir un fitxer XML i crear un objecte *Document* serien les següents:

```
1 DocumentBuilderFactory dbFactory =
2                             DocumentBuilderFactory.newInstance();
3 DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
4 Document doc = dBuilder.parse(new File("fitxer.xml"));
```

Podem reduir el nombre de variables concatenant instruccions.

```
1 Document doc = DocumentBuilderFactory.newInstance().newDocumentBuilder().parse(
      "fitxer.xml");
```

`DocumentBuilder`també instancia objectes `Document` buits que podran ser manipulats a posteriori.

```
1 DocumentBuilderFactory dbfac = DocumentBuilderFactory.newInstance();
2 DocumentBuilder docBuilder = dbfac.newDocumentBuilder();
3 Document doc = docBuilder.newDocument();
```

O bé,

```
1 Document doc = DocumentBuilderFactory.newInstance().newDocumentBuilder().
      newDocument();
```

L'escriptura de la informació continguda al DOM es pot seqüenciar en forma de text fent servir una altra utilitat de Java anomenada *Transformer*. Es tracta d'una utilitat que permet realitzar fàcilment conversions entre diferents representacions d'informació jeràrquica. És capaç, per exemple, de passar la informació continguda en un objecte Document a un fitxer de text en format XML. També seria capaç de fer l'operació inversa, però el mateix *DocumentBuilder* ja s'encarrega d'això.

Transformer és també una classe abstracta i requereix d'una *factory* per poder ser instanciada. La classe `Transformer` pot treballar amb multitud de contenidors d'informació perquè en realitat treballa amb un parell de tipus adaptadors (classes que fan compatibles jerarquies diferents) que s'anomenen `Source`i `Result`. Les classes que implementin aquestes interfícies s'encarregaran de fer compatible un tipus de contenidor específic al requeriment de la classe `Transformer`. Així, disposem de les classes DOMSource, SAXSource o StreamSorce com a adaptadors del contenidor de la font d'informació (DOM, SAX o Stream respectivament). DOMResult, SAXResult o StreamResult són els adaptadors equivalents del contenidor destí.

El codi bàsic per realitzar una transformació de DOM a fitxer de text XML seria el següent:

```
//Creació d'una instancia Transformer
Transformer trans = TransformerFactory.newInstance().newTransformer();

//Creació dels adaptadors Source i Results a partir d'un Document
//i un File.
StreamResult result = new StreamResult(file);
DOMSource source = new DOMSource(doc);
trans.transform(source, result);
```

Per tal de rebaixar la complexitat, crearem una classe que anomenarem `XmlCtrlDom`amb utilitats genèriques que ens simplifiquin el traspàs de fitxers XML a DOM o viceversa.

```
public class XmlCtrlDom {
    public static Document instanciarDocument()
        throws ParserConfigurationException{
        Document doc=null;
        doc = DocumentBuilderFactory.newInstance().newDocumentBuilder().
            newDocument();
        return doc;
    }

    public static void escriureDocumentATextXml(Document doc, File file)
      throws TransformerException {
        Transformer trans = TransformerFactory.newInstance().newTransformer();
        trans.setOutputProperty(OutputKeys.INDENT, "yes");

        StreamResult result = new StreamResult(file);
        DOMSource source = new DOMSource(doc);
        trans.transform(source, result);
      }

    public static Document instanciarDocument(File fXmlFile)
                                throws ParserConfigurationException,
                                       SAXException,
                                       IOException{
        Document doc=null;
        doc = DocumentBuilderFactory.newInstance().newDocumentBuilder().parse(
            fXmlFile);
        doc.getDocumentElement().normalize();
        return doc;
    }

    ...
}
```

L'estructura DOM

L'estructura DOM pren la forma d'un arbre, on cada part del'XML s'hi trobarà representada en forma de node. En funció de la posició en el document XML, parlarem de diferents tipus de nodes. El node principal que representa tot l'XML sencer s'anomena **document**, i les diverses etiquetes, inclosa l'etiqueta arrel, es coneixen com a nodes **element**. El contingut textual d'una etiqueta s'instancia com a node de tipus `TextElement` i els atributs com a nodes de tipus `Atribute`. Cada node específic disposa de mètodes per accedir a les seves dades concretes (nom, valor, nodes fills, node pare, etc.).

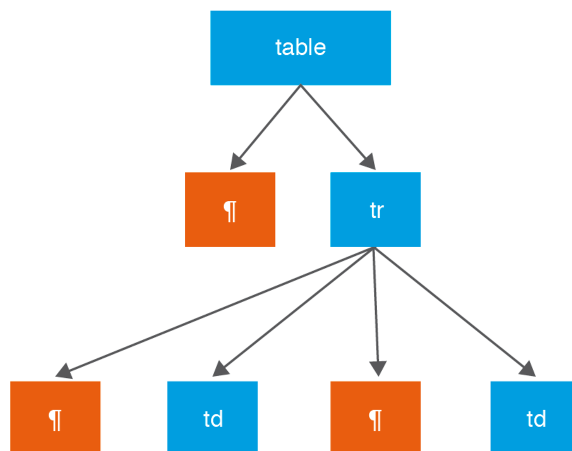
El DOM resultant obtingut des d'un XML acaba sent un còpia exacta del fitxer, però disposat de diferent manera. Tant al'XML com al DOM hi haurà informació no visible, com ara els retorns de carro, que cal tenir en compte per tal de saber processar correctament el contingut i evitar sorpreses poc comprensibles.

Imaginem que disposem d'un document XML amb el següent contingut:

```
<table>
    <tr>
        <td></td>
        <td></td>
    </tr>
</table>
```

A la figura 3.3 es mostra la representació que l'objecte DOM tindria, un cop copiat en memòria. Cal destacar que l'element *table* tindrà dos fills. En un s'hi guardarà el retorn de carro que situa l'etiqueta *tr* a la següent línia, a l'altre hi trobarem l'etiqueta *tr*. El mateix passa amb els fills de *tr*, abans de cada node *td* trobarem un retorn de carro.

Figura 3.3. Representació d'un objecte DOM amb retorns de carro

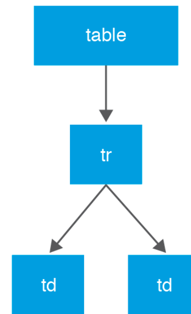


En canvi, si haguéssim partit d'un XML equivalent però sense retorns de carro, el resultat hauria estat també diferent.

```
<table><tr><td></td><td></td></tr></table>
```

El document XML anterior, sense retorns de carro, donaria la representació de l'objecte DOM que podeu observar en la figura 3.4.

Figura 3.4. Representació d'un objecte DOM sense retorns de carro



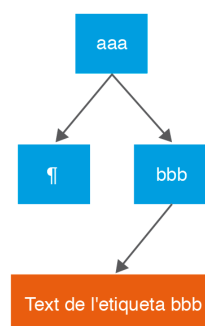
L'absència de retorns de carro en el fitxer implica també l'absència de nodes contenint els retorns de carro en l'estructura DOM.

Un altre aspecte a tenir en compte sobre el mapatge de fitxers XML és que el contingut textual de les etiquetes es plasma en el DOM com un node fill de l'etiqueta contenidora. És a dir, per obtenir el text d'una etiqueta cal obtenir el primer fill d'aquesta.

```
<aaa>
    <bbb>
        text de l'etiqueta bbb
    </bbb>
</aaa>
```

La figura 3.5 il·lustra la representació DOM d'una etiqueta que contingui text.

Figura 3.5. Representació DOM d'una etiqueta amb text



La interfície `Document` contempla un conjunt de mètodes per seleccionar diferents parts de l'arbre a partir del nom de l'etiqueta o d'un atribut identificador. Les parts de l'arbre es retornen com a objectes `Element`, els quals representen un node i tots els seus fills. D'aquesta manera, podrem anar explorant parts de l'arbre sense necessitat d'haver de passar per tots el nodes.

Per tal de facilitar l'obtenció del contingut d'un `Element` ampliarem les utilitats de la classe `XmlCtrlDom` afegint dos mètodes més.

```java
public static String getValorEtiqueta(String etiqueta,
                                      Element element) {
    Node nValue = element.getElementsByTagName(etiqueta).item(0);
    return nValue.getChildNodes().item(0).getNodeValue();
}

public static Element getElementEtiqueta(String etiqueta,
                                         Element element) {
    return (Element) element.getElementsByTagName(etiqueta).item(0);
}
```

El primer rep el nom de l'etiqueta i l'element (o node parcial de l'arbre) a partir del qual es desitja fer la cerca. Fent servir el mètode `getElementsByTagName`, aconseguirem tots els nodes que tinguin per nom el valor del paràmetre etiqueta. És a dir, ens retornarà una col·lecció de nodes. Si només existeix un únic node amb el nom del paràmetre, aquest ocuparà la primera posició de la llista. Per això hi accedim amb el mètode `item`, indicant que ens interessa el primer element (posició zero).

El segon mètode és molt semblant al primer, però en comptes de recuperar el text, obtindrem el node amb tots els fills que tingui. És útil per aplicar a nodes intermedis (no textuals).

Els objectes Element disposen de mètodes per afegir nous fills (`appendChild`) o assignar el valor a un atribut (`setAtributte`). També permeten la consulta del valor dels atributs (`getAttributte`) o la navegació pels nodes de l'arbre (`getParentNode`, per obtenir el pare; `getFirstChild/getLastChild`, per obtenir el primer/darrer fill, o `getNextSibling` per navegar de germà en germà).

L'objecte `Document` farà de *factory* per a qualsevol node del document. La creació de nous elements (etiquetes) implicarà l'ús de `createElement`. Si volem crear contingut textual caldrà cridar el mètode `createTextNode`, i si el que volem són comentaris cridarem `createComment`.

La creació de nodes no implica la ubicació d'aquest dins l'arbre. És a dir, a més de crear-los caldrà assignar-los a un pare usant `appendChild`.

3.4.3 Implementació d'exemple

L'exemple conductor que ens permetrà mostrar l'ús de la biblioteca DOM de Java serà també la implementació d'una eina que gestioni l'emmagatzematge d'un dibuix de figures geomètriques, podeu comparar aquesta implementació amb la realitzada a l'apartat "Fitxers amb formats binaris específics".

El format XML que representarà el dibuix serà el següent: l'element arrel serà *dibuix*, el qual contindrà una llista de figures etiquetades segons el seu tipus com *rectangles*, *cercles* o *grups*.

```
dibuix ((rectangle | cercle | grup)*)
```

Els rectangles es definiran per un element de tipus *figurasimple*, per *l'alçada* i per *l'amplada*:

```
rectangle (figurasimple, alcada, amplada)
```

Els cercles, en canvi, contindran també un element de tipus *figurasimple*, però a continuació només disposaran de l'element *radi*.

```
cercle (figurasimple, radi)
```

El grup contindrà només dos elements, un de tipus *figura* i una altre de tipus *colleccio*.

```
cercle (figura, colleccio)
```

La *figurasimple* serà també un element compost, mentre que *alcada* i *amplada* seran elements de tipus text contenint els valors numèrics corresponents al rectangle representat.

L'element *figurasimple* es compondrà de dos elements simples (textuals) anomenats *colorlinia* i *colorfigura.*

```
figurasimple (colorlinia, colorfigura)
```

Els elements que composin *figura* seran *posicio*, *escala* i *rotacio*. A la vegada, *posicio* estarà compost per dos elements textuals que etiquetarem amb *x* i *y*. En canvi, *escala* i *rotació* seran també elements textuals.

```
figura (posicio, escala, rotacio)
posicio (x, y)
```

Finalment, l'element *colleccio* dins l'etiqueta *grup* serà un element compost d'una llista de figures de forma semblant a com es troben definides a l'element *dibuix*.

```
colleccio ((rectangle | cercle | grup)*)
```

El procés de transferència d'informació entre objectes i document XML serà similar a la forma com hem implementat el format binari, una classe que disposi d'un conjunt de mètodes, cada un d'ells especialitzats en la conversió d'una de les classes de la jerarquia de les figures.

En aquest cas, la classe que anomenarem `XmlCtrlFigura` heretarà de `XmlCtrlDom` per poder reutilitzar les utilitats ja implementades.

```java
public class XmlCtrlFigura extends XmlCtrlDom{
    //Constants amb els noms de les etiquetes
    static final String ET_POSICIO="posicio";
    ...
    static final String ET_RADI="radi";
```

Començarem implementant l'escriptura de la classe `Figura`.

```
protected static void escriureFigura(Figura figura,
                                        Document doc,
                                        Element elemFigura){
        Element elmPosicio = doc.createElement(ET_POSICIO);
        Element nouElement = doc.createElement(ET_X);
        elmPosicio.appendChild(nouElement);
        nouElement.appendChild(doc.createTextNode(String.
            valueOf(figura.getPosicio().x)));
        nouElement = doc.createElement(ET_Y);
        elmPosicio.appendChild(nouElement);
        nouElement.appendChild(doc.createTextNode(String.
            valueOf(figura.getPosicio().y)));
        elemFigura.appendChild(elmPosicio);
        nouElement = doc.createElement(ET_ESCALA);
        nouElement.appendChild(doc.createTextNode(String.
            valueOf(figura.getEscala())));
        elemFigura.appendChild(nouElement);
        nouElement = doc.createElement(ET_ROTACIO);
        nouElement.appendChild(doc.createTextNode(String.
            valueOf(figura.getRotacio())));
        elemFigura.appendChild(nouElement);
    }
```

Del codi anterior destaquem que, per crear un element, necessitem un objecte `Document` que l'instanciï. Per això caldrà anar passant-lo als diferents mètodes que ho necessitin. L'element es crea invocant `createElementi` passant per paràmetre el nom de l'etiqueta XML associada. Si l'element és de text, cal crear-lo com un node textual fent servir `createTextNode` i passant per paràmetre el contingut textual. Els elements creats s'han d'afegir a l'estructura DOM com a fills de l'element que l'hagi de contenir (el qual haurem passat també per paràmetre juntament amb el *Document*). Usarem `appendChild`.

Els mètodes que traspassin informació de l'XML als objectes tindran la forma següent:

```
protected static void llegirFigura(Element elemFigura,
                                        Figura figura){
        int x = Integer.parseInt(getValorEtiqueta(ET_X, elemFigura));
        int y = Integer.parseInt(getValorEtiqueta(ET_Y, elemFigura));
        figura.setPosicio(x, y);
        int valorI = Integer.parseInt(getValorEtiqueta(ET_ESCALA,
                        elemFigura));
        figura.setEscala(valorI);
        float valorF = Float.parseFloat(getValorEtiqueta(ET_ROTACIO,
                        elemFigura));
        figura.setRotacio(valorF);
    }
```

Usarem les operacions implementades de `getValorEtiqueta` i de `getElementEtiqueta`, segons el cas, quan esperem un valor o un node amb fills respectivament.

De forma semblant, implementarem `escriureFiguraSimplei` `llegirFigurasSimple`:

```
protected static void escriureFiguraSimple(FiguraSimple figura,
      Document doc, Element elementFiguraSimple){
      Element elemFigura = doc.createElement(ET_FIGURA);
      escriureFigura(figura, doc, elemFigura);
      elementFiguraSimple.appendChild(elemFigura);
      Element nouElement = doc.createElement(ET_COLOR_LIN);
```

```
      nouElement.appendChild(doc.createTextNode(String.valueOf(figura.
          getColorLinia().getRGB())));
      elemFigura.appendChild(nouElement);
      nouElement = doc.createElement(ET_COLOR_FIG);
      nouElement.appendChild(doc.createTextNode(String.valueOf(figura.
          getColorFigura().getRGB())));
      elemFigura.appendChild(nouElement);
    }

    protected static void llegirFiguraSimple(Element elemFiguraSimple,
                                            FiguraSimple figura){
        Element elemFigura = getElementEtiqueta(ET_FIGURA,
                                                elemFiguraSimple);
        llegirFigura(elemFigura, figura);
        int valorI = Integer.parseInt(getValorEtiqueta(ET_COLOR_LIN,
                                                elemFiguraSimple));
        figura.setColorLinia(new Color(valorI));
        valorI = Integer.parseInt(getValorEtiqueta(ET_COLOR_FIG,
                                                elemFiguraSimple));
        figura.setColorFigura(new Color(valorI));
    }
```

L'escriptura i lectura dels objectes `Cercle`, `Rectangle` o `Grup` segueixen un patró semblant i els deixarem com a exercici a implementar, a excepció de la lectura del grup, ja que voldria parar atenció al tractament que cal donar a un node contenidor d'una llista d'elements dels quals en desconeixem el tipus. És el cas de l'etiqueta *colleccio* de l'element *grup* que conté totes les figures associades, però a priori desconeixem si es tracta de rectangles, cercles o altres grups.

Per analitzar la col·lecció continguda caldrà, en primer lloc, obtenir l'element (via `getElementEtiqueta`). Seguidament, recollirem tots els seus fills amb el mètode `getChildNodes`. Aquest mètode retorna un NodeList o col·lecció de tots els nodes fill. Recorrerem tots els fills usant un bucle. El que volem fer és recollir cada un dels fills (un cercle, un rectangle o un altre grup i fer la crida corresponent per crear la figura que representi). El problema és que no podem estar segurs que tots els nodes siguin figures. Recordeu que segons com estigui escrit l'XML, podem trobar intercalats nodes que continguin retorns de carro i que no ens interessa tractar. Per distingir els nodes tractables dels que no ho són, usarem el mètode `getNodeType`. Tots els nodes d'un DOM estan marcats amb un valor numèric que indica el tipus de node i el que conté. Ens interessa que es tracti d'un node de tipus Node.ELEMENT_NODE. És a dir, un node que conté altres nodes (ja que es tracta de figures).

```
public static Grup llegirGrup(Element element){
        Grup res=new Grup();
        Element elemFigura = getElementEtiqueta(ET_FIGURA, element);
        llegirFigura(elemFigura, res);
        Element elemColleccio = getElementEtiqueta(ET_COLLECCIO,
                                                        element);
        Noderetre_us_elsList nList = elemColleccio.getChildNodes();
        for(int i=0; i<nList.getLength(); i++){
            if(nList.item(i).getNodeType()==Node.ELEMENT_NODE){
                res.add(llegir((Element) nList.item(i)));
            }
        }
        return res;
    }
```

Els dos darrers mètodes que segueixen aconseguiran seleccionar l'operació correcta segons el tipus de figura que s'estigui escrivint o construint.

```
public static Figura llegir(Element element){
        Figura res = null;
        if(element.getTagName().equalsIgnoreCase(ET_CERCLE)){
            res = llegirCercle(element);
        }else if(element.getTagName().equalsIgnoreCase(ET_RECTANGLE)){
            res = llegirRectangle(element);
        }else if(element.getTagName().equalsIgnoreCase(ET_GRUP)){
            res = llegirGrup(element);
        }
        return res;
    }

    public static void escriure(Figura figura,
                                Document out,
                                Element element){
        Element elemFigura;
        TipusFigura tipusFigura = figura.getTipusFigura();
        if(tipusFigura == TipusFigura.CERCLE){
            elemFigura = out.createElement(ET_CERCLE);
            escriureCercle((Cercle) figura, out, elemFigura);
            element.appendChild(elemFigura);
        }else if(tipusFigura == TipusFigura.RECTANGLE){
            elemFigura = out.createElement(ET_RECTANGLE);
            escriureRectangle((Rectangle) figura, out, elemFigura);
            element.appendChild(elemFigura);
        }else if(tipusFigura == TipusFigura.GRUP){
            elemFigura = out.createElement(ET_GRUP);
            escriureGrup((Grup) figura, out, elemFigura);
            element.appendChild(elemFigura);
        }
    }
```

Ara només quedaria implementar la classe `XmlCtrlDibuix` per vinclar un *dibuix* a un fitxer XML. A banda dels constructors i els accessors, caldrà un mètode que llegeixi i escrigui l'objecte *Dibuix* a una estructura DOM i, finalment, dos mètodes més que emmagatzemin el DOM en un fitxer XML o que recuperin el contingut del fitxer i el buidin en un DOM. Respectivament, els anomenarem `emmagatzemari` `recuperar`.

```
public class XmlCtrlDibuix extends XmlCtrlDom{
    static final String ET_DIBUIX="dibuix";

    File file=null;
    private Dibuix dibuix=null;

    public XmlCtrlDibuix() {
        dibuix = new Dibuix();
    }

    public XmlCtrlDibuix(Dibuix dibuix){
        this.dibuix = dibuix;
    }

    public XmlCtrlDibuix(Dibuix dibuix, File file){
        this.file=file;
        this.dibuix = dibuix;
    }

    public File getFile(){
        return file;
    }

    public void setFile(File file) {
```

```
25            this.file=file;
26        }
27
28        public Dibuix getDibuix() {
29            return dibuix;
30        }
31
32        private void escriure(Document doc){
33            Element arrel = doc.createElement(ET_DIBUIX);
34            doc.appendChild(arrel);
35            for(Figura f: dibuix){
36                XmlCtrlFigura.escriure(f, doc, arrel);
37            }
38        }
39
40        private void llegir(Document doc){
41            Element arrel = doc.getDocumentElement();
42            NodeList nList = arrel.getChildNodes();
43            dibuix.clear();
44            for(int i=0; i<nList.getLength(); i++){
45                if(nList.item(i).getNodeType()==Node.ELEMENT_NODE){
46                    dibuix.add(XmlCtrlFigura.llegir(
47                                                (Element) nList.item(i)));
48                }
49            }
50        }
51
52        public void emmagatzemar() throws xmlDibuixException{
53            try {
54                Document doc=null;
55                doc=instanciarDocument();
56                escriure(doc);
57                escriureDocumentATextXml(doc, file);
58            } catch (ParserConfigurationException ex) {
59                throw new xmlDibuixException(ex);
60            } catch (TransformerException ex) {
61                throw new xmlDibuixException(ex);
62            }
63        }
64
65        public void recuperar() throws xmlDibuixException {
66            try {
67                Document doc=null;
68                doc=instanciarDocument(file);
69                llegir(doc);
70            } catch (ParserConfigurationException ex) {
71                throw new xmlDibuixException(ex);
72            } catch (SAXException ex) {
73                throw new xmlDibuixException(ex);
74            } catch (IOException ex) {
75                throw new xmlDibuixException(ex);
76            }
77         }
78    }
```

3.5 Binding

El *Binding* és una tècnica que consisteix a vincular classes Java amb formats específics d'emmagatzematge de manera automatitzada. Sovint un conjunt idèntic de classes pot donar lloc a múltiples esquemes XML. Això significa que normalment l'automatització de l'escriptura de dades en format XML, que en termes tècnics s'anomena *marshalling*, requereix de certa ajuda per tal de decidir quin format

s'automatitzarà d'entre els molts possibles. D'això se'n diu *mapar*, perquè ve a ser com si configuréssim un mapa on s'indiquen els vincles entre les classes i els seus atributs, i els elements i atributs XML.

En Java existeixen diverses biblioteques per gestionar el *binding*, com per exemple *JAXB*, *JiBX*, *XMLBinding*, etc. Des de la versió 6.0 s'ha incorporat en el JDK estàndard *JAXB,* unapotent biblioteca.

3.5.1 Configuració amb anotacions

JAXB utilitza *Anotacions* per aconseguir la informació extra necessària per *mapar* el *binding*. Les *Anotacions* són unes interfícies o classes de Java molt especials. Serveixen per associar informació i funcionalitat als objectes sense interferir en l'estructura del model de dades. Abans que apareguessin les *Anotacions* era necessari fer servir l'herència per poder afegir funcionalitat a una classe sense haver de codificar-la. L'herència, però, interfereix directament en el model de dades, ja que en Java no és possible l'herència múltiple. Si la classe `Rectangle`, per exemple, en correspondència al model de dades, cal que hereti de `FiguraSimple`, però a la vegada necessitem afegir a la classe rectangle una funcionalitat extra, si no fem servir *Anotacions* serà necessari escriure codi extra o fins i tot haver de modificar el model final de dades per aconseguir ambdós objectius.

Si, per contra, fem servir Anotacions, els objectes disposaran d'informació o de funcionalitat extra sense que el model de dades quedi modificat, ja que les *Anotacions* no són visibles des dels objectes, malgrat que sí són accessibles via reflexió. Les *Anotacions* poden associar-se a un paquet, a una classe, a un atribut o fins i tot a un paràmetre. Aquestes classes especials es declaren en el codi de l'aplicació anteposant el símbol @ al nom de l'*Anotació*. Quan el compilador de Java detecta una *Anotació* crea una instància i la injecta dins l'element estructural afectat (paquet, classe, mètode, atribut, etc.). Això fa que aquestes no apareguin com a atributs o mètodes propis de l'objecte, i per això diem que no interacciona amb el model de dades, però les aplicacions que ho necessitin poden obtenir la instància injectada i fer-la servir.

Les *Anotacions* poden declarar-se amb paràmetres o sense. En cas que tinguin paràmetres, aquests poden ser altres Anotacions, valors constants o valors literals, de manera que estiguin disponibles en temps de compilació (que és quan el compilador realitza la injecció).

Abans de començar a estudiar algunes de les *Anotacions* de JAXB, veurem que aquesta biblioteca és capaç també de generar, de forma automàtica, les classes necessàries per suportar la informació descrita per un Schema XML, el que venim anomenant model de l'aplicació. Les classes generades es creen amb les *Anotacions* necessàries per establir correctament els vincles entre els atributs de les classes i els elements XML.

3.5.2 Generació automàtica del model de dades

En XML, els *Schema* són documents XML que permeten definir l'estructura d'altres documents. L'estudi dels *Schema* cau per complet fora dels objectius d'aquest mòdul, però com que són una peça important en el *Binding* farem un petit repàs com a recordatori dels conceptes més importants. Els *Schema* permeten definir tipus de dades que es classifiquen en tipus de dades simples i tipus de dades complexes. Els tipus de dades simples són valors unitaris i no es poden descompondre perquè perdrien totalment el seu sentit. Generalment, es corresponen amb tipus de dades primitius, però s'amplien amb tipus específics com *dates*, *text* o classes que representin conceptualment el mateix que els valors primitius (classe `Integer`, `BigInteger`, `Float`, etc.).

Els tipus complexos es defineixen com una composició d'altres tipus (ja siguin simples o complexes), els quals aporten un sentit extra en ser tractats en conjunt. El símil per excel·lència en termes de Java són les classes, i per especificar un tipus complex es defineixen elements, cadascun dels quals representaria un dels tipus en què es descompondria el tipus complex que s'estigui definint. És a dir, podem establir una correspondència entre els elements d'un tipus complex i els atributs d'una classe. El problema és que, al mateix nivell que els elements, es poden definir també atributs XML amb una correspondència també directa amb atributs de les classes Java. Els elements XML que configurin un tipus complex es poden definir com una seqüència ordenada d'elements (*sequence*), com una tria opcional (*choice*) o bé com una combinació d'ambdues.

XMLTool4NetBeans

XMLTool4NetBeans es distribueix a la versió 7.1 de l'IDE. Si teniu una versió anterior caldrà que us l'actualitzeu usant l'eina pròpia de l'IDE o baixant-vos-el directament del portal de plugins de NetBeans (http://plugins.netbeans.org/PluginPortal/) i cercant pel nom, o accedint directament a la pàgina http://plugins.netbeans.org/plugin/40292/xmltools4netbeans. Podeu trobar informació d'ús a la pàgina de l'autor: http://blogs.oracle.com/geertjan/entry/xml_schema_editor_in_netbeans.

NetBeans disposa d'un *plugin*, anomenat **XMLTools4NetBeans**, on podem trobar un conjunt d'eines XML, entre les quals destaquem un editor d'esquemes molt potent.

El format XML que hem triat és un format còmode per fer el tractament de dades usant un DOM, ja que les dades dels diferents nivells de la jerarquia d'una figura queden totalment encapsulades en nodes independents. Aquest format, però, presenta certa dificultat d'interpretació als ulls humans en llegir el contingut XML generat, ja que les dades de figura o figura simple es tracten com a components interns de les figures específiques (`Rectangle`, `Cercle` o `Grup`), més que no pas tractar-se com a contenidors genèrics que continguin casos específics.

Aquí, en tractar-se d'un procediment automàtic de transferència de dades, podem deixar de banda l'elecció del tipus d'estructura en funció del tractament i centrar-nos més en la part conceptual. Dissenyarem un format senzill més fàcilment interpretable en què no cal plasmar la jerarquia. D'aquesta manera, independitzem el format XML de la implementació Java finalment realitzada. Així, si algun dia decidim reestructurar la jerarquia, podem continuar amb el mateix format.

Podeu fer servir el *plugin* XMLTools4NetBeans per obtenir un *Schema* que ens defineixi un format per emmagatzemar les figures del dibuix definides a l'apartat "Implementació d'un exemple" de la Secció "Parser o analitzador sintàctic".

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
3      targetNamespace="http://xml.netbeans.org/schema/dibuix"
4      xmlns:tns="http://xml.netbeans.org/schema/dibuix"
5      elementFormDefault="qualified">
6      <!--Tipus Punt per definir les coordenades
```

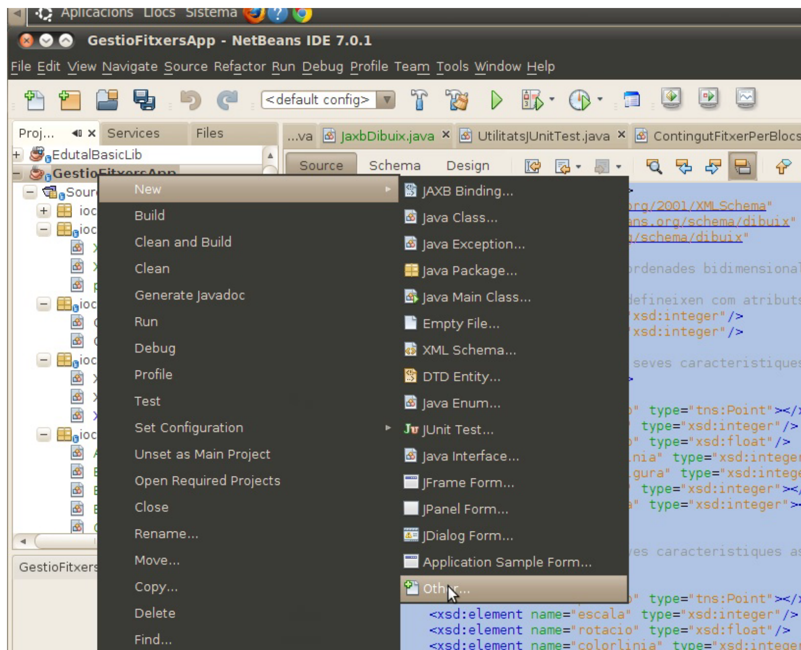
```
        bidimensional——>
    <xsd:complexType name="Point">
        <!——Les coordenades x i y és defineixen com atributs,
            no pas com elements——>
        <xsd:attribute name="x" type="xsd:integer"/>
        <xsd:attribute name="y" type="xsd:integer"/>
    </xsd:complexType>
    <!——Tipus Rectangle amb totes les seves caracteristiques
        associades——>
    <xsd:complexType name="Rectangle">
        <xsd:sequence>
            <xsd:element name="posicio" type="tns:Point"/>
            <xsd:element name="escala" type="xsd:integer"/>
            <xsd:element name="rotacio" type="xsd:float"/>
            <xsd:element name="colorlinia"
                                    type="xsd:integer"/>
            <xsd:element name="colorfigura"
                                    type="xsd:integer"/>
            <xsd:element name="alcada"
                                    type="xsd:integer"/>
            <xsd:element name="amplada"
                                    type="xsd:integer"/>
        </xsd:sequence>
    </xsd:complexType>
    <!——Tipus Cercle amb totes les seves caracteristiques
        associades——>
    <xsd:complexType name="Cercle">
        <xsd:sequence>
            <xsd:element name="posicio" type="tns:Point"/>
            <xsd:element name="escala" type="xsd:integer"/>
            <xsd:element name="rotacio" type="xsd:float"/>
            <xsd:element name="colorlinia"
                                      type="xsd:integer"/>
            <xsd:element name="colorfigura"
                                      type="xsd:integer"/>
            <xsd:element name="radi" type="xsd:integer"/>
        </xsd:sequence>
    </xsd:complexType>
    <!——Tipus que representarà una llista de figures. La
        llista podrà estar definida sense cap figura (buida)
        o be contenir un nombre indeterminat de figures——>
    <xsd:complexType name="LlistaFigures">
         <xsd:choice maxOccurs="unbounded" minOccurs="0">
             <xsd:element name="rectangle"
                                    type="tns:Rectangle"/>
             <xsd:element name="cercle" type="tns:Cercle"/>
             <xsd:element name="grup" type="tns:Grup"/>
        </xsd:choice>
    </xsd:complexType>
    <!——Tipus Grup amb totes les seves caracteristiques
        associades——>
    <xsd:complexType name="Grup">
        <xsd:sequence>
            <xsd:element name="posicio" type="tns:Point"/>
            <xsd:element name="escala" type="xsd:integer"/>
            <xsd:element name="rotacio" type="xsd:float"/>
            <!——L'element col·lecció es defineix com una
                llista de figures——>
            <xsd:element name="colleccio"
                                    type="tns:LlistaFigures"/>
        </xsd:sequence>
    </xsd:complexType>
    <!——L'element arrel cal definir—lo sense especificar
        l'atribut tipus per tal que JAXB el detecti com
        a tal. El tipus caldrà definir—lo com un element
        complex amb un únic component o la llista de
        figures——>
    <xsd:element name="dibuix">
        <xsd:complexType>
            <xsd:complexContent>
```

```
77                 <xsd:extension xmlns:tns=
78                   "http://xml.netbeans.org/schema/dibuix"
79                   base="tns:LlistaFigures"/>
80             </xsd:complexContent>
81         </xsd:complexType>
82     </xsd:element>
83 </xsd:schema>
```

Un esquema com aquest validaria documents XML semblants a:

```
1  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2  <dibuix xmlns="http://xml.netbeans.org/schema/dibuix">
3      <rectangle>
4          <posicio x="0" y="5"/>
5          <escala>1</escala>
6          <rotacio>0.0</rotacio>
7          <colorLinia>-16777216</colorLinia>
8          <colorFigura>-1</colorFigura>
9          <alcada>10</alcada>
10         <amplada>10</amplada>
11     </rectangle>
12
13     <cercle>
14         <posicio x="50" y="32"/>
15         <escala>1</escala>
16         <rotacio>0.0</rotacio>
17         <colorLinia>-16777216</colorLinia>
18         <colorFigura>-1</colorFigura>
19         <radi>24</radi>
20     </cercle>
21
22     <grup>
23         <posicio x="0" y="0"/>
24         <escala>1</escala>
25         <rotacio>0.0</rotacio>
26         <colleccio>
27             <rectangle>
28                 <posicio x="1" y="1"/>
29                 <escala>1</escala>
30                 <rotacio>0.0</rotacio>
31                 <colorLinia>-16777216</colorLinia>
32                 <colorFigura>-1</colorFigura>
33                 <alcada>100</alcada>
34                 <amplada>100</amplada>
35             </rectangle>
36             <rectangle>
37                 <posicio x="100" y="100"/>
38                 <escala>1</escala>
39                 <rotacio>0.0</rotacio>
40                 <colorLinia>-16777216</colorLinia>
41                 <colorFigura>-1</colorFigura>
42                 <alcada>10</alcada>
43                 <amplada>10</amplada>
44             </rectangle>
45             <cercle>
46                 <posicio x="50" y="55"/>
47                 <escala>1</escala>
48                 <rotacio>0.0</rotacio>
49                 <colorLinia>-16777216</colorLinia>
50                 <colorFigura>-1</colorFigura>
51                 <radi>30</radi>
52             </cercle>
53         </colleccio>
54     </grup>
55 </dibuix>
```

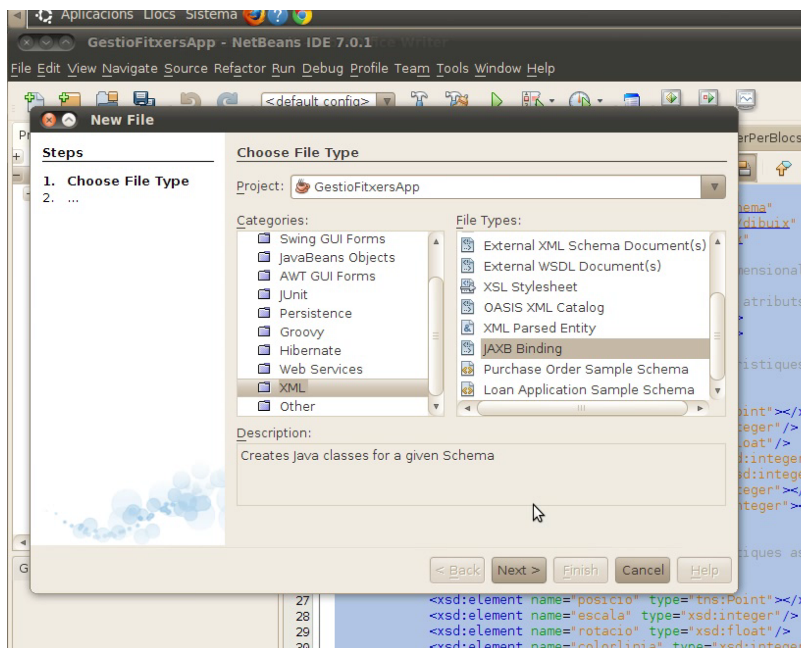
Figura 3.6. Menú per activar el Wizard de JAXB



JAXB és capaç de generar el model de dades automàticament (classes `Rectangle`, `Cercle`, `Grup` o `Dibuix`) realitzant un tractament anomenat compilació JAXB a partir de l'esquema definit més amunt. Aquest procés es pot realitzar usant comandes de consola o fent servir amb l'ajuda de l'IDE, *JAXBWizard* (figura 3.6). Sobre el projecte, cliqueu el botó dret del ratolí, seleccioneu el menú *New* i escolliu l'opció *Others*.

Això us obrirà un quadre de diàleg on caldrà que cerqueu l'opció *XML* a l'esquerra i el tipus *JAXB Binding* a la dreta (figura 3.7).

Figura 3.7. Quadre de diàleg per escollir la creació dels fitxers de configuració JAXB

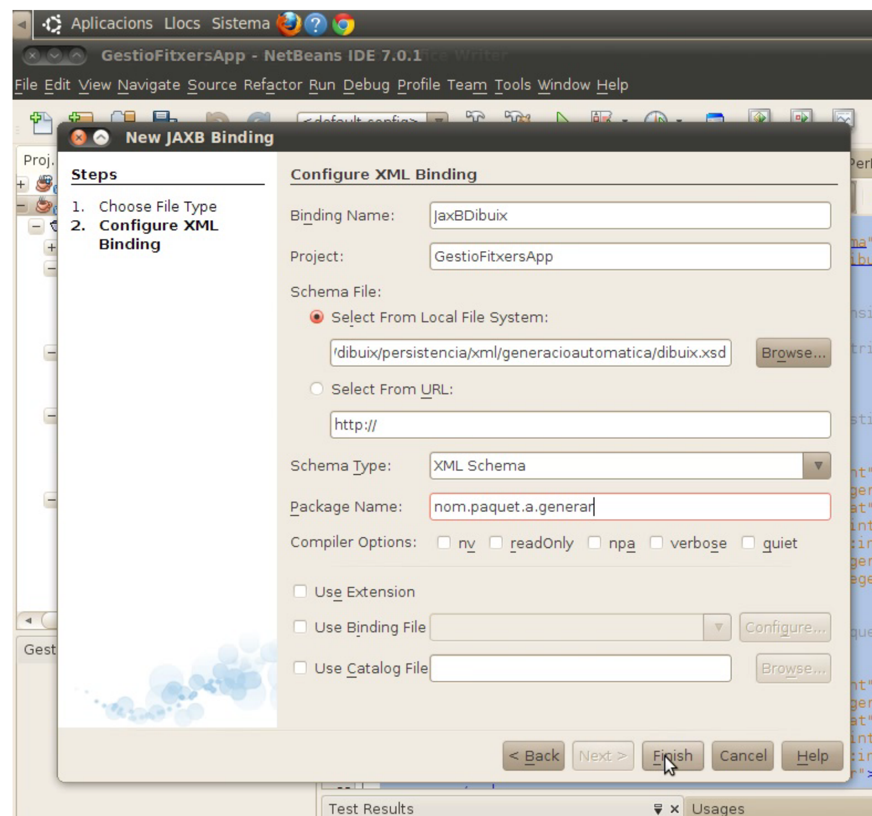


Cliqueu *Next* un cop hagueu fet la selecció correcta. D'aquesta manera, se us obrirà el darrer quadre de diàleg on caldrà que introduïu el nom de la configuració (ja que podeu generar-ne més d'una) i escolliu el fitxer esquema a partir del qual voleu fer la generació. Si cal, podeu navegar pel sistema de fitxers. Un cop escollit l'esquema, cal que introduïu el nom del paquet on desitgeu ubicar les classes generades, i finalment polseu *Finish*.

Aquest procés crearà les classes del model (les figures i el dibuix), més una classe anomenada `ObjectFactory`, la qual tindrà la funció d'instanciar els objectes de cada una de les classes del model durant el procés d'hidratació dels objectes o de traspàs de la informació des de l'XML. Si mireu el contingut, veureu també un altre fitxer anomenat *package-info.java*. No es tracta pas d'una classe, sinó d'un tipus de fitxer especial introduït a partir de la versió 5.0 de Java que permet recollir informació diversa sobre un paquet concret. Entre d'altres, admet una descripció textual sobre el propòsit general del fitxer, comentaris específics del paquet per afegir al *javadoc* i, el més important, les anotacions a nivell de paquet.

Per defecte, JAXB crearà una única *Anotació* a nivell de paquet especificant l'espai de noms que figuri al document *Schema* utilitzat per a la generació just abans de la sentència Java que identifica un paquet (figura 3.8).

FIGURA 3.8. Quadre de diàleg per configurar la generació automàtica de les classes del model a partir d'un Schema




```
1 @javax.xml.bind.annotation.XmlSchema(
2   namespace = "http://xml.netbeans.org/schema/dibuix",
3   elementFormDefault = javax.xml.bind.annotation.XmlNsForm.QUALIFIED)
4 package ioc.dam.m6.exemples.dibuix.persistencia.xml.generacioautomatica.schema;
```

Podeu observar el símbol @ com a element identificador de les *Anotacions*. Aquesta anotació, concretament, conté dos paràmetres, l'espai de noms de l'esquema i la indicació que les etiquetes descrites a l'*Schema* són qualificades i, per tant, pertanyents a l'espai de noms indicat.

La informació associada al paquet, junt amb la classe `ObjectFactory`, servirà a JAXB per crear el context de treball: una instància de *JAXContext* que usarà aquesta informació per crear uns objectes `Marshaller` o `UnMarshaller` específics per treballar amb el nostre model. La instrucció per crear el context és la següent:

```
JAXBContext context = JAXBContext.newInstance(nomPaquet);
```

L'objecte `Marshaller` l'instancia el context, fent:

```
Marshaller marshaller = context.createMarshaller();
```

Per realitzar transvasament d'informació del model al document XML serà necessari l'objecte del model corresponent a l'objecte contenidor principal de tota la informació (és a dir, l'equivalent al qual serà l'arrel del document XML) i un OutputStream associat al fitxer d'emmagatzematge.

```
marshaller.marshal(objecteRoot, fitxerOutputStream);
```

En el nostre model de figures, l'objecteRoot seria el dibuix.

La instància *UnMarshaller* també la crearà el context:

```
Unmarshaller unmarshaller = context.createUnmarshaller();
```

Per obtenir un objecte contenidor a partir de les dades emmagatzemades en un XML caldrà passar per paràmetre un `FileInputStream` del fitxer. El mètode `unmarshal`gestionarà tota la creació d'objectes necessaris per realitzar el transvasament d'informació cap al model. La instanciació dels objectes es realitzarà fent servir la classe `ObjectFactory`. El mètode `unmarshal`retornarà un objecte corresponent al contenidor principal i l'equivalent a l'arrel del document XML.

```
objectRoot = unmarshaller.unmarshal(fitxerInputStream);
```

Podeu fer la prova instanciant un objecte Dibuix al qual li afegiu unes quantes figures.

Vigileu que les instàncies siguin realment de les classes generades per JAXB i no pas les usades a l'apartat "Implementació d'un exemple" de la Secció "Parser o analitzador sintàctic" i que formen part de la biblioteca del mòdul. Recordeu que, de moment, el vincle s'ha establert amb les classes generades de forma automàtica.

3.5.3 Ús de JAXB amb un model de disseny propi

Normalment, el models de dades solen ser la peça clau de les aplicacions, i generar-los automàticament a partir d'un Schema no és, de ben segur, la millor manera d'optimitzar-lo. Es pot modificar l'Schema per aconseguir millors resultats, però a vegades és preferible plantejar les coses al revés. És a dir, partint de les classes, fem

les anotacions oportunes per aconseguir generar XML que validin un determinat Schema.

Segurament, la generació automàtica pot ser útil quan sigui necessari disposar ràpidament d'un model de classes Java a partir d'un model XML existent. Però si el punt de partida són les classes Java, la solució passa per escriure notacions sobre el model o sobre alguna classe derivada.

Cal que entengueu la derivació com a herència.

Si partim d'un model dissenyat totalment per nosaltres caldrà crear, a banda de les classes del model, la classe `ObjectFactory`i el fitxer *package-info.java*. L'`ObjectFactory`tindrà almenys un mètode de creació per a cada classe del model. Si fos necessari, una classe podria disposar de més d'un mètode d'instanciació, de la mateixa manera que fos necessari tenir diversos constructors.

Si, en canvi, partíssim d'un model derivat d'una altre al qual no hi tinguéssim accés, perquè fos un biblioteca de tercers, perquè no disposéssim de les fonts o per qualsevol altre motiu, només podríem anotar les classes derivades. Com ja hem vist, les *Anotacions* tenen diferents àmbits d'influència segons el tipus. Hi ha anotacions de paquets que afecten totes les seves classes, però hi ha també anotacions de classes especifiques, el rang de les quals afectaria només a la classe on s'ha definit i als seus components, mètodes, atributs, etc.

Un problema que trobarem quan treballem amb herència usant JAXB és que les *Anotacions* de classes afecten només la classe específica on s'han escrit les anotacions, i no serveixen per a les superclasses. No és possible, per exemple, anotar en la classe `Cercle` un atribut de `Figura`. Si necessitem anotar l'atribut caldrà fer-ho dins la mateixa classe `Figura`. Però si no hi tinguéssim accés, existeixen diverses solucions. Aquí plantejarem la més laboriosa, però també la que permet un major grau de configuració. La tècnica consisteix, d'una banda, a evitar que JAXB faci cap exploració de les classes que no puguem anotar. De l'altra, a dotar a les classes derivades d'accessors (mètodes `get`i `set`) propis per accedir als atributs heretats.

És possible optar per solucions intermèdies si la seriació per defecte que fa JAXB ja ens anés bé. En aquest cas, només caldria derivar la classe que representi l'etiqueta arrel per poder anotar-la com a tal.

Anotacions bàsiques

Per introduir les anotacions bàsiques de JAXB continuarem amb l'exemple del dibuix de figures geomètriques. Implementarem dues versions: en una, crearem el model des de zero i escriurem *anotacions* en qualsevol de les classes creades. A la segona versió, partirem del model que trobem a la biblioteca de manera que per anotar-lo ens calgui crear un model derivat. Farem les implementacions en paral·lel, i així copsarem millor les diferències.

Preparació dels paquets del model
Pel que fa a la implementació des de zero, en primer lloc caldrà crear el paquet. Per exemple:

```
ioc.dam.m6.exemples.dibuix.persistencia.xml.model
```

Per al model derivat, el paquet podria ser:

```
ioc.dam.m6.exemples.dibuix.persistencia.xml.modelderivat
```

També caldrà evitar que JAXB explori les classes originals de la biblioteca impedint-ne la seriació a l'XML. En el nostre projecte crearem un paquet amb el mateix nom que el paquet on es trobi el model original (el de la biblioteca). Un cop creat, hi ubicarem únicament un fitxer *package-info.java,* en el qual anotarem la deshabilitació dels accessors fent servir `@XmlAccessorType`, al qual li passarem per paràmetre el valor de la constant `@XmlAccessType.NONE`, la qual indicarà que les classes del paquet associat no seran accessibles des de JAXB.

```
@javax.xml.bind.annotation.XmlAccessorType(
  value= javax.xml.bind.annotation.XmlAccessType.NONE)
package ioc.dam.m6.exemples.dibuix;
```

Podreu crear un fitxer *package-info.java* clicant *new File*, escollint la categoria Java i seleccionant el tipus de fitxer *Java Package Info*.

Per a la creació del model, en la versió de la implementació total (des de zero) caldrà crear les mateixes classes (`Cercle`, `Rectangle`, `Grup`, `Figura`, `Dibuix`, etc.) que el model de la biblioteca.

Per al model derivat, caldrà implementar `XmlDibuix`, `XmlCercle`, `XmlRectangl` i `XmlGrup` heretant de les classes corresponents i implementant els accessors adequats. Veiem com a exemple `XmlCercle`:

```
public class XmlCercle extends Cercle {

    public XmlCercle() {
    }

    public XmlCercle(int x, int y, int radi) {
        super(x, y, radi);
    }

    @Override
    public Point getPosicio() {
        return super.getPosicio();
    }

    @Override
    public void setPosicio(Point posicio) {
        super.setPosicio(posicio);
    }

    @Override
    public Color getColorFigura() {
        return super.getColorFigura();
    }

    @Override
    public void setColorFigura(Color colorFigura) {
        super.setColorFigura(colorFigura);
    }

    @Override
    public Color getColorLinia() {
        return super.getColorLinia();
    }

```

```
    @Override
    public void setColorLinia(Color colorLinia) {
        super.setColorLinia(colorLinia);
    }

    @Override
    public int getEscala() {
        return super.getEscala();
    }

    @Override
    public void setEscala(int escala) {
        super.setEscala(escala);
    }

    @Override
    public float getRotacio() {
        return super.getRotacio();
    }

    @Override
    public void setRotacio(float rotacio) {
        super.setRotacio(rotacio);
    }

    @Override
    public int getRadi() {
        return super.getRadi();
    }

    @Override
    public void setRadi(int radi) {
        super.setRadi(radi);
    }
}
```

De forma semblant, caldrà implementar `XmlRectagle` i `XmlGrup`. D'aquesta darrera cal comentar que l'atribut `elements`necessita també accessors per tal que JAXB hi pugui accedir.

```
public class XmlGrup extends Grup{
    public ArrayList<Figura> getElements(){
        return elements;
    }

    public void setElements(ArrayList<Figura> list){
        elements=list;
    }

    ...
}
```

Finalment, a la `classeXmlDibuix` hi afegirem un nou mètode per copiar la llista de figures d'un dibuix a un altre.

```
public class XmlDibuix extends Dibuix {
    public XmlDibuix() {
    }

    public List<Figura> getFigures(){
        return this;
    }

    public void copiarDesDe(Dibuix dibuix){
        this.clear();
        this.addAll(dibuix);
```

```
    }
}
```

Per a ambdues versions, i amb l'objectiu que JAXB reconegui les classes que ha de manipular i *mapar*, caldrà crear un arxiu *package-info.java* i un `ObjectFactory`a cada paquet.

Al fitxer amb la informació del paquet hi escriurem l'anotació `@XmlSchema`, que ens permetrà definir un espai de noms (http://xml.ioc.edu/schema/dibuix, per exemple) i indicar si els noms són o no qualificats. També indicarem de forma exclusiva en la versió del model derivat, que l'accés a les dades vinculades amb l'XML es realitzarà per mitjà d'accessors (mètodes `getiset`) usant el valor de la constant *XmlAccessType.PROPERTY*, mentre que el tipus d'accés per al model propi serà directe als atributs. Usarem el valor `XmlAccessType.FIELD`.

El contingut del *package-info.java* del model derivat és el següent:

```
@javax.xml.bind.annotation.XmlSchema(
  namespace ="http://xml.ioc.edu/schema/dibuix",
  elementFormDefault = javax.xml.bind.annotation
                                   .XmlNsForm.QUALIFIED)
@javax.xml.bind.annotation.XmlAccessorType(
    value= javax.xml.bind.annotation
                               .XmlAccessType.PROPERTY)
package ioc.dam.m6.exemples.dibuix.persistencia.xml.modelderivat;
```

I el del *package-info.java* del model propi, aquest:

```
@javax.xml.bind.annotation.XmlSchema(
    namespace = "http://xml.ioc.edu/schema/dibuix",
    elementFormDefault = javax.xml.bind
                            .annotation.XmlNsForm.QUALIFIED)
@javax.xml.bind.annotation.XmlAccessorType(
    value=javax.xml.bind.annotation.XmlAccessType.FIELD)
package ioc.dam.m6.exemples.dibuix.persistencia.xml.model;
```

Seguidament, caldrà crear i adaptar les classes *factory* que per defecte usa JAXB. Crearem en cada paquet una classe `ObjectFactory` que anotarem amb `@XmlRegistry`. Aquesta anotació indica a JAXB que es tracta d'una classe instanciadora i que, per tant, analitzant els seus mètodes es poden obtenir totes les classes usades durant la vinculació.

En el paquet del model propi, la classe `ObjectFactory` tindrà mètodes per generar rectangles, cercles, grups o dibuixos. Els mètodes instanciadors es compondran del prefix "*create*" i d'un sufix amb el nom que l'XML usarà com a referència d'aquesta classe:

```
@XmlRegistry
public class ObjectFactory {

    public ObjectFactory() {
    }

    public Grup createGrup() {
        return new Grup();
    }

    public Rectangle createRectangle() {
```

```
        return new Rectangle();
    }

    public Rectangle createRectangle(int x, int y,
                                    int amplada, int alcada) {
        return new Rectangle(x, y, amplada, alcada);
    }

    public Cercle createCercle() {
        return new Cercle();
    }

    public Cercle createCercle(int x, int y, int radi) {
        return new Cercle(x, y, radi);
    }

    public Dibuix createDibuix() {
        return new Dibuix();
    }
}
```

Per facilitar la codificació de crear objecte de tipus rectangle i cercle, afegirem dos mètodes més que acceptin paràmetres d'inicialització. Aquest mètodes no podran substituir els instanciadors per defecte, ja que JAXB usa sempre instanciadors sense paràmetres.

De forma semblant, en el paquet del model derivat:

```
@XmlRegistry
public class ObjectFactory {

    public ObjectFactory() {
    }

    public XmlGrup createGrup() {
        return new XmlGrup();
    }

    public XmlRectangle createRectangle() {
        return new XmlRectangle();
    }

    public XmlRectangle createRectangle(int x, int y, int amplada,
        int alcada) {
        return new XmlRectangle(x, y, amplada, alcada);
    }

    public XmlCercle createCercle() {
        return new XmlCercle();
    }

    public XmlCercle createCercle(int x, int y, int radi) {
        return new XmlCercle(x, y, radi);
    }

    public XmlDibuix createDibuix() {
        return new XmlDibuix();
    }
}
```

Anotacions de classe

L'anotació `@XmlElementRoot` serveix per indicar quina classe o quines classes són candidates a usar-se com a node arrel del document XML. Les instàncies d'aquestes classes poden accedir de manera directa o indirecta a tota la informació que s'emmagatzemarà al'XML. Cal recordar que XML no admet documents amb arrels múltiples i, per tant, no és possible seriar les dades a partir de dos o més objectes. En el nostre cas, la classe `arrel` correspon a `Dibuix`. Afegiu sobre la declaració de la classe l'`Anotació`.

```
1 @XmlRootElement(name="dibuix")
2 public class Dibuix extends ArrayList<Figura>{
3     ...
```

@XmlElementRoot accepta el paràmetre `nom`, el qual indica el nom que tindrà l'etiqueta arrel en el document XML. Feu el mateix a la classe `XmlDibuix`.

Emmagatzemeu els canvis i obriu la classe `Figura`. Afegirem una notació per indicar l'ordre en què voldrem escriure els seus atributs.

```
1 @XmlType(propOrder={"posicio","escala","rotacio"})
2 public abstract class Figura {
3     ...
```

L'Anotació `@XmlType` permet, entre d'altres coses, especificar l'ordre en què s'escriuran els seus camps. Observeu a la sintaxi com les Anotacions reben un conjunt de dades: el conjunt de dades es tanca entre claus, i per separar cada una de les dades s'intercala una coma.

Aquesta *Anotació* és opcional. De fet, si no la poséssim, tant el *marshaller* com l'*unmarshaller* funcionarien sense cap problema. De tota manera, es tracta d'una *anotació* important perquè sovint els XML han de complir DTD o Schemes força restrictius que requereixen un ordre determinat de les etiquetes.

Farem quelcom semblant a la classe `FiguraSimple` i `Rectangle`.

```
1 @XmlType(propOrder = {"colorLinia","colorFigura"})
2 public class FiguraSimple extends Figura{
3     ...
4 
5 @XmlType(propOrder = {"alcada","amplada"})
6 public class Rectangle extends FiguraSimple{
7     ...
```

No és necessari definir l'ordre dels atributs de la classe `Cercle` ni la classe `Grup`, perquè només en tenen un.

Les operacions equivalents al paquet del model derivat haurien de poder definir el mateix ordre, però com que no tenim accés a les classes derivades caldrà fer les anotacions a la pròpia classe, que és on hi trobarem els accessors (`gets`i `sets`). A més, caldrà indicar que el tipus XML que aquesta classe representa no coincideix amb el seu nom.

```
@XmlType(name="Rectangle",
      propOrder={"posicio","rotacio","escala","colorFigura",
                  "colorLinia", "alcada","amplada"})
public class XmlRectangle extends Rectangle{
    ...
```

Cal realitzar la mateixa operació amb totes i cada una de les classes derivades:

```
@XmlType(name="Cercle",
      propOrder={"posicio","rotacio","escala", "colorFigura",
                  "colorLinia","radi"})
public class XmlCercle extends Cercle {
    ...

@XmlType(name="Grup",
      propOrder={"posicio","rotacio","escala","elements"})
public class XmlGrup extends Grup{
    ...
```

Anotacions de camps, mètodes o accessors

De vegades caldrà matisar vincles específics pels elements de les classes. Podem, per exemple, evitar que el valor d'un determinat camp es traspassi a l'XML. Això ho aconseguirem marcant-lo amb la notació `@XmlTransient`. És el que farem amb el camp TipusFigura de la classe `Figura` del model propi.

```
@XmlType(propOrder={"posicio","escala","rotacio"})
public abstract class Figura {
    @XmlTransient
    private TipusFigura tipusFigura=TipusFigura.INDETERMINADA;
    private Point posicio=null;
    ...
```

En el model derivat no cal fer-ho, perquè ja hem impedit l'accés a nivell del paquet. Malgrat tot, caldrà definir altres anotacions. Com que no hi ha accés directe als camps, JAXB no sabrà quin nom posar a les etiquetes de l'XML i, per defecte, JAXB usarà sempre el nom del camp associat. En el model derivat, caldrà indicar a l'Anotació el nom desitjat a l'etiqueta. Les Anotacions s'han de fer només a un dels dos accessors, i per convenció solen posar-se en els mètodes de lectura (`get`), però s'accepten a qualsevol dels dos.

Les propietats comunes caldrà repetir-les a totes les classes:

```
...

    @XmlElement(name="posicio")
    @Override
    public Point getPosicio() {
        return super.getPosicio();
    }
    ...

    @XmlElement(name="escala")
    @Override
    public int getEscala() {
        return super.getEscala();
    }
    ...
```

```
16
17    @XmlElement(name="rotacio")
18    @Override
19    public float getRotacio() {
20        return super.getRotacio();
21    }
22    ...
23
24    @XmlElement(name="colorFigura")
25    @Override
26    public Color getColorFigura() {
27        return super.getColorFigura();
28    }
29    ...
30
31    @XmlElement(name="colorLinia")
32    @Override
33    public Color getColorLinia() {
34        return super.getColorLinia();
35    }
36    ...
```

I les específiques, a les classes corresponents:

```
1  @XmlElement(name="alcada")
2      @Override
3      public int getAlcada() {
4          return super.getAlcada();
5      }
6      ...
7
8      @XmlElement(name="amplada")
9      @Override
10     public int getAmplada() {
11         return super.getAmplada();
12     }
13     ...
```

El tractament de col·leccions també s'especifica a nivell d'element o de propietat. Per defecte, les col·leccions Java es plasmaran al'XML com una seqüència d'elements descendents directes de la classe contenidora. És a dir, en el cas dels *grups*, que contenen una llista de *figures*, si no s'indica res més, la seqüència de figures es penjarà directament de l'etiqueta *grup*. No és això el que volem. De fet, es vol fer dependre la seqüència de *figures* d'un element extra anomenat *colleccio*, el qual formaria part de l'element grup i contindria les *figures* de forma directa. JAXB disposa de l'*Anotació* `@XmlElementWrapper` per indicar la necessitat d'una etiqueta contenidora extra.

D'altra banda, els elements de la llista de figures a vegades seran rectangles, d'altres cercles i d'altres grups. La manera d'associar un tipus de dada amb una etiqueta es pot especificar aquí usant l'Anotació `@XmlElements`, a la qual se li indicarà usant una col·lecció de `@XmlElement` quines classes s'associaran a quines etiquetes.

```
1  public class Grup extends Figura{
2      @XmlElementWrapper(name="colleccio", required=true)
3      @XmlElements({
4          @XmlElement(name="rectangle",type=Rectangle.class),
5          @XmlElement(name="cercle",type=Cercle.class),
6          @XmlElement(name="grup",type=Grup.class)
7      })
8      ArrayList<Figura> elements = new ArrayList<Figura>();
9      ...
```

El mateix cal fer a la classe XmlGrup, però associant les classes corresponents.

```
@XmlElementWrapper(name="colleccio", required=true)
    @XmlElements({
        @XmlElement(name="rectangle",type=XmlRectangle.class),
        @XmlElement(name="cercle",type=XmlCercle.class),
        @XmlElement(name="grup",type=XmlGrup.class)
    })
    public ArrayList<Figura> getElements(){
        return elements;
    }
```

La classe Dibuix és també una llista de característiques molt similars a la classe Grup. Caldrà, doncs, una anotació semblant. Abans, però, haurem de salvar un problema: l'Anotació @XmlElements només pot ser definida a nivell de mètode o d'atribut i no pas de classe. Com que Dibuix hereta d'ArrayList, no disposa de cap atribut contenidor, la solució passa per crear un accessor de lectura on es pugui especificar la notació:

```
@XmlRootElement(name="dibuix")
public class Dibuix extends ArrayList<Figura>{
    @XmlElements({
        @XmlElement(name="rectangle",type=Rectangle.class),
        @XmlElement(name="cercle",type=Cercle.class),
        @XmlElement(name="grup",type=Grup.class)
    })
    public ArrayList<Figura> getFigures(){
        return this;
    }
    ...
```

Cal, també, realitzar una anotació idèntica a la classe XmlDibuix, substituint les classes originals per les derivades corresponents.

```
@XmlElements({
    @XmlElement(name="rectangle",type=XmlRectangle.class),
    @XmlElement(name="cercle",type=XmlCercle.class),
    @XmlElement(name="grup",type=XmlGrup.class)
})
public List<Figura> getFigures(){
    return this;
}
```

Rectificació de les anotacions a nivell de paquet

Malauradament, un cop s'han realitzat totes aquestes modificacions, si fem la prova intentant seriar una instància de Dibuix ens saltarà una excepció. Això és degut al fet que el nostre model utilitza dues classes pròpies de Java que JAXB no sap *mapar*. Ens referim a Point i a Color. JAXB necessita que totes les classes del model tinguin un constructor per defecte (constructor sense paràmetres) i que els atributs que calgui seriar tinguin accessors de lectura i escriptura (get i set). La classe Color no disposa de constructor per defecte, i la classe Point té un accessor de tipus *double* corresponent a un atribut de tipus enter i un accessor anomenat getLocation que no es correspon a cap atribut.

Per solucionar aquests problemes, JAXB disposa d'unes anotacions que permeten declarar classes o tipus alternatius, així com la forma de passar d'un tipus a un

altre. Ens referim a @XmlJavaTypeAdapter i a la classe XmlAdapter. La primera és l'Anotació amb la qual s'informarà de quines classes han de fer d'adaptadors. La segona és una classe abstracta i parametritzada amb els tipus que caldrà intercanviar, d'on qualsevol adaptador haurà de derivar.

Per exemple, en passar al'XML, volem que les figures escriguin el color com si fos un valor numèric. Crearem una Classe que hereti de XmlAdapter, que haurà d'implementar els seus dos mètodes abstractes anomenats unmarshal i marshal. El mètode unmarshalpermet convertir el valor obtingut des del'XML a un valor de tipus vàlid en el llenguatge Java. En el nostre cas, rebrà un enter per paràmetre i el transformarà a Color.

```
public class ColorAdapter extends XmlAdapter<Integer, Color> {

    @Override
    public Color unmarshal(Integer v) throws Exception {
        return new Color(v.intValue());
    }

    @Override
    public Integer marshal(Color v) throws Exception {
        return new Integer(v.getRGB());
    }

}
```

El mètode marshal fa l'operació inversa: rep per paràmetre el valor provinent de la classe Java i el converteix en un valor adequat per emmagatzemar al'XML.

Farem el mateix amb la classe Point. La diferència és, però, que els punts no poden transformar-se en un tipus simple, ja que sempre es necessiten dues coordenades. La classe alternativa la crearem nosaltres mateixos assegurant-nos que compleix els requisits de JAXB:

```
public class Posicio implements Serializable{
    private int coordenadaX;
    private int coordenadaY;
    public Posicio() {
    }

    public Posicio(Point p){
        this.coordenadaX=p.x;
        this.coordenadaY=p.y;
    }

    @XmlAttribute(name="x")
    public int getCoordenadaX() {
        return coordenadaX;
    }

    public void setCoordenadaX(int coordenadaX) {
        this.coordenadaX = coordenadaX;
    }

    @XmlAttribute(name="y")
    public int getCoordenadaY() {
        return coordenadaY;
    }

    public void setCoordenadaY(int coordenadaY) {
        this.coordenadaY = coordenadaY;
    }
}
```

També implementarem el seu adaptador:

```
public class PointAdapter extends XmlAdapter<Posicio, Point> {

    @Override
    public Point unmarshal(Posicio v) throws Exception {
        return new Point(v.getCoordenadaX(), v.getCoordenadaY());
    }

    @Override
    public Posicio marshal(Point v) throws Exception {
        return new Posicio(v);
    }

}
```

Finalment, per tal que els canvis tinguin efecte, caldrà declarar-ho en forma d'Anotacions. Això ho farem a nivell de paquet. La notació rep dos paràmetres: el paràmetre *type* identifica la classe incompatible, i el paràmetre *value* especificarà per quina classe caldrà substituir-la. Per tal que es puguin definir tants adaptadors com sigui necessari, aquests es passaran per paràmetre de l'anotació `XmlJavaTypeAdapters`.

```
@javax.xml.bind.annotation.XmlSchema(
   namespace = "http://xml.netbeans.org/schema/dibuix",
   elementFormDefault = javax.xml.bind.annotation.XmlNsForm.QUALIFIED)

@javax.xml.bind.annotation.adapters.XmlJavaTypeAdapters({
    @javax.xml.bind.annotation.adapters.XmlJavaTypeAdapter(
        value=ioc.dam.m6.exemples.dibuix.persistencia.xml.binding.
            ColorAdapter.class,
        type=java.awt.Color.class),
    @javax.xml.bind.annotation.adapters.XmlJavaTypeAdapter(
        value=ioc.dam.m6.exemples.dibuix.persistencia.xml.binding.
            PointAdapter.class,
        type=java.awt.Point.class)
})

package ioc.dam.m6.exemples.dibuix.persistencia.xml.binding.model;
```

Implementació del controlador

Tal com hem fet en anteriors ocasions, crearem una classe `Controlador` que aglutini un conjunt d'utilitats per facilitar les crides a la seriació i deseriació en format XML usant les eines JAXB. Bàsicament, la Utilitat que construirem ha de facilitar la creació d'un context específic, així com l'escriptura des del model al'XML i la lectura des del'XML al model.

Aquesta classe l'anomenarem `BindingCtrl`. Contindrà una instància d'Utilitats, un `File` referenciant el fitxer d'emmagatzematge XML i una cadena representant el nom del paquet on es trobaran les classes del model de dades. Crearem un mètode *init* que inicialitzarà el controlador creant un JAXBContext. També disposarà d'utilitats per realitzar la seriació/deseriació a l'XML.

```
public class BindingCtrl {
    protected Utilitats utilitats = new Utilitats();
    protected File file = null;
    protected boolean init = false;
    protected JAXBContext context;
```

```
6       protected String nomPaquet=null;
7   
8       public BindingCtrl() {
9       }
10  
11      public BindingCtrl(String ctx) {
12          nomPaquet = ctx;
13      }
14  
15      public File getFile() {
16          return file;
17      }
18  
19      public void init() throws JAXBException {
20          if(nomPaquet!=null){
21              context = JAXBContext.newInstance(nomPaquet);
22          }else{
23              context = JAXBContext.newInstance(classFactory);
24          }
25          init=true;
26      }
27  
28      public void setFile(File file) {
29          this.file = file;
30      }
31  
32      protected void escriu(Object object) throws BindingCtrlException{
33          FileOutputStream out = null;
34          try {
35              if(!init){
36                  init();
37              }
38              Marshaller marshaller = context.createMarshaller();
39              marshaller.setProperty(Marshaller.JAXB_ENCODING, "UTF-8");
40              marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT,
41                                          true);
42              out = new FileOutputStream(file);
43              marshaller.marshal(object, out);
44          } catch (FileNotFoundException ex) {
45              throw new BindingCtrlException(ex);
46          } catch (JAXBException ex) {
47              throw new BindingCtrlException(ex);
48          } finally {
49              utilitats.intentarTancar(out);
50          }
51      }
52  
53      protected Object llegeix() throws BindingCtrlException {
54          Object object = null;
55          FileInputStream in=null;
56          try {
57              if(!init){
58                  init();
59              }
60              Unmarshaller unmarshaller = context.createUnmarshaller();
61              in = new FileInputStream(file);
62              object = unmarshaller.unmarshal(in);
63          } catch (FileNotFoundException ex) {
64              throw new BindingCtrlException(ex);
65          } catch (JAXBException ex) {
66              throw new BindingCtrlException(ex);
67          } finally {
68              utilitats.intentarTancar(in);
69          }
70          return object;
71      }
72  }
```

A partir d'aquesta classe, per herència podem crear controladors específics per cada model concret. Per exemple, la del model derivat s'implementarà fent el següent:

```
public class ModelDerivatCtrlDibuix extends BindingCtrl{
    Dibuix root;

    public ModelDerivatCtrlDibuix(){
        this(new Dibuix());
    }

    public ModelDerivatCtrlDibuix(Dibuix dibuix){
        super(dibuix.getClass().getName().substring(0,
                    dibuix.getClass().getName().lastIndexOf(".")));
        this.root = dibuix;
    }

    public ModelDerivatCtrlDibuix(Dibuix dibuix, File file){
        this(dibuix);
        this.file=file;
    }

    public void emmagatzemar() throws xmlDibuixException{
        try {
            escriu(root);
        } catch (BindingCtrlException ex) {
            throw new xmlDibuixException(ex);
        }
    }

    public void recuperar() throws xmlDibuixException {
        try {
            root.clear();
            root.addAll((Dibuix) this.llegeix());
        } catch (BindingCtrlException ex) {
            throw new xmlDibuixException(ex);
        }
    }

    public Dibuix getDibuix() {
        return root;
    }

}
```

Persistència en BDR-BDOR-BDOO

Josep Cañellas Bornas

Accés a dades

Índex

Introducció 5

Resultats d'aprenentatge 7

1 Gestió de connectors 9
1.1 El desfasament objecte-relacional . . . 9
1.1.1 Model relacional . . . 10
1.1.2 Model orientat a l'objecte . . . 11
1.1.3 Desfasaments . . . 13
1.2 Connectors . . . 14
1.2.1 ODBC . . . 14
1.2.2 JDBC . . . 17
1.3 Iniciació a l'API JDBC . . . 23
1.3.1 Càrrega de controladors . . . 24
1.3.2 Establint connexió . . . 25
1.3.3 Fent peticions SQL bàsiques . . . 28
1.3.4 Exemple senzill . . . 31
1.4 JDBC Avançat . . . 37
1.4.1 Tractament d'errors en aplicacions JDBC . . . 37
1.4.2 Millora del rendiment . . . 41

2 Eines de mapatge objecte-relacional (ORM) 45
2.1 Concepte de mapatge (objecte-relacional) . . . 45
2.2 Eines de mapatge . . . 46
2.2.1 Tècniques de mapatge . . . 46
2.2.2 Llenguatge de consulta . . . 47
2.2.3 Tècniques de sincronització . . . 47
2.3 JPA (Java Persistence API) . . . 48
2.3.1 Característiques generals del JPA . . . 49
2.4 Peces bàsiques del JPA . . . 49
2.4.1 Entitats . . . 49
2.4.2 El gestor d'entitats . . . 50
2.4.3 Unitats de persistència . . . 50
2.4.4 El fitxer XML que conté el mapatge . . . 52
2.4.5 Transaccions i excepcions . . . 53
2.5 Ús de JPA amb EclipseLink sobre NetBeans i PostgreSQL . . . 53
2.6 Metadades per definir la persistència . . . 58
2.6.1 Marcant entitats . . . 59
2.6.2 Generació automàtica de la clau primària . . . 60
2.6.3 Marcant relacions . . . 64
2.6.4 Modificant les opcions per defecte de JPA . . . 67
2.6.5 Relacions unidireccionals i bidireccionals . . . 72
2.6.6 Objectes incrustats (Embedded Objects) . . . 79

2.6.7 Col·leccions d'objectes bàsics i objectes incrustats . . . 86
2.6.8 Identificadors compostos . . . 96
2.6.9 Herència . . . 103
2.7 Funcionalitat del gestor d'entitats . . . 112
2.7.1 Creació d'una unitat de persistència per configurar la connexió a l'SGBD . . . 112
2.7.2 Obtenció d'un EntityManager . . . 113
2.7.3 Gestió d'entitats a l'EntityManager . . . 114
2.8 El llenguatge de consulta JPQL . . . 118
2.8.1 Parametrització de sentències . . . 120
2.8.2 Consultes predefinides o Named Queries . . . 121
2.8.3 JPQL en detall . . . 123

3 Bases de dades objecte-relacionals i orientades a objectes 131
3.1 Definició de dades en sistemes Objecte-Relacionals . . . 131
3.1.1 Suport de PostgreSQL als tipus definits per SQL99 . . . 132
3.2 Manipulació de dades en sistemes de gestió Objecte-Relacionals . . . 134
3.2.1 Tipus especials de dades . . . 134
3.2.2 Ús d'arrays en sentències DML . . . 135
3.2.3 Ús de tipus estructurats en sentències DML . . . 137
3.2.4 Tractament d'objecte en aplicacions que usin JDBC 2.0 o superior . . . 139
3.3 Bases de Dades Orientades a Objecte . . . 142
3.3.1 Ús d'ObjectDB amb NetBeans i JPA . . . 144
3.3.2 Ús del servidor i el client d'ObjectDB . . . 147

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.

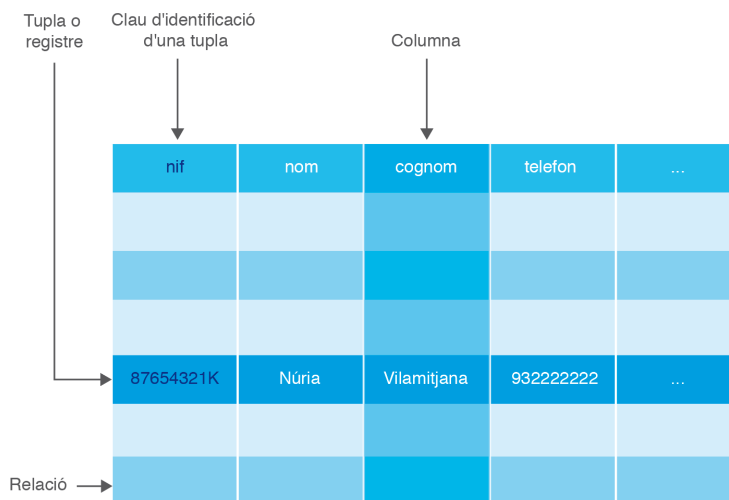
El problema és que els models conceptuals són representacions mentals fruit d'un procés d'abstracció. No hi ha una única forma de plasmar-los o representar-los fora del nostre cervell. Sovint fem servir aproximacions esquemàtiques que poden acostar-se força a la representació mental, però fins i tot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

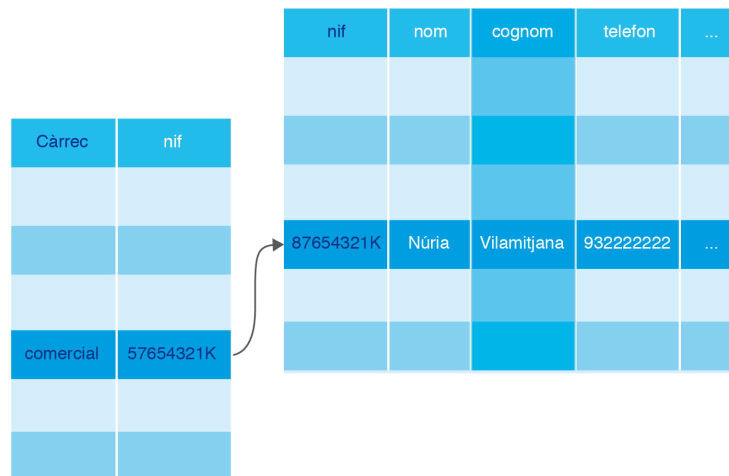
El model relacional és una forma d'organitzar les dades d'una aplicació agrupant-les segons la relació que es pugui establir entre elles d'acord amb el model. Aquesta agrupació es materialitza en forma de taula, on distingim les columnes o conjunt de dades que representen un mateix concepte del model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

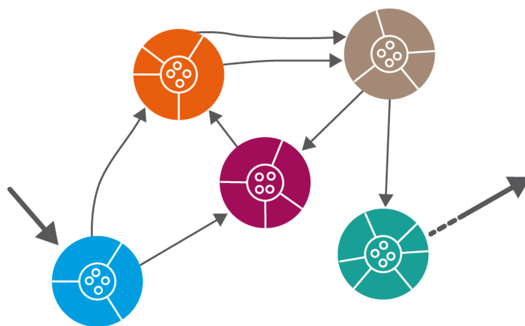
La tecnologia dels objectes ha fet en els darrers temps un esforç molt important per regular i estandarditzar les representacions esquemàtiques dels models conceptuals. El resultat d'aquests esforços ha conduït a un llenguatge esquemàtic però molt expressiu anomenat UML.

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

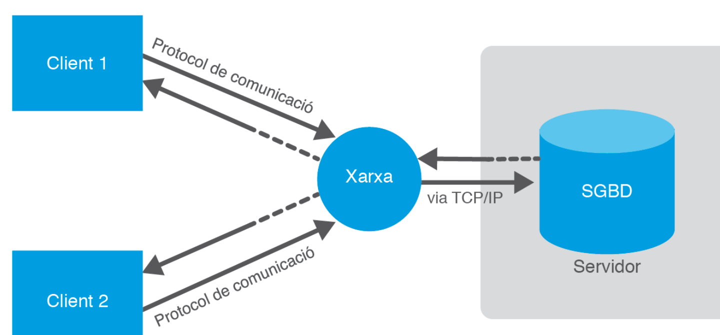
La història de les bases de dades relacionals ha transcorregut íntimament lligada a la història de les aplicacions d'arquitectura distribuïda i en particular, arquitectures client-servidor. Fa poc més d'un quart de segle, la persistència de les aplicacions formava part del desenvolupament d'aquestes, 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.

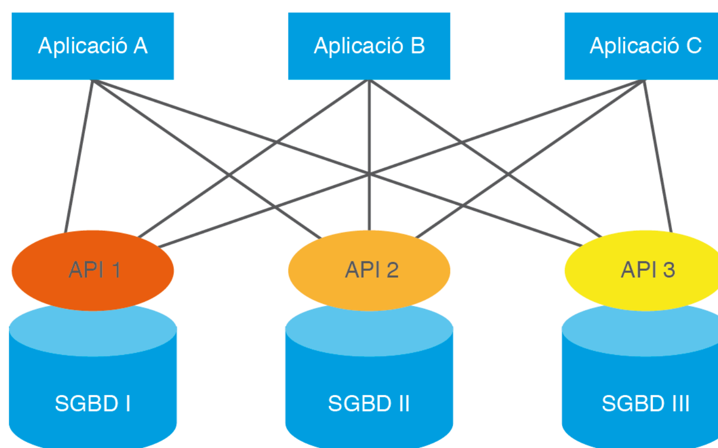
A poc a poc, el programari al voltant de les bases de dades va créixer espectacularment intentant donar resposta a un màxim ventall de demandes a traves de sistemes altament configurables. És el que avui dia es coneix com a *middleware* o capa intermèdia de persistència. És a dir, el conjunt d'aplicacions, 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).

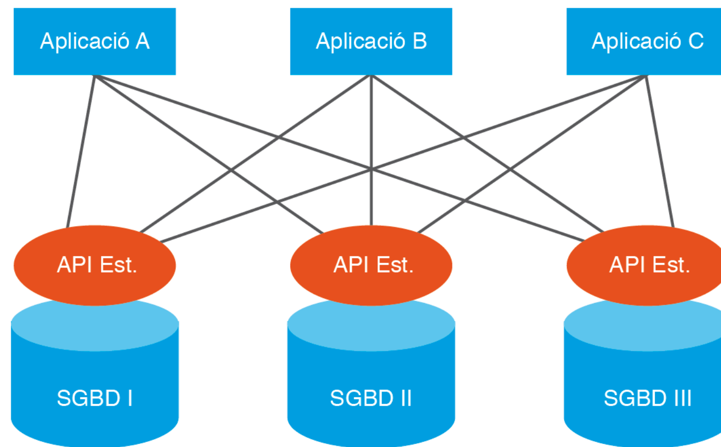
Figura 1.5. Sistema de connexió propietari



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

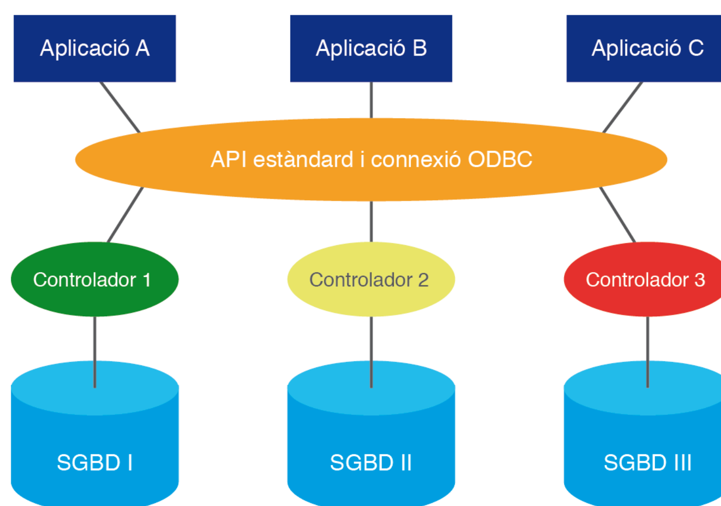
El grup anomenat SQL Access Group, en el qual hi participaven prestigioses empreses del sector com Oracle, Informix, Ingres, DEC, Sun o HP, va definir un API universal amb independència del llenguatge de desenvolupament i la base de dades a connectar (figura 1.6).

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.

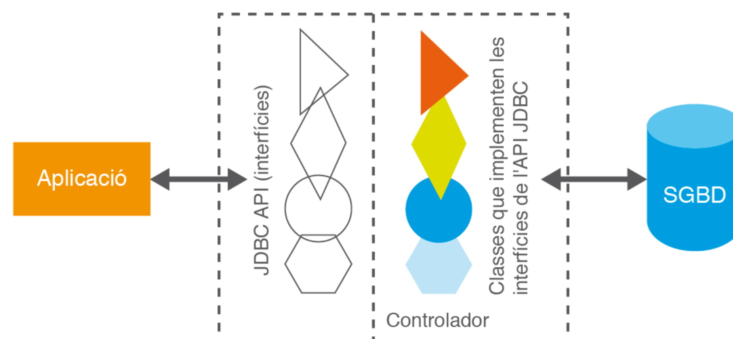
Actualment, la gran majoria d'SGBD disposen de *drivers* JDBC, però en cas d'haver de treballar amb un sistema que no en tingui, si disposa de controlador ODBC, podrem fer servir el pont JDBC-ODBC per aconseguir la connexió des de Java.

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.

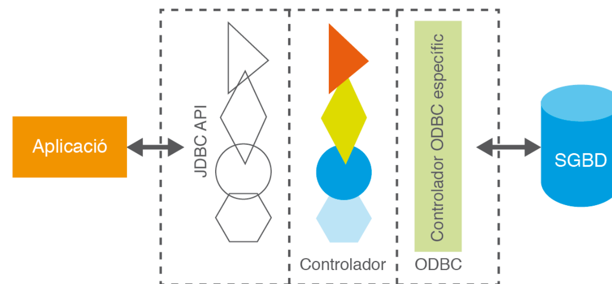
Tipus de controladors

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

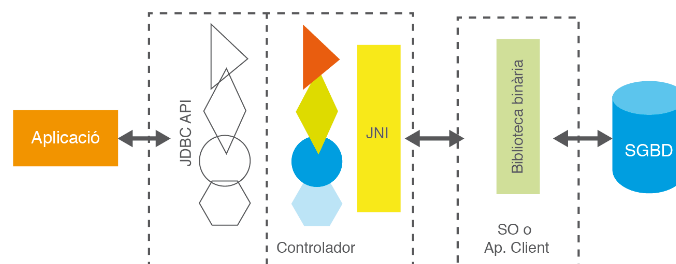
especificacions de l'API JDBC i la tecnologia concreta utilitzada. El més conegut és el controlador pont JDBC-ODBC, però n'hi ha d'altres, com JDBC-OLE DB. La seva principal raó de ser és la de permetre usar una tecnologia molt estesa i assegurar així la connexió amb pràcticament qualsevol font de dades. 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



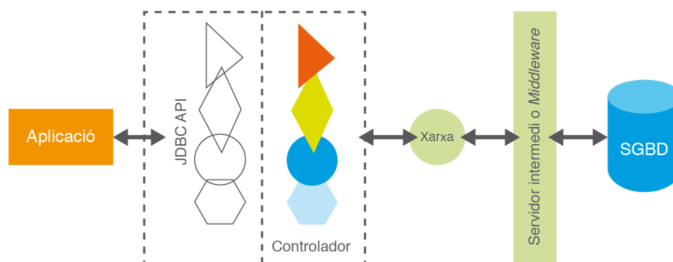
2. Tipus II. Controladors de **Java amb API parcialment nadiu** (*Native-API partly Java driver*). S'anomenen també simplement *nadius*. Com el seu nom indica, estan formats d'una part codificada en Java i una altra part que usa biblioteques binàries instal·lades en el sistema operatiu. Aquest tipus 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



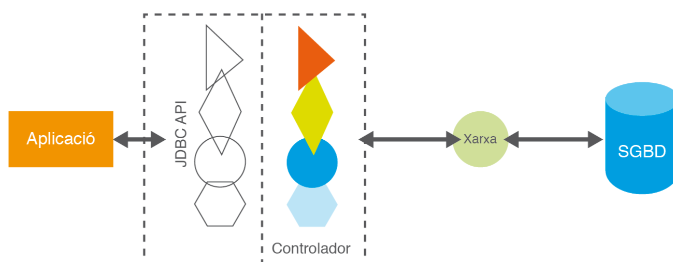
3. Tipus III. Controladors de **Java via protocol de xarxa**. Es tracta d'un controlador escrit totalment en Java que tradueix les crides JDBC a un protocol de xarxa contra un servidor intermedi (anomenat comunament *Middleware*) que pot estar connectat a diversos SGBD. Aquest tipus de *driver* presenta 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 puro 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.

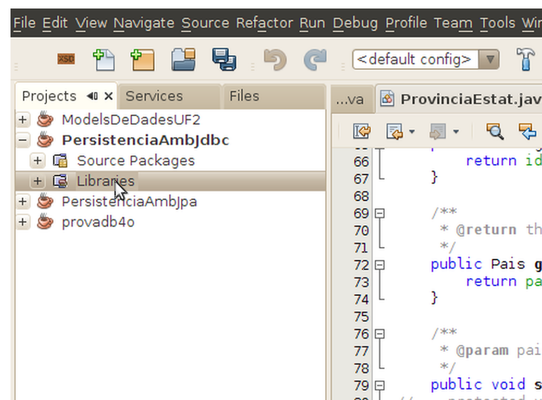
Un cop verificat l'accés al sistema gestor de dades, caldrà obtenir el controlador JDBC del sistema gestor, en el nostre cas PostgreSQL. Generalment, cada fabricant posarà a disposició dels seus usuaris els diferents tipus de controladors que tingui per als seus productes. És habitual que disposi de 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.

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.

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



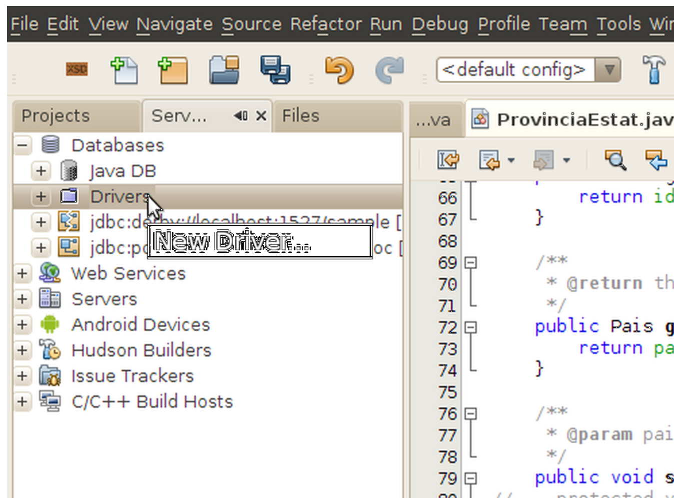
Per fer-ho, caldrà situar el cursor del ratolí a Libraries, clicar el botó dret, escollir add JAR/Folder... i navegar pel sistema de fitxers fins seleccionar el fitxer desitjat.

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*.

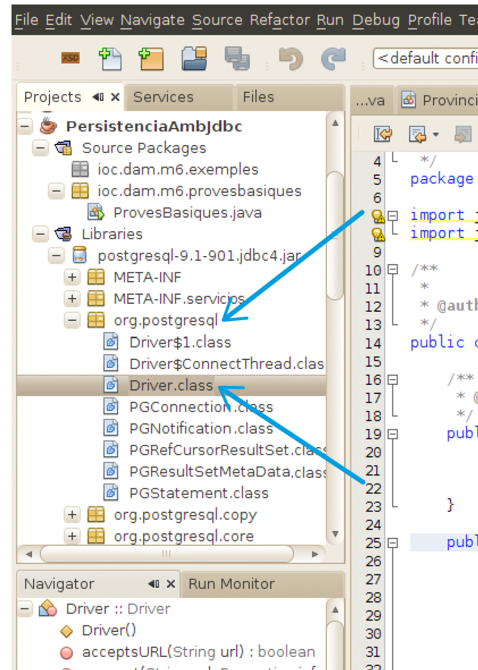
Normalment, la màquina virtual descobreix la localització exacta de la ruta on es troba la classe a carregar analitzant les sentències *import*. En general és una forma força útil i eficient de detectar la ubicació de les classes d'una aplicació. El problema, però,és que per poder-lo usar necessitem 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`

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

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

El mètode `forName` de la classe `Class`pot 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*.

```
public void primeraProva(){
    try {
        Class.forName("org.postgresql.Driver");

    } catch (ClassNotFoundException ex) {
        System.out.println("No s'ha trobat el controlador JDBC");
    }
}
```

1.3.2 Establint connexió

Cal recordar que l'API JDBC, a banda d'algunes classes especifiques, 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:

```
1 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:

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

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

```
1 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*:

```
1 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`:

```
1 String url="jdbc:postgresql://localhost:5432/ioc_proves";
2 String usuari="ioc";
3 String password="ioc";
4 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;
```

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

    System.out.println("provaDeConnexio()");

    try {
        //Carreguem el controlador en memòria
        Class.forName("org.postgresql.Driver");
    } catch (ClassNotFoundException ex) {
        System.out.println("No s'ha trobat el controlador JDBC ("
                                   + ex.getMessage() +")");
        //Si no tenim controlador no podem fer res més. Sortim.
        return;
    }

    try{
         //Obtenim una connexió des de DriverManager
        con = DriverManager.getConnection(url, usuari, password);
        System.out.println("Connexió realitzada usant"
              + " DriverManager");
        con.close();

    } catch (SQLException ex) {
        System.out.println("Error " + ex.getMessage());
    }

    try{
         //Obtenim el Driver del controlador des de DriverManager
        driver = DriverManager.getDriver(url);
         //configurem l'usuari i la contrasenya
        Properties properties = new Properties();
        properties.setProperty("user", usuari);
        properties.setProperty("password", password);
         //Obtenim una connexió des de la instància de Driver
        con = driver.connect(url, properties);
        System.out.println("Connexió realitzada usant Driver");
        con.close();
    } catch (SQLException ex) {
        System.out.println("Error " + ex.getMessage());
    }
}
```

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`.

```
...

    Statement statement = con.createStatement();
    statement.executeUpdate(sentenciaSQL);
    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:

```
1  CREATE TABLE tasques(
2     id INTEGER NOT NULL,
3     descripcio VARCHAR(300) NOT NULL,
4     data_inici DATE,
5     data_final DATE NOT NULL,
6     finalitzada BOOLEAN DEFAULT false,
7     CONSTRAINT pk_clauPrimaria PRIMARY KEY (id)
8  );
```

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

```
1  public void crearTaula(){
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 = "CREATE TABLE tasques(\n"
16                  + "id INTEGER NOT NULL, \n"
17                  + "descripcio VARCHAR(300) NOT NULL, \n"
18                  + "data_inici DATE, \n"
19                  + "data_final DATE NOT NULL, \n"
20                  + "finalitzada BOOLEAN DEFAULT false, \n"
21                  + "CONSTRAINT pk_clauPrimaria PRIMARY KEY (id)"
22                  + ")";
23
24          statement.executeUpdate(sentenciaSQL);
25
26
27      } catch (SQLException ex) {
28              System.out.println("Error " + ex.getMessage());
29      } catch (ClassNotFoundException ex) {
30          System.out.println("No s'ha trobat el controlador JDBC ("
31                                             + ex.getMessage() +")");
32      }finally{
33          try {
34              if(statement!=null && !statement.isClosed()){
35                      statement.close();
36              }
37          } catch (SQLException ex) {/*llàstima!*/}
38          try {
39              if(con!=null && !con.isClosed()){
40                      con.close();
41              }
42          } catch (SQLException ex) {/*llàstima!*/}
43      }
44  }
```

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`.

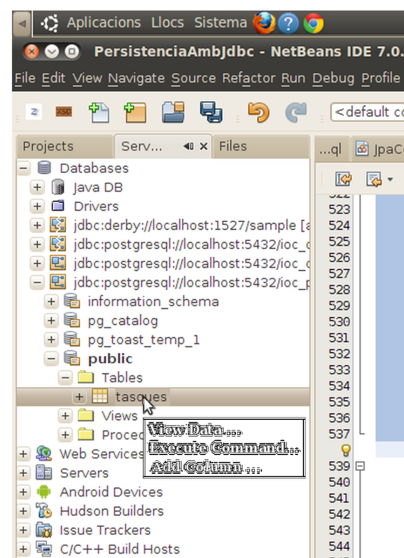
```
1   public void insereixTasques(){
2       Connection con=null;
3       Statement statement = null;
4       String sentenciaSQL=null;
5
6       try {
7           Class.forName("org.postgresql.Driver");
8
9           String url = "jdbc:postgresql://localhost:5432/ioc_proves";
10          con = DriverManager.getConnection(url, "ioc", "ioc");
11
12
13          statement = con.createStatement();
14
15          sentenciaSQL = "INSERT INTO TASQUES VALUES (1, "
16                    + "'Comprar pomes', '2012-02-15', '2012-02-16')";
17          statement.executeUpdate(sentenciaSQL);
18          sentenciaSQL = "INSERT INTO TASQUES VALUES (2, "
19                   + "'Estudiar examen', '2012-05-2', '2012-05-18')";
20          statement.executeUpdate(sentenciaSQL);
21          sentenciaSQL = "INSERT INTO TASQUES VALUES (3, "
22            + "'Compar bitllet Sidney', '2012-02-15', '2012-05-20')";
23          statement.executeUpdate(sentenciaSQL);
24          sentenciaSQL = "INSERT INTO TASQUES VALUES (4, "
25                + "'Anar a Australia', '2012-06-01', '2012-08-25')";
26          statement.executeUpdate(sentenciaSQL);
27          sentenciaSQL = "INSERT INTO TASQUES VALUES (5, "
28                     + "'Revisió cotxe', '2012-04-8', '2012-04-16')";
29          statement.executeUpdate(sentenciaSQL);
30          sentenciaSQL = "INSERT INTO TASQUES VALUES (6, "
```

```java
                + "'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`:

```
public void modificarTasques(){
    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 = "UPDATE TASQUES SET FINALITZADA=true "
                     + "WHERE ID=1 OR ID=3 OR id=5";
        statement.executeUpdate(sentenciaSQL);
        sentenciaSQL = "UPDATE TASQUES SET data_final='2012-06-5' "
                     + "where id=7";
        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!*/}
    }
}
```

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.

```
1  try{
2      sentenciaSQL = "INSERT INTO PAIS VALUES('Marroc')";
3      PreparedStatement stm = con.prepareStatement(sentenciaSql);
4      stm.executeUpdate();
5  }finally{
6      try {
7          if(stm!=null && !stm.isClosed()){
8                  stm.close();
9          }
10     } catch (SQLException ex) {
11         Logger.getLogger(ProvesBasiques.class.getName()).log(
12                                           Level.SEVERE, null, ex);
13     }
14 }
```

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:

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

Veiem un exemple:

```
1  try {
2          autocommit=con.getAutoCommit();
3          PreparedStatement stm = con.prepareStatement(
4                "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.

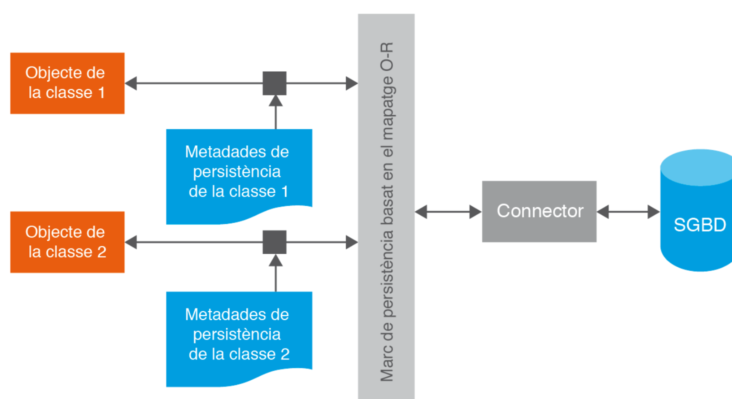
Les eines de mapatge objecte-relacional (ORM) intenten aprofitar la maduresa i l'eficiència de les bases de dades relacionals minimitzant tant com sigui possible el desfasament objecte-relacional. Es tracta de biblioteques i marcs de programació que defineixen un format per expressar múltiples situacions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

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.

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.

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`.

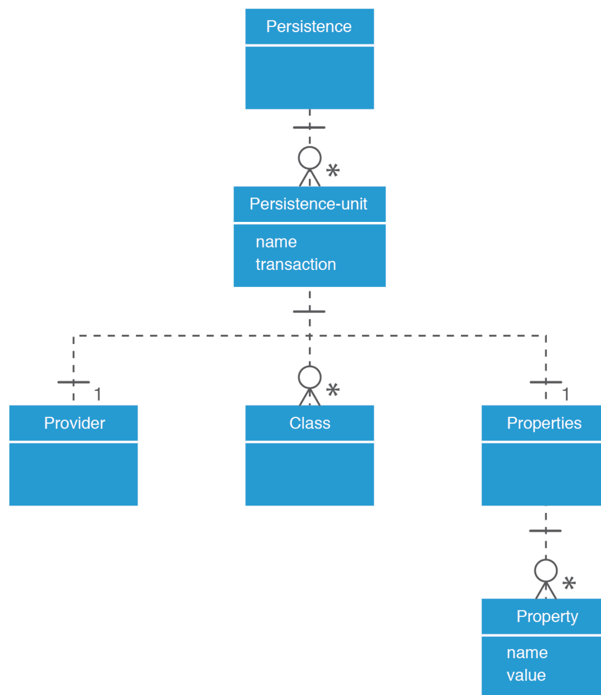
El primer cop que s'instanciï un `EntityManger` es connectarà a l'SGBD i comprovarà si existeixen totes les taules necessàries per mantenir la persistència de les entitats que aquest `EntityManger` controli. En cas que falti alguna, es generaran les sentències de creació adequades d'acord amb les metadades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

```
4     "http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
5     <persistence-unit name="UnitatDePersistenciaPersistenciaEmpreses"
6     transaction-type="RESOURCE_LOCAL">
7       <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
8       <class>ioc.dam.m6.exemples.comandes.Comercial</class>
9       <class>ioc.dam.m6.exemples.comandes.Client</class>
10      <properties>
11        <property name="javax.persistence.jdbc.url"
12      value="jdbc:postgresql://localhost:5432/ioc_comandes"/>
13        <property name="javax.persistence.jdbc.password"
14                                                  value="ioc"/>
15        <property name="javax.persistence.jdbc.driver"                    value="
            org.postgresql.Driver"/>
16        <property name="javax.persistence.jdbc.user" value="ioc"/>
17      </properties>
18    </persistence-unit>
19    <persistence-unit
20          name="UnitatDePersistenciaPersistenciaProductes"
21      transaction-type="RESOURCE_LOCAL">
22      <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
23      <class>ioc.dam.m6.exemples.comandes.Producte</class>
24      <class>ioc.dam.m6.exemples.comandes.Magatzem</class>
25      <properties>
26        <property name="javax.persistence.jdbc.url"
27          value="jdbc:postgresql://localhost:5432/ioc_comandes"/>
28        <property name="javax.persistence.jdbc.password"
29              value="ioc"/>
30        <property name="javax.persistence.jdbc.driver"
31      value="org.postgresql.Driver"/>
32        <property name="javax.persistence.jdbc.user" value="ioc"/>
33      </properties>
34    </persistence-unit>
35  </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`.

```
1  EntityManagerFactory emfProd =
2          Persistece.createEntityManagerFactory(
3                  "UnitatDePersistenciaPersistenciaProductes");
4
5  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:

```
1  <persistence ...>
2     <persistence-unit>
3          ...
4          <mapping-file>META-INF/comandes.xml</mapping-file>
5          ...
6     </persistence-unit>
7  </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*.

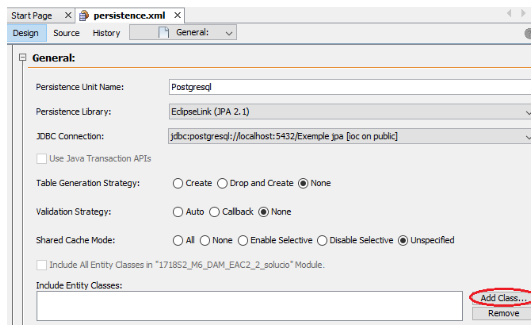
En aquest apartat s'indica com fer-ho.

- 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...

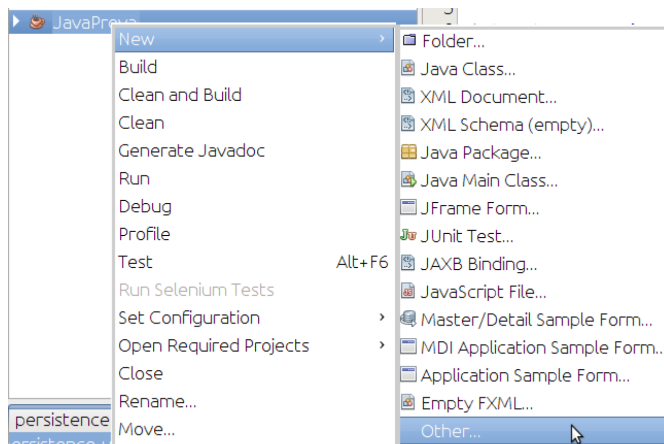
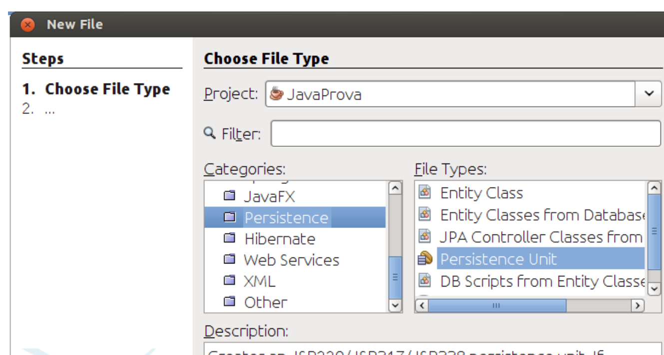
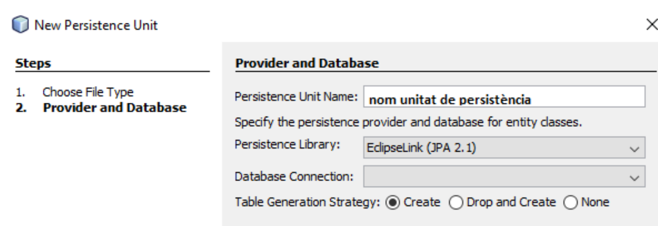


FIGURA 2.6. Selecció del tipus de fitxer Persistence Unit.



- 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*.

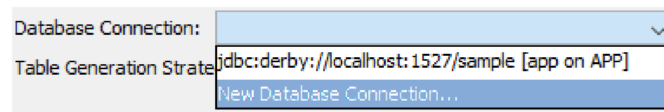
FIGURA 2.7. Finestra de configuració d'una nova unitat de persistència.



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

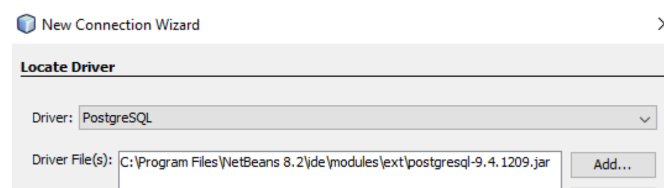
- Triar l'opció *New Database Connection...* al desplegable etiquetat *Database Connection*, com es veu a la figura 2.8

FIGURA 2.8. Nova connexió a una base de dades.



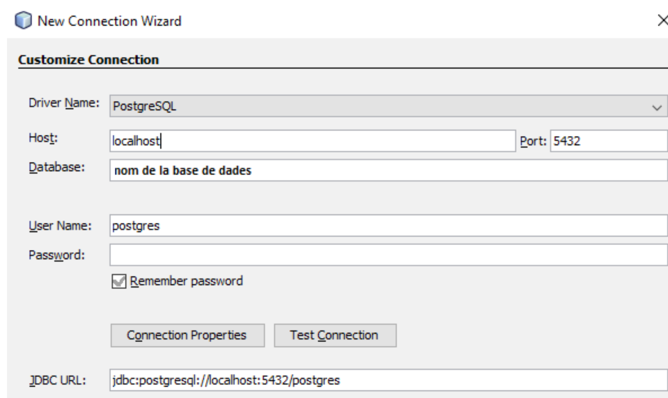
- A la següent pantalla, cal triar *PostreSQL* com a *Driver*, segons es veu a la figura 2.9, i, a continuació, fer clic a *Next*.

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



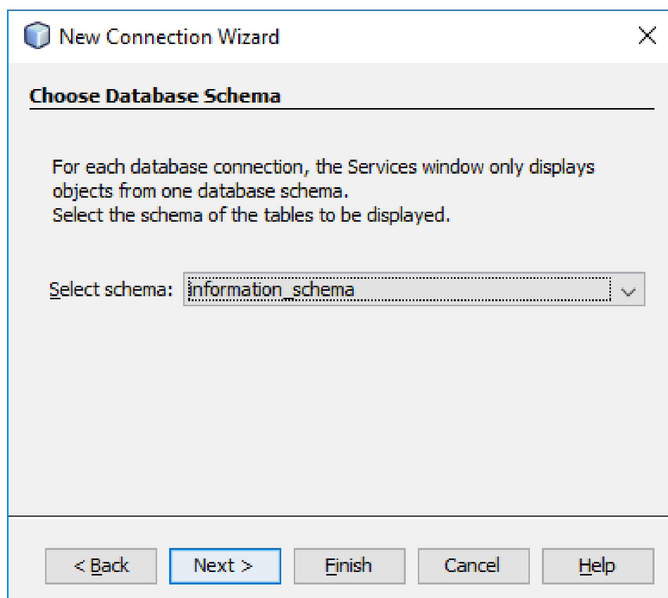
- A la nova finestra, que es veu a la figura 2.10, caldrà acabar la configuració de la connexió amb les dades:
  - ***Host***: si el servidor és a la mateixa màquina on treballem (és habitual quan es fa desenvolupament), es pot deixar el valor que surt per defecte, *localhost*; en cas contrari, caldria 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.



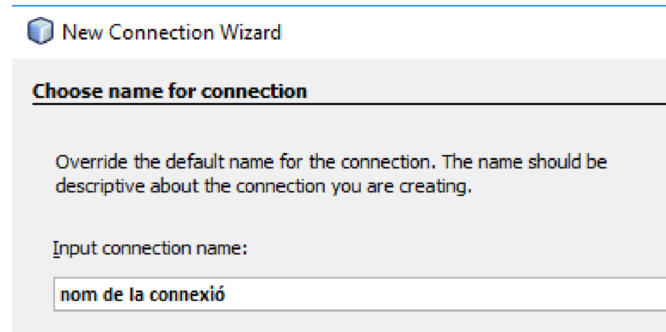
- Al quadre de text etiquetat amb JDBC URL ens apareix la URL de la connexió a partir de les dades que hem entrat als camps anteriors. Abans de passar a la següent pantalla, podem comprovar si la configuració és correcta amb el botó *Test Connection*. Quan ho sigui, caldrà fer clic al botó *Next*. Ens 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:

```
1  @Entity
2  public class UnitatDeMesura implements Serializable {
3      @Id
4      private String simbol;
5      private String descripcio;
6      @Transient
7      private Integer hashSimbol;
8
9      public UnitatDeMesura() {
10     }
11     public UnitatDeMesura(String simbol, String descripcio) {
12         this.simbol = simbol;
13         this.descripcio = descripcio;
14     }
15
16   ...
17 }
```

Exemple usant XML:

```
1  <entity-mappings ...>
2
3    ...
```

```
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.

```
@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:

```
<entity-mappings ...>

...

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

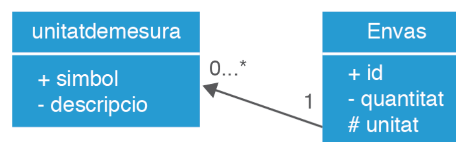
```
        <attributes>
            <id name="id">
                <generated–value strategy="TABLE"/>
            </id>
        </attributes>
    </entity>

...

</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).

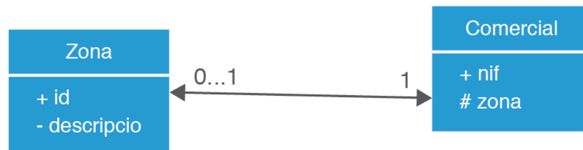
La diferència entre *OneToOne* i *ManyToOne* és més conceptual que pràctica, ja que ambdós (utilitzats per defecte) produeixen la mateixa conversió: una clau forana a la taula on s'emmagatzemarà la relació.

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:

```
<entity-mappings ...>

    ...

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

    ...

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

...

</entity-mappings>
```

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

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

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

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

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

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

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

Primer fent servir anotacions:

```
1  @Entity
2  public class Sector implements Serializable {
3      @Id
4      private String id;
5      private String descripcio;
6      @ManyToMany
7      private List<Producte> productes=new ArrayList<Producte>();
8  
9      protected Sector() {
10     }
11 
12     public Sector(String id) {
13         this.id = id;
14     }
15 
16     public Sector(String id, String descripcio) {
17         this.id = id;
18         this.descripcio = descripcio;
19     }
20 
21     public String getId() {
22         return id;
23     }
24 
25     protected void setId(String id) {
26         this.id = id;
27     }
28 
29     public String getDescripcio() {
30         return descripcio;
31     }
32 
33     public void setDescripcio(String descripcio) {
34         this.descripcio = descripcio;
35     }
36 }
```

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`.

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

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

    ...
}
```

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

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

```
<entity-mappings ...>

    ...

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

...

</entity-mappings>
```

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

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

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

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

```
@Entity
@Table(uniqueConstraints={
    @UniqueConstraint(columnNames={"tipus", "quantitat", "unitat_simbol"})
})
```

```
5 public class Envas implements Serializable {
6     ...
7 }
```

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`.

```
1  <entity-mappings ...>
2      ...
3
4      entity class="ioc.dam.m6.exemples.comandes.Envas "
5                  metadata-complete="true">
6          <table>
7              <unique-constraint>
8                  <column-name>tipus</column-name>
9                  <column-name>quantitat</column-name>
10                 <column-name>unitat_simbol</column-name>
11             </unique-constraint>
12         </table>
13         ...
14     </entity>
15     ...
16
17 </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.

```
1  @Entity
2  public class Comercial implements Serializable {
3      @Id
4      @Column(length=15)
5      private String nif;
6      private String nom;
7      @Basic(fetch= FetchType.LAZY)
8      private String contracte;
9      @OneToOne(fetch= FetchType.LAZY)
10     private Zona zona;
11     ...
12 }
```

Si volem plasmar-ho en format XML:

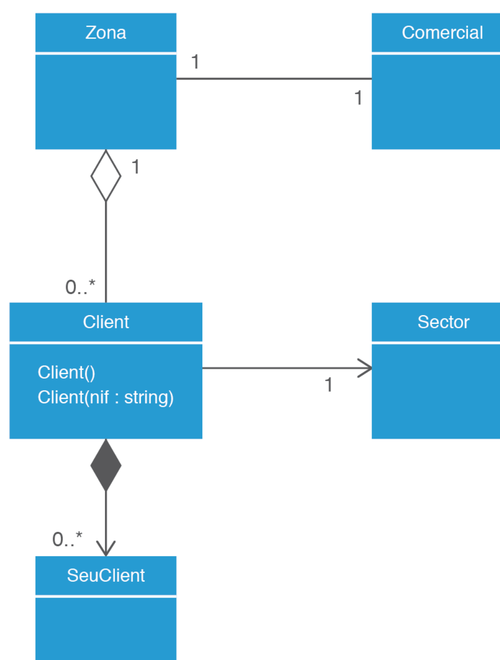
```
1  <entity-mappings ...>
2
3      ...
4
5      <entity class="ioc.dam.m6.exemples.comandes.Comercial"
6                  metadata-complete="true">
7          <attributes>
8              <id name="nif">
9                  <column length="15" />
10             </id>
11             <basic name="contracte" fetch="LAZY" />
12             <one-to-one name="zona" fetch="LAZY" />
13         </attributes>
14     </entity>
15
16     ...
17
18 </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*).

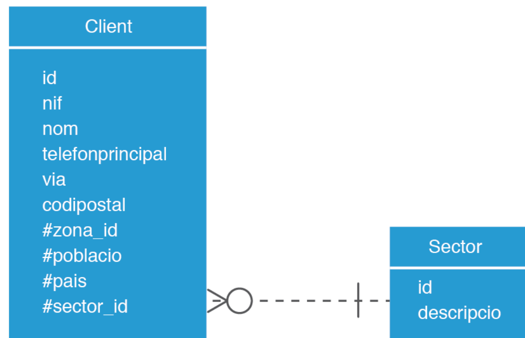
Figura 2.15. Diagrama de classes parcial de l'aplicació exemple



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.

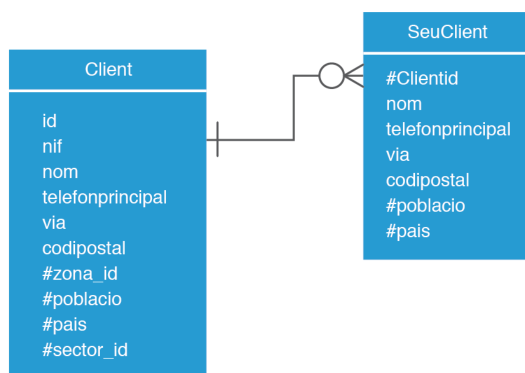
La implementació implicarà un atribut de tipus `Sector` qualificat com a *OneToOne* o *ManyToOne:*

```
@OneToOne(fetch= FetchType.LAZY)
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*.

```
1  @Entity
2  public class Client extends Empresa {
3      @OneToMany(mappedBy="client", fetch= FetchType.LAZY)
4      private Set<SeuClient>seus=new HashSet<SeuClient>();
5    ...
6  }
7
8  ...
9
10 @Entity @Access(AccessType.PROPERTY)
11 public class SeuClient implements Serializable{
12     @ManyToOne
13     private Client client;
14   ...
15 }
```

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>();
  ...
}
```

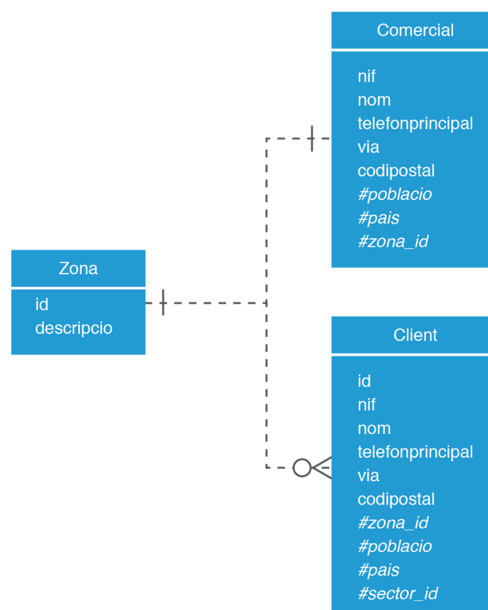
L'atribut `cascade` admet el valor `CascadeType.ALL` per indicar que la classe client s'encarregarà de tota la persistència de les seves seus.

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>
```

```
45   </entity>
46
47   <entity class="ioc.dam.m6.exemples.comandes.SeuClient "
48               metadata–complete="true">
49     <attributes>
50       ...
51       <many–to–one name="client"/>
52       ...
53     </attributes>
54   </>
55
56   <entity class="ioc.dam.m6.exemples.comandes.Sector "
57               metadata–complete="true">
58     <attributes>
59       <id name="id">
60         <column length="50" />
61       </id>
62       ...
63     </attributes>
64   </entity>
65
66   ...
67
68 </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.

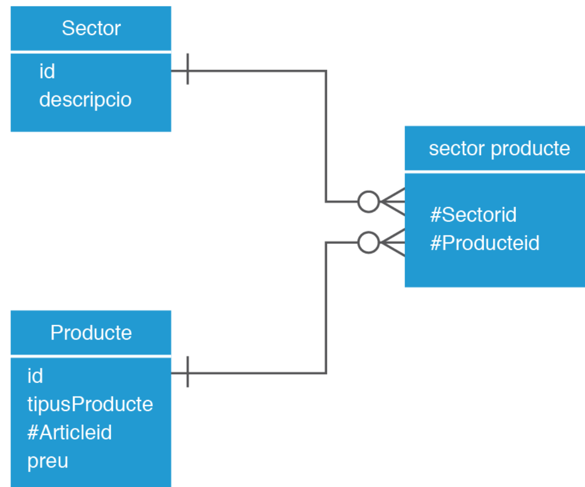
Un exemple de relació *ManyToMany*el podem trobar a la relació entre la classe `Sector` i la classe `Producte`.

```
1  @Entity
2  public class Sector implements Serializable {
3      @Id
4      private String id;
5      private String descripcio;
6      @ManyToMany(fetch= FetchType.LAZY, cascade= CascadeType.ALL)
7      private List<Producte> productes=new ArrayList<Producte>();
8
9    ...
10 }
```

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*.

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

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

2.6.6 Objectes incrustats (Embedded Objects)

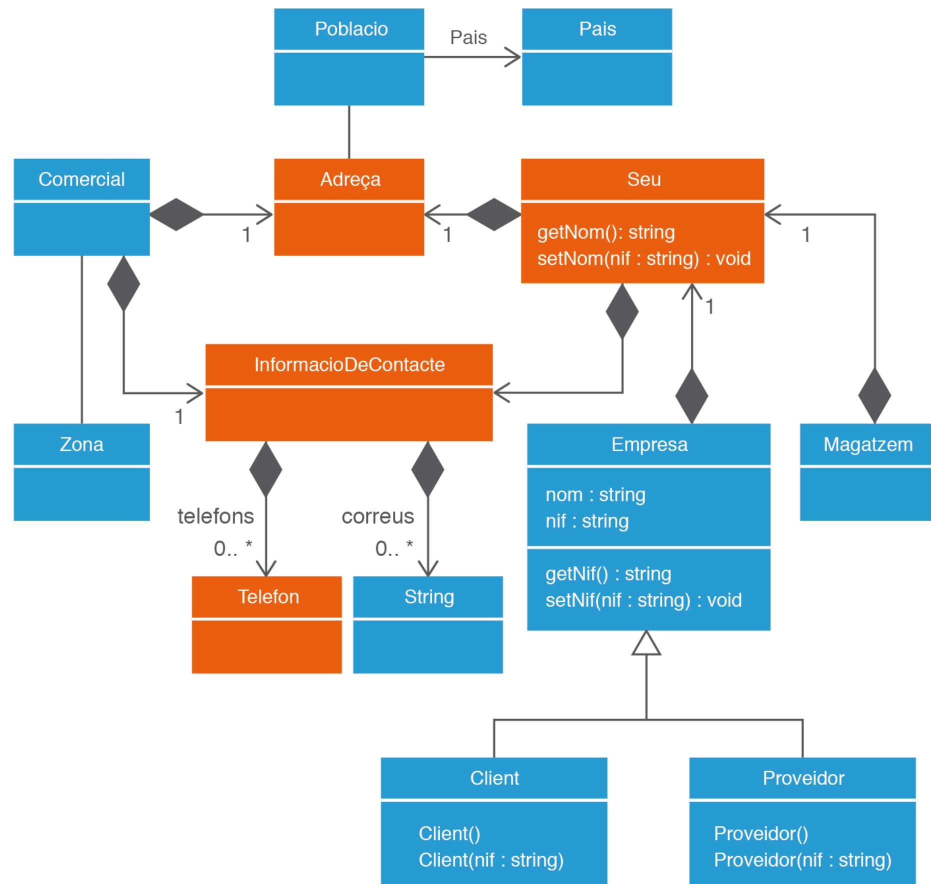
Anomenem objectes incrustats a aquells objectes no marcats com a entitats que estan continguts, directament o indirecta, en una entitat.

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

Comercial
nif nom telefonprincipal via codipostal #poblacio #pais #zona_id

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

Observeu la classe `Adreca`:

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

La classe s'ha de qualificar d'*Embeddable* usant l'anotació, però els seus atributs es tracten de la mateixa manera que si fossin atributs d'una entitat. En primer lloc es limita la mida de *codipostal* i seguidament s'especifica la relació entre l'entitat que contingui `Adreca` (sigui la que sigui) i l'entitat `Poblacio`. Aquesta entitat necessita dues columnes com a clau forana, el nom de la població i el nom del

país. A la taula *POBLACIO*, la columna que referencia el nom de la població s'anomena NOM, i la que referencia el nom del país s'anomena NOM_PAIS. Per evitar problemes hem decidit canviar els noms per uns de més significatius (POBLACIO i PAIS respectivament) fent servir les anotacions `JoinColumns` i `JoinColumn`. Com podeu constatar a la figura 2.21, 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;
    }
  ...
```

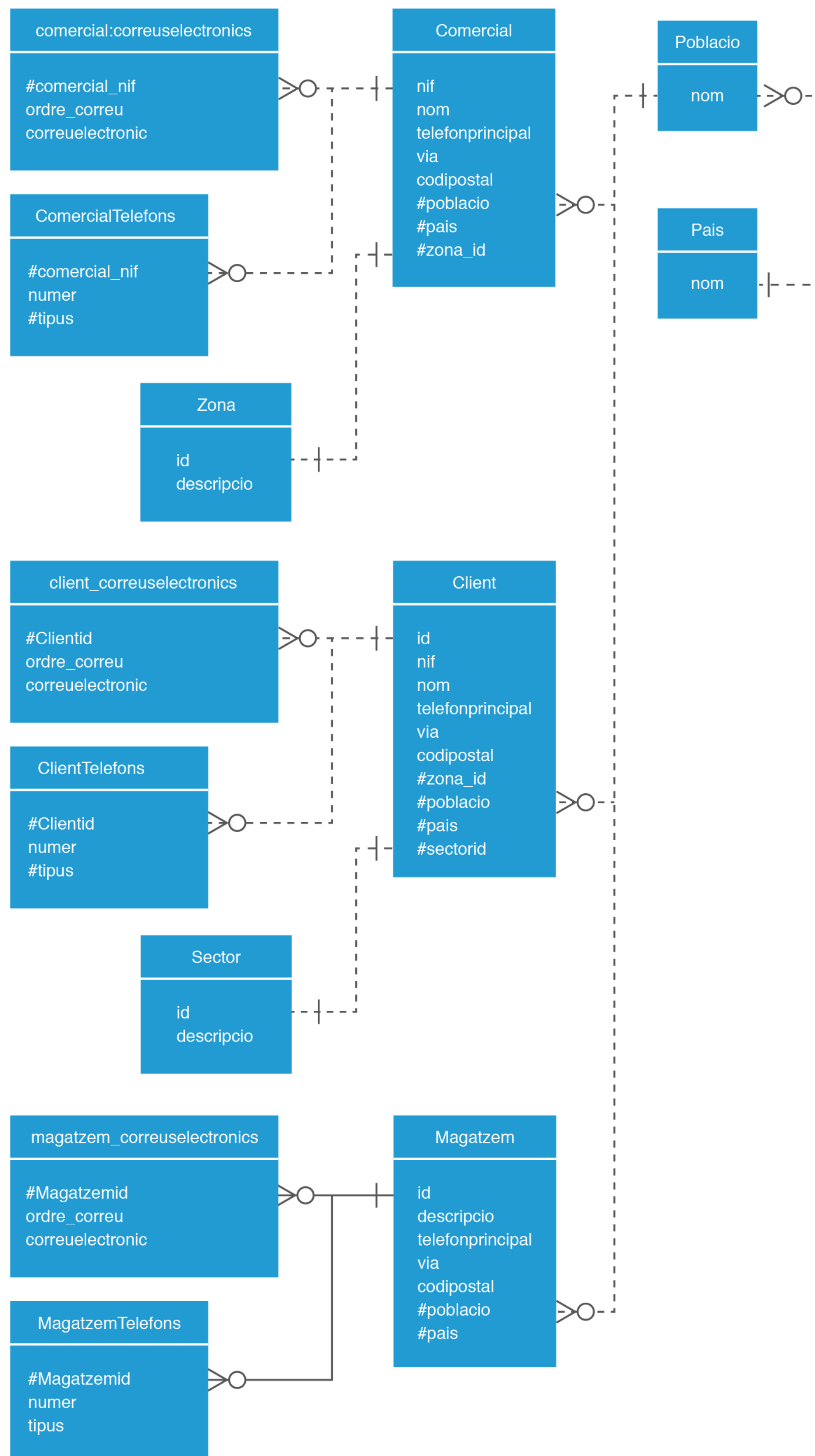
Donat un objecte `Comercial`, caldria afegir el seu codi postal fent :

```
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">
```

```
12          <join-column name="POBLACIO"
13            referenced-column-name="NOM"/>
14          <join-column name="PAIS"
15            referenced-column-name="NOM_PAIS"/>
16        </many-to-one name>
17      </attributes>
18    </embeddadble>
19
20    <entity class="ioc.dam.m6.exemples.comandes.Comercial "
21      metadata-complete="true">
22      <attributes>
23        <id name="nif">
24          <column length="15" />
25        </id>
26        <embedded name="adreca" />
27        <embedded name="informacioDeContacte" />
28        <one-to-one name="zona" fetch="LAZY" />
29      </attributes>
30    </entity>
31
32    <embeddadble class="ioc.dam.m6.exemples.comandes.Seu "
33      metadata-complete="true">
34      <attributes>
35        <embedded name="adreca" />
36        <embedded name="informacioDeContacte" />
37      </attributes>
38    </embeddadble>
39
40    <entity class="ioc.dam.m6.exemples.comandes.Magatzem "
41      metadata-complete="true">
42      <attributes>
43        <id name="id">
44          <column length="20" />
45        </id>
46        <embedded name="seu">
47          <attribute-override name="nom">
48            <column name="DESCRIPCIO" />
49          </attribute-override>
50        </embedded>
51      </attributes>
52    </entity>
53    ...
54
55  </entity-mappings>
```

2.6.7 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.

```
1 @ElementCollection(fetch= FetchType.LAZY)
2 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.

```
1  public void afegirCorreuElectronic(String correu){
2      correusElectronics.add(correu);
3  }
4
5  public void eliminaCorreuElectronic(String correu){
6      correusElectronics.remove(correu);
7  }
8
9  public Iterator<String> getCorreusElectronics(){
10     return correusElectronics.iterator();
11 }
12
13 public void incremetaPrioritatCorreuElectronic(int pos){
14     if(pos>=1){
15         String aux = correusElectronics.get(pos−1);
16         correusElectronics.set(pos−1, correusElectronics.get(pos));
17         correusElectronics.set(pos, aux);
18     }
19 }
20
21 public void decremetaPrioritatCorreuElectronic(int pos){
22     if(pos<correusElectronics.size()){
23         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.

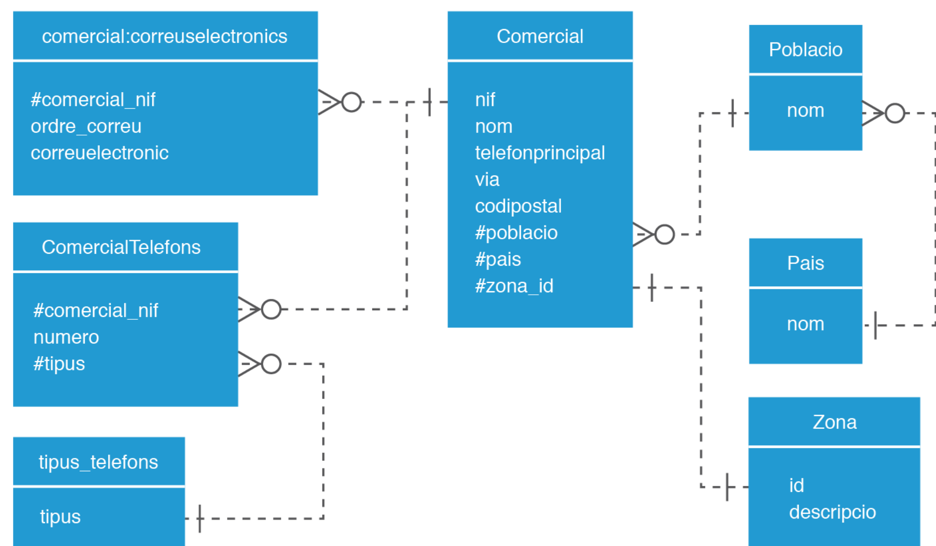
L'anotació `OrderColumn` indica a JPA que a la taula on s'emmagatzemi la col·lecció hi haurà una columna de més on s'emmagatzemarà l'ordre de la col·lecció, amb un valor de tipus numèric.

```
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.

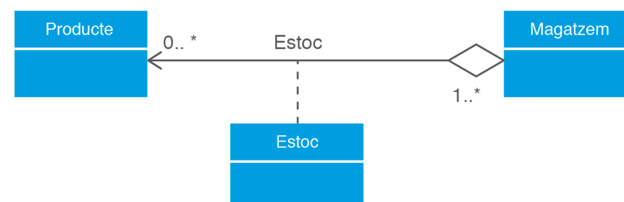
Map amb la clau de tipus entitat i valor de tipus bàsic o Embeddable

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 `Map`com una simple col·lecció d'objectes especificant la notació `ElementCollection`.

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

Il·lustrarem aquest cas per mitjà de la classe `Magatzem`, que conté una col·lecció de tipus `Map` amb l'estoc de cada producte ubicat en algun lloc del magatzem propietari (figura 2.24). 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`.

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

Com a valor ens interessarà guardar la quantitat de producte existent en estoc en el magatzem propietari del `Map` i també el lloc del magatzem on es troba

ubicat el producte. Imaginarem que els magatzems es troben dividits en seccions identificades per una cadena de caràcter.

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

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

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

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

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

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

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

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

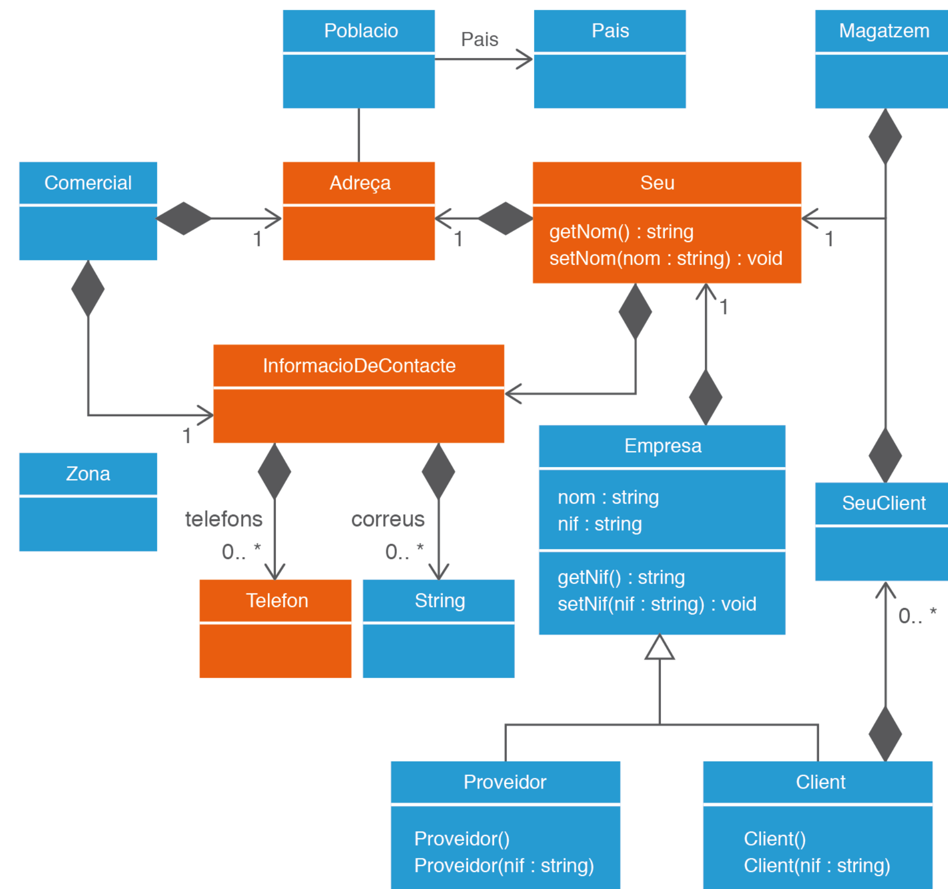
Map amb valor de tipus entitat

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

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

Però a més, la classe client permet emmagatzemar un conjunt de seus. S'ha considerat que en la relació comercial d'una empresa és important conèixer les diferents seus que cada client pugui tenir. Per això, a banda de les dades heretades a través de l'empresa i que constituirien el que podem anomenar la seu central del client, el model incorpora també una col·lecció on poder guardar un nombre indeterminat de seus (una sucursal, un magatzem, una botiga, unes oficines, una fàbrica, etc.). Els elements de la col·lecció són del tipus `SeuClient` (figura 2.25). 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">
```

```
    <attributes>
      <id name="nom" />
      <many-to-one name="client" id="true" />
      <embedded name="adreca" />
      <embedded name="informacioDeContacte" />
    </attributes>
  </entity>
  <embeddadble class="ioc.dam.m6.exemples.comandes.Estoc " />

       <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-mappings>
```

2.6.8 Identificadors compostos

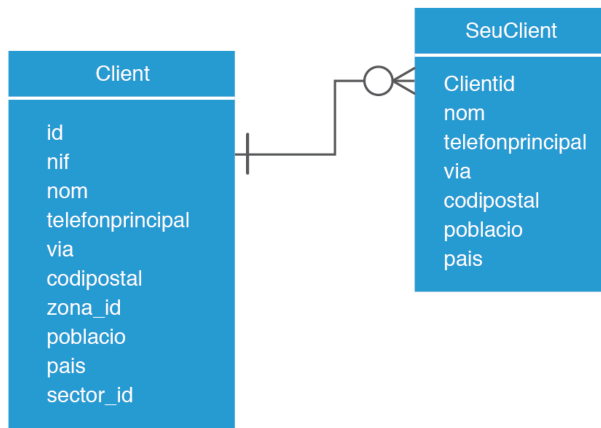
Les claus primàries compostes per diversos camps són força habituals en la majoria d'aplicacions. En aquest apartat veurem les dues respostes que ofereix JPA per tractar aquelles entitats que requereixin identificadors compostos.

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
```

```java
    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:

```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.

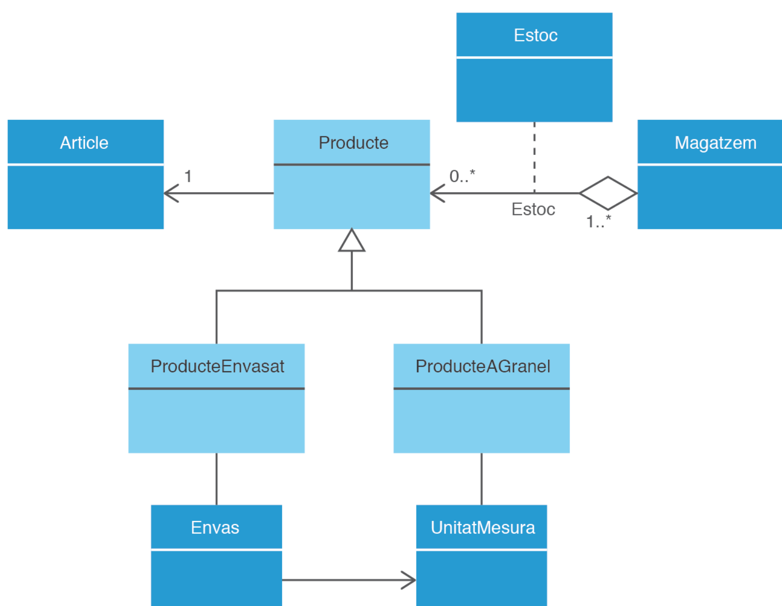
Concepte d'entitats en una jerarquia

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.

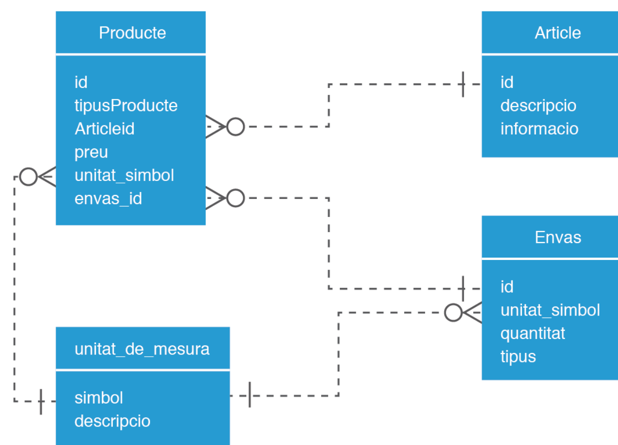
Una taula única per a tota la jerarquia

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




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

    public Producte() {
    }

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

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
```

```
        this.id = id;
    }

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

    public Article getArticle() {
        return article;
    }

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

    public double getPreu() {
        return preu;
    }

    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:

```
1  <entity–mappings ...>
2  
3    ...
4  
5    <entity class="ioc.dam.m6.exemples.comandes.Producte "
6      metadata–complete="true">
7      <inheritance strategy="SINGLE TABLE"/>
8      <discriminator–column name="TIPUS_PRODUCTE"
9            discriminator–type="INTEGER" />
10     <discriminator–value> 0 </discriminator–value>
11     ...
12   </entity>
13 
14   <entity class="ioc.dam.m6.exemples.comandes.ProducteAGranel "
15     metadata–complete="true">
16     <discriminator–value> 1 </discriminator–value>
17     ...
18   </entity>
19 
20   <entity class="ioc.dam.m6.exemples.comandes.ProducteEnvasat "
21       metadata–complete="true">
22     <discriminator–value> 2 </discriminator–value>
23     ...
24   </entity>
25 
26   ...
27 
28 </entity–mappings>
```

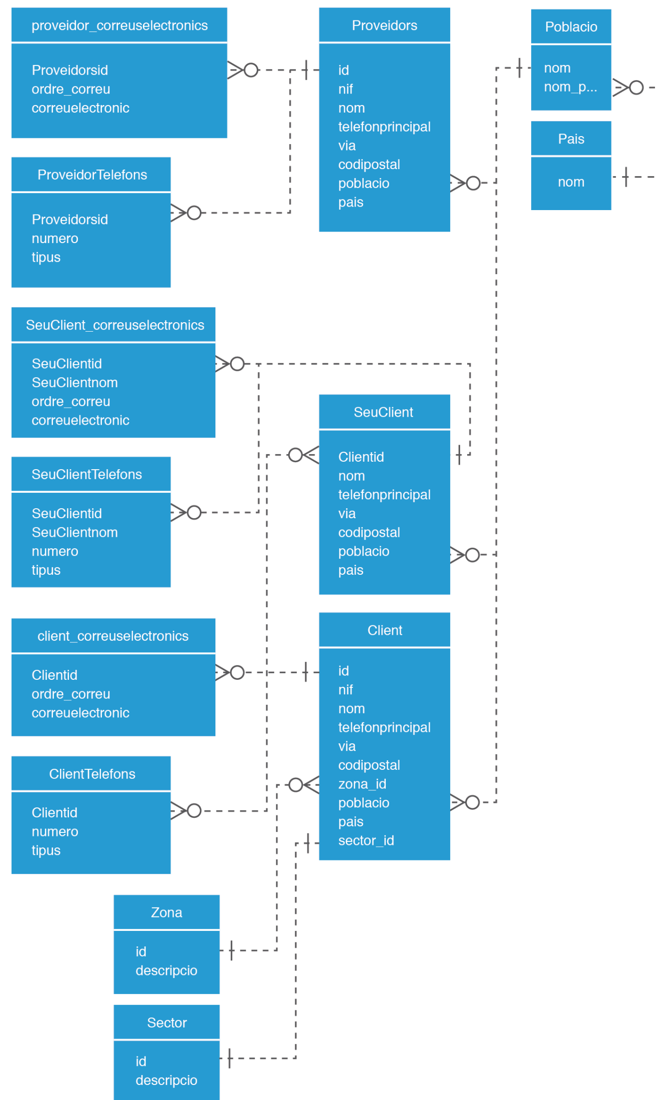
Una taula per a cada entitat

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

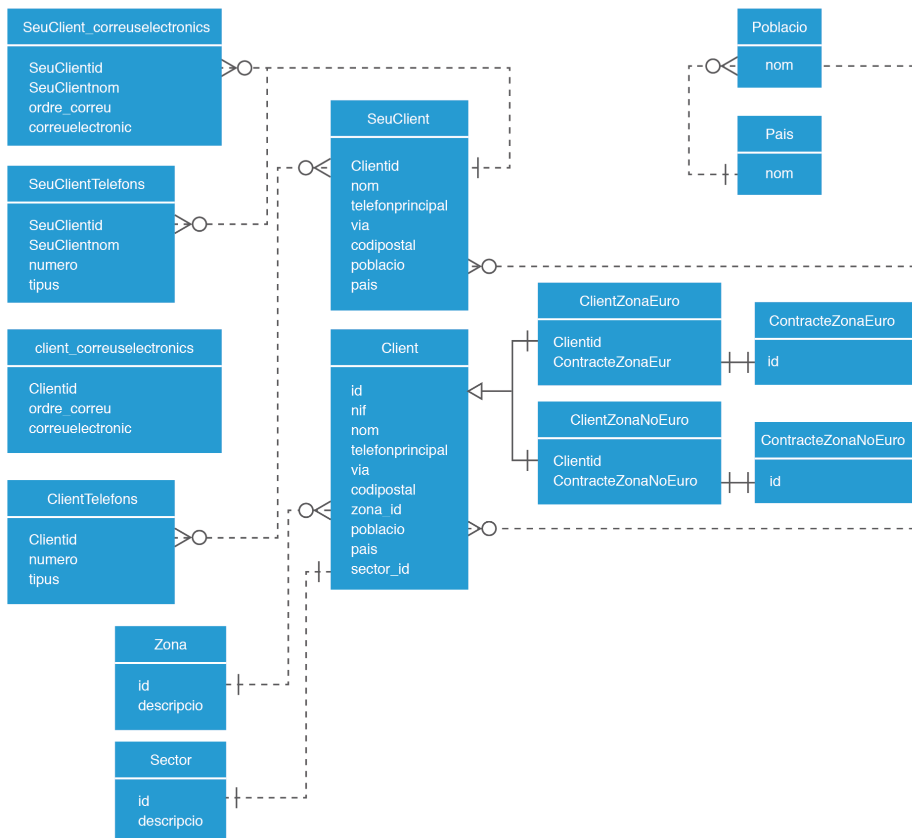
La darrera estratègia seria útil en cas que la jerarquia estigués formada per una classe principal que calgués recuperar sovint més un conjunt de classes derivades que especialitzessin la classe principal i només calgués recuperar en moments puntuals. Imaginem que necessitem una jerarquia en la qual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

```

```
3    ...
4
5     <entity class="ioc.dam.m6.exemples.comandes.Client "
6      metadata–complete="true">
7    <inheritance strategy="JOINED"/>
8    <discriminator–column name="tipus_client"
9          discriminator–type="INTEGER" />
10   ...
11    </entity>
12
13    <entity class="ioc.dam.m6.exemples.comandes.ClientZonaEuro "
14     metadata–complete="true">
15   <discriminator–value> 0 </discriminator–value>
16   ...
17    </entity>
18
19    <entity class="ioc.dam.m6.exemples.comandes.ClientZonaNoEuro "
20     metadata–complete="true">
21   <discriminator–value> 1 </discriminator–value>
22   ...
23  </entity>
24
25  ...
26
27 </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

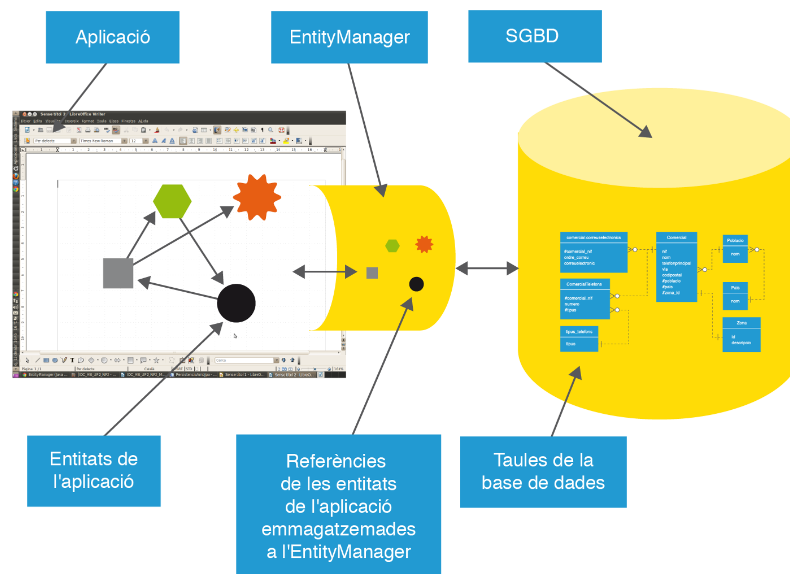
Per encarar amb èxit una aplicació gestionada per JPA caldrà donar a l'`EntityManager` un paper central en el tractament de les entitats. De fet, podem considerar-lo com el representant o interlocutor local de l'SGBD a la nostra aplicació.

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 `Zona`i 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({
```

```
3         @NamedQuery(name="Client.clientsDUnSector",
4         query= "SELECT c "
5         + "FROM Client c "
6         + "WHERE c.sector.id=:sector"),
7         @NamedQuery(name="Client.clientPerNif",
8         query= "SELECT c "
9         + "FROM Client c "
10        + "WHERE c.nif=:nif"),
11        @NamedQuery(name="Client.clientPerNom",
12        query= "SELECT c "
13        + "FROM Client c "
14        + "WHERE c.seuEmpresa.nom=:nom")
15 })
16 public class Client extends Empresa {
17   ...
18 }
```

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

```
1  <entity-mappings ...>
2
3    ...
4
5    <entity class="ioc.dam.m6.exemples.comandes.Client "
6      metadata-complete="true">
7      <inheritance strategy="JOINED"/>
8      <discriminator-column name="tipus_client"
9            discriminator-type="INTEGER" />
10     ...
11     <named-query name="Client.clientsDUnSector">
12       <query>
13         SELECT c
14         FROM Client c
15         WHERE c.sector.id=:sector
16       </query>
17     </named-query>
18     <named-query name="Client.clientPerNif">
19       <query>
20         SELECT c
21         FROM Client c
22         WHERE c.nif=:nif
23       </query>
24     </named-query>
25     <named-query name="Client.clientPerNom">
26       <query>
27         SELECT c
28         FROM Client c
29         WHERE c.seuEmpresa.nom=:nom
30       </query>
31     </named-query>
32   </entity>
33   ...
34
35 </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:

```java
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:

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

La taula 2.2 us donarà una idea de les principals operacions suportades per les expressions WHERE de JPQL.

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

Operació	Descripció	Exemple
=, >, <, >=, <=, <>	Comparen dues expressions situades a banda i banda de l'operació d'acord amb el símbol matemàtic que representin.	`p.preu *0.20 >= c.descompte` Compara si el 20% del preu de l'entitat *p* és major o igual al descompte de l'entitat *c*.
BETWEEN	Compara si una expressió numèrica es troba dins d'un rang definit.	`p.preu BETWEEN 10 AND 100` Compara si el preu de l'entitat *p* es troba entre 10 i 100 (ambdós inclosos).

LIKE	Compara si una expressió de caràcters segueix un determinat patró. El patró es defineix a partir de seqüències de caràcters en combinació amb caràcters especials anomenats comodins. Hi ha dos caràcters comodí: _ i %. El primer representa qualsevol caràcter i el segon qualsevol expressió de caràcters de mida indefinida.	`LOWER(a.nom) LIKE '%poma%'`
		Compara si el nom de l'entitat *a* conté la seqüència `poma`.
		`c.adreca.poblacio.pais LIKE 'E%'`
		Compara si el país de la població de l'adreça de l'entitat *c* comença per la lletra *E*.
		`c.nif LIKE '_12345678'`
		Compara si la numeració del NIF de l'entitat *c* es correspon a *12345678* amb independència de la lletra inicial.

IN	Compara si una expressió es correspon amb algun dels valors continguts entre parèntesis.
	El conjunt de valor pot ser una seqüència de literals o bé els valors retornats per una subconsulta.
		`c.adreca.poblacio.nom IN ('Barcelona', 'Sabadell', 'Badalona')`
		Compara si el nom de la població de l'adreça de l'entitat *c* es correspon amb *Barcelona*, *Sabadell* o *Badalona*.
		`c.zona IN (SELECT cl.zona FROM Client cl WHERE cl.sector.id=:sector)`
		Compara si la zona de l'entitat *c* es correspon amb la zona d'aquells clients que tenen per sector el valor passat pel paràmetre que respon al nom de *sector*.

IS EMPTY IS NOT EMPTY	Compara si un valor és *null* o no.	`z.descripcio IS NOT EMPTY` Compara si la descripció de l'entitat *z* no presenta un valor *null*.
. . .		

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:

```sql
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.

```
1  ResultSet rs = stmt.executeQuery(
2      "SELECT correus_electronics FROM client WHERE id=" + client.getId());
3  while (rs.next()) {
4      Array arraySqlCorreus = rs.getArray(1);
5      String[] correus = (String[])arraySqlCorreus.getArray();
6      for (int i = 0; i < correus.length; i++) {
7          ...
8      }
9  }
```

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:

```
1  INSERT INTO CLIENT (id, correus_electronics)
2      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:

```
1  PreparedStatement comStm = null;
2          ...
```

```
3  StringBuilder comStr = new StringBuilder();
4         comStr.append("INSERT INTO client (id, correus_electronics) ");
5         comStr.append("VALUES(?, ?)");
6    ...
7  comStm = getConnexio().prepareStatement(comStr.toString());
8
9  comStm.setInteger(1, entitat.getId());
10 Array correu = getConnexio().createArrayOf("varchar",
11     entitat.getInformacioDeContacte().getArrayCorreusElectronics());
12 comStm.setArray(2, correu);
13
14 comStm.executeUpdate();
15 ...
```

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.

```
1  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:

```
1  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.

```
1  PreparedStatement comStm = null;
2          ...
3  StringBuilder comStr = new StringBuilder();
4         comStr.append("UPDATE CLIENT SET correus_electronics[1] = ? ");
5         comStr.append("WHERE id=?");
6    ...
7  comStm = getConnexio().prepareStatement(comStr.toString());
8
9  comStm.seString(camp, entitat.getInformacioDeContacte()
10                                        .getArrayCorreusElectronics()[0]);
11 comStm.setInteger(1, entitat.getId());
12
13 comStm.executeUpdate();
14 ...
```

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

```
1  PreparedStatement comStm = null;
2          ...
3  StringBuilder comStr = new StringBuilder();
4         comStr.append("UPDATE CLIENT SET correus_electronics = ? ");
5         comStr.append("WHERE id=?");
6    ...
7  comStm = getConnexio().prepareStatement(comStr.toString());
8
```

```
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
```

```
10 );
11
12 CREATE TYPE t_adreca AS(
13   via VARCHAR(255),
14   codipostal VARCHAR(10),
15   poblacio poblacio
16 );
17
18 CREATE TABLE comercial(
19   nif VARCHAR(15) NOT NULL,
20   nom VARCHAR(255),
21   adreca t_adreca,
22   info_contacte t_info_contacte,
23   zona_id VARCHAR(10),
24   CONSTRAINT comercial_pkey PRIMARY KEY (nif),
25   CONSTRAINT fk400731df1cf26051 FOREIGN KEY (zona_id)
26       REFERENCES zona (id) MATCH SIMPLE
27       ON UPDATE NO ACTION ON DELETE NO ACTION
28 );
```

Un exemple d'inserció podria ser el següent:

```
1 INSERT INTO COMERCIAL (nif, nom, adreca, info_contacte, zona_id)
2   values('12365478F', 'Comercial 1',
3          ('carrer fum', '08009', ('Barcelona', 'Espanya')),
4          ('93333333', ARRAY['aaa@a.cat', 'bbb@bb.es', 'ccc@m.es'],
5          CAST(ARRAY[('fix', '93333333'), ('mbl','666666666')]
6                 AS t_telefon ARRAY))
7          , 'BCN');
```

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

```
1 INSERT INTO COMERCIAL (nif, nom, adreca, info_contacte, zona_id)
2   values('12365478F', 'Comercial 1',
3          ('carrer fum', '08009', ('Barcelona', 'Espanya')),
4          ('93333333', ARRAY['aaa@a.cat', 'bbb@bb.es', 'ccc@m.es'],
5          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.

```
1 UPDATE CLIENT
2 SET seu.info_contacte.telefons = CAST(ARRAY[('fix', '93333333'),
3          ('mbl','666666666')] AS t_telefon[])
4 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:

```
public class ComercialSQLData extends Comercial implements SQLData{
    private String sqlTypeName;

    // Com que per defecte es crea una instància d'Adreca i d'
    //InformacioDeContacte, aquí cal assegurar que les instàncies seran
    // respectivament de tipus AdrecaSQLData i InformacioContacteSQLData
    public ComercialSQLData() {
        setAdreca(new AdrecaSQLData());
        setInformacioDeContacte(new InformacioContacteSQLData());
    }

    // Com que per defecte es crea una instància d'Adreca i d'
    //InformacioDeContacte, aquí cal assegurar que les instàncies seran
    // respectivament de tipus AdrecaSQLData i InformacioContacteSQLData
    public ComercialSQLData(String nif) {
        super(nif);
        setAdreca(new AdrecaSQLData());
        setInformacioDeContacte(new InformacioContacteSQLData());
    }

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

    public void readSQL(SQLInput sqli, String typeName)
                                              throws SQLException {
        sqlTypeName=typeName;
        setNif(sqli.readString());
        setNom(sqli.readString());
        setAdreca((Adreca) sqli.readObject());
        setInformacioDeContacte(
        (InformacioDeContacte) sqli.readObject());
        setZona((Zona) sqli.readObject());
    }

    public void writeSQL(SQLOutput sqlo) throws SQLException {
        sqlo.writeString(getNif());
        sqlo.writeString(getNom());
        sqlo.writeObject((AdrecaSQLData) getAdreca());
        sqlo.writeObject(
                (InformacioContacteSQLData) getInformacioDeContacte());
        sqlo.writeObject((ZonaSQLData) getZona());
    }
}
```

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.

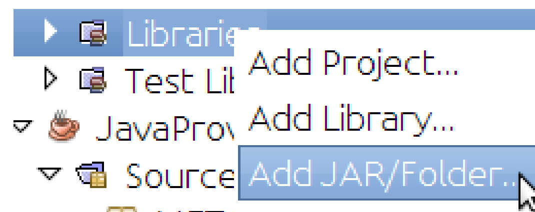
A tots dos casos, el sistema és transparent al programador. Aquest només ha d'ocupar-se de fer les crides adients als mètodes de la biblioteca, posar les anotacions necessàries i, al cas de JPA, definir correctament la unitat de persistència al fitxer *persistence.xml*. La resta de feina la fan els mètodes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

3.3.1 Ús d'ObjectDB amb NetBeans i JPA

Per utilitzar ObjectDB amb NetBeans cal seguir els següents passos:

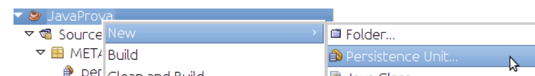
- Descomprimir el fitxer *.zip* que ens hem descarregat del web d'ObjectDB.
- Crear un projecte de Java del tipus *Java Application*.
- Afegir al projecte la biblioteca *objectdb.jar* de la següent manera: fent clic amb el botó secundari a la carpeta de biblioteques, seleccionant *Add / Jar Folder* i 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....

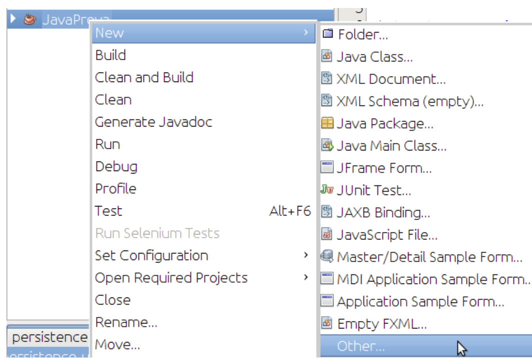
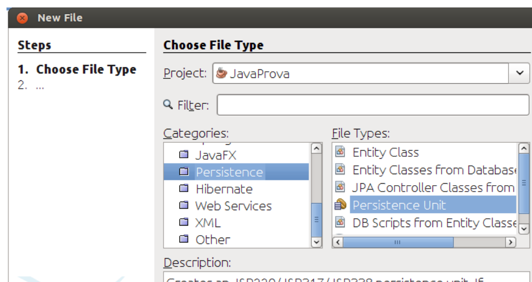


FIGURA 3.4. Selecció del tipus de fitxer Persistence Unit



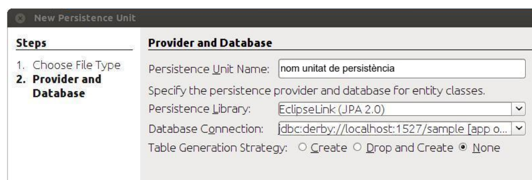
- Configurar la unitat de persistència amb els valors que s'especifiquen a continuació. A la figura :figura 3.5 es veu un possible resultat.

  - *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.

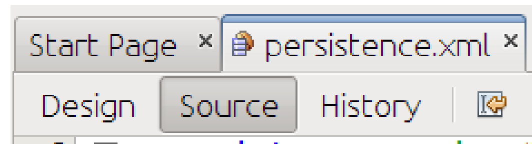
  - *Table Generation Strategy*: es pot seleccionar *None*, ja que no s'ha de crear cap taula.

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

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

per

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

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

    ```
    1 <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

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

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

    ```
    1 <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.

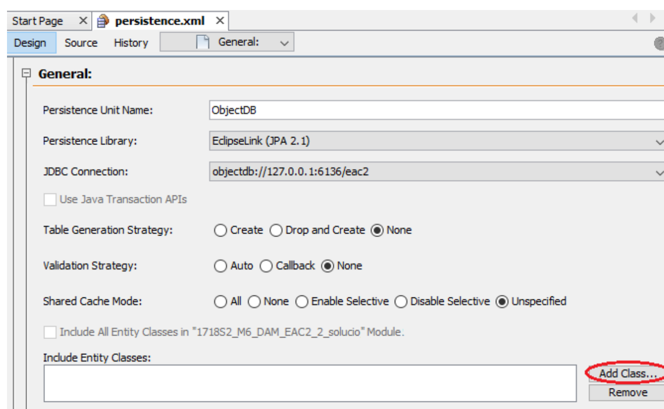
ObjectDB, en ser una base de dades orientada a objectes, no necessita les anotacions de JPA que indiquen com realitzar el mapatge d'objecte a relacional. Si en troba alguna, no dona error per compatibilitat amb l'especificació, però **la ignora**.
Algunes d'aquestes anotacions són:

- 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

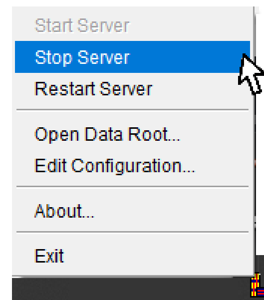


A bit.ly/2mDW7Wn trobareu la documentació oficial sobre totes les eines i utilitats de la base de dades ObjectDB.

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.



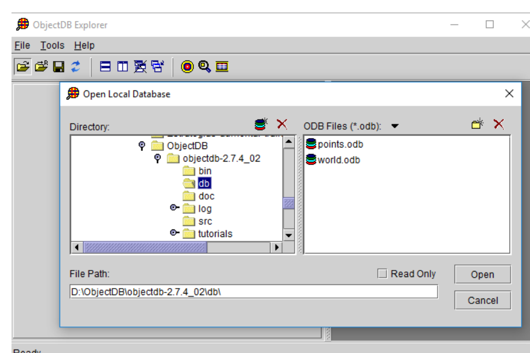
El servidor ObjectDB requereix un fitxer de configuració. Per defecte, aquest es troba a la carpeta de la instal·lació, és a dir, just a l'arrel del resultat de descomprimir el fitxer que cal descarregar-se d'Internet. En cas que no n'hi hagués cap, la pròpia biblioteca *.jar* (que no deixa de ser un 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:

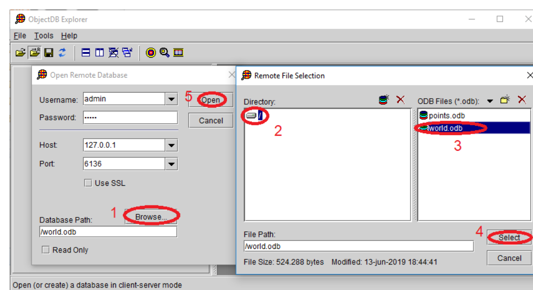
- Obrir una base de dades directament indicant la seva ubicació. Es fa clicant la icona *Open Embedded*, seleccionant-la a la finestra que s'obre. A la figura 3.9 podeu veure l'operació: la icona *Open Embedded* apareix polsada i la finestra de selecció del fitxer oberta. Al quadre gran de l'esquerra 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.



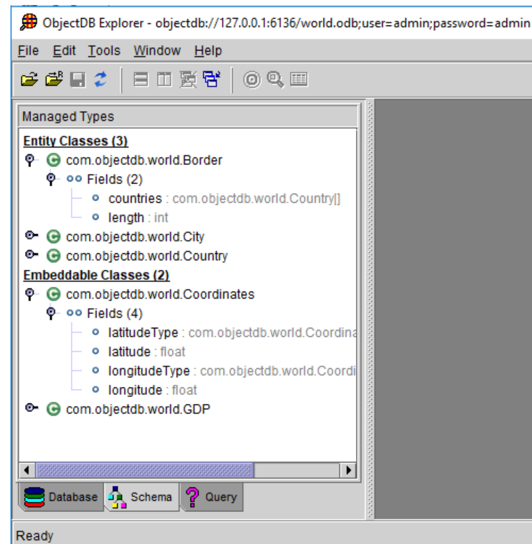
- Tancar una connexió. Pot fer-se bé amb l'opció *File* / *Close connection* o bé tancant el programa.

- 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.

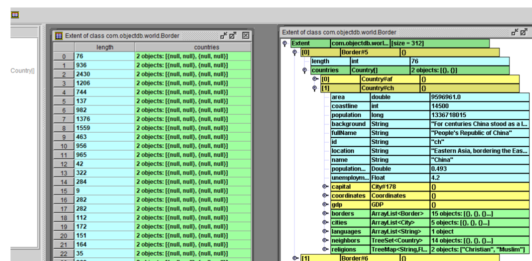
- Visualitzar l'estructura de les dades. Un cop oberta una base de dades directament des del programa o a través del servidor, la columna esquerra queda amb la pestanya *Schema* activada i s'omple amb la informació *Managed Types*, que mostra l'estructura de les dades. Concretamnet i com es veu a la 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.

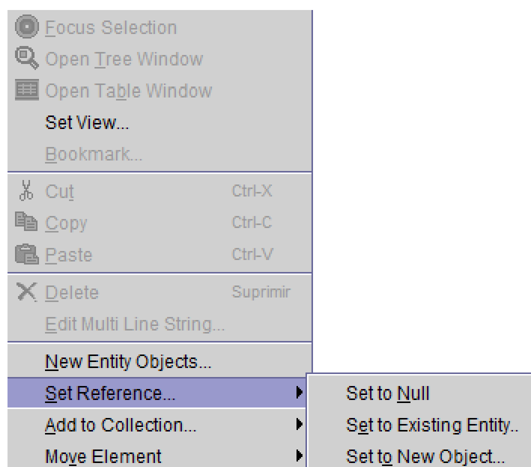
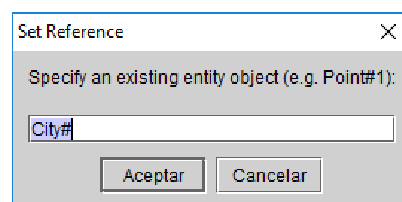


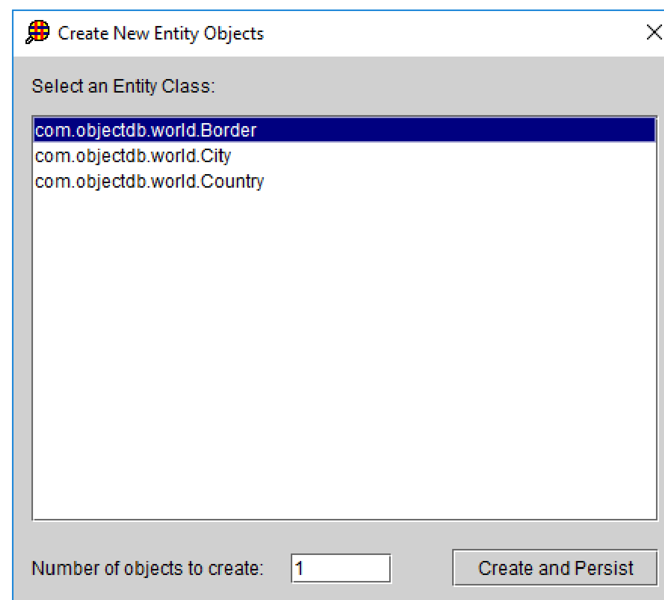
Figura 3.14. Referenciar una entitat de la base de dades.



- 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.

Persistència en BD natives XML

Isidre Guixà Miranda

Accés a dades

Índex

Introducció 5

Resultats d'aprenentatge 7

1 BD-XML natives. API Java específica de l'SGBD 9
- 1.1 Estratègies d'emmagatzematge d'XML . . . 9
- 1.2 SGBD-XML natives vers SGBD predecessors . . . 13
  - 1.2.1 SGBD jeràrquics . . . 13
  - 1.2.2 SGBD relacionals . . . 13
  - 1.2.3 SGBD orientats a objectes . . . 17
  - 1.2.4 SGBD-XML habilitades . . . 19
  - 1.2.5 SGBD-XML natives . . . 20
- 1.3 Biblioteques Java específiques de l'SGBD-XML natives . . . 25
  - 1.3.1 API Java de l'SGBD BaseX . . . 26
  - 1.3.2 API Java de l'SGBD Sedna . . . 33

2 API Java estàndards per a BD-XML natives 43
- 2.1 API XQJ . . . 43
  - 2.1.1 Establiment de connexió . . . 44
  - 2.1.2 Sentències XQuery d'execució immediata . . . 49
  - 2.1.3 Variables lligades . . . 57
  - 2.1.4 Sentències XQuery preparades . . . 59
  - 2.1.5 XQJ per processar documents XML . . . 61

Introducció

La informàtica és una branca de la tecnologia que té com a objectiu el tractament automàtic de la informació mitjançant dispositius electrònics. Això provoca la necessitat d'emmagatzemar la informació (fer-la persistent) i, en conseqüència, van aparèixer en un primer moment els sistemes gestors de fitxers (SGF), els quals van evolucionar fins als anomenats sistemes gestors de bases de dades (SGBD).

En el moment actual, els SGBD són part fonamental dels sistemes informàtics i, segons la tècnica que empren per emmagatzemar la informació, apareixen diversos tipus de bases de dades: jeràrquiques, relacionals, objecte-relacionals, orientades a objectes i XML natives. Per tant, els desenvolupadors de programari han de conèixer com accedir a les dades emmagatzemades en els diversos sistemes existents, objectiu del mòdul professional en el qual s'insereix aquesta unitat.

L'objectiu d'aquesta unitat és l'accés a les dades emmagatzemades en SGBD-XML natives, és a dir, aprendre a desenvolupar programes que accedeixin a BD-XML natives, que són bases de dades que emmagatzemen documents en format XML.

El llenguatge XML permet emmagatzemar informació en un format estructurat, que és més o menys organitzat segons el tipus d'informació (no és el mateix estructurar el contingut d'un llibre, que conté uns pocs capítols molt extensos, que una comanda de venda, que conté unes dades concretes: client, data, productes, quantitats, preus, import...).

Per aconseguir el nostre objectiu necessitem prendre dues decisions: quin(s) llenguatge(s) de programació emprar i sobre quin(s) SGBD-XML natives practicar.

L'accés als SGBD s'acostuma a fer actualment a través d'una interfície de programació d'aplicacions (API) facilitada pels fabricants de l'SGBD en diversos llenguatges. Així, ara mateix, pels diversos SGBD podem trobar API per als llenguatges C++, Java, Perl, .NET i d'altres. És clar que el temps no ens permet abastar tots els llenguatges i, per tant, hem hagut de decidir-nos per un: Java.

Un dels motius pels quals hem escollit Java resideix en el fet que, a banda de ser un llenguatge actualment molt utilitzat, hi ha en el mercat dues API estàndards, en aquest llenguatge, per accedir a SGBD-XML natives.

Ja hem comentat que els fabricants dels SGBD faciliten API d'accés per a diversos llenguatges. Si aquestes API són particulars de cada fabricant, és impossible desenvolupar programes que permetin l'accés a diversos SGBD (atès que cada SGBD facilita una API particular). Per tant, l'existència d'API estàndards és fonamental, ja que els programes basats en una API estàndard haurien de poder accedir a qualsevol SGBD que suporti aquesta API.

Pel que fa als SGBD-XML natives sobre els quals practicar, hi ha molts productes en el mercat, tots ells amb diferències que fan difícil una elecció. Al final hem optat per utilitzar tres SGBD-XML natives diferents (BaseX, Sedna i eXist-db), que cobreixen la majoria de possibilitats que ens podem trobar si hem de treballar, a la vida professional, amb un SGBD-XML natives.

Aquesta unitat està dividida en dos apartats. El primer, "BD-XML natives, API Java específica de l'SGBD", introdueix els conceptes bàsics a conèixer en referència als SGBD-XML natives i inicia el desenvolupament de programes en Java que accedeixin a SGBD utilitzant API específiques (no estàndards) de l'SGBD. El material web incorpora uns annexos que expliquen el procés d'instal·lació dels tres SGBD-XML natives i també els llenguatges de gestió de documents XML (XQuery per a consultes i llenguatges per a actualització XML) que l'alumne ja hauria de conèixer.

El segon apartat, *API Java estàndards per a BD-XML natives*, ens endinsa, com el seu nom indica, en el desenvolupament de programes Java utilitzant la principal API Java estàndard, XQJ, que forma part de les especificacions del W3C (*World Wide Web Consortium*). Aquest apartat també introdueix l'altra API Java estàndard existent,XML-DB, però sense tractar el desenvolupament de programes que la utilitzin, ja que que avui en dia aquesta API ha quedat obsoleta.

El seguiment d'aquesta unitat pressuposa que l'alumne és coneixedor del llenguatge XML i dels llenguatges per gestionar la informació de documents XML, XPath i XQuery per fer consultes en documents XML, així com alguna de les versions de llenguatges existents per modificar el contingut de documents XML (XUpdate, Update de P. Lehti, XQUF). Aquests coneixements s'adquireixen a les unitats 1 i 2 del mòdul professional *Llenguatges de marques i sistemes de gestió de la informació*. Cal tenir en compte que en els annexos del material web hi ha diversos exemples d'utilització d'instruccions dels llenguatges per gestionar la informació de documents XML. L'alumne també ha de conèixer el llenguatge Java. Aquests coneixements s'adquireixen en el mòdul professional *Programació*.

Per tal d'assolir un bon aprenentatge, cal estudiar els continguts en l'ordre indicat, sense saltar-se cap apartat, i quan es fa referència a algun material web, adreçar-s'hi i dur-lo a la pràctica. Els programes d'exemple que incorpora el material, cal analitzar-los amb deteniment i posar-los en execució amb els valors que es proposen i amb altres valors que pugui intuir l'alumne per comprovar-ne el correcte funcionament. Una vegada estudiats a fons els programes d'exemple, és necessari desenvolupar les activitats web.

Resultats d'aprenentatge

En finalitzar aquesta unitat l'alumne/a:

1. Desenvolupa aplicacions que gestionen la informació emmagatzemada en bases de dades natives XML avaluant i utilitzant classes específiques.

   - Valora les avantatges i inconvenients d'utilitzar una base de dades nativa XML.
   - Estableix la connexió amb la base de dades.
   - Desenvolupa aplicacions que fan consultes sobre el contingut de la base de dades.
   - Afegeix i elimina col·leccions de la base de dades.
   - Desenvolupa aplicacions per afegir, modificar i eliminar documents XML de la base de dades.

1. BD-XML natives. API Java específica de l'SGBD

L'XML és, segons el seu impulsor World Wide Web Consortium (W3C), un format simple basat en text per representar informació estructurada: documents, dades, configuracions, llibres, transaccions, factures i molt més. Va ser derivat d'un format estàndard més antic, anomenat SGML, amb la finalitat de ser més adequat per a la seva utilització en la web.

L'XML, avui en dia, és un dels formats més utilitzats per a l'intercanvi d'informació estructurada: entre els programes, entre les persones, entre ordinadors i persones, tant a nivell local com a través de les xarxes. El fet que la informació s'intercanviï en format XML ha implicat l'aparició de mecanismes que permetin enregistrar dita informació en format XML, de manera que no sigui necessari efectuar traduccions a altres formats.

L'emmagatzematge de la informació en format XML no s'ha fet d'una única manera, sinó que han aparegut diverses estratègies d'emmagatzematge que ens interessa conèixer, així com els avantatges i els inconvenients d'utilitzar les implementacions basades en les diverses estratègies.

Les bases de dades natives XML han estat el resultat d'una de les estratègies per emmagatzemar XML. Ens interessa, d'elles, refrescar les seves principals característiques per poder atacar el nostre objectiu, que no és cap altre que desenvolupar aplicacions que gestionin la informació emmagatzemada en elles.

1.1 Estratègies d'emmagatzematge d'XML

L'XML és un llenguatge que permet tenir informació estructurada, ja sigui per a intercanvi entre plataformes o, simplement, per facilitar un ràpid accés a continguts. La seva àmplia acceptació ha dut la necessitat d'idear mecanismes per poder emmagatzemar, de manera fàcil i àgil, grans volums de documents XML.

Per enfrontar-nos a la necessitat d'emmagatzematge i poder definir la millor estratègia cal fer una petita reflexió sobre els tipus de documents XML que ens podem trobar:

1. Documents centrats en dades (*data-centric*), pensats per a l'intercanvi entre plataformes. Solen ser documents amb estructures regulars i ben definides. Les dades que transmeten són partícules atòmiques ben definides, i en moltes ocasions són actualitzables. Acostumen a tenir com a origen una base de dades i, en conseqüència, no és gaire important la seva persistència com a document XML.

Així, per exemple, un document XML centrat en les dades podria ser el següent:

```
<clients>
  <client>
    <codi>10</codi>
    <raoSocial>Components Informàtics</raoSocial>
  </client>
  <client>
    <codi>20</codi>
    <raoSocial>Institut Obert de Catalunya</raoSocial>
  </client>
</clients>
```

2. Documents centrats en el document(*document-centric*), amb una estructura irregular. L'origen i el destí acostuma a estar en les persones i solen estar fets a mà. N'hi ha que poden arribar a tenir un cert format, però no és estricte ni definit i en tal cas parlem de dades semiestructurades. Exemples de documents centrats en el document són els llibres, els correus electrònics, els anuncis, etc.

A continuació veiem un exemple de document XML centrat en el document. Es tracta d'un document descriptiu d'un producte (llibre) en què l'autor ha inserit diverses marques per facilitar-ne la consulta i la formatació en un navegador. Podeu comprovar-ho copiant el contingut en un fitxer amb extensió `.xml` i obrir-lo amb un navegador.

```
<SèrieCòmic>

<Introducció>
La sèrie <NomCòmic>Mortadel·lo i Filemó</NomCòmic> de <Autor>Francisco Ibáñez
     Talavera</Autor> creada i desenvolupada <Any>a partir de 1958</Any> <Resum
     >narra les peripècies de dos agents de l'organització secreta TIA (Tècnics
      d'Investigació Aeroterràquia)</Resum>.
</Introducció>

<ComentariDetallat>

<Paràgraf>La sèrie va néixer amb el nom de <i>Mortadel·lo i Filemó, agència d'
     informació</i>, on Filemó és el cap d'una agència de detectius i té a
     Mortadel·lo com a empleat i únic ajudant. El 1969 ingressen a les files de
      la TIA, una desastrosa agència secreta que els permet parodiar les histò
     ries d'espies i s'incorporen nous personatges.</Paràgraf>
<Paràgraf>En qualsevol de les seves èpoques, la sèrie destaca pel seu humor
     extremadament <Link URL="http://es.wikipedia.org/wiki/Slapstick">slapstick
     </Link>, de manera que els personatges pateixen constantment contratemps
     com caigudes des de grans altures, explosions, aixafaments per tota mena d
     'objectes pesats (pianos, caixes fortes, etc.) sense que les conseqüències
      dels mateixos acostumin durar més d'una vinyeta. </Paràgraf>

<Personatges>
<List>
<Item><Link URL="mortadelo.html">Mortadel·lo</Link>Agent alt i prim,
     completament calb <i>gràcies</i> a un invent del professor Bacteri per fer
      créixer el cuir cabellut. Té un nas allargat i duu ulleres i un vestit
     negre. Es comenta que la característica que l'ha portat a la fama és que
     sempre s'està emprovant disfresses, amb les quals pot transformar—se, per
     a sorpresa de tothom, en qualsevol cosa, des d'un elefant fins a una
     formiga, des de Superman fins a l'home invisible. Al principi duia un
     paraigües negre penjat del braç i lluïa un barret d'on sempre extreia les
     disfresses.</Item>
<Item><Link URL="filemo.html">Filemó</Link>És més baix que en Mortadel·lo, i és
      el seu cap. Té dos pèls al cap, i duu una camisa blanca i uns pantalons
     vermells, encara que en els primers anys de la seva publicació duia una
     gran pipa a la boca i un abillament similar al de Sherlock Holmes. És qui
     se sol endur les garrotades quan alguna cosa surt malament (és a dir,
     gairebé sempre).</Item>
<Item><Link URL="elSuper.html">El Super</Link>És el superintendent (de nom Viç
```

```
     ens) de la TIA, el qual encomana les missions a Mortadel·lo i Filemó. Duu
     un poblat bigoti i un vestit blau. En paraules del propi Ibáñez, és una cr
     ítica contra els anteriors caps que havia tingut quan treballava al banc
     .</Item>
17 <Item><Link URL="bacteri.html">Professor Bacteri</Link>és el típic professor
     guillat. Crea els invents més inversemblants, però gairebé sempre fan el
     contrari del que es pretenia. Té una gran barba negra, que els agents o el
      Súper prenen amb ferotgia contra la seva persona quan els seus invents
     surten malament.</Item>
18 <Item><Link URL="ofelia.html">Ofèlia</Link>És la secretària, una dona rossa i
     grassa enamorada d'en Mortadel·lo (però no corresposta) i quasi sempre
     utilitzada com a mofa.</Item>
19 <Item><Link URL="irma.html">Irma</Link>És l'altra secretària, encara que no
     surt tant en el còmic com l'Ofèlia (de fet va deixar de ser dibuixada anys
      enrere). És molt maca, i tant en Mortadel·lo com en Filemó estan bojos
     per ella... la qual cosa provoca gags interessants amb l'Ofèlia.</Item>
20 </List>
21 </Personatges>
22
23 <Paràgraf>Les històries de Mortadel·lo i Filemó s'han adaptat al cinema amb
     personatges reals, com en la <i>La gran aventura de Mortadel·lo i Filemó
     (2003)</i> dirigida per Javier Fesser i protagonitzada per Benito Pocino
     en el paper de Mortadel·lo i Pepe Viyuela en el paper de Filemó.</Paràgraf
     >
24 <Paràgraf>Una nova pel·lícula, <i>Mortadel·lo i Filemó. Missió: Salvar la Terra
     </i>, es va estrenar el 2008 coincidint amb el cinquantè aniversari de l'
     aparició dels personatges.</Paràgraf>
25
26 </ComentariDetallat>
27
28 </SèrieCòmic>
```

La classificació de documents en *data-centric* i *document-centric* no és sempre directa i clara i, en múltiples ocasions, el contingut estarà barrejat o serà difícil de catalogar en un o altre tipus. Així, podem tenir documents centrats en dades (com una comanda o una factura) on alguna de les dades tingui codificació lliure (per exemple, part de la descripció de les línies) i documents centrats en el document (com un manual d'usuari) amb una part regular amb format estricte i ben definit (com el nom de l'autor i la data de revisió). Malgrat això, la caracterització dels documents en *data-centric* i *document-centric* s'utilitza per ajudar a decidir quina és la millor estratègia d'emmagatzematge.

Ens cal, doncs, conèixer les tècniques d'emmagatzematge existents per, una vegada conegudes, decidir quina tècnica és més adequada segons la tipologia de documents a emmagatzemar i la gestió que en pretenguem.

Tècnicament hi ha tres possibles estratègies per emmagatzemar els documents XML:

1. Enregistrament directe en el sistema d'arxius del sistema operatiu. Opció molt pobra, ja que les funcionalitats que permet fer sobre el document queden limitades i definides pel sistema operatiu. No permet efectuar operacions sobre el contingut i només es permet el moviment del document com a una unitat. Si es tracta d'una petita quantitat de dades guardades en uns pocs documents XML, aquesta opció pot funcionar, però no és gens recomanable per gestionar un volum elevat de documents XML.
2. Enregistrament en un dels tipus de bases de dades tradicional: relacional, jeràrquic o orientat a objectes. Aquesta opció obliga a una transformació

del document XML cap al model que correspongui (relacional, jeràrquic u orientat a objectes), de manera que les dades s'emmagatzemen segons el model de l'SGBD.

3. Enregistrament en una base de dades XML nativa. Aquesta opció permet enregistrar directament el document XML a la base de dades sense cap tipus de transformació. Les bases de dades XML natives han estat dissenyades especialment per a l'emmagatzematge d'XML.

Una **base de dades XML** nativa és una base de dades dissenyada especialment per a l'emmagatzematge d'XML. L'abreviatura en català és BD-XML nativa. L'abreviatura anglesa és NXD.

Com a regla general, els documents de tipologia *data-centric* s'emmagatzemen en una base de dades tradicional (relacional, orientada a objectes o jeràrquica). Això es pot fer per mitjà d'eines de tercers o per les capacitats incorporades a la mateixa base de dades. En aquest últim cas, es diu que la base de dades està habilitada per a XML (*XML-enabled*). En canvi, els documents de tipologia *document-centric* s'emmagatzemen en una base de dades XML nativa o en un sistema de gestió de continguts.

Un sistema de gestió de continguts és una aplicació dissenyada per gestionar documents, construïda normalment damunt una base de dades nativa XML.

Aquestes regles no són absolutes. D'una banda, els documents *data-centric* —sobretot si es tracta de dades semiestructurades— es poden emmagatzemar en bases de dades natives. De l'altra, els documents *document-centric* poden ser emmagatzemats en bases de dades tradicionals en aquells casos en què es precisen poques funcionalitats específiques XML.

En el moment en què va aparèixer la necessitat d'emmagatzemar documents XML, l'emmagatzematge de dades estava centrat en els SGBD relacionals, jeràrquics o orientats a objectes i van sorgir dues tendències:

- D'una banda, els grans SGBD existents van començar a introduir funcionalitats XML en els seus SGBD, donant lloc a l'aparició dels SGBD-XML habilitades.
- Altrament, van començar a aparèixer els SGBD-XML natives.

Avui en dia, els límits entre els SGBD-XML habilitades i els SGBD-XML natives s'estan desdibuixant, ja que les versions actuals de molts SGBD tradicionals (relacionals, jeràrquics i orientats a objectes) ja incorporen capacitats natives d'XML i alguns SGBD-XML natives faciliten l'emmagatzematge de fragments de documents XML en bases de dades externes (usualment relacionals).

1.2 SGBD-XML natives vers SGBD predecessors

Els SGBD predecessors dels SGBD-XML natives a tenir en compte són els relacionals, jeràrquics i orientats a objectes, existents abans de la irrupció en el món de les bases de dades dels SGBD-XML natives.

Per entendre l'aparició dels SGBD-XML natives és molt interessant veure quines possibilitats facilitava cadascun dels SGBD predecessors, així com els seus inconvenients.

1.2.1 SGBD jeràrquics

Els SGBD jeràrquics (anomenats SGBDJ a partir d'ara) poden semblar una eina vàlida per emmagatzemar dades XML a causa de la naturalesa jeràrquica dels seus elements. Van aparèixer en els anys seixanta i emmagatzemen els seus elements en una jerarquia de nodes (arbre). Cada node conté dades d'identificació i un conjunt de subnodes d'un cert tipus. L'accés a les dades és sempre molt predictible i, en conseqüència, els accessos i actualitzacions estan optimitzats. La sintaxi de les consultes és molt semblant al llenguatge XPath. El principal problema d'intentar emmagatzemar dades XML en un SGBDJ és la manca de flexibilitat en l'estructura de la base de dades, ja que no es pot modificar l'estructura de la base de dades sobre la marxa, cosa bàsica en documents XML.

1.2.2 SGBD relacionals

Els SGBD relacionals (anomenats SGBDR a partir d'ara) també podrien esdevenir un receptacle per a dades XML. Tenim tres possibilitats bàsiques d'emmagatzematge d'XML en aquests SGBD:

1. Emmagatzematge del document XML complet, com a text, en una columna CLOB o BLOB.

Aquesta estratègia és una bona opció quan el document XML conté informació estàtica que només serà modificada quan el document complet sigui reemplaçat, i aquesta és la primera opció que van facilitar els primers SGBDR per començar a donar suport a l'emmagatzematge XML. L'emmagatzematge de dades XML, seguint aquesta opció, és simple d'implementar, ja que no es necessita efectuar cap transformació del document cap al model relacional, però presenta enormes deficiències a l'hora de realitzar recerques d'informació dins el document o intentar la indexació de dades.

2. Gestió (consulta i modificació) del document XML des de les eines de l'SGBDR, però emmagatzematge dins el sistema d'arxius del sistema operatiu.

BLOB i CLOB

Els BLOB (Binary Large Objects) és una col·lecció de dades binàries emmagatzemades en un SGBD com a una entitat simple. S'utilitzen normalment per emmagatzemar àudio, imatges o objectes multimèdia, tot i que també poden ser utilitzats per emmagatzemar textos. Els CLOB (Character Large Object) són una col·lecció de dades textuals emmagatzemades en un SGBD com a una entitat simple. Es diferencien dels BLOB perquè poden gestionar codificació de caràcters.

Incrustar un objecte en SGBD ofimàtic

Si heu utilitzat algun SGBD ofimàtic, l'emmagatzematge del document XML complet en una columna CLOB o BLOB seria similar a l'opció d'incrustar un objecte, que acostumen a facilitar els SGBD ofimàtics.

Aquesta opció s'acostuma a utilitzar quan el nombre de documents XML és petit i difícilment la informació que conté és actualitzable (tot i que es pot permetre l'actualització). Presenta limitacions en escalabilitat, flexibilitat a l'hora de l'emmagatzematge i recuperació i, evidentment, deficiències de seguretat, ja que les dades resideixen fora de la base de dades com a arxius externs.

Vincular o enllaçar un arxiu

Si heu utilitzat algun SGBD ofimàtic, l'emmagatzematge del document XML en el sistema d'arxius seria similar a l'opció de vincular o enllaçar un arxiu del sistema d'arxius del sistema operatiu, que acostumen a facilitar els SGBD ofimàtics.

3. Transformació (mapatge) de l'estructura de dades del document XML cap al model relacional, per emmagatzemar les dades en taules. La idea principal d'aquesta opció és observar l'estructura del document XML, amb els elements i atributs que hi apareixen i efectuar la conversió al model relacional seguint un algorisme. La taula 1.1 presenta una versió simplificada de les conversions a efectuar.

En la secció de Referències de la web hi trobareu l'enllaç a articles científics que versen sobre la transformació entre XML i el model relacional.

TAULA 1.1. Correspondència d'elements XML amb elements relacionals.

Esquema XML	Esquema relacional
Element	Taula
Atribut/Element imbricat	Columna
Atribut `ID`	Clau primària
`IDREF`/Element imbricat	Clau secundària
`#REQUIRED`, `#IMPLIED`	`NOT NULL, NULL`

Val a dir que, evidentment, l'observació cal efectuar-la sobre l'XSD o el DTD que valida els documents a mapar, per assegurar que el mapatge és correcte davant qualsevol instància de document XML que verifiqui l'XSD o el DTD.

DTD

El DTD (Document Type Definition) d'un document XML és un document que defineix tant els elements del document XML com les relacions que es donen entre ells. L'XSD (XML Schema Definition) és posterior al DTD, està escrit en XML i permet noves característiques (definir tipus de dades, utilitzar espais de noms, definir intervals de valors per als atributs i els elements...); en definitiva, l'XSD aporta un major potencial semàntic que el DTD.

Per il·lustrar aquest mapatge, observem el següent document XML de tipologia *data-centric,* per procedir a la generació del model relacional que ens permeti emmagatzemar les seves dades. Per simplificar el procés, analitzem directament el document, tot i que en un cas real caldria analitzar l'XSD o el DTD.

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <ComandesVenda>
3     <ComandaVenda>
4        <Número>1234</Número>
5        <Client>Indústries Fèrriques</Client>
6        <Data>29/10/2011</Data>
7        <Línia Número="1">
8           <Article>PIL001</Article>
9           <Quantitat>12</Quantitat>
10          <Preu>12.85</Preu>
11       </Línia>
12       <Línia Número="2">
13          <Article>CAT010</Article>
14          <Quantitat>30</Quantitat>
15          <Preu>5.25</Preu>
16       </Línia>
17    </ComandaVenda>
18    <ComandaVenda>
19       <Número>1235</Número>
20       <Client>Eines del Berguedà</Client>
21       <Data>30/10/2011</Data>
22       <Línia Número="1">
23          <Article>XQT301</Article>
24          <Quantitat>14</Quantitat>
25          <Preu>23.2</Preu>
26       </Línia>
27    </ComandaVenda>
28 </ComandesVenda>
```

Després d'una simple anàlisi del document, observem que ens calen dues taules (`ComandaVenda` i `ComandaVendaLínia`) per poder emmagatzemar la informació:

```
ComandaVenda(Número, Client, Data)
ComandaVendaLínia(Comanda, Número, Article, Quantitat, Preu)
  on Comanda referencia ComandaVenda(Número)
```

Com podeu veure, es tracta d'una situació molt simple i no del tot documentada, atès que el document XML no aporta l'XSD ni el DTD. Així, no podem estar segurs dels camps identificadors a les taules, que suposadament serien el `Número` a la taula `ComandaVenda` i la parella (`Comanda, Número`) a la taula `ComandaVendaLínia`.

Tot i que el mecanisme sembla simple, no sempre és senzill fer aquesta conversió, ja que el model relacional i l'XML parteixen de conceptes força diferents:

- El model relacional està basat en dades bidimensionals sense jerarquia ni ordre, mentre que el model XML està basat en arbres jeràrquics on l'ordre és rellevant.
- En un document XML hi pot haver dades repetides, mentre el model relacional fuig de les repeticions.
- Les relacions i les estructures dins dels documents XML no sempre són òbvies.
- I, a més, què passa si necessitem tenir el document XML de nou? Fer el procés invers no sempre és trivial. Un dels conceptes difícils és determinar quines dades eren atributs i quines eren elements.

Intenteu, per exemple, generar el model relacional per a un document XML de tipologia *document-centric*, com el següent:

```
<SèrieCòmic>

<Introducció>
La sèrie <NomCòmic>Mortadel·lo i Filemó</NomCòmic> de <Autor>
     Francisco Ibáñez Talavera</Autor> creada i desenvolupada <Any
     >a partir de 1958</Any> <Resum>narra les peripècies de dos
     agents de l'organització secreta TIA (Tècnics d'Investigació
     Aeroterràquia)</Resum>.
</Introducció>

<ComentariDetallat>

<Paràgraf>La sèrie va néixer amb el nom de <i>Mortadel·lo i Filem
     ó, agència d'informació</i>, on Filemó és el cap d'una agè
     ncia de detectius i té a Mortadel·lo com a empleat i únic
     ajudant. El 1969 ingressen a les files de la TIA, una
     desastrosa agència secreta que els permet parodiar les histò
     ries d'espies i s'incorporen nous personatges.</Paràgraf>
<Paràgraf>En qualsevol de les seves èpoques, la sèrie destaca pel
      seu humor extremadament <Link URL="http://es.wikipedia.org/
     wiki/Slapstick">slapstick</Link>, de manera que els
     personatges pateixen constantment contratemps com caigudes
     des de grans altures, explosions, aixafaments per tota mena d
     'objectes pesats (pianos, caixes fortes, etc.) sense que les
     conseqüències dels mateixos acostumin durar més d'una vinyeta
     . </Paràgraf>
```

```
11
12 <Personatges>
13 <List>
14 <Item><Link URL="mortadelo.html">Mortadel·lo</Link>Agent alt i
       prim, completament calb <i>gràcies</i> a un invent del
       professor Bacteri per fer créixer el cuir cabellut. Té un nas
        allargat i duu ulleres i un vestit negre. Es comenta que la
       característica que l'ha portat a la fama és que sempre s'està
        emprovant disfresses, amb les quals pot transformar–se, per
       a sorpresa de tothom, en qualsevol cosa, des d'un elefant
       fins a una formiga, des de Superman fins a l'home invisible.
       Al principi duia un paraigües negre penjat del braç i lluïa
       un barret d'on sempre extreia les disfresses.</Item>
15 <Item><Link URL="filemo.html">Filemó</Link>És més baix que en
       Mortadel·lo, i és el seu cap. Té dos pèls al cap, i duu una
       camisa blanca i uns pantalons vermells, encara que en els
       primers anys de la seva publicació duia una gran pipa a la
       boca i un abillament similar al de Sherlock Holmes. És qui se
        sol endur les garrotades quan alguna cosa surt malament (és
       a dir, gairebé sempre).</Item>
16 <Item><Link URL="elSuper.html">El Super</Link>És el
       superintendent (de nom Viçens) de la TIA, el qual encomana
       les missions a Mortadel·lo i Filemó. Duu un poblat bigoti i
       un vestit blau. En paraules del propi Ibáñez, és una crítica
       contra els anteriors caps que havia tingut quan treballava al
        banc.</Item>
17 <Item><Link URL="bacteri.html">Professor Bacteri</Link>és el tí
       pic professor guillat. Crea els invents més inversemblants,
       però gairebé sempre fan el contrari del que es pretenia. Té
       una gran barba negra, que els agents o el Súper prenen amb
       ferotgia contra la seva persona quan els seus invents surten
       malament.</Item>
18 <Item><Link URL="ofelia.html">Ofèlia</Link>És la secretària, una
       dona rossa i grassa enamorada d'en Mortadel·lo (però no
       corresposta) i quasi sempre utilitzada com a mofa.</Item>
19 <Item><Link URL="irma.html">Irma</Link>És l'altra secretària,
       encara que no surt tant en el còmic com l'Ofèlia (de fet va
       deixar de ser dibuixada anys enrere). És molt maca, i tant en
        Mortadel·lo com en Filemó estan bojos per ella... la qual
       cosa provoca gags interessants amb l'Ofèlia.</Item>
20 </List>
21 </Personatges>
22
23 <Paràgraf>Les històries de Mortadel·lo i Filemó s'han adaptat al
       cinema amb personatges reals, com en la <i>La gran aventura
       de Mortadel·lo i Filemó (2003)</i> dirigida per Javier Fesser
        i protagonitzada per Benito Pocino en el paper de Mortadel·
       lo i Pepe Viyuela en el paper de Filemó.</Paràgraf>
24 <Paràgraf>Una nova pel·lícula, <i>Mortadel·lo i Filemó. Missió:
       Salvar la Terra</i>, es va estrenar el 2008 coincindint amb
       el cinquantè aniversari de l'aparició dels personatges.</Parà
       graf>
25
26 </ComentariDetallat>
27
28 </SèrieCòmic>
```

El problema no només radica a aconseguir el model relacional, sinó també a poder reconstruir el document XML a partir de la informació emmagatzemada en taules.

1.2.3 SGBD orientats a objectes

La darrera opció d'emmagatzemar documents XML en SGBD predecessors dels SGBD-XML natives passa per la utilització dels SGBD orientats a objectes (anomenats SGBDOO a partir d'ara).

En aquest cas, es tracta d'analitzar el document (millor dit, l'XSD o el DTD) i efectuar un mapatge XML-OO basat en interrelacions entre classes, tot dissenyant el diagrama de classes adequat.

En la secció Referències del web hi trobareu l'enllaç a articles científics que versen sobre la transformació entre XML i el model orientat a objectes.

De forma molt simplificada, el mapatge XML-OO consisteix en el fet que cada tipus d'element XML de més alt nivell es modela com una classe d'objectes, i els atributs XML es modelen com a propietats de les classes.

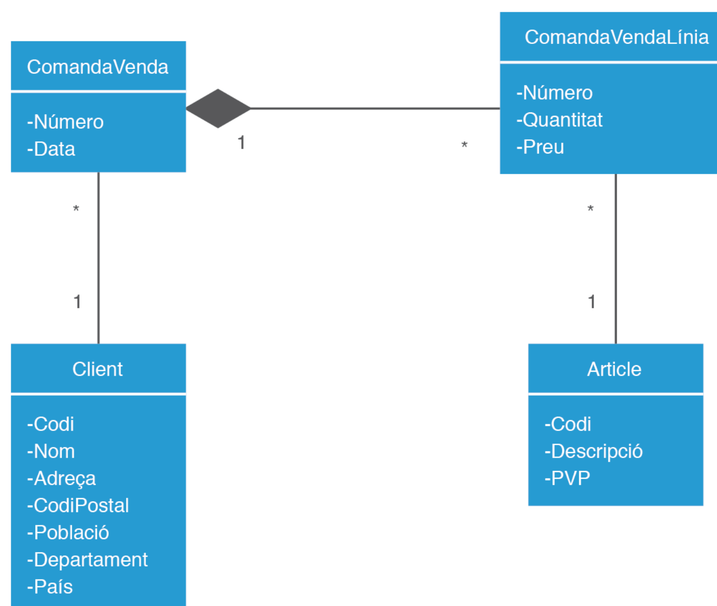
Per il·lustrar aquest mapatge, observem el següent document XML de tipologia *data-centric* per procedir a la generació del model orientat a objecte que ens permeti emmagatzemar les seves dades. Per simplificar el procés, analitzem directament el document, tot i que en un cas real caldria analitzar l'XSD o el DTD.

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <Vendes>
3     <ComandesVenda>
4        <ComandaVenda>
5           <Número>1234</Número>
6           <Client>100</Client>
7           <Data>29/10/2011</Data>
8           <Línia Número="1">
9              <Article>PIL001</Article>
10             <Quantitat>12</Quantitat>
11             <Preu>12.85</Preu>
12          </Línia>
13          <Línia Número="2">
14             <Article>CAT010</Article>
15             <Quantitat>30</Quantitat>
16             <Preu>5.25</Preu>
17          </Línia>
18       </ComandaVenda>
19       <ComandaVenda>
20          <Número>1235</Número>
21          <Client>200</Client>
22          <Data>30/10/2011</Data>
23          <Línia Número="1">
24             <Article>XQT301</Article>
25             <Quantitat>14</Quantitat>
26             <Preu>23.2</Preu>
27          </Línia>
28       </ComandaVenda>
29    </ComandesVenda>
30   <Articles>
31       <Article Codi="PIL001">
32          <Descripció></Descripció>
33          <PVP>13.15</PVP>
34       </Article>
35       <Article Codi="CAT010">
36          <Descripció></Descripció>
37          <PVP>6.15</PVP>
38       </Article>
39       <Article Codi="XQT301">
40          <Descripció></Descripció>
41          <PVP>25.00</PVP>
```

```
42          </Article>
43       </Articles>
44       <Clients>
45          <Client Codi="100">
46             <Nom>Indústries Fèrriques</Nom>
47             <Adreça>Carrer Muntanya, 5</Adreça>
48             <CodiPostal>08700</CodiPostal>
49             <Població>Igualada</Població>
50             <Departament>Barcelona</Departament>
51             <País>ES</País>
52          </Client>
53          <Client Codi="200">
54             <Nom>Eines del Berguedà</Nom>
55             <Adreça>Gran Via, 36</Adreça>
56             <CodiPostal>08600</CodiPostal>
57             <Població>Berga</Població>
58             <Departament>Barcelona</Departament>
59             <País>ES</País>
60          </Client>
61       </Clients>
62    </Vendes>
```

Després de l'anàlisi del document podem arribar a la conclusió que el diagrama de classes de la figura 2.1 ens podria servir per emmagatzemar la informació.

Figura 1.1. Diagrama de classes



Una vegada trobat el diagrama de classes adequat, cal efectuar-ne la implementació que pertoqui en l'SGBD orientat a objectes. La mateixa tècnica es pot utilitzar per, a partir del diagrama de classes, efectuar-ne la implementació en un SGBD objecte-relacional.

1.2.4 SGBD-XML habilitades

El model de dades d'XML presenta característiques que l'apropen a qualsevol dels SGBD predecessors dels SGBD-XML natives (SGBDJ, SGBDR i SGBDOO). Les tres tipologies presenten punts a favor i en contra per ser utilitzades en l'emmagatzematge de documents XML:

1. El fet que els elements d'un document XML puguin contenir dades heterogènies ens apropa als SBGDOO, però el problema apareix quan l'XSD o el DTD no permeten flexibilitat d'ubicació dels elements (és a dir, quan l'ordre és rellevant).
2. La rellevància en la ubicació dels elements ens apropa als SGBDJ.
3. Els SGBDR, tan estesos en l'actualitat, són adequats per a documents *data-centric* en els quals no hi hagi gran complexitat.

Els grans fabricants d'SGBD, davant la irrupció de l'XML, van realitzar esforços per incorporar informació XML en els seus productes, donant lloc als SGBD-XML habilitades. Sense voler deixar de banda els altres SGBD, potser els SGBDR-XML habilitades són els que, atesa la seva gran estesa, han evolucionat més.

Les extensions XML que han anat incorporant els SGBDR han tingut crítiques que han constituït el punt de partida per al disseny dels SGBD-XML natives. Les crítiques inclouen:

- **Pèrdua d'estructura**. Si la jerarquia de dades XML és complexa, la seva conversió -en cas de SGBDR- a un conjunt de taules produeix una gran quantitat d'aquestes o de columnes dins de cada taula amb valors nuls. Es poden relacionar les taules resultants per mantenir l'estructura jeràrquica d'XML, però sol ser un procés complicat per a estructures de dades XML complexes.
- **Poca flexibilitat de les consultes**. És molt possible que es vulguin fer consultes sobre qualsevol element o propietat d'XML, però podria no ser possible si l'element no està inclòs en cap índex.
- **Poca velocitat d'accés**. Segons la manera com les dades XML estan emmagatzemades físicament, pot resultar força lent recuperar-les en una estructura relacional.
- **Impossible utilització directa de tecnologies XML**, com les consultes XPath, XSLT, XQL o XQuery.

Tot i que aquestes crítiques han afavorit l'aparició dels SGBD-XML natives, cal tenir en compte que els grans fabricants d'SGBD han continuat amb l'evolució de les funcionalitats XML que facilitaven els primers SGBD-XML habilitades, arribant al moment actual en el qual faciliten prestacions d'SGBD-XML natives. Això alimenta la controvèrsia: cal considerar-los SGBD-XML natives?

1.2.5 SGBD-XML natives

Què és un SGBD-XML natives? En principi, aquest terme va ser creat per fer referència als SGBD dissenyats especialment i única per a l'emmagatzematge de documents XML. Avui en dia, aquesta definició porta controvèrsia, atès que:

1. No serien SGBD-XML natives aquells que, sent predecessors dels SGBD-XML natives (relacionals, jeràrquics i orientats o objectes), incorporen emmagatzematge XML natiu, és a dir, inclouen l'emmagatzematge en format XML.
2. Podrien ser considerats SGBD aquells que, tot i facilitar l'emmagatzematge XML natiu, no compleixen les característiques ACID que ha de proporcionar tot SGBD.

El terme ACID en el món de les bases de dades

En el món de les bases de dades, el terme ACID és un acrònim de les característiques necessàries que ha de facilitar l'SGBD perquè un conjunt d'instruccions puguin ser considerades una transacció. Aquestes característiques són atomicitat (atomicity), consistència (consistency), aïllament (isolation) i durabilitat (durability), que en llengua anglesa donen lloc a l'acrònim. Les seves característiques són:

- **Atomicitat**: propietat que assegura que l'operació s'ha realitzat en la seva totalitat o no s'ha realitzat. Per tant, davant un error del sistema no pot quedar a mitges.
- **Consistència**: també anomenada integritat, és la propietat que assegura que no es perdrà mai la integritat de la base de dades.
- **Aïllament**: propietat que assegura que una operació no pot afectar d'altres, és a dir, l'execució de dues transaccions sobre la mateixa informació, que siguin independents i no provoquin cap error.
- **Durabilitat**: propietat que assegura que una vegada executada una operació, aquesta serà persistent i no es podrà desfer en cap cas, ni davant una fallada del sistema.

Davant de tot això, el concepte d'SGBD-XML natives ha evolucionat i va més enllà de quedar definit per una única característica.

Un **SGBD-XML natives** és un sistema de gestió de la informació que ha de:

- Definir un model lògic per a un document XML (en contraposició als SGBD que defineixen el model per les dades) i enregistrar i recuperar els documents segons aquest model. Com a mínim, el model ha d'incloure elements, atributs, `PCDATA` i l'ordre del document.
- Mantenir una relació transparent amb el mecanisme subjacent d'emmagatzematge, incorporant les característiques ACID de qualsevol SGBD.
- Incloure un nombre arbitrari de nivells de dades i complexitat.
- Permetre les tecnologies de consulta i transformació pròpies d'XML: XPath, XSLT, XQL, XQuery, etc., com a vehicle principal de consulta i gestió.

- Permetre la introducció d'informació *data-centric*, *document-centric* i mixta.

El segon punt de la definició porta implícita la idea que un SGBD-XML natives no té per què tenir cap model físic d'emmagatzematge subjacent específic, sinó que es podria construir damunt un SGBDR, un SGBDOO, un SGBDJ, o un sistema propi d'emmagatzematge. Això sembla contradictori amb els problemes i crítiques que presenten per a la gestió de documents XML els SGBD predecessors dels SGBD-XML natives.

La realitat és que el camp de les bases de dades està dividit en dos: un, abanderat per les grans empreses d'SGBD (Oracle Corporation, IBM Corporation, Microsoft Corporation), que argumenten que els seus productes (inicialment XML habilitats) han solucionat tots els problemes inicials i que faciliten emmagatzematge XML natiu; l'altre, defensat per les empreses que utilitzen un model físic de dades propi, ideat específicament per a l'emmagatzematge XML, que destaquen els problemes dels primers. De ben segur que els dos tipus d'SGBD conviuran durant molt de temps a causa, entre d'altres raons, de la gran quantitat de dades que es troben emmagatzemades en SGBDR.

Atès, doncs, que en el moment actual hi ha poca diferència entre els inicialment anomenats SGBD-XML natives i les extensions XML que proporcionen els SGBD no XML, el terme "natiu/va" és àmpliament utilitzat per designar els productes especialitzats en solucions de BD-XML.

Funcionalitats usuals de les BD-XML natives

En el mercat trobem, actualment, molts productes SGBD-XML natives, però no tots faciliten les mateixes prestacions. Ens cal conèixer les principals característiques a avaluar en les BD-XML natives per tal de poder escollir el producte que millor s'adapti a les nostres necessitats:

- Emmagatzematge dels documents XML en col·leccions. Les col·leccions juguen, en les BD-XML natives, el paper de les taules en les BDR. Els documents s'acostumen a agrupar, en funció de la informació que contenen, en col·leccions que, a la vegada, haurien de poder contenir altres col·leccions. No tots els SGBD-XML natives faciliten aquest mecanisme de la forma que s'ha presentat i, a més, n'hi ha que confonen el concepte "col·lecció" amb el concepte "bases de dades". En realitat, un SGBD-XML hauria de possibilitar la gestió de diverses BD i cada BD hauria de possibilitar definir col·leccions i, a la vegada, cada col·lecció hauria de poder contenir altres col·leccions. A la realitat, ens trobem SGBD-XML que permeten aquesta organització, però n'hi ha que només permeten una única BD, la qual sí permet col·leccions; altres permeten diverses BD però no permeten col·leccions i, en aquest cas, haurem d'utilitzar diferents BD si ens interessa organitzar la informació en col·leccions.
- Validació de documents. Caldria que l'SGBD permetés la introducció d'XSD o DTD per validar els documents XML que gestionem a la BD.

- Tecnologies de consulta i transformació pròpies d'XML: XPath, XSLT, XQL, XQuery, etc., com a vehicle principal de consulta i gestió. Per definició, els SGBD-XML natives han d'incorporar aquestes tecnologies. Cal veure quines en facilita i la seva versió.

- Estratègies d'actualització. Evidentment, és molt interessant poder actualitzar els documents XML d'una BD-XML nativa. Els actuals SGBD-XML natives permeten, sense cap problema, l'actualització d'un document XML via substitució total. Però cal anar més lluny i veure la possibilitat d'actualitzar (inserir, modificar, eliminar) dades dels documents XML sense haver de substituir el document sencer. En aquest punt, els SGBD-XML natives estan treballant per assolir millors prestacions, que passen per implementar algun dels llenguatges de modificació de les dades dels documents XML que han anat sorgint:

  - Llenguatge XUpdate, promogut pel grup XML:DB (any 2000). Avui en dia, aquest estàndard ja és obsolet.
  - Llenguatge Update, promogut per Patrick Lehti (any 2001).
  - Llenguatge XQUF, promogut per W3C (any 2011).

- Indexació XML. La indexació en els SGBD-XML natives és un dels punts febles a tenir en compte a l'hora d'avaluar l'SGBD. En el món de les BDR, a l'hora de definir el model relacional de les dades, es determina quins són els índexs adequats perquè les consultes més emprades siguin eficients. En les BD-XML natives no és tan fàcil, perquè qualsevol estructura present en XML pot formar part d'una consulta. Es podria fer una indexació completa de tots els elements presents en una BD-XML nativa per aconseguir les consultes més ràpides, però això porta dos problemes: en primer lloc, la sobrecàrrega necessària per mantenir els índexs portaria a actualitzacions molt lentes, i en segon lloc hi ha el fet que no es coneix quines etiquetes i atributs tindrà un document XML abans d'introduir-lo a la base de dades. Atès que l'estructura d'un document XML i la naturalesa de les consultes és impredictible, és mot difícil planificar la indexació per endavant. Per tant, les BD-XML natives utilitzen tècniques d'indexació adaptatives (depenent del fabricant de l'SGBD) com, per exemple, una combinació d'indexació controlada per l'administrador de la BD-XML nativa, junt amb tècniques avançades de recuperació de textos d'alt rendiment.

- Interfície d'usuari. És molt interessant que l'SGBD aporti una (com a mínim) interfície d'usuari agradable i intuïtiva, que faciliti una ràpida utilització. En els moments actuals és d'agrair l'existència d'una interfície web que no precisi de cap instal·lació en una màquina client.

- Protocols d'accés. Un altre punt interessant a tenir en compte és conèixer els protocols d'accés que permet l'SGBD, ja que ens facilitaran diferents mecanismes de connectivitat amb l'SGBD per tal de poder accedir a la informació que emmagatzemen les BD-XML natives. Entre els protocols d'accés actuals típics de trobar en els SGBD-XML natives tenim WebDAV, REST, SOAP i XML-RPC.

Protocols d'accés

Web-based Distributed Authoring and Versioning (**WebDAV**) és un conjunt de mètodes basats en el protocol HTTP que proporciona funcionalitats per crear, canviar i moure documents en un servidor remot (típicament un servidor web). S'utilitza sobretot per permetre l'edició dels documents que serveix un servidor web, però també es pot aplicar a sistemes d'emmagatzematge generals basats en web, que puguin ser accedits des de qualsevol lloc.

La majoria de sistemes operatius moderns proporcionen suport per a WebDAV, fent que els arxius d'un servidor WebDAV apareguin emmagatzemats en una unitat local. Tot i que va néixer per controlar també les versions dels documents, aquesta funcionalitat no s'ha implementat i, probablement, haurien de canviar-li el nom i deixar-lo com a WebDA.

Representational State Transfer (**REST**) és un estil d'arquitectura del programari per a sistemes hipermèdia distribuïts que permet obtenir continguts (informació) des d'un lloc web mitjançant la lectura d'una pàgina web (protocol HTTP) que conté codi XML que descriu i inclou la informació. REST es basa únicament en la utilització del protocol HTTP i el llenguatge XML, i no necessita de cap protocol d'intercanvi de missatges.

Així, per exemple, REST podria ser utilitzat per un editor en línia per posar a disposició dels subscriptors continguts sindicats. Periòdicament, l'editor prepararia i activaria una pàgina web que inclouria els continguts amb l'XML descriptor dels continguts. Els subscriptors només haurien de conèixer l'adreça URL de la pàgina, on hi accedirien amb un navegador web, obtindrien la informació i la podrien formatar i usar adequadament per als seus propòsits. REST no és un estàndard reconegut pel W3C, però és àmpliament utilitzat.

Simple Object Access Protocol (**SOAP**) és un protocol estàndard (sota la tutela del W3C) que defineix com dos objectes en diferents processos es poden comunicar via intercanvi de dades XML. És molt habitual la seva utilització per accedir a serveis web (web services). SOAP té diferents tipus de missatges, però els que més es fan servir són els que segueixen el patró de crida remota a aplicacions (RPC-Remote Procedure Call),en què el client fa una petició (request) al servidor i aquest respon immediatament amb un missatge (response) que conté la resposta a la petició del client. SOAP no deixa de ser una evolució del protocol de comunicació XML-RPC.

XML-RPC és un protocol RPC (*Remote Procedure Call*) que utilitza XML per codificar les crides i el protocol HTTP com a mecanisme de transport de les dades.

- Interfícies de programació d'aplicacions. Un darrer apartat que cal considerar a l'hora de decantar-nos per un SGBD-XML natives és conèixer quines API facilita. Normalment, els SGBD faciliten una API pròpia i simple (baix nivell) amb la qual podrem interactuar amb llenguatges com Java i C++. A banda de l'API pròpia de cada SGBD, interessa saber si hi ha implementacions de les API estàndards per a accés a BD-XML natives, com són XML:DB (ja obsoleta) i XQJ.

API estàndards per a accés a BD-XML natives

XML:DB, també anomenada **XAPI**, és una API ideada pel grup XML:DB, iniciativa apareguda l'any 2000 per intentar aconseguir un mecanisme estàndard d'accés a les BD-XML natives, de manera similar al mecanisme JDBC per a les BDR, davant els mecanismes propietaris que cada SGBD-XML natives es veia obligat a dissenyar. El grup està inactiu des del 2003. Per tant, l'API es considera obsoleta.

XQuery API for Java (**XQJ**) és una interfície de programació d'aplicacions Java pensada per utilitzar el llenguatge XQuery (i també el llenguatge XUpdate) per obtenir informació de BD-XML natives, de manera similar a com JDBC és una API pensada per utilitzar el llenguatge SQL per accedir a BDR. XQJ va néixer el 2003 i la seva versió definitiva ha estat publicada el 2009. També és coneguda com a JSR 225,ja que ha estat dissenyada com un projecte JCP (Java Community Process). De moment (versió 1.7 de Java), no forma part de la biblioteca estàndard de classes Java.

Avantatges i inconvenients dels SGBD-XML natives

Els avantatges que s'adjudiquen les BD-XML natives, en comparació a les BD no XML, són els següents:

- Faciliten accés i emmagatzematge d'informació en format XML sense necessitat de codi addicional ni cap tipus de mapatge.
- La majoria d'SGBD-XML natives incorporen un motor de cerca d'alt rendiment.
- És molt senzill afegir nous documents XML.
- Permeten emmagatzemar dades heterogènies.
- Conserven la integritat dels documents (es poden recuperar en el seu estat inicial).

Per contra, els inconvenients que s'associen als SGBD-XML natives són aquests:

- La gran quantitat d'espai necessari per emmagatzemar el mateix document XML com a format de representació de la informació, a causa del fet que les etiquetes poden suposar el 75% de la informació d'un document XML. I això és,sense cap mena de dubte, innecessari en guardar molts documents validats per un mateix XSD o DTD.
- El fet que les BD-XML natives només puguin guardar i retornar dades en format XML.
- En emmagatzemar la informació en format XML es fa molt complicat poder generar noves estructures a partir de la informació existent com, per exemple, aconseguir càlculs estadístics.
- Les dificultats d'indexació del contingut d'una base de dades, que ha de permetre la reducció del temps necessari (d'ordre seqüencial a ordre logarítmic) per trobar certs elements clau.
- Les pobres facilitats per modificar el contingut dels documents XML emmagatzemats sense haver de substituir tot el document.

Els dos darrers inconvenients (indexació-actualització) són cavalls de batalla dels SGBD i de ben segur que s'anirà avançant en aquest camp fins que deixin de ser inconvenients.

Estratègies d'emmagatzematge dels SGBD-XML natives

Els SGBD-XML natives no tenen cap model d'emmagatzematge físic subjacent concret i, en conseqüència, poden ser construïts sobre bases de dades relacionals, jeràrquiques, orientades a objectes o en formats d'emmagatzematge propietaris. Així doncs, podem trobar:

- Emmagatzematge basat en text (arxius de text). Emmagatzemen el document XML sencer en forma de text i proporcionen alguna funcionalitat de bases de dades per accedir-hi. Apliquen, com a molt, tècniques de compressió per reduir l'espai d'emmagatzematge i mantenen índexs addicionals per augmentar l'eficiència d'accés a la informació. Es poden definir sobre BD o sistemes d'arxius:
  - Possibilitat simple: emmagatzemar el document com un BLOB en una base de dades relacional o mitjançant un fitxer i proporcionar alguns índexs sobre el document que accelerin l'accés a la informació.
  - Possibilitat sofisticada: emmagatzemar el document en un magatzem adequat amb índexs, suport per a transaccions...
- Emmagatzematge basat en un model. Defineixen un model de dades lògic (com DOM) per a l'estructura jeràrquica dels documents XML i emmagatzemen els documents d'acord amb aquest model utilitzant el model d'emmagatzematge físic que es desitgi (mapatge a BDR, mapatge a BDOO...). Així, tenim:
  - Possibilitat 1: traduir el DOM al model relacional.
  - Possibilitat 2: traduir el DOM al model OO.
  - Possibilitat 3: utilitzar un magatzem creat especialment per a aquesta finalitat.
- Emmagatzematge desenvolupat específicament per a la gestió de documents XML.

Arribats a aquest punt, ens adonem que els SGBD-XML natives no deixen d'utilitzar, per al seu emmagatzematge, les estratègies que directament ens podem plantejar per emmagatzemar XML en SGBD no XML (mapatge al model relacional, mapatge al model orientat a objectes...). L'avantatge d'utilitzar els SGBD-XML natives és que l'estratègia d'emmagatzematge que utilitzen no afecta (és transparent) els usuaris de la BD, els quals es limiten a treballar amb documents XML i amb els llenguatges XML que facilita l'SGBD, i mai han de pensar a efectuar cap tipus de mapatge.

Productes SGBD-XML natives en el mercat

Els SGBD-XML natives existents en el mercat són molts i presentar-los tots es converteix en una tasca difícil.

Als Annexos del web hi trobareu l'annex "Productes SGBD-XML natives en el mercat", amb un ampli recull de productes.

1.3 Biblioteques Java específiques de l'SGBD-XML natives

Els diversos SGBD-XML natives faciliten biblioteques per permetre l'accés a l'SGBD des dels llenguatges més populars (Java, C++, .NET, Perl, PHP, etc.), però no tots els SGBD faciliten les biblioteques per a tots els llenguatges. Així

doncs, per saber quines biblioteques facilita cada SGBD caldrà analitzar la seva documentació. D'altra banda, tenint en compte que Java és un dels llenguatges més utilitzats actualment, és força normal que la majoria d'SGBD facilitin les seves biblioteques per poder desenvolupar aplicacions en aquest llenguatge que accedeixin a les BD-XML natives.

Les BD-XML natives són relativament modernes i, en els darrers anys, des del 2000, hi ha hagut diversos intents de generar un estàndard de connexió Java per als SGBD-XML natives. El primer intent va ser l'API XML:DB (o XAPI), promogut pel grup XML:DB entre els anys 2000 i 2003. Posteriorment, entre el 2003 i el moment actual, ha anat prosperant, com a segon intent, l'API XQJ, sota la tutela de JCP (*Java Community Process*). Molts SGBD-XML natives han proporcionat o proporcionen biblioteques XML:DB i/o XQJ, de manera que si desenvolupem aplicacions utilitzant aquestes biblioteques, aquestes aplicacions podran accedir indistintament als diversos SGBD que proporcionen implementacions per a aquestes API.

La utilització de les API estàndards facilita la reutilització de les aplicacions per a diferents SGBD, però constitueixen una capa més de programari, ja que normalment els SGBD faciliten una biblioteca pròpia per accedir a l'SGBD i les API estàndards són una capa intermèdia entre la biblioteca pròpia de l'SGBD i l'aplicació que utilitza les API estàndard. Pot ser interessant, doncs, conèixer l'API pròpia que facilita l'SGBD. Concretament, l'API Java que proporcionen els SGBD-XML BaseX i Sedna.

1.3.1 API Java de l'SGBD BaseX

Als Annexos del web trobareu l'annex "Introducció a l'SGBD-XML natives BaseX" amb les indicacions per instal·lar aquest SGBD i tenir-hi una primera presa de contacte.

A la pàgina web oficial de BaseX, (basex.org) a l'apartat "Desenvolupament", es facilita el codi de clients per a diversos llenguatges. En concret, pot ser interessant fer una ullada al client Java (`BaseXClient.java`), que va acompanyat d'exemples d'utilització.

L'SGBD-XML natives BaseX proporciona un protocol client/servidor que permet escriure clients en diversos llenguatges de programació. El desenvolupament de clients per accedir a BaseX sobrepassa l'objectiu d'aquest apartat, on simplement volem utilitzar el client Java que se'ns facilita.

Entre els clients que BaseX facilita per a diversos llenguatges de programació, a nosaltres ens podria interessar el client per a Java (`BaseXClient.java`). BaseX, però, també facilita una extensa API Java, la qual incorpora un client equivalent al facilitat per la classe `BaseXClient`.

Nosaltres utilitzarem les classes incorporades a l'API Java que facilita la versió 7.1 de BaseX i que trobarem en diversos paquets inclosos en el fitxer `basex.jar` ubicat a la carpeta arrel d'on s'hagi efectuat la instal·lació de l'SGBD.

La instal·lació de BaseX no incorpora la documentació (arxiu javadoc) d'aquesta API, la qual es pot trobar a la pàgina web oficial de BaseX i també als annexos de la web, a l'apartat "Material per desenvolupament sobre SGBD-XML natives BaseX".

L'API Java de BaseX facilita molts paquets i classes. En aquest moment ens interessa fer una ullada a les necessàries per desenvolupar programes clients de l'SGBD. En concret:

- Classe `org.basex.server.ClientSession`, que ens facilita el mecanisme per establir una sessió client amb un servidor BaseX i un seguit de mètodes per executar accions en la sessió establerta:
  - Mètodes constructors, per establir una sessió client amb un servidor.
  - Mètode `close()` per tancar la sessió.
  - Mètode `create()` per crear bases de dades.
  - Mètode `add()` per afegir un document a la base de dades oberta.
  - Mètode `replace()` per substituir un document a la base de dades oberta.
  - Mètode `query()` per crear un objecte `ClientQuery` a partir d'una consulta emmagatzemada en una `String`, a punt de ser executada.
  - Mètodes `execute()`, heretats de la classe abstracta `Session`, pensats per executar qualsevol de les ordres que admet BaseX. En concret, l'utilitzarem per obrir la base de dades amb la qual vulguem treballar, atès que no es facilita cap mètode específic per a aquesta tasca.
  - Mètode `info()`, heretat de la classe abstracta `Session`, per obtenir informació de la darrera ordre executada.
- Classe `org.basex.server.ClientQuery`, pensada per executar consultes XPath, XQuery i XQUF a la base de dades activa. Per això ens facilita els mètodes:
  - Mètode `execute()`, que executa la consulta i retorna el resultat complet de la consulta.
  - Mètode `info()`, per obtenir informació de la darrera consulta executada.
  - Mètode `close()`, per tancar la consulta.
- Classe `org.basex.core.BaseXException` per gestionar algunes de les excepcions que llença BaseX.

Podeu obtenir un llistat de les ordres que admet un servidor BaseX tot posant en marxa la consola BaseX Client que el procés d'instal·lació deixa a l'arbre de programes, entrant-hi amb usuari i contrasenya `admin` i demanant ajuda via l'ordre `help`.

Per executar qualsevol programa que utilitzi els paquets `org.basex.core` i `org.basex.core.cmd` haureu de tenir el fitxer `basex.jar` inclòs en el `CLASSPATH` actiu.

Cal tenir clara la diferència entre l'execució d'una `query` (XPath, XQuery i XQUF) i l'execució d'una ordre de consola de les que proporciona BaseX. El mètode `ClientSession.execute()` està pensat per executar una ordre de consola, mentre que el mètode `ClientQuery.execute()` està ideat per executar una `query`.

Amb tota aquesta informació, ja estem en condicions de desenvolupar alguns programes Java que interactuïn amb les bases de dades d'un servidor BaseX. Els exemples següents estan basats en el servidor BaseX instal·lat seguint les indicacions del material web.

En el servidor BaseX instal·lat seguint les instruccions de l'apartat "Introducció a l'SGBD-XML natives BaseX" dels annexos de la web, hi tenim instal·lada una base de dades de nom `mondial`. Hi basarem els exemples desenvolupats en aquest apartat.

Exemple de connexió a un servidor BaseX per executar-hi una consulta senzilla

El programa `BX_Client01.java` visualitza els noms dels continents emmagatzemats en el document `mondial.xml` de la base de dades `mondial`.

Trobareu el fitxer `BX_Client01.java` dins el paquet de codi font a l'annex "Material per a desenvolupament sobre SGBD-XML natives BaseX" dels Annexos del web.

Trobareu el fitxer `BX_Client02.java` dins el paquet de codi font a l'annex "Material per a desenvolupament sobre SGBD-XML natives BaseX" dels Annexos del web.

Exemple de connexió a un servidor BaseX per procedir a crear-hi una base de dades buida

El programa `BX_Client02.java` mostra com es pot crear una base de dades buida. Per fer-ho s'utilitza el mètode `ClientSession.create()`, que obliga a introduir un primer document XML que afegim a la base de dades.

L'execució d'aquest programa introduint el nom d'una base de dades ja existent, obté:

```
G:\>java -Dfile.encoding=cp850 proves.BX_Client02 mondial
Ja existeix una BD amb aquest nom.
```

L'execució d'aquest programa introduint el nom d'una base de dades inexistent, obté:

```
G:\>java -Dfile.encoding=cp850 proves.BX_Client02 novaBD
No existeix cap BD amb aquest nom.
Procedim...
Database 'novaBD' created in 299.87 ms.
```

Ordres d'un servidor BaseX

Podeu obtenir un llistat de les ordres que admet un servidor *BaseX*, tot posant en marxa la consola *BaseX Client* que el procés d'instal·lació deixa a l'arbre de programes, entrant-hi amb usuari i contrasenya `admin` i demanant ajuda via l'ordre `help`.

Per crear una base de dades (buida o no), a més del mètode `ClientSession.create()`, disposem de l'ordre *create database* que pot ser invocada pel mètode *ClientSession.execute()*. Només cal substituir el codi:

```
session.create(args[0],new ByteArrayInputStream("".getBytes()));
```

per:

```
session.execute ("create database "+args[0]);
```

Trobareu el fitxer `BX_Client03.java` dins el paquet de codi font a l'annex "Material per a desenvolupament sobre SGBD-XML natives BaseX" dels Annexos del web.

Exemple de connexió a un servidor BaseX per executar-hi consultes sobre la base de dades mondial, a introduir per l'usuari

El programa `BX_Client03.java` facilita la possibilitat que l'usuari introdueixi qualsevol consulta admesa per BaseX sobre la base de dades `mondial`.

Comprovem el funcionament d'aquest programa davant diverses instruccions a introduir per l'usuari. L'execució en qualsevol cas és:

```
G:\>java -Dfile.encoding=cp850 proves.BX_Client03
Introdueixi la instrucció a executar...
Per finalitzar, introdueixi una línia buida (en blanc):
```

Vegem quina és la resposta davant les següents instruccions:

Consulta simple (una sola línia)

```
//continent/@name/string()

Consulta executada:
//continent/@name/string()

Resultats:
Europe Asia America Australia/Oceania Africa

Informació de la consulta executada:
Hit(s): 5 Items
Updated: 0 Items
Total Time: 122.48 ms
```

Consulta complexa (instrucció FLWOR)

```
for $i in //continent
return $i/@name/string()

Consulta executada:
for $i in //continent
```

```
6  return $i/@name/string()
7
8  Resultats:
9  Europe Asia America Australia/Oceania Africa
10
11 Informació de la consulta executada:
12 Hit(s): 5 Items
13 Updated: 0 Items
14 Total Time: 14.2 ms
```

Intent d'execució d'una instrucció que no és consulta (query)

```
1  create database xxx
2
3  La instrucció introduïda o no és executable com a consulta,
4  o té algun error sintàctic.
5  Error reportat pel servidor:
6  org.basex.core.BaseXException: Stopped at line 1, column 7:
7  [XPST0003] Unexpected end of query: 'database xxx'.
8          at org.basex.server.ClientQuery.exec(ClientQuery.java:93)
9          at org.basex.server.ClientQuery.execute(ClientQuery.java
               :58)
10         at proves.BX_Client03.main(BX_Client03.java:55)
```

En l'SGBD BaseX, els conceptes bases de dades i col·lecció són equivalents. BaseX ens facilita la funció `collection()` per fer referència a la base de dades oberta.

Enumerar els documents existents a la base de dades

```
1  for $doc in collection()
2  return document-uri($doc)
3
4  Consulta executada:
5  for $doc in collection()
6  return document-uri($doc)
7
8  Resultats:
9  mondial.xml
10
11 Informació de la consulta executada:
12 Hit(s): 1 Item
13 Updated: 0 Items
14 Total Time: 23.9 ms
```

Llista de totes les bases de dades, amb els seus documents

```
1  <collection nom="{$i}">
2  {
3  for $doc in collection($i)
4  return <document>{document-uri($doc)}</document>
5  }
6  </collection>
7
8  Consulta executada:
9  for $i in db:list()
10 return
11 <collection nom="{$i}">
12 {
13 for $doc in collection($i)
14 return <document>{document-uri($doc)}</document>
15 }
16 </collection>
17
18 Resultats:
19 <collection nom="mondial">
20 <document>mondial.xml</document>
21 </collection>
22
23 Informació de la consulta executada:
24 Hit(s): 1 Item
```

```
Updated: 0 Items
Total Time: 79.79 ms
```

BaseX facilita la funció `db:list()` per obtenir un llistat de les bases de dades.

Enumerar les bases de dades existents

```
db:list()

Consulta executada:
db:list()

Resultats:
mondial

Informació de la consulta executada:
Hit(s): 1 Item
Updated: 0 Items
Total Time: 4.2 ms
```

Transaccions en XQUF

XQUF no facilita instruccions específiques per gestionar transaccions i s'entén que cada execució d'una instrucció XQUF és una transacció. En cas que vulguem executar en una transacció diverses instruccions, cal posar-les en seqüència, separades per una coma.

Inserir nous nodes (seguint sintaxi XQUF)

```
insert node <institut nom="IOC">IOC</institut>
after //continent[@name="Asia"]

Consulta executada:
insert node <institut nom="IOC">IOC</institut>
after //continent[@name="Asia"]

Resultats:

Informació de la consulta executada:
Hit(s): 0 Items
Updated: 1 Item
Total Time: 991.29 ms
```

Podem comprovar, via BaseX GUI, que la inserció s'ha dut a terme.

Canviar nodes (seguint sintaxi XQUF)

```
replace node //institut[@nom="IOC"]
with
<cicle>
<codi>DAM</codi>
<nom>Desenvolupament d'aplicacions multiplataforma</nom>
</cicle>

Consulta executada:
replace node //institut[@nom="IOC"]
with
<cicle>
<codi>DAM</codi>
<nom>Desenvolupament d'aplicacions multiplataforma</nom>
</cicle>

Resultats:

Informació de la consulta executada:
Hit(s): 0 Items
Updated: 1 Item
Total Time: 32.58 ms
```

Podem comprovar, via BaseX GUI, que la substitució s'ha dut a terme.

Canviar el valor de nodes (seguint sintaxi XQUF)

```
replace value of node //institut[@nom="IOC"]
with "Institut Obert de Catalunya"
```

```
3
4 Consulta executada:
5 replace value of node //institut[@nom="IOC"]
6 with "Institut Obert de Catalunya"
7
8 Resultats:
9
10 Informació de la consulta executada:
11 Hit(s): 0 Items
12 Updated: 1 Item
13 Total Time: 156.83 ms
```

Podem comprovar, via BaseX GUI, que la substitució s'ha dut a terme.

Reanomenar nodes (seguint sintaxi XQUF)

```
1 rename node //cicle as "cf"
2
3 Consulta executada:
4 rename node //cicle as "cf"
5
6 Resultats:
7
8 Informació de la consulta executada:
9 Hit(s): 0 Items
10 Updated: 1 Item
11 Total Time: 20.58 ms
```

Podem comprovar, via BaseX GUI, que el canvi de nom s'ha dut a terme.

Eliminar nodes (seguint sintaxi XQUF)

```
1 delete node //cf
2
3 Consulta executada:
4 delete node //cf
5
6 Resultats:
7
8 Informació de la consulta executada:
9 Hit(s): 0 Items
10 Updated: 1 Item
11 Total Time: 29.3 ms
```

Podem comprovar, via BaseX GUI, que l'eliminació s'ha dut a terme.

Exemple de connexió a un servidor BaseX per executar-hi ordres a introduir per l'usuari

El programa `BX_Client04.java` facilita la possibilitat que l'usuari introdueixi qualsevol ordre admesa per BaseX.

El programa permet introduir diverses instruccions multilínia, que han de finalitzar amb una línia buida (en blanc). Per finalitzar el programa cal introduir una instrucció buida (en blanc).

Comprovem el funcionament d'aquest programa, tot introduint la seqüència d'instruccions per aconseguir:

1. Obrir la base de dades `mondial`.
2. Exportar la base de dades a un arxiu XML.
3. Crear una base de dades anomenada `ioc`.
4. Obrir la base de dades `ioc`.
5. Afegir a la base de dades l'arxiu abans assolit via exportació.

Ordres d'un servidor BaseX

Podeu obtenir un llistat de les ordres que admet un servidor BaseX, tot posant en marxa la consola BaseX Client que el procés d'instal·lació deixa a l'arbre de programes, entrant-hi amb usuari i contrasenyaadmin i demanant ajuda via l'ordre help.

Trobareu el fitxer `BX_Client04.java` dins el paquet de codi font a l'annex "Material per a desenvolupament sobre SGBD-XML natives BaseX" dels Annexos del web.

6. Tancar la base de dades.

La seqüència d'execució és la següent:

```
G:\>java -Dfile.encoding=cp850 proves.BX_Client04
Introdueixi la instrucció a executar...
Per finalitzar, introdueixi una línia buida (en blanc)
Per finalitzar el programa, introdueixi instrucció buida
open mondial

Informació final del servidor:
Database 'mondial' was opened in 3.15 ms.

Introdueixi la instrucció a executar...
Per finalitzar, introdueixi una línia buida (en blanc)
Per finalitzar el programa, introdueixi instrucció buida
export C:\mondial.xml

Informació final del servidor:
Database 'mondial' was exported in 539.49 ms.

Introdueixi la instrucció a executar...
Per finalitzar, introdueixi una línia buida (en blanc)
Per finalitzar el programa, introdueixi instrucció buida
create database ioc

Informació final del servidor:
Database 'ioc' created in 167.53 ms.

Introdueixi la instrucció a executar...
Per finalitzar, introdueixi una línia buida (en blanc)
Per finalitzar el programa, introdueixi instrucció buida
open ioc

Informació final del servidor:
Database 'ioc' was opened in 3.13 ms.

Introdueixi la instrucció a executar...
Per finalitzar, introdueixi una línia buida (en blanc)
Per finalitzar el programa, introdueixi instrucció buida
add C:\mondial.xml

Informació final del servidor:
Path "mondial.xml" added in 1447.69 ms.

Introdueixi la instrucció a executar...
Per finalitzar, introdueixi una línia buida (en blanc)
Per finalitzar el programa, introdueixi instrucció buida
close

Informació final del servidor:
Database 'ioc' was closed.

Introdueixi la instrucció a executar...
Per finalitzar, introdueixi una línia buida (en blanc)
Per finalitzar el programa, introdueixi instrucció buida
```

Podem comprovar, via BaseX GUI, l'existència de la nova base de dades amb l'arxiu mondial.xml com a contingut.

1.3.2 API Java de l'SGBD Sedna

Als Annexos del web trobareu l'annex "Introducció a l'SGBD-XML natives Sedna" amb les indicacions per instal·lar aquest SGBD i tenir-hi una primera presa de contacte.

L'SGBD-XML natives Sedna proporciona un protocol client/servidor que permet escriure clients en diversos llenguatges de programació. El desenvolupament de clients per accedir a Sedna sobrepassa l'objectiu d'aquest apartat, on simplement volem utilitzar el client Java que se'ns facilita.

Centrem-nos, doncs, en el client Java que facilita la versió 3.5 de Sedna, que resideix en el paquet `ru.ispras.sedna.driver`, que trobareu en el fitxer `sednadriver.jar` ubicat a la carpeta `driver\java\lib` del directori on hagueu instal·lat Sedna. Així mateix, la documentació d'aquest paquet es troba a la carpeta `driver\java\javadoci` només cal obrir l'arxiu `index.html` per accedir-hi des de qualsevol navegador. I si teniu curiositat per conèixer el codi font del paquet, també el teniu disponible a la carpeta `driver\java\src`.

Per executar qualsevol programa que utilitzi el paquet `ru.ispras.sedna.driver`, haureu de tenir el fitxer `sednadriver.jar` inclòs en el `CLASSPATH` actiu.

Observarem que el paquet `ru.ispras.sedna.driver` proporciona un reduït conjunt de classes i interfícies. Per poder utilitzar un client amb un SGBD ens cal saber com establir i gestionar connexions i com executar-hi instruccions de consulta i actualització.

En primer lloc ens cal establir connexions amb servidors Sedna. Per això disposem del mètode `getConnection(...)` de la classe `DatabaseManager`:

```
1 static SednaConnection getConnection (
2     java.lang.String urlString,
3     java.lang.String dbName,
4     java.lang.String login,
5     java.lang.String password)
6     throws DriverException
```

El paràmetre `urlString` ha de contenir el nom o la IP de la màquina que conté el servidor Sedna al qual ens volem connectar. Pot contenir el port pel qual està escoltant el servidor Sedna (en format `ip:port` o `nom:port`). En cas de no indicar cap port, s'intenta establir la connexió pel port 5050 (port utilitzat normalment per Sedna).

La resta de paràmetres són obvis: `dbName` per indicar el nom de la base de dades a la qual ens volem connectar, `login` per indicar el nom de l'usuari i `password` per indicar la seva contrasenya.

Sedna suporta la funció estàndard `trace()` del llenguatge XQuery, que permet efectuar un seguiment dels valors que el motor XQuery va processant. El client Java de Sedna mostra, per la sortida estàndard, els missatges que la funció `trace` genera en els punts on s'hagi incorporat en el codi. Si es vol mantenir la funció `trace` en el codi, però desactivar la seva funcionalitat, el client Java de Sedna ens proporciona el mètode `setTraceOutput()` de la interfície `SednaConnection`.

Perquè la connexió s'estableixi sense problemes, cal que el servidor Sedna estigui engegat i escoltant pel port pel qual estem intentant establir la connexió, i que la base de dades a la qual ens volem connectar estigui també engegada. El client Java de Sedna que estem estudiant no proporciona cap mecanisme d'engegada i/o aturada del servidor ni de les bases de dades existents en el servidor, i tampoc permet crear-hi ni eliminar-hi bases de dades.

Una vegada ja tenim una connexió establerta, disposarem d'un objecte `SednaConnection` que ens permetrà anar executant les diverses accions sobre la base de dades.

A nivell de connexió (interfície `SednaConnection`), ens cal saber els diversos mètodes de què disposem:

- Mètode `isClose()`, per saber si la connexió s'ha tancat o es manté oberta.
- Mètode `close()`, per tancar la connexió.
- Mètode `begin()` per obrir una transacció. És obligatori tenir una transacció oberta per poder executar consultes i actualitzacions sobre la base de dades.
- Mètode `rollback()` per tancar una transacció sense enregistrar cap dels canvis que s'hagin pogut efectuar durant la transacció.
- Mètode `commit()` per tancar una transacció enregistrant tots els canvis que s'hagin pogut efectuar durant la transacció.
- Mètode `createStatement()` per crear un objecte `SednaStatement` que ens permet executar consultes i actualitzacions sobre la base de dades.
- Mètode `setTraceOutput()` per activar o desactivar el funcionament de la funció XQuery estàndard `trace()`.
- Mètode `setDebugMode()` per activar/desactivar el mode depuració.

Sedna facilita el mètode getDebugInfo() de la classe DriverException per obtenir informació quan una operació falla, sempre que la sessió tingui activat el mode depuració, cosa que es pot gestionar amb el mètode setDebugMode de la interfície SednaConnection.

- Mètode `setReadonlyMode()` per canviar el mode "només lectura" per a la següent transacció.

Sedna crea les transaccions en mode actualització, a no ser que s'indiqui "només lectura" amb el mètode `setReadonlyMode()` de la interfície SednaConnection. En cas que es tingui la seguretat que en una transacció no s'efectuarà cap instrucció d'actualització de la base de dades, és millor activar el mode "només lectura", ja que es guanyarà efectivitat perquè les operacions no es queden en espera si hi ha altres transaccions que estan modificant les mateixes dades.

La interfície `SednaStatement` ens proporciona diversos mètodes:

- Mètodes `execute()`, per executar consultes XQuery que produeixen un resultat i altres instruccions (actualitzacions, creació/eliminació de col·leccions, càrrega/eliminació de documents). El propi mètode ens retorna un valor booleà que ens indica el tipus d'instrucció executada:
  - `true`, indicador que s'ha executat una consulta i que podrem obtenir els resultats amb el mètode `getSerializedResult()`.
  - `false`, indicador que s'ha executat una instrucció que no és una consulta; en aquest cas, obtindrem un error si intentem executar el mètode `getSerializedResult()`.
- Mètode `getSerializedResult()`, per obtenir el resultat (objecte `SednaSerializedResult`) del darrer mètode `execute()` sempre i quan correspongui a una consulta.
- Mètodes `loadDocument()`, per carregar documents a la base de dades.

Els programes Java que executen consultes sobre bases de dades XML han de poder rebre els resultats de les consultes per tal d'efectuar-ne la gestió que correspongui. En el cas dels resultats facilitats pel servidor Sedna, haurem d'utilitzar els mètodes de la interfície `SednaSerializedResult`. Aquesta interfície únicament proporciona dos mètodes `next()`, que permeten anar iterant sobre l'objecte `SednaSerializedResult` obtingut amb el mètode `SednaStatement.getSerializedResult()`.

En els exemples de consultes que segueixen, atès que interessa visualitzar els resultats per la consola, acostumarem a utilitzar el següent mètode:

```
/* Mètode per mostrar resultats per la sortida estàndard
 * Retorna el nombre d'elements de la consulta
 */
  private static int mostrarResultats(SednaStatement st)
    throws DriverException {
    int comptador = 0;
    String item;
    System.out.println("\nResultats:\n");
    SednaSerializedResult pr = st.getSerializedResult();
    while ((item = pr.next()) != null) {
      comptador++;
      System.out.print(item);
      // No utilitzem "println" perquè cada "item", a partir del 2n,
      // comença amb "\n" i això ja provoca el salt de línia
    }
    System.out.println();
    return comptador;
  }
}
```

Amb tota aquesta informació ja estem en condicions de desenvolupar alguns programes Java que interactuïn amb les bases de dades d'un servidor Sedna. Els exemples següents estan basats en el servidor Sedna instal·lat seguint les indicacions del material web.

En el servidor Sedna instal·lat seguint les instruccions de l'annex"Introducció a l'SGBD-XML natives Sedna" dels Annexos del web, hi tenim instal·lada una base de dades de nom `mondial`.Hi basarem els exemples desenvolupats en aquest apartat.

Exemple de connexió a un servidor Sedna per executar-hi una consulta senzilla.

El programa `SE_Client01.java` visualitza els noms dels continents emmagatzemats a la base de dades mundial.

L'execució d'aquest programa obté:

```
G:\>java –Dfile.encoding=cp850 proves.SE_Client01
Executant consulta: doc('mondial.xml')//continent/name/text()
Resultats:

Europe
Asia
Australia/Oceania
Africa
America

S'han obtingut 5 resultats.
```

Trobareu el fitxer `SE_Client01.java` dins el paquet de codi font a l'annex "Material per a desenvolupament sobre SGBD-XML natives Sedna" dels Annexos del web.

Exemple de connexió a un servidor Sedna per executar-hi una consulta FLWOR i comprovar el funcionament de la funció trace().

El programa `SE_Client02.java` visualitza els noms dels països existents a la base de dades `mondial`, acompanyats de la seva població. Aprofitem aquest exemple per mostrar el funcionament de la funció `trace()` del llenguatge XQuery i la possibilitat d'activar-la o desactivar-la des d'un programa client Java.

Trobareu el fitxer `SE_Client02.java` dins el paquet de codi font a l'annex "Material per a desenvolupament sobre SGBD-XML natives Sedna" dels Annexos del web.

L'execució d'aquest programa obté:

```
G:\>java -Dfile.encoding=cp850 proves.SE_Client02 traceOff
Executant consulta:
for $i in trace(doc("mondial.xml")//country,'###')
return concat($i/name[1],"-",$i/population)
Resultats:

Albania-3249136
Greece-10538594
Macedonia-2104035
Serbia-7379339
Montenegro-672180
Kosovo-1804838
...
Sao Tome and Principe-144128
Seychelles-77575

S'han obtingut 238 resultats.
```

En aquest cas la funció trace() que forma part de la consulta XQuery no s'ha manifestat en el resultat, perquè l'execució s'ha indicat amb traceOff.

Si repetim l'execució amb traceOn, obtenim un resultat similar al següent:

```
G:\>java -Dfile.encoding=cp850 proves.SE_Client02 traceOn
Executant consulta:
for $i in trace(doc("mondial.xml")//country,'###')
return concat($i/name[1],"-",$i/population)
Resultats:

### <country car_code="AL" area="28750"
...
</country>
Albania-3249136### <country car_code="GR" area="131940"
...
</country>
Greece-10538594### <country car_code="MK" area="25333"
...
</country>
Macedonia-2104035### <country ...
...
...
</country>
Seychelles-77575

S'han obtingut 238 resultats.
```

Fixem-nos que com la funció `trace()` està aplicada sobre la consulta `doc("mondial.xml")//country`, amb prefix `###`, veiem per la consola el contingut de cada element country recuperat pel mètode `SednaSerializedResult.next()`, precedit pel prefix `###`, abans que puguem fer res amb aquest element. La utilització d'aquesta funció pot ajudar a depurar les consultes, tot i que és més fàcil fer la depuració executant la consulta directament en una terminal de Sedna o des de la GUI SednaAdmin que no pas des de dins d'un programa Java.

Trobareu el fitxer `SE_Client03.java` dins el paquet de codi font a l'annex "Material per a desenvolupament sobre SGBD-XML natives Sedna" dels Annexos del web.

Exemple de connexió a un servidor Sedna per executar-hi qualsevol instrucció (consulta i gestió de documents i col·leccions) a introduir per l'usuari

El programa `SE_Client03.java` facilita la possibilitat que l'usuari introdueixi qualsevol instrucció admesa per Sedna (consultes, actualitzacions, càrrega i eliminació de documents, creació i eliminació de col·leccions...) sobre la base de dades mondial.

Comprovem el funcionament d'aquest programa davant diverses instruccions a introduir per l'usuari. L'execució en qualsevol cas és:

```
G:\>java –Dfile.encoding=cp850 proves.SE_Client03
Introdueixi la instrucció a executar...
Per finalitzar, introdueixi una línia buida (en blanc):
```

Vegem quina és la resposta davant les següents instruccions:

Consulta simple (una sola línia)

```
doc('mondial.xml')//continent/name

Executant instrucció:
doc('mondial.xml')//continent/name

Resultats:

<name>Europe</name>
<name>Asia</name>
<name>Australia/Oceania</name>
<name>Africa</name>
<name>America</name>

S'han obtingut 5 resultats.
```

Consulta complexa (instrucció FLWOR)

```
for $i in doc('mondial.xml')//continent
return $i/name

Executant instrucció:
for $i in doc('mondial.xml')//continent
return $i/name

Resultats:

<name>Europe</name>
<name>Asia</name>
<name>Australia/Oceania</name>
<name>Africa</name>
<name>America</name>

S'han obtingut 5 resultats.
```

Càrrega del document ”G:\DOCENCIA\books.xml” amb nom ”llibres.xml” dins l'arrel de la base de dades

```
LOAD 'G:\DOCENCIA\books.xml' 'llibres.xml'

Executant instrucció:
LOAD 'G:\DOCENCIA\books.xml' 'llibres.xml'

Instrucció executada
```

Podem comprovar, via SednaAdmin, que la càrrega s'ha dut a terme.

Intent de càrrega del document ”G:\DOCENCIA\books.xml” amb nom ”llibres.xml” (ja existent) dins l'arrel de la base de dades

```
LOAD 'G:\DOCENCIA\books.xml' 'llibres.xml'

Executant instrucció:
LOAD 'G:\DOCENCIA\books.xml' 'llibres.xml'

ru.ispras.sedna.driver.DriverException: SEDNA Message: ERROR
    SE2001
Document with the same name already exists.
```

```
Details: llibres

        at ru.ispras.sedna.driver.NetOps.bulkLoad(NetOps.java
            :202)
        at ru.ispras.sedna.driver.SednaStatementImpl.
            executeResponseAnalyze(SednaStatementImpl.java:155)
        at ru.ispras.sedna.driver.SednaStatementImpl.execute(
            SednaStatementImpl.java:114)
        at ru.ispras.sedna.driver.SednaStatementImpl.execute(
            SednaStatementImpl.java:38)
        at proves.SE_Client03.main(SE_Client03.java:50)
```

Crear una col·lecció anomenada xxx

```
create collection 'xxx'

Executant instrucció:
create collection 'xxx'

Instrucció executada
```

Sedna facilita el document de sistema `$collections` amb la llista de totes les col·leccions existents a la base de dades.

Podem comprovar, via SednaAdmin, que la creació s'ha dut a terme.

Enumerar les col·leccions existents a la base de dades

```
doc('$collections')

Executant instrucció:
doc('$collections')

Resultats:

<collections>
<collection name="$modules"/>
<collection name="xxx"/>
</collections>

S'ha obtingut 1 resultats.
```

O també:

```
doc('$collections')//collection

Executant instrucció:
doc('$collections')//collection

Resultats:

<collection name="$modules"/>
<collection name="xxx"/>

S'han obtingut 2 resultats.
```

O també:

```
doc('$collections')//collection/@name

Executant instrucció:
doc('$collections')//collection/@name

Resultats:

<name="$modules"/>
<name="xxx"/>

S'han obtingut 2 resultats.
```

Intent de creació de la col·lecció xxx (ja existent)

```
create collection 'xxx'

ru.ispras.sedna.driver.DriverException: SEDNA Message: ERROR
    SE2002
Collection with the same name already exists.
Details: xxx

        at ru.ispras.sedna.driver.SednaStatementImpl.
            executeResponseAnalyze(SednaStatementImpl.java:145)
        at ru.ispras.sedna.driver.SednaStatementImpl.execute(
            SednaStatementImpl.java:114)
        at ru.ispras.sedna.driver.SednaStatementImpl.execute(
            SednaStatementImpl.java:38)
        at proves.SE_Client03.main(SE_Client03.java:50)
```

Sedna no proporciona, en l'actual versió, la possibilitat de reanomenar un document introduït a la base de dades.

Reanomenar la col·lecció xxx a ioc

```
RENAME COLLECTION 'xxx' INTO 'ioc'

Executant instrucció:
RENAME COLLECTION 'xxx' INTO 'ioc'

Instrucció executada
```

Podem comprovar, via SednaAdmin, que el canvi de nom s'ha dut a terme.

Sedna facilita el document de sistema `$documents` amb la llista de totes les col·leccions amb els documents de cada col·lecció i els documents ubicats a l'arrel de la base de dades.

Llistat de tots els documents existents a la base de dades

```
doc('$documents')

Executant instrucció:
doc('$documents')

Resultats:

<documents>
  <document name="$db_security_data"/>
  <collection name="$modules"/>
  <collection name="ioc">
    <document name="mondial.xml"/>
  </collection>
  <document name="llibres.xml"/>
  <document name="mondial.xml"/>
</documents>

S'ha obtingut 1 resultats.
```

Càrrega del document mondial (el mateix que en el procés d'instal·lació i configuració de Sedna es va introduir a la base de dades) dins la col·lecció de nom ioc, amb nom mondial.xml

```
LOAD '...\mondial.xml' 'mondial.xml' 'ioc'

Executant instrucció:
LOAD '...\mondial.xml' 'mondial.xml' 'ioc'

Instrucció executada
```

Podem comprovar, via SednaAdmin, que la càrrega s'ha dut a terme.

Obtenir els documents de la col·lecció ioc

```
doc('$documents')//collection[@name='ioc']/document

Executant instrucció:
```

```
4 doc('$documents')//collection[@name='ioc']/document
5
6 Resultats:
7
8 <document name="mondial.xml"/>
9
10 S'ha obtingut 1 resultats.
```

Eliminar el document llibres.xml de l'arrel de la base de dades

```
1 DROP DOCUMENT 'llibres.xml'
2
3 Executant instrucció:
4 DROP DOCUMENT 'llibres.xml'
5
6 Instrucció executada
```

Podem comprovar, via SednaAdmin, que l'eliminació s'ha dut a terme.

Eliminar el document mondial.xml de la col·lecció ioc

```
1 DROP DOCUMENT 'mondial.xml' IN COLLECTION 'ioc'
2
3 Executant instrucció:
4 DROP DOCUMENT 'mondial.xml' IN COLLECTION 'ioc'
5
6 Instrucció executada
```

Podem comprovar, via SednaAdmin, que l'eliminació s'ha dut a terme.

L'ordre `drop collection`''elimina una col·lecció malgrat que tingui documents, que també són eliminats.

Eliminar la col·lecció ioc

```
1 DROP COLLECTION 'ioc'
2
3 Executant instrucció:
4 DROP COLLECTION 'ioc'
5
6 Instrucció executada
```

Podem comprovar, via SednaAdmin, que l'eliminació s'ha dut a terme.

Inserir nous nodes (seguint sintaxi Update de Sedna)

```
1 UPDATE insert <institut nom="IOC">IOC</institut>
2 following doc("mondial.xml")//continent[name="Asia"]
3
4 Executant instrucció:
5 UPDATE insert <institut nom="IOC">IOC</institut>
6 following doc("mondial.xml")//continent[name="Asia"]
7
8 Instrucció executada
```

Podem comprovar, via SednaAdmin, que la inserció s'ha dut a terme.

Canviar el valor de nodes (seguint sintaxi Update de Sedna, on cal canviar tot el node)

```
1 UPDATE replace $n in doc("mondial.xml")//institut[@nom="IOC"]
2 with
3 <institut nom="IOC">Institut Obert de Catalunya</institut>
4
5 Executant instrucció:
6 UPDATE replace $n in doc("mondial.xml")//institut[@nom="IOC"]
7 with
8 <institut nom="IOC">Institut Obert de Catalunya</institut>
9
10 Instrucció executada
```

Podem comprovar, via SednaAdmin, que la substitució s'ha dut a terme.

Canviar nodes (seguint sintaxi Update de Sedna)

```
UPDATE replace $n in doc("mondial.xml")//institut[@nom="IOC"]
with
<cicle>
  <codi>DAM</codi>
  <nom>Desenvolupament d'aplicacions multiplataforma</nom>
</cicle>

Executant instrucció:
UPDATE replace $n in doc("mondial.xml")//institut[@nom="IOC"]
with
<cicle>
  <codi>DAM</codi>
  <nom>Desenvolupament d'aplicacions multiplataforma</nom>
</cicle>

Instrucció executada
```

Podem comprovar, via SednaAdmin, que la substitució s'ha dut a terme.

Reanomenar nodes (seguint sintaxi Update de Sedna)

```
UPDATE rename doc("mondial.xml")//cicle on cf

Executant instrucció:
UPDATE rename doc("mondial.xml")//cicle on cf

Instrucció executada
```

Podem comprovar, via SednaAdmin, que el canvi de nom s'ha dut a terme.

Eliminar nodes (seguint sintaxi Update de Sedna)

```
UPDATE delete doc("mondial.xml")//cf

Executant instrucció:
UPDATE delete doc("mondial.xml")//cf

Instrucció executada
```

Podem comprovar, via SednaAdmin, que l'eliminació s'ha dut a terme.

2. API Java estàndards per a BD-XML natives

Les dades emmagatzemades a les BD han de poder ser accessibles des d'aplicacions desenvolupades en diferents llenguatges i, per aquest motiu, els SGBD es veuen obligats a facilitar interfícies de programació per als llenguatges de programació més emprats. Així, els fabricants d'SGBD, coneixedors de la tecnologia emprada en el seu producte, faciliten una API per permetre-hi l'accés, desenvolupada de la manera més eficient possible per al seu producte, fet que comporta l'aparició d'un problema: l'API proporcionada per a cada SGBD és pròpia i diferent de les API dels altres SGBD i, en conseqüència, les aplicacions desenvolupades queden lligades a l'SGBD i els programadors han de conèixer un munt d'API diferents, tantes com nombre d'SGBD diferents hagin d'enllaçar.

Davant l'anarquia d'API existent per atacar els SGBD d'un determinat tipus, acostumen a aparèixer intents per estandarditzar el mecanisme i proporcionar una API estàndard. En el cas de les BD-XML natives hi ha hagut dos processos d'estandardització per al llenguatge Java, que han donat lloc a dues API estàndards: XML:DB (també anomenada XAPI) i XQueryAPI for Java (XQJ).

Actualment hi ha molts SGBD-XML natives que faciliten la implementació de les dues API. Sembla, però, que l'API XQJ és la que està destinada a evolucionar, atès que està promoguda pel W3C, mentre que l'API XML:DB, anterior en el temps, està aturada des del 2003. Per tant, es pot considerar ja obsoleta.

2.1 API XQJ

XQuery API for Java (**XQJ**) és una interfície de programació d'aplicacions Java pensada per utilitzar el llenguatge XQuery per obtenir informació de BD-XML natives, de manera semblant a com JDBC és una API pensada per utilitzar el llenguatge SQL per accedir a BDR.

XQJ va néixer el 2003 i la seva versió definitiva ha estat publicada el 2009. També és coneguda com a JSR 225 ja que ha estat dissenyada com un projecte JCP (Java Community Process). De moment (versió 1.7 de Java), no forma part de la llibreria estàndard de classes Java.

Nombrosos SGBD-XML natives faciliten la connectivitat des de Java mitjançant aquesta XQJ, de manera que podem desenvolupar aplicacions que accedeixin a SGBD-XML natives via XQJ amb l'única particularitat d'haver d'utilitzar la implementació de l'API que facilita cada SGBD.

A continuació desenvoluparem aplicacions Java que connecten contra SGBD-XML natives via XQJ i en comprovarem la seva correcta execució en tres SGBD-XML natives: eXist-db, BaseX i Sedna. Per això necessitem tenir instal·lats els tres SGBD i disposar de les implementacions XQJ per a cada SGBD.

Als Annexos del web trobareu els apartats "Introducció a l'SGBD-XML natives" per als SGBD eXist-db, BaseX i Sedna, amb les indicacions per instal·lar aquests SGBD i tenir-hi una primera presa de contacte.

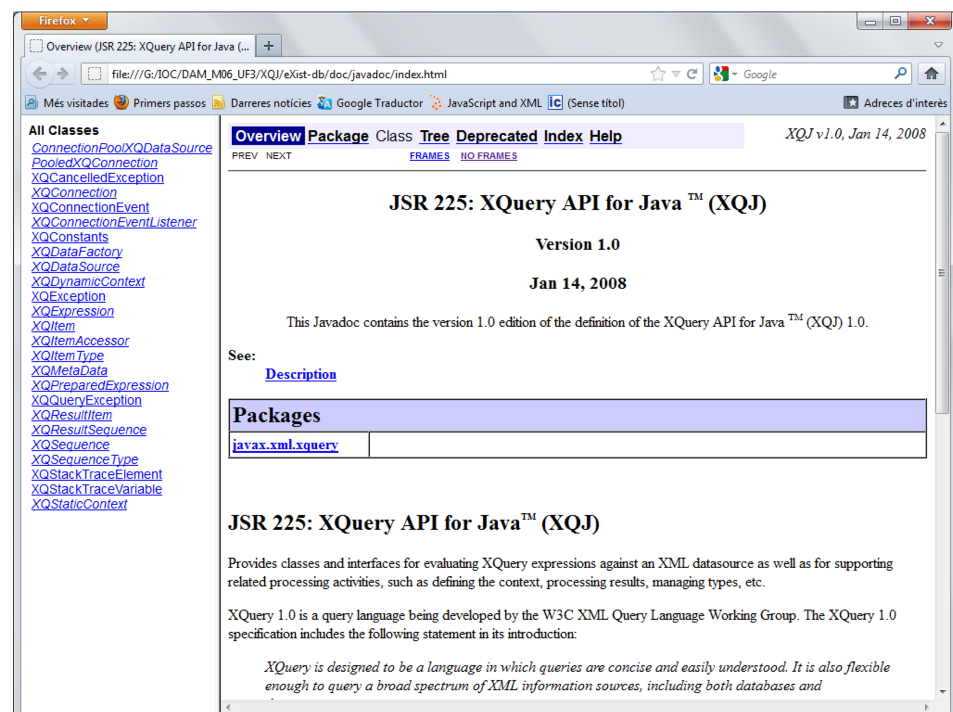
Als Annexos del web trobareu els apartats "Material per a desenvolupament sobre SGBD-XML natives utilitzant l'API XQJ" on hi ha una llibreria XQJ, desenvolupada per Charles Foster, per als SGBD eXist-db i Sedna.

Si fem una ullada a la documentació (`javadoc` de la figura 2.1) de l'API XQJ, veurem que està constituïda per un gran nombre d'interfícies i unes poques classes. Això és així perquè cada SGBD ha de proveir les classes que implementen les interfícies que dictamina l'API. En el desenvolupament d'aplicacions amb aquesta API utilitzarem sempre referències a aquestes interfícies i mai utilitzarem directament referències a les classes subministrades per l'SGBD. D'aquesta manera aconseguirem dissenyar aplicacions que es puguin utilitzar per gestionar BD en diversos SGBD que implementen l'API XQJ.

Llibreria XQJ per a l'SGBD BaseX

L'SGBD BaseX (versió 7.1) porta inclosa una llibreria XQJ, que presenta algunes anomalies. En el moment de confeccionar aquests materials els responsables de BaseX han informat que Charles Foster està desenvolupant una llibreria XQJ per a BaseX, semblant a les desenvolupades per a eXist-db i Sedna. Cal, doncs, estar atents a l'aparició d'aquesta llibreria per utilitzar-la en lloc de la que incorpora BaseX.

Figura 2.1. Documentació de l'API XQJ



Així doncs, ens interessa conèixer les interfícies que aporta l'API XQJ i emprar-les en el disseny d'aplicacions que accedeixin als SGBD-XML natives (concretament: eXist-db, BaseX i Sedna).

La documentació (`javadoc`) de l'API XQJ es pot trobar en els arxius `zip` corresponents a l'API XQJ per a eXist-db i Sedna i també directament a la web del projecte JSR 225.

2.1.1 Establiment de connexió

Les aplicacions que accedeixen a BD necessiten, com a primer pas per poder gestionar les dades de la BD, establir la connexió amb la BD a gestionar. L'API XQJ facilita dues interfícies apropiades per aconseguir aquest objectiu:

1. `XQDataSource`, fàbrica per obtenir objectes `XQConnection`

2. `XQConnection`, per referenciar connexions (sessions) amb un SGBD específic. Tota connexió s'assoleix, forçosament, a través d'un objecte `XQDataSource`

Així, per obtenir una connexió, cal seguir l'esquema següent:

```
XQDataSource xqs = new ClasseEspecíficaQueGeneraXQDataSource();
XQConnection con = xqs.getConnection(llistaParàmetres);
```

La primera línia del codi anterior mostra com crear un objecte `XQDataSource` específic de l'SGBD al qual ens volem connectar. La segona línia mostra com establir la connexió i, com a llista de paràmetres, haurem d'incloure els adequats a l'SGBD.

Per cada SGBD-XML natives amb què vulguem connectar via XQJ, ens cal saber la classe específica facilitada per l'SGBD que implementa la interfície `XQDataSource`. La taula 2.1 recull les classes adequades per als SGBD amb els quals practiquem.

Taula 2.1. Classes que implementen la interfície XQDataSource en diversos SGBD.

SGBD	Classe
eXist-db	`net.xqj.exist.ExistXQDataSource`
BaseX	`org.basex.api.xqj.BXQDataSource`
Sedna	`net.xqj.sedna.SednaXQDataSource`

Llibreria XQJ per a l'SGBD BaseX

La classe de BaseX indicada a la taula 2.1 és la que porta integrada BaseX (versió 7.1) i que presenta algunes anomalies. En el moment de confeccionar aquests materials, els responsables de BaseX han informat que Charles Foster està desenvolupant una llibreria XQJ per a BaseX, similar a les desenvolupades per eXist-db i Sedna. Cal, doncs, està atent a l'aparició d'aquesta llibreria per utilitzar-la en lloc de la que incorpora BaseX.

Així doncs, si volem assolir un programa per connectar-nos amb un SGBD eXist-db, caldria escriure quelcom similar al següent:

```
XQDataSource xqs = new net.xqj.exist.ExistXQDataSource();
XQConnection con = xqs.getConnection(llistaParàmetres);
```

El codi anterior és codi específic per a l'SGBD eXist-db, i ens interessa, si és possible, escriure codi que sigui independent, en el major grau possible, de l'SGBD. Per tant, ens interessa obtenir l'objecte `XQDataSource` de forma genèrica, sense cridar cap mètode específic de l'API subministrada per l'SGBD. Per aconseguir-ho escriurem el següent:

```
String driver = "nomClasseEspecíficaQueGeneraXQDataSource";
XQDataSource xqs = (XQDataSource)Class.forName(driver).newInstance();
XQConnection con = xqs.getConnection(llistaParàmetres);
```

En el codi anterior, la cadena `driver` ha de contenir el nom de la classe específica de l'SGBD que genera l'objecte `XQDataSource` (taula 2.1), però l'avantatge respecte la situació anterior és que el contingut de la cadena `driver` no té per què residir en el codi font del programa, sinó que es pot llegir d'un fitxer de configuració o es pot recollir via paràmetre, entre d'altres possibilitats.

Una vegada es disposa de l'objecte `XQDataSource` ja estem en condicions de crear un objecte `XQConnection` per establir la sessió de treball amb l'SGBD i això ho aconseguim amb el mètode `getConnection()` de la interfície `XQDataSource`. L'API XQJ facilita tres sobrecàrregues del mètode `getConnection()`:

```
XQConnection getConnection(java.sql.Connection con)
                           throws XQException;
XQConnection getConnection() throws XQException;
XQConnection getConnection(java.lang.String username,
                           java.lang.String passwd)
                           throws XQException;
```

Les tres modalitats intenten establir una connexió contra l'SGBD de diferents maneres:

1. La primera, utilitzant una connexió JDBC ja existent.
2. La segona, sense aportar cap informació (paràmetre).
3. La tercera opció, indicant l'usuari i la contrasenya (paràmetres).

Si hem treballat amb SGBD relacionals estem acostumats a indicar, quan volem establir la connexió, entre d'altres paràmetres: la màquina (IP o nom), el port, la base de dades, l'usuari i la contrasenya. Observant les modalitats anteriors, estarem d'acord que les opcions segona i tercera semblen incompletes, ja que no indiquem alguns dels paràmetres claus: màquina i port on resideix l'SGBD i nom de la base de dades. A la segona opció, fins i tot, no s'indiquen usuari i contrasenya.

No desesperem! La solució passa per saber que la interfície `XQDataSource` facilita el mètode `setProperty()`, que ens permet definir un seguit de propietats abans d'intentar establir la connexió amb el mètode `getConnection()`:

```
void setProperty(java.lang.String name,
                 java.lang.String value)
                 throws XQException;
```

Entre les propietats que permet establir el mètode `setProperty()` hi trobem el nom de la màquina, el port, el nom de la base de dades, l'usuari i la contrasenya. No tots els SGBD-XML tenen les mateixes propietats o utilitzen idèntics noms per a una mateixa propietat. Ens cal saber, doncs, les propietats que cada SGBD admet per poder-les assignar amb el mètode `setProperty()` abans d'establir la connexió amb el mètode `getConnection()` i per aconseguir-ho, la interfície `XQDataSource` facilita el mètode següent:

```
java.lang.String[] getSupportedPropertyNames();
```

El programa `XQJ01.java` ens serveix per detectar les propietats suportades pels SGBD-XML eXist-db, BaseX i Sedna. Observeu, amb atenció, el requeriment per a la correcta execució, incorporat a la capçalera del codi font.

Trobareu el fitxer `XQJ01.java` dins el paquet de codi font a l'apartat "Material per a desenvolupament sobre SGBD-XML natives utilitzant l'API XQJ" dels annexos del web.

L'execució del programa ens facilita la següent informació:

```
G:\>java -Dfile.encoding=cp850 proves.XQJ01
Propietats de XQDataSource en el SGBD eXist-db
        description
        logLevel
        loginTimeout
```

```
6          serverName
7          port
8          user
9          password
10 Propietats de XQDataSource en el SGBD BaseX
11         user
12         password
13 Propietats de XQDataSource en el SGBD Sedna
14         description
15         logLevel
16         loginTimeout
17         serverName
18         databaseName
19         port
20         user
21         password
22         description
```

Analitzem les propietats suportades per l'objecte`XQDataSource` subministrat per cadascun dels tres SGBD indicats.

Comencem per l'SGBD BaseX, ja que únicament facilita les propietats usuari i contrasenya. Com és que no ens permet indicar la màquina, el port i el nom de la base de dades? Doncs perquè l'API XQJ subministrada per BaseX (versió 7.1) no permet la connexió client/servidor i, en conseqüència, només permet la connexió contra el servidor BaseX local, motiu pel qual no es necessita indicar la màquina ni el port.

Fixem-nos que BaseX tampoc ens permet indicar la base de dades amb la qual ens connectarem. Recordem que BaseX no implementa el concepte de col·lecció i que en aquest SGBD, els conceptes "base de dades" i "col·lecció" són equivalents.

Les propietats suportades pels SGBD eXist-db i Sedna són molt més semblants. En ambdós hi veiem que podem indicar la màquina (IP o nom), el port, l'usuari i la contrasenya. Sedna ens permet, també, indicar el nom de la base de dades. L'SGBD eXist-db permet gestionar connexions però hi ha una única base de dades i, per tant, les propietats per establir connexió no permeten indicar-ne la base de dades. Dels tres SGBD emprats en aquest material, Sedna és l´únic que permet gestionar diverses bases de dades amb diverses col·leccions en cadascuna d'elles.

Una vegada conegudes les propietats suportades pels `XQDataSource` dels diversos SGBD, ja estem en condicions d'intentar establir-hi connexió, emprant el mètode `XQDataSource.getConnection()`.

Aquest mètode retorna, si tot és correcte, un objecte que implementa la interfície `XQConnection`, el qual utilitzarem per executar qualsevol acció XQuery o Update sobre la BD. En finalitzar el programa cal recordar sempre de tancar la connexió amb el mètode `XQConnection.close()`, per tal d'alliberar tots els recursos assignats pel sistema operatiu per mantenir la connexió.

La interfície `XQConnection` facilita mètodes per crear expressions sobre les quals executarem sentències XQuery/Update, i els mètodes per validar (`commit()`) o retrocedir (`rollback()`) els canvis duts a terme durant una transacció.

Per defecte, una connexió opera en mode `auto-commit`, fet que significa que cada instrucció Update és executada i validada en una transacció individual, Aquesta forma de treballar es pot desactivar amb el mètode `XQConnection.setAutoCommit()`, i en tal situació, la transacció finalitzarà efectuant una crida al mètode `commit()` o `rollback()`, donant lloc a l'inici d'una nova transacció. No hi ha, doncs, una instrucció específica per indicar l'inici de transacció.

El programa `XQJ02.java` mostra com establir connexió amb cadascun dels tres SGBD indicats. Cal tenir present que aquest programa:

- Obliga a passar per paràmetre el nom de l'SGBD.
- Considera que l'usuari i la contrasenya per a cada SGBD són els que el procés d'instal·lació facilita per defecte (`admin/admin` per a eXist-db i BaseX i `SYSTEM/MANAGER` per a Sedna).
- Per als SGBD eXist-db i Sedna, que són els SGBD que suporten l'arquitectura client/servidor, es considera que el servidor resideix a la mateixa màquina des d'on s'executarà el programa (`localhost`) i que el port pel qual s'estableix la comunicació TCP/IP és el que el procés d'instal·lació facilita per defecte (`8080` per a eXist-db i `5050` per a Sedna).
- Per a l'SGBD Sedna, intenta establir connexió amb una BD de nom `mondial`. Si no es disposa d'aquesta BD, cal indicar el nom de BD que correspongui.

Trobareu el fitxer `XQJ02.java` dins el paquet de codi font a l'apartat "Material per a desenvolupament sobre SGBD-XML natives utilitzant l'API XQJ" dels Annexos del web.

L'execució del programa finalitza amb error (convenientment documentat) si:

- No s'indica, a l'hora d'executar el programa, cap SGBD.
- S'indica, a l'hora d'executar el programa, un nom d'SGBD no suportat.
- No es troba, en el `CLASSPATH` actiu, la llibreria XQJ corresponent a l'SGBD-XML natives amb el qual es vol connectar.
- En el cas de l'SGBD Sedna, el servidor està aturat o la base de dades indicada està aturada, o el port és erroni.
- En el cas dels SGBD BaseX i Sedna, l'usuari o la contrasenya són erronis.

El programa anterior no es queixa, per a l'SGBD eXist-db, si el servidor és aturat o el port o l'usuari o la contrasenya són erronis, i el corresponent error es posposa al moment en què s'intenta executar una instrucció XQuery o Update a través de la connexió. Aquest funcionament és causat pel fet que l'SGBD eXist-db treballa amb connexions no persistents.

Així doncs, haurem de tenir present que a eXist-db, el fet que el mètode `XQDataSource.getConnection()` no llenci una excepció, no és garantia que la connexió s'hagi establert correctament.

Observem, també, que l'execució sobre BaseX (versió 7.1) ha de ser en local i no precisa que el servidor BaseX estigui engegat, atès que no utilitza l'arquitectura client/servidor.

2.1.2 Sentències XQuery d'execució immediata

Tots els SGBD acostumen a facilitar dos mecanismes d'execució de sentències contra la BD, utilitzant el llenguatge que correspongui (SQL en SGBDR i SGBDOR, OQL/OML en SGBDOO i XQuery/Update en SGBD-XML):

1. Execució immediata d'una sentència, utilitzada quan la sentència s'ha d'executar una única vegada.
2. Execució de sentències paramètriques o preparades, utilitzada quan la mateixa sentència (o sentència similar amb alguns valors diferents) s'ha d'executar repetidament i consisteix a escriure la sentència com una plantilla que conté algunes variables que s'aniran substituint a cada execució.

L'API XQJ facilita els dos mecanismes a través de les interfícies:

- `XQExpression`, per a l'execució immediata de sentències.
- `XQPreparedExpression`, per a l'execució de sentències paramètriques.

Centrem-nos, en aquest moment, en l'execució immediata de sentències proporcionada per la interfície `XQExpression`.

Per executar una sentència de manera immediata, crearem un objecte `XQExpression` a partir del mètode `XQConnection.createExpression()`. Disposem de dues sobrecàrregues d'aquest mètode:

```
1 XQExpression createExpression() throws XQException;
2 XQExpression createExpression(XQStaticContext properties)
3                                throws XQException;
```

En ambdues sobrecàrregues s'obté un objecte `XQExpression` que podrem utilitzar per executar sentències XQuery/Update de manera immediata. La segona sobrecàrrega permet indicar les propietats del context estàtic que es tindran en compte en avaluar l'expressió, mentre que la primera sobrecàrrega pren com a context estàtic l'associat a la connexió.

La interfície `XQStaticContext` proveeix un conjunt de mètodes `get` i `set` per recuperar i establir les propietats d'un context estàtic. La interfície `XQConnection` facilita els mètodes `getStaticContext()` i `setStaticContext()` per recuperar i establir el context estàtic de la connexió.

Context d'una expressió XPath/XQuery/Update

El context d'una expressió és tota aquella informació que pot incidir en el resultat de l'avaluació de la mateixa i pot ser de dos tipus: estàtic i dinàmic.

El context estàtic està constituït per tota la informació disponible durant l'anàlisi estàtic, és a dir, abans de l'execució de l'expressió, com per exemple l'abast de les variables que apareixen dins l'expressió. El context dinàmic està constituït per tota la informació disponible quan s'està executant l'expressió, per exemple, l'element que s'està avaluant actualment, els valors actuals de les variables, la data i hora actuals, etc. Es pot establir una analogia dels conceptes "context estàtic i dinàmic" en les expressions XPath amb els conceptes "temps de compilació i d'execució" en els llenguatges de programació.

Una vegada tinguem l'objecte `XQExpression`, podrem a través d'ell executar consultes i altres ordres de manera immediata. Recordem que a XQuery cal distingir, com a SQL, les sentències "consulta" que poden retornar un conjunt de resultats que caldrà processar, de les sentències "no consulta" que permeten executar una ordre (inserció, eliminació o actualització i fins i tot ordres específiques del'SGBD), de la qual se'ns informa, com a molt, de l'èxit o fracàs de la seva execució. Per aquest motiu la interfície `XQExpression` distingeix els mètodes `executeQuery` per a les "consultes" dels mètodes `executeCommand` per a les "no consultes":

```
1 void executeCommand(java.lang.String cmd)throws XQException;
2 void executeCommand(java.io.Reader cmd) throws XQException;
3 XQResultSequence executeQuery(java.lang.String query)
4                               throws XQException;
5 XQResultSequence executeQuery(java.io.Reader query)
6                               throws XQException;
7 XQResultSequence executeQuery(java.io.InputStream query)
8                               throws XQException;
```

A més dels mètodes anteriors, disposem del mètode `close()` per tancar l'expressió, alliberant tots els recursos associats quan ja no sigui necessària. L'execució del mètode `close()` sobre la connexió tanca totes les expressions definides sobre ella. També disposem del mètode `isClosed()` per poder esbrinar si una expressió està oberta o tancada:

```
1 void close() throws XQException;
2 boolean isClosed();
```

Fixem-nos que el mètode `executeQuery()` retorna, si tot va bé, un objecte `XQResultSequence`, interfície que deriva de la interfície `XQSequence`, la qual ens facilita un conjunt de mètodes per avaluar la seqüència de resultats obtinguts amb el mètode `executeQuery()`:

Mireu la documentació de la interfície `XQSequence` per veure una llista completa dels mètodes que facilita.

La interfície `XQResultSequence` representa una seqüència d'elements seguint la definició d'XDM (XQuery 1.0 and XPath 2.0 Data Model, de W3C). Una ullada a la documentació d'aquesta interfície ens mostra que disposem de mètodes per:

- Processar els seus elements: `next()`, `previous()`, `getItem()`, `getPosition()`, `count()`, `first()`, `last()`, `isFirst()`, `isLast()`, `isBeforeFirst()`, `isAfeterLast()` i un conjunt de mètodes get per obtenir el contingut d'un element en el format adequat (`getBoolean()`, `getByte()`, `getDouble()`...).

- Obtenir una versió seriada de la seqüència com un objecte `String` (mètode `getSequenceAsString()`).
- Obtenir una versió seriada de la seqüència (mètode `getSequenceAsStream()`) com un objecte `XMLStreamReader` (interfície de l'API StAX de Java, que permet la iteració, cap endavant, d'un document XML en mode lectura, utilitzant els mètodes `next()` i `hasNext()`).

Segons quina sigui la gestió que haguem d'efectuar, utilitzarem uns o altres mètodes. Així, davant un programa de sentència oberta (en el qual l'usuari pugui introduir la sentència a executar), només podrem pensar en mostrar el resultat a través de la versió seriada cap `String` o via objecte `Transformer` a partir de la seriació cap a `XMLStreamReader`. I davant un programa de sentència tancada (sentència perfectament coneguda en escriure el programa), atès que es coneix la forma de la resposta, podem pensar a efectuar un tractament específic.

Per desenvolupar programes Java que gestionin documents XML és necessari conèixer les API DOM, SAX i StAX facilitades per Java. També és recomanable conèixer l'API JDOM (projecte jdom.org) alternativa a les API facilitades per Java.

A continuació veurem alguns programes de sentència tancada i, finalment, un programa de sentència oberta i els executarem, tots ells, sobre els SGBD eXist-db, Sedna i BaseX. Perquè sigui possible, cal tenir instal·lat:

- A eXist-db, una col·lecció de nom `xqj` amb l'arxiu `mondial.xml`.
- A BaseX, una base de dades de nom `xqj` amb l'arxiu `mondial.xml`.
- A Sedna, una base de dades de nom `db`, amb una col·lecció de nom `xqj` amb l'arxiu `mondial.xml`.

Als Annexos del web trobareu l'annex "Material per a desenvolupament sobre SGBD-XML natives utilitzant l'API XQJ" que inclou l'arxiu `mondial.xml` i les instruccions d'instal·lació en els diversos SGBD per poder executar els programes aquí desenvolupats.

Exemple de programa amb sentència tancada de tipus "consulta" i d'execució immediata

El programa `XQJ03.java` mostra els noms de tots els països de l'arxiu `mondial.xml`.

Trobareu el fitxer `XQJ03.java` dins el paquet de codi font a l'Annex "Material per a desenvolupament sobre SGBD-XML natives utilitzant l'API XQJ" dels Annexos del web.

Comprovem el funcionament d'aquest programa executant-lo per cadascun dels tres SGBD. L'execució, en qualsevol cas, és la següent:

```
G:\>java −Dfile.encoding=cp850 proves.XQJ03 nomSGBD
Connexió establerta amb SGBD ...
Executant instrucció:
...
Resultats:
<name>Albania</name>
<name>Greece</name>
<name>Macedonia</name>
...
<name>Seychelles</name>
```

Podem comprovar que el resultat és idèntic en els tres SGBD i que en tots ells obtenim el nom dels països entre les etiquetes `<name>` i `</name>`. En cas que només vulguem obtenir el nom del país sense les etiquetes, podem pensar en utilitzar la funció `text()` al final de la sentència XQuery:

```
/mondial/country/name/text()
```

Aquesta sintaxi funciona perfectament en el llenguatge XQuery als tres SGBD, però l'API XQJ no la tracta com esperem i els resultats són molt diferents. Per tant, si ens cal obtenir únicament el nom, podem efectuar dos tractaments:

1. Cercar la subcadena existent entre les etiquetes `<name>` i `</name>`.

2. Passar cada `XQItem` de la seqüència `XQResultSequence` cap a `XMLStreamReader` (via mètode `getItemAsStream()`) i llavors efectuar-ne un tractament com a objecte `XMLStreamReader`.

El següent tros de programa (incorporat en el programa `XQJ04.java`) mostra les instruccions a executar per mostrar únicament els noms dels països (sense etiquetes) passant per objectes `XMLStreamReader`:

```
System.out.println("\nResultats:");
while (xqrs.next()) {
  XMLStreamReader xsr = xqrs.getItemAsStream();
  for(; xsr.hasNext(); xsr.next()) tractarNom(xsr);
}
```

on

```
static void tractarNom(XMLStreamReader reader) {
  if (reader.getEventType() ==
      XMLStreamConstants.CHARACTERS)
    System.out.println(reader.getText());
}
```

Comprovem el funcionament d'aquest programa executant-lo per a cadascun dels tres SGBD. L'execució, en qualsevol cas, és:

```
G:\>java –Dfile.encoding=cp850 proves.XQJ04 nomSGBD
Connexió establerta amb SGBD ...
Executant instrucció:
...
Resultats:
Albania
Greece
Macedonia
...
Seychelles
```

Trobareu el fitxer `XQJ04.java` dins el paquet de codi font a l'Annex "Material per a desenvolupament sobre SGBD-XML natives utilitzant l'API XQJ" dels Annexos del web.

Exemple de programa amb sentència tancada de tipus "no consulta" (insert) i d'execució immediata

Trobareu el fitxer `XQJ05.java` dins el paquet de codi font a l'annex "Material per a desenvolupament sobre SGBD-XML natives utilitzant l'API XQJ" dels Annexos del web.

El programa `XQJ05.java` insereix un node `country` a l'inici de tots els països (és a dir, immediatament abans del `country[1]`).

Es tracta d'una instrucció d'actualització de la informació existent en un fitxer XML. Recordem que el llenguatge XQuery és, inicialment, de només consulta i que hi ha diverses extensions Update d'aquest llenguatge:

- L'extensió promoguda per P. Lehti, implementada a eXist-db i Sedna. Les instruccions d'aquesta extensió són interpretades, per l'API XQJ, com a ordres i, per tant, cal executar-les amb el mètode `XQExpression.executeCommand()`.
- L'extensió XQUF promoguda per W3C, implementada a BaseX. Les instruccions d'aquesta extensió són interpretades, per l'API XQJ, com a instruccions XQuery i, en conseqüència, cal executar-les amb el mètode `XQExpression.executeQuery()`.

Sentències FLW per filtratge

En ocasions, pot ser necessari filtrar la informació a actualitzar (`insert`, `update` o `delete`) amb sentències FLW. En aquest cas, caldrà consultar la sintaxi a emprar en cadascuna de les extensions Update existents per a XQuery: documentació de P. Lehti (extensió Update d'eXist-db i Sedna) o documentació W3Q (extensió XQUF de BaseX).

El programa, doncs, actua d'una manera o altra segons l'SGBD al qual es connecta.

Comprovem el funcionament d'aquest programa executant-lo per cadascun dels tres SGBD. L'execució, en qualsevol cas, és la següent:

Trobareu el fitxer `XQJ06.java` dins el paquet de codi font a l'annex "Material per a desenvolupament sobre SGBD-XML natives utilitzant l'API XQJ" dels Annexos del web.

```
G:\>java –Dfile.encoding=cp850 proves.XQJ05 nomSGBD
Connexió establerta amb SGBD ...
Executant instrucció:
...
Instrucció executada
```

Podem comprovar, a través de la interfície gràfica que correspongui (eXist Client Shell, BaseX GUI o SednaAdmin), que la inserció s'ha dut a terme.

Exemple de programa amb sentència tancada de tipus "no consulta" (update) i d'execució immediata

El programa `XQJ06.java` canvia el nom dels països que tenen $$ com a car_code.

Comprovem el funcionament d'aquest programa, executant-lo per cadascun dels tres SGBD. L'execució, en qualsevol cas, és la següent:

```
1 G:\>java –Dfile.encoding=cp850 proves.XQJ06 nomSGBD
2 Connexió establerta amb SGBD ...
3 Executant instrucció:
4 ...
5 Instrucció executada
```

Podem comprovar, a través de la interfície gràfica que correspongui (eXist Client Shell, BaseX GUI o SednaAdmin), que la modificació s'ha dut a terme.

Trobareu el fitxer `XQJ07.java` dins el paquet de codi font a l'apartat "Material per a desenvolupament sobre SGBD-XML natives utilitzant l'API XQJ" dels Annexos del web.

Exemple de programa amb sentència tancada de tipus "no consulta" (delete) i d'execució immediata

El programa XQJ07.java elimina els països que tenen $$ com a car_code.

Comprovem el funcionament d'aquest programa, executant-lo per cadascun dels tres SGBD. L'execució, en qualsevol cas, és la següent:

```
1 G:\>java –Dfile.encoding=cp850 proves.XQJ07 nomSGBD
2 Connexió establerta amb SGBD ...
3 Executant instrucció:
4 ...
5 Instrucció executada
```

Podem comprovar, a través de la interfície gràfica que correspongui (eXist Client Shell, BaseX GUI o SednaAdmin), que l'eliminació s'ha dut a terme.

Recordem que les instruccions `UPDATE` que aporten els SGBD eXist-db i Sedna són considerades "no consulta", mentre que les que aporta l'SGBD BaseX (XQUF) són considerades "consulta".

Els tres exemples anteriors han consistit en l'execució d'una sentència "no consulta" corresponent a l'actualització de les bases de dades (`insert`,`update` i `delete`, respectivament). Hi ha, però, altres tipus d'ordres, segons l'SGBD, que també es poden executar des de l'API XQJ (gestió de col·leccions, gestió de documents...). Atès que no tots els SGBD les permeten i que la sintaxi és molt diferent d'uns a altres, no té cap interès desenvolupar programes que puguin servir per a diversos SGBD. És convenient, però, que el lector n'efectuï proves en els diversos SGBD.

Les API client específiques d'alguns SGBD-XML natives faciliten mecanismes per saber, abans de procedir a l'execució, si una sentència correspon a una "consulta" (retorna un conjunt de resultats a processar) o a una "no consulta" (executa una ordre). L'SGBD Sedna n'és un exemple.

Observem, també, que en els tres exemples anteriors s'han executat instruccions d'actualització de la base de dades sense validar (`commit`) els canvis i aquests han esdevingut permanents. Això ha estat així perquè, si no s'indica el contrari, les connexions de l'API XQL neixen amb la propietat `auto-commit` activada.

Trobareu el fitxer `XQJ08.java` dins el paquet de codi font a l'apartat "Material per a desenvolupament sobre SGBD-XML natives utilitzant l'API XQJ" dels Annexos del web.

Per finalitzar l'execució de sentències immediates, presentem dos programes de sentència oberta (l'usuari introdueix la sentència que cal executar): un per executar sentències "consulta" i l'altre per executar sentències "no consulta". L'usuari ha de ser conscient que està executant el programa adequat, segons el tipus de consulta.

Exemple de programa amb sentència oberta de tipus "consulta" i d'execució immediata

El programa `XQJ08.java` demana a l'usuari, per la consola, la instrucció a executar i procedeix a executar-la com a "consulta". Recordem que, malauradament, la interfície `XQExpression` no facilita cap mètode per deduir si una sentència és "consulta" o "no consulta". Si no es produeix error, el programa intenta mostrar la seqüència de resultats de dues maneres:

1. Com a arxiu XML, a partir de la seriació de la seqüència de resultats a través del mètode `XQResultSequence.getSequenceAsStream()` i amb l'ajut de la classe `Transformer`.

2. Com a objecte `String`, a partir de la seriació de la seqüència de resultats a través del mètode `XQResultSequence.getSequenceAsString()`.

Per tal de poder veure el resultat de dues maneres, cal tornar a executar la sentència abans de la segona visualització.

En cas que la seqüència de resultats (`XQResultSequence`) contingui més d'un valor, perquè la seriació cap a `XMLStreamReader` funcioni correctament cal que l'API XQJ emboliqui tota la seqüència dins un node de més alt nivell. Aquest funcionament és correcte en les interfícies desenvolupades per Charles Foster per als SGBD eXist-db i Sedna, però no és així en l'API XQJ integrada en BaseX (versió 7.1), de manera que la visualització XML dels resultats de les consultes amb més d'un resultat, en BaseX, només mostra el primer resultat. Per això també mostrem la solució en format `String`, per constatar que la instrucció s'executa correctament i que el que falla és la seriació cap a `XMLStreamReader`.

Comprovem el funcionament d'aquest programa executant una mateixa instrucció de consulta (amb la sintaxi adequada a cada SGBD) per cadascun dels tres SGBD. Demanem, per exemple, els noms de les autonomies de l'Estat espanyol.

Execució en eXist-db:

```
G:\>java -Dfile.encoding=cp850 proves.XQJ08 eXist-db
Introdueixi la instrucció a executar...
Per finalitzar, introdueixi línia buida (en blanc):
doc("xqj/mondial.xml")/mondial/country[@car_code='E']/province/
    name

Connexió establerta amb SGBD exist-db
Executant instrucció "consulta":
doc("xqj/mondial.xml")/mondial/country[@car_code='E']/province/
    name

Visualització de resultats com a XML:
<?xml version="1.0" encoding="UTF-8"?>
<r:result xmlns:r="http://www.xqj.net/">
  <name>Andalusia</name>
  <name>Aragon</name>
  <name>Asturias</name>
  <name>Balearic Islands</name>
  <name>Basque Country</name>
  <name>Canary Islands</name>
  <name>Cantabria</name>
  <name>Castile and Leon</name>
  <name>Castile La Mancha</name>
  <name>Catalonia</name>
  <name>Estremadura</name>
  <name>Galicia</name>
  <name>Madrid</name>
  <name>Murcia</name>
  <name>Navarre</name>
  <name>Rioja</name>
  <name>Valencia</name>
</r:result>
```

```
31
32 Repetim execució:
33 doc("xqj/mondial.xml")/mondial/country[@car_code='E']/province/
       name
34
35 Visualització de resultats com a String:
36 <name>Andalusia</name> <name>Aragon</name> <name>Asturias</name>
       <name>Balearic Islands</name> <name>Basque Country</name> <
       name>Canary Islands</name><name>Cantabria</name> <name>
       Castile and Leon</name> <name>Castile La Mancha</name><name>
       Catalonia</name> <name>Estremadura</name> <name>Galicia</name
       > <name>Madrid</name> <name>Murcia</name> <name>Navarre</name
       ><name>Rioja</name><name>Valencia</name>
```

Execució en Sedna:

```
1  G:\>java -Dfile.encoding=cp850 proves.XQJ08 Sedna
2  Introdueixi la instrucció a executar...
3  Per finalitzar, introdueixi línia buida (en blanc):
4  doc("mondial.xml","xqj")/mondial/country[@car_code='E']/province/
       name
5
6  Connexió establerta amb SGBD Sedna
7  Executant instrucció "consulta":
8  doc("mondial.xml","xqj")/mondial/country[@car_code='E']/province/
       name
9
10 Visualització de resultats com a XML:
11 <?xml version="1.0" encoding="UTF-8"?>
12 <r:result xmlns:r="http://www.xqj.net/">
13   <name>Andalusia</name>
14   <name>Aragon</name>
15   <name>Asturias</name>
16   <name>Balearic Islands</name>
17   <name>Basque Country</name>
18   <name>Canary Islands</name>
19   <name>Cantabria</name>
20   <name>Castile and Leon</name>
21   <name>Castile La Mancha</name>
22   <name>Catalonia</name>
23   <name>Estremadura</name>
24   <name>Galicia</name>
25   <name>Madrid</name>
26   <name>Murcia</name>
27   <name>Navarre</name>
28   <name>Rioja</name>
29   <name>Valencia</name>
30 </r:result>
31
32 Repetim execució:
33 doc("mondial.xml","xqj")/mondial/country[@car_code='E']/province/
       name
34
35 Visualització de resultats com a String:
36 <name>Andalusia</name> <name>Aragon</name> <name>Asturias</name>
       <name>Balearic Islands</name> <name>Basque Country</name> <
       name>Canary Islands</name><name>Cantabria</name> <name>
       Castile and Leon</name> <name>Castile La Mancha</name><name>
       Catalonia</name> <name>Estremadura</name> <name>Galicia</name
       > <name>Madrid</name> <name>Murcia</name> <name>Navarre</name
       ><name>Rioja</name><name>Valencia</name>
```

Execució en BaseX:

```
1  G:\>java -Dfile.encoding=cp850 proves.XQJ08 BaseX
2  Introdueixi la instrucció a executar...
3  Per finalitzar, introdueixi línia buida (en blanc):
```

```
for $doc in collection("xqj")
where contains(document-uri($doc),"mondial.xml")
return $doc/mondial/country[@car_code='E']/province/name

Connexió establerta amb SGBD Basex
Executant instrucció "consulta":
for $doc in collection("xqj")
where contains(document-uri($doc),"mondial.xml")
return $doc/mondial/country[@car_code='E']/province/name

Visualització de resultats com a XML:
<?xml version="1.0" encoding="UTF-8"?>
<name>Andalusia</name>

Repetim execució:
for $doc in collection("xqj")
where contains(document-uri($doc),"mondial.xml")
return $doc/mondial/country[@car_code='E']/province/name

Visualització de resultats com a String:
<name>Andalusia</name>
<name>Aragon</name>
<name>Asturias</name>
<name>Balearic Islands</name>
<name>Basque Country</name>
<name>Canary Islands</name>
<name>Cantabria</name>
<name>Castile and Leon</name>
<name>Castile La Mancha</name>
<name>Catalonia</name>
<name>Estremadura</name>
<name>Galicia</name>
<name>Madrid</name>
<name>Murcia</name>
<name>Navarre</name>
<name>Rioja</name>
<name>Valencia</name>
```

Podem comprovar que la visualització XML en BaseX (versió 7.1) no és correcta.

Exemple de programa amb sentència oberta de tipus "no consulta" i d'execució immediata

El programa `XQJ09.java` demana a l'usuari, per la consola, la instrucció a executar i procedeix a executar-la com a "no consulta". Recordem que, malauradament, la interfície `XQExpression` no facilita cap mètode per deduir si una sentència és "consulta" o "no consulta". Si no es produeix error, el programa informarà que la sentència ha estat executada.

Comprovem el funcionament d'aquest programa executant una mateixa instrucció (amb la sintaxi adequada a cada SGBD) per cadascun dels tres SGBD. Procedim, per exemple, a crear una col·lecció (base de dades a BaseX) a l'arrel de la base de dades activa. Per conèixer la sintaxi adequada per aconseguir el nostre objectiu, en cadascun dels tres SGBD, ens cal consultar la corresponent documentació.

Trobareu el fitxer `XQJ09.java` dins el paquet de codi font a l'apartat "Material per a desenvolupament sobre SGBD-XML natives utilitzant l'API XQJ" dels Annexos del web.

Execució en eXist-db:

```
G:\>java -Dfile.encoding=cp850 proves.XQJ09 eXist-db
Introdueixi la instrucció a executar...
Per finalitzar, introdueixi línia buida (en blanc):
xmldb:create-collection("/","xqjbis")

Connexió establerta amb SGBD exist-db
Executant instrucció "no consulta":
xmldb:create-collection("/","xqjbis")

Instrucció executada
```

Podem comprovar, a través de la interfície gràfica eXist Client Shell,que la instrucció s'ha dut a terme.

Execució en Sedna:

```
G:\>java –Dfile.encoding=cp850 proves.XQJ09 Sedna
Introdueixi la instrucció a executar...
Per finalitzar, introdueixi línia buida (en blanc):
create collection "xqjbis"

Connexió establerta amb SGBD Sedna
Executant instrucció "no consulta":
create collection "xqjbis"

Instrucció executada
```

Podem comprovar, a través de la interfície gràfica SednaAdmin (prèvia desconnexió-connexió a la base de dades), que la instrucció s'ha dut a terme.

Execució en BaseX:

```
G:\>java –Dfile.encoding=cp850 proves.XQJ09 BaseX
Introdueixi la instrucció a executar...
Per finalitzar, introdueixi línia buida (en blanc):
create database xqjbis

Connexió establerta amb SGBD BaseX
Executant instrucció "no consulta":
create database xqjbis

Instrucció executada
```

Podem comprovar, a través de la interfície gràfica BaseX GUI, que la instrucció s'ha dut a terme.

2.1.3 Variables lligades

En les aplicacions Java que accedeixen a bases de dades, ja sigui via SQL o via XQuery, una de les pitjors implementacions que pot efectuar un programador és utilitzar el contingut d'una variable emplenada per l'usuari directament en una expressió SQL o XQuery, ja que hi ha usuaris finals maliciosos que poden injectar codi SQL o XQuery maliciós per aconseguir informació privilegiada o per efectuar actualitzacions no desitjades.

Trobareu el fitxer `XQJ10.java` dins el paquet de codi font a l'apartat "Material per a desenvolupament sobre SGBD-XML natives utilitzant l'API XQJ" dels Annexos del web.

Exemple de programa que admet injecció XQuery

El programa `XQJ10.java` mostra els noms dels països democràtics que han aconseguit la independència a partir d'una determinada data a introduir per l'usuari.

Comprovem el funcionament d'aquest programa executant-lo per cadascun dels tres SGBD i suposem que l'usuari introdueix una data, com s'indica. L'execució, en qualsevol cas, és la següent:

```
G:\>java –Dfile.encoding=cp850 proves.XQJ10 nomSGBD
Països democràtics independents a partir de data...
Introdueixi data [dddd–mm–yy] o res per finalitzar:
1992–01–01
Connexió establerta amb SGBD ...
Executant instrucció:
```

```
7   ...
8   Resultats:
9   Serbia – parliamentary democracy – 2006–06–05
10  Montenegro – parliamentary democracy – 2006–06–03
11  Czech Republic – parliamentary democracy – 1993–01–01
12  Slovakia – parliamentary democracy – 1993–01–01
13  Bosnia and Herzegovina – emerging democracy – 1992–04–01
14  Timor–Leste – parliamentary democracy – 2002–05–20
```

Observem, però, com l'usuari pot injectar codi XQuery per aconseguir tots els països, democràtics i no democràtics, i a partir de qualsevol data:

```
1   G:\>java –Dfile.encoding=cp850 proves.XQJ09 nomSGBD
2   Països democràtics amb independència a partir de data...
3   Introdueixi data [dddd–mm–yy] o res per finalitzar:
4   1992–01–01' or '1'='1
5   Connexió establerta amb SGBD ...
6   Executant instrucció:
7   ...
8   Resultats:
9   Albania – emerging democracy – 1912–11–28
10  Greece – parliamentary republic – 1829–01–01
11  Macedonia – emerging democracy – 1991–09–17
12  ...
13  Seychelles – republic – 1976–06–29
```

Per tant, l'usuari ha aconseguit informació que no tenia per què obtenir i el problema es produeix perquè el programador ha utilitzat el contingut de la variable desdeData directament dins la sentència XQuery, sense cap tipus de comprovació del seu contingut.

El fet d'actuar així també provoca errors en cas que l'usuari no introdueixi una data en el format indicat. Comproveu què succeeix quan l'usuari introdueix 01/01/1992 o qualsevol cadena alfanumèrica.

Per evitar els problemes d'injecció de codi i per assegurar que les variables que s'utilitzen en les expressions XQuery corresponguin a tipus adequats, XQuery permet declarar variables externes (sentència `declare` prèvia) i en el programa haurem de lligar les variables externes definides a la sentència XQuery amb els valors que corresponguin, normalment introduïts per l'usuari, mitjançant els mètodes `bind` que facilita la interfície `XQDynamicContext`.

Trobareu el fitxer `XQJ11.java` dins el paquet de codi font a l'apartat "Material per a desenvolupament sobre SGBD-XML natives utilitzant l'API XQJ" dels Annexos del web.

Exemple de programa que utilitza variables externes en XQuery prevenint la injecció de codi i errors de tipus

El programa `XQJ11.java` mostra els noms dels països democràtics que han aconseguit la independència a partir d'una determinada data a introduir per l'usuari. El programa controla que la data sigui correcta i no permet injecció de codi.

En aquest cas, és impossible l'execució del programa introduint valors erronis com a data o intentant injecció de codi XQuery.

Al programa anterior s'utilitza el mètode `bindAtomicValue()` de la interfície `XQDynamicContext` per assignar el contingut de la variable `String desdeData` a la variable externa `fromDate` de la sentència XQuery:

```
1   xqe.bindAtomicValue(new QName("fromDate"),
2                       desdeData,
3                       conn.createAtomicType(XQItemType.XQBASETYPE_DATE)
4                      );
```

on `xqe` correspon a l'expressió que executarem i que ha de contenir la variable externa `$fromDate` declarada en el prolog de la instrucció. A eXist-db, per exemple:

```
xqe.executeQuery(
   "declare variable $fromDate external; "+
   "for $i in doc('xqj/mondial.xml')/mondial/"+
   "country[contains(government,'democracy') and "+
   "indep_date >= $fromDate] "+
   "return concat($i/name,' − ',$i/government,' − ',$i/indep_date))";
```

Fixem-nos en el mètode `bindAtomicValue()`:

- El primer paràmetre és el nom de la variable externa de l'objecte `XQExpression`.
- El segon paràmetre és el valor `String` que es vol assignar a la variable externa.
- El tercer paràmetre ha de ser un objecte `XQItemType` corresponent al tipus amb el qual s'ha d'interpretar el valor del segon paràmetre per assignar-lo a la variable del primer paràmetre.

Mireu la documentació de les interfícies `XQDynamicContext` i `XQItemType`.

La interfície `XQItemType` defineix el conjunt de tipus pels quals es poden tenir objectes `XQItemType`, els quals s'han de crear amb el mètode `createAtomicType()` de la interfície `XQDataFactory`.

La interfície `XQDynamicContext` conté mètodes similars a `bindAtomicValue()`, com `bindInt()`, `bindByte()`, `bindDouble()`, `bindDocument()`, `bindBoolean()`... que es diferencien de `bindAtomicValue()` en el segon paràmetre, que ha de ser del tipus que indica el nom del mètode.

2.1.4 Sentències XQuery preparades

Tots els SGBD acostumen a facilitar dos mecanismes d'execució de sentències contra la BD, utilitzant el llenguatge que correspongui (SQL en SGBDR i SGBDOR, OQL en SGBDOO i XQuery/Update en SGBD-XML):

1. Execució immediata d'una sentència, utilitzada quan la sentència s'ha d'executar una única vegada.
2. Execució de sentències paramètriques o preparades, utilitzada quan la mateixa sentència (o sentència similar amb alguns valors diferents) s'ha d'executar repetidament i consistent a escriure la sentència com una plantilla que conté algunes variables que s'aniran substituint a cada execució.

L'API XQJ facilita els dos mecanismes a través de les interfícies:

- `XQExpression`, per a l'execució immediata de sentències.
- `XQPreparedExpression`, per a l'execució de sentències paramètriques.

Centrem-nos, en aquest moment, en l'execució de sentències paramètriques proporcionada per la interfície `XQPreparedExpression`.

Per executar una sentència preparada crearem un objecte `XQPreparedExpression` a partir del mètode `XQConnection.prepareExpression()`. Disposem de sis sobrecàrregues d'aquest mètode:

```
1  XQPreparedExpression prepareExpression(java.lang.String consulta)
2                         throws XQException;
3  XQPreparedExpression prepareExpression(java.lang.String consulta,
4                         XQStaticContext properties) throws XQException;
5  XQPreparedExpression prepareExpression(java.io.InputStream consulta)
6                         throws XQException;
7  XQPreparedExpression prepareExpression(java.io.InputStream consulta,
8                         XQStaticContext properties) throws XQException;
9  XQPreparedExpression prepareExpression(java.io.Reader consulta)
10                        throws XQException;
11 XQPreparedExpression prepareExpression(java.io.Reader consulta,
12                        XQStaticContext properties) throws XQException;
```

En totes aquestes sobrecàrregues s'obté un objecte `XQPreparedExpression` que portarem a execució amb el mètode `XQPreparedExpression.executeQuery()`. Tres de les sis sobrecàrregues permeten indicar les propietats del context estàtic que es tindran en compte en avaluar l'expressió, mentre que la resta de sobrecàrregues prenen com a context estàtic l'associat a la connexió.

Les propietats del context estàtic es poden gestionar amb mètodes `get` i `set` de la interfície `XQStaticContext`. La interfície `XQConnection` facilita els mètodes `getStaticContext()` i `setStaticContext()` per recuperar i establir el context estàtic de la connexió.

Els objectes `XQPreparedExpression` estan pensats per definir una plantilla de sentència "consulta" ideada per ser executada múltiples vegades, canviant a cada execució el valor de les variables que incorpora. Així doncs, no té cap sentit crear un objecte `XQPreparedExpression` sense variables.

Les variables que s'utilitzen en les sentències preparades són sempre variables enllaçades per prevenir injecció de codi i per garantir la correctesa dels tipus dels valors assignats a les variables. Aquestes variables han d'estar declarades en el prolog de la instrucció XQuery a executar, i han d'haver estat emplenades abans de procedir a l'execució de la sentència.

La interfície `XQPreparedExpression` disposa de diversos mètodes `get` per obtenir informació sobre el context estàtic i el context dinàmic (variables enllaçades). Disposa, també, del mètode `close()` per tancar l'expressió, alliberant tots els recursos associats quan ja no sigui necessària. L'execució del mètode `close()` sobre la connexió tanca totes les expressions definides sobre aquesta. També disposem del mètode `isClosed()` per poder esbrinar si una expressió està oberta o tancada:

```
1 void close() throws XQException;
2 boolean isClosed();
```

Fixem-nos que el mètode `executeQuery()` retorna, si tot va bé, un objecte `XQResultSequence`, interfície que deriva de la interfície `XQSequence`, la qual ens facilita un conjunt de mètodes per avaluar la seqüència de resultats obtinguts amb el mètode `executeQuery()`, com el mètode `XQExpression.executeQuery()`.

Exemple de programa amb sentència XQuery preparada

El programa `XQJ12.java` mostra el rànquing dels països segons la mortalitat infantil (nombre de nens menors d'un any morts per cada mil naixements), en intervals d'amplada vint ([0,20[, [20,40[, [40,60[,...).

Trobareu el fitxer `XQJ12.java` dins el paquet de codi font a l'apartat "Material per a desenvolupament sobre SGBD-XML natives utilitzant l'API XQJ" dels annexos del web.

Per aconseguir el resultat desitjat, el programa prepara una sentència XQueryque cerca els països amb mortalitat infantil en un determinat interval i executa aquesta instrucció tantes vegades com intervals d'amplada 20 hi ha entre 0 i 1000, és a dir, 50 vegades.

Comprovem el funcionament d'aquest programa executant-lo per cadascun dels tres SGBD. L'execució, en qualsevol cas, és la següent:

```
G:\>java –Dfile.encoding=cp850 proves.XQJ12 nomSGBD
Connexió establerta amb SGBD ...
Interval [0.0,20.0[:
        2.2 - Andorra
        2.4 – Man
        ...
        6.3 - Spain
        ...
        19.6 – Qatar
Interval [20.0,40.0[:
        20 – Saint Lucia
        ...
...
Interval [140.0,160.0[:
        145.82 – Western Sahara
        149.7 – Afghanistan
```

El codi presentat conté la sentència:

```
// En BaseX (versió 7.1), cal tornar a preparar la sentència
   if (args[0].equalsIgnoreCase("BaseX"))
      xqpe = conn.prepareExpression(cad);
```

És a dir, a la versió 7.1 de BaseX cal repetir la preparació de la sentència abans de cada execució. Això és, evidentment, un mal funcionament del mecanisme en aquest SGBD. Comproveu el funcionament si comenteu aquesta sentència condicional.

2.1.5 XQJ per processar documents XML

L'API XQJ permet processar documents XML que no tenen per què estar emmagatzemats a la base de dades, mitjançant els mètodes `bind` adequats per a documents que facilita la interfície `XQDynamicContext`.

Concretament, ens interessa conèixer els mètodes següents:

```
void bindDocument(javax.xml.namespace.QName varName,
                  java.io.Reader value,
                  java.lang.String baseURI,
                  XQItemType type) throws XQException;
void bindDocument(javax.xml.namespace.QName varName,
                  java.io.InputStream value,
                  java.lang.String baseURI,
                  XQItemType type) throws XQException;
```

```
void bindDocument(javax.xml.namespace.QName varName,
                  javax.xml.stream.XMLStreamReader value,
                  XQItemType type) throws XQException;
void bindDocument(javax.xml.namespace.QName varName,
                  javax.xml.transform.Source value,
                  XQItemType type) throws XQException;
```

Els dos primers mètodes esperen, en el segon paràmetre `value`, la font XML a gestionar, ja sigui com a objecte `Reader` o com a objecte `InputStream`, la qual serà analitzada i recuperada com un node document. El tercer paràmetre `baseURI` és opcional (per tant, pot ser `null`) i pot ser utilitzat per resoldre URI relatives o inclòs en missatges d'error. El quart paràmetre ha de ser `null` o `XQITEMKIND_DOCUMENT_ELEMENT` o `XQITEMKIND_DOCUMENT_SCHEMA_ELEMENT`.

Els dos darrers mètodes esperen, en el segon paràmetre `value`, un document XML, ja sigui com a objecte `XMLStreamReader` o com a objecte `Source`. Recordem que `XQResultSequence.getSequenceAsStream()` obté el resultat d'una consulta com a objecte `XMLStreamReader`.

Als annexos de la web trobareu l'apartat "Material per a desenvolupament sobre SGBD-XML natives utilitzant l'API XQJ" que inclou l'arxiu `mondial.zip`, que conté els fitxers `mondial.xml` i `mondial.dtd`.

Trobareu el fitxer `XQJ13.java` dins el paquet de codi font a l'apartat "Material per a desenvolupament sobre SGBD-XML natives utilitzant l'API XQJ" dels Annexos del web.

Exemple de programa que utilitza els motors XQuery dels SGBD per processar un fitxer XML resident al sistema de fitxers

Volem executar una instrucció XQuery sobre el fitxer `mondial.xml` existent en el sistema de fitxers del sistema operatiu, és a dir, fora de cap BD-XML. Concretament, volem cercar els noms de les autonomies d'Espanya.

Per poder executar el programa (`XQJ13.java`) necessitem tenir el fitxer `mondial.xml` en una carpeta del nostre sistema d'arxius. Atès que el fitxer incorpora, en la seva segona línia, una referència al fitxer `mondial.dtd` corresponent, ens caldrà disposar d'aquest fitxer en la mateixa carpeta.

Comprovem el funcionament d'aquest programa executant-lo per cadascun dels tres SGBD. L'execució, en qualsevol cas, hauria de ser la següent:

```
G:\>java —Dfile.encoding=cp850 proves.XQJ12 nomSGBD
Connexió establerta amb SGBD ...
<name>Andalusia</name>
<name>Aragon</name>
<name>Asturias</name>
<name>Balearic Islands</name>
<name>Basque Country</name>
<name>Canary Islands</name>
<name>Cantabria</name>
<name>Castile and Leon</name>
<name>Castile La Mancha</name>
<name>Catalonia</name>
<name>Estremadura</name>
<name>Galicia</name>
<name>Madrid</name>
<name>Murcia</name>
<name>Navarre</name>
<name>Rioja</name>
<name>Valencia</name>
```

El resultat és així en BaseX, però en eXist-db i Sedna apareixen errors diversos.

A eXist-db, l'error fa referència al fet que no troba el fitxer `mondial.dtd` indicat a la segona línia del fitxer `mondial.xml`. Si es comenta aquesta línia, l'error canvia però eXist-db continua donant un error diferent, que desapareix en eliminar la segona línia del document XML. Llavors s'obté el resultat esperat. Sembla que es tracta d'un error de l'API XQJ per a eXist-db.

A Sedna es genera l'excepció `ArrayOutOfBoundsException`, independentment que la segona línia del document `mondial.xml` hi sigui o estigui comentada, o no hi sigui.

En conseqüència, observem que els motors XQuery dels SGBD ens poden ajudar a gestionar arxius XML externs a les BD a través de l'API XQJ.

Però cal estar alerta, ja que, per defecte, l'API XQJ intenta emmagatzemar tot el document XML en memòria, tal com fa el DOM (Document Object Model), i això pot provocar problemes de memòria quan el document XML a processar té una grandària considerable (de l'ordre de gigabytes).

Per evitar-ho i assegurar-se que XQJ no actuï d'aquesta manera, XQJ proporciona l'enllaç diferit, que permet no carregar l'arxiu en memòria i accedir-hi en el moment de procés, segons les necessitats de la sentència a executar.

La documentació de la interfície `XQDynamicContext` detalla el funcionament de l'enllaç immediat i diferit.

Per evitar l'enllaç immediat (opció per defecte) cal establir l'enllaç diferit, cosa que s'assoleix fent el següent:

1. Crear un nou objecte `XQStaticContext` basat en el context estàtic actual:
   ```
   XQStaticContext propietats = conn.getStaticContext();
   ```
2. Establir la propietat "mode d'enllaç" a diferit:
   ```
   propietats.setBindingMode(XQConstants.BINDING_MODE_DEFERRED);
   ```
3. Establir les noves propietats com a context a emprar, fet que es pot activar en tres punts diferents:
   (a) Per a una determinada sentència d'execució immediata:
   ```
   XQExpression xqe = conn.createExpression
   (propietats);
   ```
   (b) Per a una determinada sentència preparada:
   ```
   String sentencia = "declare...";
   XQPreparedExpression xqpe = conn.prepareExpression(sentencia,
   propietats);
   ```
   (c) Per a totes les noves expressions (canvi global):
   ```
   conn.setStaticContext(propietats);
   ```

Trobareu el fitxer `XQJ14.java` dins el paquet de codi font a l'apartat "Material per a desenvolupament sobre SGBD-XML natives utilitzant l'API XQJ" dels annexos del web.

Exemple de programa que utilitza els motors XQuery dels SGBD per processar un fitxer XML resident al sistema de fitxers, amb enllaç diferit.

Aquest exemple (`XQJ14.java`) és la repetició de l'exemple anterior, per obtenir els noms de les autonomies d'Espanya a partir del fitxer `mondial.xml` resident en el sistema d'arxius, utilitzant enllaç diferit.

L'execució en BaseX continua funcionant correctament. I a eXist-db i Sedna, prèvia eliminació de la segona línia de l'arxiu `mondial.xml`, també funciona. És a dir, a Sedna ha desaparegut l'excepció `ArrayIndexOut-OfBoundsException` existent quan no estava activat l'enllaç diferit.

Components d'accés a dades

Josep Cañellas Bornas

Índex

Introducció 5

Resultats d'aprenentatge 7

1 Aproximació al concepte de component de programari 9
1.1 Definició de component . . . 9
1.2 Fases de construcció dels components de programari . . . 10
1.2.1 Especificació de components de programari . . . 10
1.2.2 Implementació de components de programari . . . 12
1.2.3 Empaquetatge dels components . . . 13
1.2.4 Desplegament dels components de programari . . . 14
1.3 Principals problemes a resoldre durant la construcció d'un component . . . 16
1.3.1 Ús d'interfícies i classes . . . 16
1.3.2 Accés a les interfícies del component . . . 17
1.3.3 Granularitat dels components . . . 21
1.3.4 Descomposició dels components en classes . . . 22
1.3.5 Components i reutilització . . . 23
1.3.6 Adaptació dels components . . . 24
1.4 Creació de components amb Java . . . 27
1.4.1 JavaBeans . . . 27
1.4.2 Esdeveniments . . . 29
1.4.3 Propietats . . . 33
1.5 Components per a la persistència . . . 43
1.5.1 Esdeveniments associats als components d'accés a dades . . . 44

2 Exemples d'implementació de components de persistència 49
2.1 Especificació de les interfícies d'accés al component de persistència . . . 50
2.1.1 Organització de les interfícies d'accés al component . . . 52
2.1.2 Estratègia façana en el disseny de components de persistència . . . 53
2.1.3 Estratègia fàbrica en el disseny de components de persistència . . . 58
2.2 Implementació dels components de persistència basats en JDBC . . . 64
2.2.1 Processos comuns d'accés a dades . . . 64
2.2.2 Creació de taules en el'SGBD . . . 69
2.2.3 Implementació de la interfície amb rol de façana . . . 73
2.2.4 Implementació de la interfície amb rol instanciador . . . 85
2.3 Empaquetatge del component de persistència . . . 93
2.4 Desplegament del component de persistència . . . 96

Introducció

En aquest unitat anomenada “Components d’accés a dades”, treballarem per primer cop el concepte de component. Els components no són exclusius de la persistència, ja que es poden aplicar a qualsevol part d’una aplicació. De fet, és un concepte que apareix a les darreries del segle XX i es comença a implementar durant aquesta darrera dècada. És, doncs, una tecnologia força innovadora que intenta copiar els grans avantatges quel’enginyeria industrial i l’arquitectura han sabut portar a terme amb tant d’èxit.

Tant a la indústria mecànica com a l’arquitectura, els processos de construcció d’elements complexos es divideixen de forma que sigui possible crear peces independents que acabin formant part de l’element final, però que permetin una construcció en paral·lel de múltiples components per tal que sigui més fàcil automatitzar i rendibilitzar els processos de construcció de les peces, que sigui possible reutilitzar el mateix tipus de peces en diverses construccions, de manera que es pugui abaratir el seu cost final i que es puguin encastar fàcilment amb la resta de peces quan calgui acabar de construir l’element final.

Sota aquestes prerrogatives, la indústria del software ha volgut assentar també les bases de la construcció d’aplicacions a partir de components de programari més petits. Malauradament, les aplicacions no són peces físiques, i la construcció de components porta associada una certa complexitat que cal conèixer per tal de saber com superar-la.

En l’apartat “Aproximació al concepte de component de programari”, després de veure el concepte de component i analitzar-ne les característiques, la discussió s’encamina cap als components de persistència. Es tracta de components específics encarregats de l’emmagatzematge i recuperació de les dades de les aplicacions.

L’apartat “Exemples d’implementació de components de persistència” implementa un exemple on es posen en pràctica els conceptes estudiats en tot el mòdul sencer. Es tracta d’un exemple complex que l’estudiant haurà de seguir i implementar mentre el llegeix per tal de poder copsar tots els processos que de la implementació d’un component de certa complexitat se’n poden desprendre. Per raons d’espai la implementació no es realitza totalment, però seria convenient que l’intenteu completar tant com el temps us permeti.

En aquest apartat veurem d’una forma pràctica com implementar components de persistència d’una aplicació. Per raons d’espai de pàgines i temps d’estudi, resulta impossible crear tots els components de persistència d’una aplicació sencera. Amb ànim de guanyar temps partirem d’un exemple que ja coneixeu, l’aplicació comercial que heu vist a la unitat “Persistència en BDR-BDOR-BDOO”, a l’apartat on s’estudiava la persistència usant l’eina de mapatge JPA. En aquell apartat es desenvolupava un exemple d’una aplicació comercial i de control d’estocs.

Tanmateix, no desenvoluparem tota la persistència de l'aplicació, sinó que ens centrarem en els aspectes més significatius que il·lustrin la implementació d'un component de persistència.

Suposarem que ja s'ha fet l'anàlisi del model de dades i que està documentat i implementat. De fet, podeu trobar el codi i una descripció del model al'annex "Biblioteques usades en els exemples" de la unitat "Components d'accés a dades".

També és important que realitzeu les dues activitats proposades. De nou aquí es tracta de la implementació d'un model lleugerament més simple que el de la lectura, però amb una complexitat estructural prou elevada com per haver d'aplicar les tècniques estudiades.

Resultats d'aprenentatge

En acabar aquesta unitat, l'alumne:

1. Programa components d'accés a dades identificant les característiques que ha de posseir un component i utilitzant eines de desenvolupament.

    - Valora les avantatges i inconvenients d'utilitzar programació orientada a components.
    - Identifica eines de desenvolupament de components.
    - Programa components que gestionen informació emmagatzemada en fitxers.
    - Programa components que gestionen, mitjançant connectors, informació emmagatzemada en bases de dades.
    - Programa components que gestionen informació utilitzant mapatge objecte relacional.
    - Programa components que gestionen informació emmagatzemada en bases de dades objecte relacionals i orientades a objectes.
    - Programa components que gestionen informació emmagatzemada en una base de dades nativa XML.
    - Prova i documenta els components desenvolupats.
    - Integra els components desenvolupats en aplicacions.

1. Aproximació al concepte de component de programari

A mida que ens calgui enfrontar-nos a aplicacions cada cop més complexes, ens adonarem que el nombre de classes començarà a créixer desmesuradament i que aconseguir cohesionar-les totes aplicant criteris de reutilització, eficiència i qualitat és una tasca realment difícil.

Per intentar apaivagar aquesta dificultat, va aparèixer al principi d'aquest segle el concepte de component de programari. La idea és tractar el programari com si fos un producte mecànic.

Abans de començar, per exemple, la construcció industrial d'un determinat prototipus de cotxe, els enginyers busquen sistemes de construcció que redueixin el temps de fabricació i facilitin el posterior manteniment i reposició de peces.

En la indústria mecànica aquests sistemes es basen a fer construccions parcials, organitzades de tal manera que totes es puguin realitzar a la vegada i que, un cop acabat llur assemblatge, doni com a resultat el producte final. Cada una d'aquestes construccions s'anomena *component*. Així podem veure que el motor, les rodes i tots els seus mecanismes interns, els seients, les portes, capós, quadres de comandaments, etc., poden considerar-se components perquè disposen d'un sistema de construcció independent i, un cop acabats, poden acoblar-se entre ells generant el producte finalment acabat.

De forma semblant, si fos possible organitzar tot el conjunt de classes que componen una aplicació en components independents, de manera que fos possible la creació concurrent de cada peça i s'aconseguís un assemblatge final fàcil i eficient, ens trobaríem davant d'una possible solució al problema de la construcció d'aplicacions complexes.

1.1 Definició de component

Seguint amb l'exemple del cotxe, amb el qual hem introduït el concepte de component, aprofitarem per discutir quan una agrupació de peces podem considerar-la component i quan no.

Establint un paral·lelisme amb l'exemple del cotxe, direm que els **components de programari** són aquelles unitats executables i independents que componen una aplicació, les quals tenen una funcionalitat i una forma d'interactuar perfectament definides, de manera que el seu assemblatge amb la resta de components es pugui fer sense haver de modificar el codi intern de cap d'ells i, a més, en qualsevol moment sigui possible la substitució per un altre component equivalent (amb una funcionalitat definida de forma idèntica).

Segurament, la forma més evident de components de programari que podeu reconèixer a primer cop d'ull són els anomenats *plugins* o *extensions*. Qui no ha instal·lat un diccionari al navegador, o no ha afegit un connector multimèdia per visualitzar determinats formats de vídeo?

Els *plugins* són unitats executables i independents que poden formar part d'una aplicació amb la qual interaccionen. La seva incorporació no requereix cap modificació del codi de l'aplicació ni de la resta de components. La instal·lació és molt fàcil de dur a terme i en qualsevol moment podem substituir el *plugin* per un altre de més eficient o per una versió superior del mateix.

Els *plugins* són components molt independents que afegeixen funcionalitat a l'aplicació, però que no són necessaris per a l'execució de la resta de funcionalitats. Però no tots els components presenten tanta independència, alguns formen part del nucli funcional de l'aplicació i, per tant, el seu rendiment afectarà l'execució de tota l'aplicació. No és possible executar l'aplicació sense ells i si s'eliminen, s'han de substituir per d'altres equivalents.

Si mirem enrere, ens podem fixar en els Drivers JDBC com un clar exponent. És tracta de components de baix nivell a partir dels quals construïm altres components més específics de cada aplicació, però són components al cap i a la fi que compleixen tots els requisits exposats. Són unitats executables independents que formen part de les aplicacions, que responen a una funcionalitat perfectament definida i es poden assemblar i intercanviar fàcilment per altres d'equivalents.

De fet, a la unitat "Persistència en fitxers" tots els exemples implementats són components,ja que responen també a les característiques esmentades.

1.2 Fases de construcció dels components de programari

La definició que fins ara hem donat de component de programari pot servir per fer-nos una idea del concepte, però no ens ajuda gaire a saber què hem de fer per construir en últim terme un component.

Com qualsevol peça de programari, els components s'hauran d'**especificar**, **implementar**, **empaquetar** i **desplegar** en alguna aplicació i sota alguna plataforma concretes. Cada un d'aquests aspectes comporta característiques específiques que val la pena tenir en compte si volem construir un producte de qualitat.

1.2.1 Especificació de components de programari

Per especificació dels components, cal entendre totes aquelles descripcions que permetin dur a terme la resta de fases posteriors. És a dir, la descripció detallada de les operacions que el component haurà de suportar, del conjunt d'objectes que integrin el component, de les operacions d'altres components que sigui necessari

cridar durant l'execució, de la forma com caldrà empaquetar el component i quins elements formaran part del paquet o com s'haurà de fer la instal·lació i la configuració del component per aconseguir un assemblatge correcte, etc.

De totes les descripcions esmentades, les que fan referència a la descripció detallada de les operacions són d'una importància cabdal, perquè amb elles hem d'assegurar la independència de cada component, l'assemblatge final i la capacitat de crear diversos components equivalents i intercanviables.

Malgrat que les especificacions s'acostumen a fer amb llenguatges independents que suportin tant descripcions textuals com diagramàtiques, com per exemple UML, cada llenguatge de programació acostuma a permetre, plasmar totalment o parcial, aquesta informació en el codi per tal que sigui més fàcil la fase d'implementació.

El llenguatge Java fa servir dos elements per plasmar la descripció detallada de les operacions suportades pel component. D'una banda, fa servir **interfícies** per descriure la sintaxi de les operacions i assegurar la independència i l'equivalència dels components.

Les interfícies determinen de quines operacions disposarà el component i quina serà la seva sintaxi. Posteriorment, caldrà implementar les interfícies mitjançant classes que codificaran tots els mètodes, però l'ús d'interfícies permet poder fer diverses implementacions diferents sense perdre interoperabilitat amb la resta de components i aplicació.

L'ús d'interfícies Java no són un requisit dels components. De fet, el que reclama el component és només una sintaxi clara i única per a cada operació, i això es pot fer també implementant directament les classes. Les interfícies, però, presenten un gran avantatge sobre les classes: permeten fer servir diverses versions d'un mateix component a la vegada, en una mateixa aplicació.

D'altra banda, Java disposa també dels comentaris de documentació per descriure textualment la funcionalitat dels mètodes d'una classe. Quan els comentaris de documentació es fan servir en una interfície, la seva descripció es fa extensible a qualsevol de les seves implementacions.

Malgrat que la documentació del codi és important a l'hora de fer qualsevol desenvolupament, cobra una importància cabdal quan es tracta de les interfícies d'un component, ja que una bona documentació facilita i clarifica molt la funcionalitat que caldrà aconseguir en les diferents implementacions que durant tot el cicle de vida del component es vagin fent. També defineix com altres components i aplicacions hauran de fer servir el component implementat. És per això que inclourem sempre a la documentació de les interfícies els següents aspectes:

- Descriurem detalladament l'entrada de dades de cada operació.
- Descriurem detalladament el tipus de resultat i com s'obté o es calcula a partir de les dades inicials. No es tracta de fer una codificació a la documentació, sinó una descripció detallada que no doni peu a interpretacions errònies.

- Descriurem les condicions sota les quals es produirà algun error i explicarem quin tipus d'excepció s'obtindrà en cada cas.
- Descriurem quins requeriments són necessaris per assegurar el bon funcionament del de cada operació i si es coneixen, descriurem també aquelles condicions adverses que no siguin capaces d'assegurar que la resposta del component sigui la correcta. És evident que la complexitat de certs components impedirà poder fer una implementació que controli absolutament totes les diferents situacions des de les quals seria possible accedir a alguna de les operacions del component. Per això cal delimitar quan sigui necessari, les condicions adequades de funcionament.

Imaginem que un component disposa d'una operació per calcular el nombre de vendes que un comercial hagi fet durant un cert període de temps. Imaginem que, si no es donen unes dates correctes, l'operació no fos capaç de donar una resposta fiable. Caldria indicar a la documentació que el component funcionarà correctament si les dates són correctes i coherents. Aquesta documentació permetrà als desenvolupadors que facin servir el component en el desenvolupament d'alguna aplicació prestar especial atenció a les dates que es passin a la operació.

1.2.2 Implementació de components de programari

La implementació dels components fa referència a la codificació d'una especificació. Com ja s'ha comentat, una mateixa especificació pot ser implementada de diverses maneres. Els motius de realitzar implementacions diferents poden ser molt diversos. Imagineu que la mateixa aplicació, la fan servir empreses que necessiten treballar de forma distribuïda i altres que amb una aplicació local en tenen prou. Si la funcionalitat d'ambdues aplicacions fos la mateixa, podríem fer dues implementacions d'aquells components que fossin sensibles a treballar de forma distribuïda, de manera que seria possible disposar d'una aplicació local o distribuïda segons la combinació de components que féssim.

En altres ocasions, pot no existir una forma ideal d'implementar un component, i en funció de les condicions on calgui fer la instal·lació hauríem de decantar-nos per una o altraalternativa. Sovint hi ha processos que es poden accelerar a costa de malbaratar espai de memòria i/o disc dur, però no totes les empreses estan disposades a ampliar els seus recursos de hardware i, per tant, en funció de la quantitat de recursos disponibles podríem compondre una aplicació més austera o més malgastadora.

De vegades, les implementacions només pretenen adaptar els components a una plataforma determinada i així podem disposar de components específics per a Windows, Linux, sistemes web, etc.

En Java, la implementació s'haurà de fer amb una o més classes que implementin totes les interfícies definides en el component. Malgrat que és possible implementar totes les interfícies en una sola classe, cal anar molt en compte de no crear

megaclasses que absorbeixin tota la funcionalitat del component. És important repartir-la entre diverses classes per tal de rebaixar la complexitat.

1.2.3 Empaquetatge dels components

Una de les característiques bàsiques de qualsevol component fa referència a la unitat. Per poder ser distribuïts, combinats i integrats en diferents aplicacions, és necessari organitzar tots els recursos que formen el component en una unitat que es pugui seriar fàcilment per tal de ser copiada allà on faci falta.

Generalment, distingirem dos tipus de recursos:

1. Els recursos que conformaran el component funcional (executables, imatges, dissenys d'interfícies gràfiques, etc.).
2. Els recursos d'instal·lació i configuració necessaris per adaptar el component a una aplicació concreta (configuració de les connexions específiques al'SGBD, configuració dels informes impresos, dels sistemes de registres o d'altres configuracions que el component requereixi).

Els primers acostumen a empaquetar-se junts en un únic fitxer o en un nombre reduït, per tal de facilitar la seva distribució. El codi executable s'acostuma a organitzar en fitxers de biblioteques dinàmiques. Sovint és possible incorporar altres recursos, com per exemple imatges o icones dins la mateixa biblioteca per tal de reduir el nombre de fitxers implicats. Els recursos de configuració, en canvi, solen mantenir-se en fitxers independents per tal de facilitar la seva modificació.

Quan els components s'implementen amb altres llenguatges interpretats o pseudocompilats com Java, també caldrà empaquetar d'alguna manera el component, però depèn de les utilitats que cada llenguatge disposa.

Empaquetatge del codi Java

Com ja sabeu, el llenguatge Java no crea un únic fitxer binari executable, sinó que compila per separat cada classe. Les classes compilades es vinculen durant el procés d'execució a la pròpia màquina virtual. Per aquesta raó, Java no necessita crear un únic fitxer executable.

Tot i això, per tal de mantenir el codi dels components agrupats, Java acostuma a empaquetar totes les classes compilades en un únic fitxer de tipus JAR. Es tracta d'un fitxer de format .zip constituït almenys per les classes compilades i per un fitxer descriptiu anomenat manifest.mf i ubicat a la carpeta interna del fitxer JARanomenada *meta-inf*.

El fitxer manifest.mf és útil perquè conté informació sobre el component, fins i tot és possible indicar on es troben els altres components i biblioteques requerides per aquest (`classpath` necessari durant l'execució).

Els fitxers JAR poden contenir també fitxers de recursos que no siguin específicament de codi, per exemple d'imatges, o altres formats que siguin necessaris durant l'execució de les classes dels components.

La majoria d'entorns de desenvolupament, com NetBeans, usen una utilitat anomenada Ant per automatitzar gran part de les tasques requerides a l'hora de construir una aplicació, biblioteca o component. L'eina Ant es configura per mitjà d'un fitxer XML, el qual permet definir una sèrie de tasques que es poden executar de forma independent i que NetBeans invoca durant la compilació, l'empaquetatge o l'execució. Cada tasca es defineix com una seqüència d'accions simples que en invocar la tasca s'executen una darrera l'altra. En tractar-se d'un fitxer XML, resulta molt senzill fer modificacions per tal d'adaptar les accions a les necessitats especifiques que un determinat projecte tingui. Generalment, les tasques definides per defecte són suficients, però de vegades, si el component disposa d'elements externs al JAR(fitxers de configuració, documentació, etc.), caldrà afegir alguna acció per copiar aquests fitxers al directori destí on es generarà tot el component.

1.2.4 Desplegament dels components de programari

Per desplegament dels components de programari cal entendre la instal·lació o còpia dels recursos que conformen el component en el lloc adequat per a integrar-lo a l'aplicació de la qual ha de formar part, deixant-lo operatiu.

Sovint, el component a desplegar conté múltiples recursos que caldrà copiar en diferents ubicacions del sistema on es faci el desplegament. Probablement requereixi configuració específica, que serà necessari detallar. En cas que el component a desplegar requereixi de la instal·lació d'altres components, s'haurà de comprovar que aquests ja estiguin instal·lats al sistema i activar el seu desplegament en cas que no ho estiguin. A més, és possible que calgui fer petites modificacions de la configuració de l'aplicació que incorpori el component per tal de fer operatiu el nou component.

Fer tots aquests canvis manualment pot suposar un cost molt important quan el desplegament s'hagi de repetir en diverses estacions de treball o sistemes informàtics. Per minimitzar aquest cost, la indústria ha creat diverses solucions per automatitzar aquests canvis. Generalment, es tracta de crear paquets que continguin els canvis a realitzar i aprofitar alguna eina de desempaquetatge per automatitzar-los.

Normalment, els propis sistemes operatius disposen d'un format propi d'empaquetatge per expressar les accions a realitzar durant un desplegament de software. Per exemple, el sistema operatiu Windows treballa amb el format MSI (*microsoft installer*), mentre que Linux sol treballar amb paquets RPM (*redhat package manager*) si la plataforma deriva d'una distribució RedHat o bé amb paquets `deb` si deriva d'una distribució Debian.

Cada sistema operatiu disposarà també de l'eina que executi les accions indicades en un paquet d'instal·lació concret, de manera que el desplegament consisteixi només a executar un paquet específic per cada component.

El format dels paquets sol ser textual i s'hi poden identificar comandes i seccions descrivint els requeriments i les accions del desplegament que l'eina d'instal·lació interpretarà i executarà.

Existeixen també eines que ajuden a crear els paquets de desplegament. Per exemple, Linux disposa d'una eina anomenada Debreate que ajuda a partir d'una interfície gràfica a crear paquets Debian. Windows té també multitud d'equivalents gràfics per generar paquets MSI. Entre d'altres, trobem Advanced Installer o Emco Msi Package Builder.

En general, totes aquestes eines permeten seleccionar un conjunt de fitxers que seran copiats durant el desplegament a una ruta especificada (ja sigui expressada de forma absoluta o relativa), la relació de requeriments a comprovar abans del desplegament (generalment indicant també la versió mínima requerida), els fitxers de comandes a executar durant el desplegament, etc.

El principal problema de crear els paquets de desplegament amb aquests formats és que depenen de la plataforma on caldrà desplegar-los. És a dir, en cas d'haver de desplegar-los en més d'una plataforma, serà necessari crear un paquet d'instal·lació per cada una d'elles.

El problema, però, es complica si es desconeix la plataforma on caldrà instal·lar el component. De fet, quan les actualitzacions es fan des d'una adreça pública no sempre es coneix la plataforma on es desplegarà el component, i malgrat que es pugui disposar d'una adreça diferent per a cada plataforma, no deixa de ser una complexitat afegida. És per això que també existeixen formes d'empaquetar els components amb independència de la plataforma.

És possible usar un empaquetatge estàndard (per exemple, el format ZIP o el format TAR) per aconseguir aquesta independència. Tot i això, aquest tipus d'empaquetatge només serveix per traslladar els recursos del component al dispositiu on s'hagi de fer el desplegament i realitzar la còpia pertinent. No és possible realitzar un control de les dependències ni una execució de cap fitxer de comandes `batch`. Per tant, si optem per aquesta opció és possible que sigui necessària la intervenció d'un administrador per completar el desplegament.

Aquest tipus de fitxers es poden crear automàticament durant el desenvolupament dels components fent servir eines de gestió automàtica de tasques durant el desenvolupament del programari. Ens referim a les eines associades a qualsevol llenguatge que permeten la compilació, la creació d'executables, la generació de paquets i la còpia de fitxers a rutes locals o remotes. Si usem el llenguatge C o C++ disposarem de l'eina Make. En canvi, si fem servir Java, l'eina de gestió automatitzada de tasques s'anomena Ant.

Existeixen altres utilitats per gestionar el desplegament dels components i les aplicacions amb independència de la plataforma usada. Es tracta d'eines força complexes que permeten gestionar el desplegament de tot un projecte de programari.

La utilitat Maven n'és un exemple. Es tracta d'un gestor de projectes Java que treballa amb un reposador de components i biblioteques tant pròpies del projecte com externes. La utilitat permet controlar els diferents components que formin part d'un projecte, així com les dependències que aquests tinguin, tenint en compte fins i tot les diferents versions que d'un component o d'una biblioteca hi pogués haver.

Maven també permet sincronitzar diferents instal·lacions amb el reposador principal del projecte. Cada cop que el reposador principal sigui actualitzat, Maven serà capaç de sincronitzar cada instal·lació incorporant només els canvis que siguin necessaris.

1.3 Principals problemes a resoldre durant la construcció d'un component

Un cop descrita cada fase de la construcció de components, analitzarem els principals problemes que solen sorgir durant la construcció i apuntarem aquelles solucions que habitualment es fan servir per evitar-los.

1.3.1 Ús d'interfícies i classes

Ja s'ha comentat que en Java és possible declarar interfícies per definir la sintaxi de les operacions del component sense necessitat d'haver-les de codificar. Això presenta el gran avantatge de poder realitzar diverses implementacions de les mateixes operacions, de forma que disposem de diverses alternatives d'un mateix component amb capacitat de treballar plegades en una mateixa aplicació si fes falta.

L'ús d'interfícies, però, incrementa una mica el grau de complexitat de la implementació, ja que fa créixer el nombre d'elements a posar en joc; com a mínim, la interfície i una classe que la implementi. Què passaria, però, si un component estigués format per un nombre considerable d'interfícies i sabéssim que mai arribarien a conviure en una mateixa aplicació diverses alternatives del mateix component? Òbviament, incrementaríem la complexitat del component sense necessitat.

L'eliminació d'interfícies pot ser una opció vàlida només per a aplicacions petites o per a components d'ús específic i molt ben localitzat. Fixeu-vos que les aplicacions mitjanes o grans rarament es modifiquen senceres. Si l'ús d'un mateix component s'estén en diversos mòduls o parts d'una aplicació, la substitució del component en només una de les parts, implicarà la coexistència de diverses alternatives i per tant es farà inevitable l'ús d'interfícies.

L'ús d'interfícies Java assegura també la creació de components estàndards ben definits. Quan el component que desitgem implementar estigui cridat a convertir-se en estàndard de múltiples aplicacions, serà també important usar interfícies per assegurar la interoperabilitat i obligar les implementacions a complir l'estàndard.

Sovint, però, la majoria de solucions opten per un punt intermedi. És a dir, es fan servir interfícies però només les justes per assegurar la interoperabilitat. Normalment els components acaben tenint poques interfícies i moltes més classes, la majoria internes (és a dir, que mai es fan servir fora del component).

Fixem-nos, per exemple, en un component: JDBC. Si consultem el nombre de classes que conté qualsevol de les versions d'un *driver* d'alguna de les bases de dades relacionals actuals, comprovarem que és enorme; en canvi, les interfícies obligades per JDBC són moltes menys (`Driver`, `Connection`, `Statement`, `PreparedStatement`, `ResulSet`, i alguna més de menor importància).

Com a conclusió, podem establir que aquells objectes que traspassin la frontera del component i hagin de ser utilitzats per altres s'obtindran normalment sota l'aparença d'interfície per assegurar la interoperabilitat. En canvi, els objectes interns no necessitaran cap interfície, sinó que referenciaran directament la classe implementadora.

Com veurem més endavant, en casos d'objectes molt consolidats amb una funcionalitat bàsica de contenidor d'informació, podran implementar-se sense interfície per reduir la complexitat de la codificació malgrat que traspassin la frontera dels components.

1.3.2 Accés a les interfícies del component

Prenent com a bona la suposició que els components encapsularan les seves operacions en interfícies, se'ns presenta un problema que caldrà resoldre d'alguna manera. Si desconeixem les classes que finalment acabaran implementant les interfícies del component, com podrem instanciar-les?

A continuació exposarem diverses solucions. Malauradament, no n'hi ha una millor que l'altra, sinó que depenent de les circumstàncies i el context haurem d'escollir la que considerem millor.

Ús del mètode forName de la classe Class

Quan s'ha estudiat l'estàndard JDBC ja s'ha vist la utilització d'aquesta utilitat que aquí recordarem i ampliarem. El llenguatge Java, per poder fer crides als mètodes d'un objecte, necessitarà carregar en memòria la classe on es troben implementats.Habitualment la localització de les classes cridades es realitza mitjançant les sentències import. Recordeu que les sentències import indiquen la ruta on es troben les diferents classes (mitjançant el paquet) que seran cridades en alguna part del codi. Si desconeixem el nom i el paquet de la classe durant la fase d'implementació no serà possible fer servir la sentència import.

Vegeu l'annex "Parametrització d'aplicacions. Tècniques de configuració" per recordar algunes formes de configuració del programari.

Tot i així, Java habilita també una forma que permet carregar classes en memòria de forma dinàmica. És a dir, sense necessitat d'haver de conèixer el nom de la classe durant la codificació. El nom de la classe es resoldrà durant l'execució, i per tant pot afegir-se habitualment com un element de configuració en un fitxer, en una variable de sistema, etc.

La classe `Class` disposa d'un mètode estàtic anomenat `forName`, que a partir d'una cadena de caràcters amb el nom complet d'una classe és capaç de cercar-la i carregar-la en memòria. Imaginem que el sistema tingui una variable anomenada `nomClasse` de tipus `String` amb el nom de la classe a carregar. La crida correcta seria:

```
Class.forName(nomClasse);
```

La càrrega en memòria és important, però no resol encara la instanciació d'objectes de la classe. Per fer-ho, necessitarem invocar `newInstance`. Es tracta d'un mètode que crea un objecte fent servir el constructor per defecte (sense paràmetres) de la classe carregada en memòria. És important, doncs, que la classe concreta disposi d'un mètode constructor per defecte, en cas contrari la invocació de `newInstance` provocarà error.

L'objecte creat serà de la classe que l'instància, però com que mentre codifiquem desconeixem el seu nom, la variable que el referenciï haurà de ser del mateix tipus que la interfície, que sí coneixem.

Imaginem que disposem d'un component que contingui una interfície anomenada `Gestor` i que es trobi implementada per una classe o més classes. Imaginem també que el nom d'una d'aquestes classes s'especifica en un fitxer de propietats anomenat `Gestor.properties`, sota el nom de la propietat configFile. Per poder obtenir un objecte de tipus `Gestor` haurem d'escriure el següent:

```
...
//Llegeix i carrega les propietats
Properties properties = new Properties();
properties.load(new FileReader("Gestor.properties"));
// assigna el nom de la classe a la variable nomClasse
String nomClasse = properties.getProperty("configFile");

//Instancia un objecte i l'assigna a la variable gestor
Gestor gestor=(Gestor)class.forName(nomClasse).newInstance();
```

Òbviament, fent servir aquest sistema podríem intentar d'instanciar objectes de classes que no existissin en el sistema, o no implementessin la interfície, o... És per això que necessitarem embolcallar les crides entre sentències `try-catch`.

```
try {
  //Llegeix i carrega les propietats
  Properties properties = new Properties();
  properties.load(new FileReader("Gestor.properties"));

  // assigna el nom de la classe a la variable nomClasse
  String nomClasse=properties.getProperty("configFile");

  //Instancia un objecte i l'assigna a la variable gestor
  Gestor gestor = (Gestor)
              Class.forName(nomClasse).newInstance();
```

```
12 }catch (ClassNotFoundException ex) {
13    //La classe demanada no existeix al sistema
14    ...
15 }catch (ClassCastException ex) {
16    //La classe demanada no implementa la interfície Gestor
17    ...
18 }catch (InstantiationException ex) {
19    //La classe demanada no disposa de constructor per defecte
20    ...
21 }catch (IllegalAccessException ex) {
22    //La classe demanada no disposa de constructor per defecte
23    ...
24 }catch (IOException ex) {
25    //No es troba el fitxer de propietats per obtenir
26    //el nom de la classe
27    ...
28 }
```

Com es pot veure, és una solució molt flexible però també molt genèrica i per tant exposada a errades de configuració. Per això no sempre s'opta per aquesta solució.

És possible concretar més la instanciació dinàmica d'objectes embolcallant les diferents instanciacions en una classe específica que tingui com a objectiu la creació d'objectes. Recordeu que aquesta mena de classes s'anomenen `Factory`. D'aquesta manera, cada cop que calgui crear un objecte nou n'hi haurà prou d'invocar el mètode instanciador específic.

Depenent de les necessitats, codificarem la classe `Factory` de diferent manera. Si no ens cal mantenir diverses versions del component a l'aplicació i coneixem quina classe instanciarà la interfície, podrem implementar una classe `Factory` constituïda de mètodes que creïn instàncies directament. Imaginem que la classe `ComponentXFactory` sigui la classe encarregada d'instanciar els objectes de la interfície anomenada `ComponentXInterficieN` a través de la classe `ComponentXInterficieNImpl`. El mètode `newComponentXInterficieN` mostra una forma fàcil de codificar-la:

```
1 public class ComponentXFactory{
2     public ComponentXInterficieN newComponentXInterficieN(){
3         return new ComponentXInterficieNImpl();
4     }
5     ...
6 }
```

Sovint els mètodes d'aquestes `Factory` acostumen a ser *static*, perquè les instàncies no disposen d'un estat, sinó que el resultat obtingut depèn únicament de la codificació del mètode.

En cas que la classe implementadora de la interfície no sigui coneguda, o en cas que sigui necessari mantenir diverses versions d'un mateix component, optarem per una `Factory` configurable. La classe disposarà d'un atribut per a cada tipus d'objecte a instanciar, corresponent a cada una de les interfícies del component, que identificarà la classe implementadora.

D'aquesta manera es podran mantenir a l'aplicació tantes instàncies d'una classe `Factory` com sigui necessari. La tècnica d'instanciació usarà els mètodes `forName` i `newInstance` que ja s'han vist. En aquests casos, l'objectiu

principal d'una classe `Factory` és facilitar la instanciació invocant un mètode senzill i lliure d'errors o amb un tractament d'errors més especific. Imaginem de nou que necessitem construir una classe `Factory` anomenada també `ComponentXFactory`, encarregada d'instanciar els objectes de la interfície anomenada `ComponentXInterficieN`.Aquest cop, però, desconeixem la classe que permetrà la instanciació.

```
public class ComponentXFactory{
  private Class classeComponentXInterficieN;
  ...

  //iniciar
  public void iniciar(String nomClasseComponentXInterficieN,
                          ... ){
    try {
      //Instancia una classe per poder instanciar objectes
      classeComponentXInterficieN =
          Class.forName(nomClasseComponentXInterficieN);
    }catch (ClassNotFoundException ex) {
      //La classe demanada no existeix al sistema
      ...
    }
  }
```

Dins la mateixa classe, el mètode `newComponentXInterficieN` s'encarregarà d'instanciar els objectes a partir de la classe configurada.

```
public ComponentXInterficieN newComponentXInterficieN(){
    ComponentXInterficieN ret = null;
    try{
      ret =  classeComponentXInterficieN.newInstance();
    }catch (ClassCastException ex) {
      //La classe demanada no implementa la interfície
      ...
    }catch (InstantiationException ex) {
      //La classe demanada no té constructor per defecte
      ...
    }catch (IllegalAccessException ex) {
      //La classe demanada no té constructor per defecte
      ...
    }

    return ret;
  }
  ...
}
```

A l'aplicació es podran configurar diferents instàncies d'un mateix component creant per a cada una un objecte `Factory` i configurant-los específicament, fent servir el mètode `iniciar`. El nom de les classes es podria obtenir fàcilment a partir d'un fitxer de propietats. Exemple:

```
...
ComponentXFactory factoryDelCompXVer1 =
                new ComponentXFactory();
ComponentXFactory factoryDelCompXVer2 =
                    new ComponentXFactory();
...


public void cofiguracio(){

  try {
```

```
12      //Llegeix i carrega les propietats
13      Properties properties = new Properties();
14      properties.load(new FileReader("ComponentX.properties"));
15
16      //Assigna els noms de les classes en variables
17      String nomVersioModul1=properties.getProperty("versMod1");
18      String nomVersioModul2=properties.getProperty("versMod2");
19      ...
20
21      //Iniciar les instàncies de factory
22       factoryDelCompXMod1.iniciar(nomVersioModul1, ...);
23       factoryDelCompXMod2.iniciar(nomVersioModul2, ...);
24
25      //Configurar els diferents mòduls que usin el components.
26      //Cada mòdul rebrà la versió del component que requereixi.
27      modul1.setComponentXINterficieN(factoryDelCompXMod1.
28                      newComponentXInterficieN());
29
30      modul2.setComponentXINterficieN(factoryDelCompXMod2.
31                      newComponentXInterficieN());
32    }catch (IOException ex) {
33      //No es troba el fitxer de propietats per obtenir
34      //el nom de la classe
35      ...
36    }
37  }
```

Després, des de qualsevol mètode d'algun dels mòduls es podrà fer una crida a cada operació de la interfície. Treballaran amb les mateixes operacions, però instanciaran classes diferents.

De vegades, aquesta tècnica fins i tot s'aplica en l'obtenció de les instàncies de les classes `Factory`. Sobretot si es tracta d'algun component d'una alta reusabilitat, o amb vocació d'estàndard. En comptes de definir una classe `Factory` es definirà una interfície que tindrà aquest paper i podrà ser implementada per diferents classes. Quelcom semblant a una `fàbrica de fàbriques`.

1.3.3 Granularitat dels components

Reprenguem de nou el paral·lelisme amb els productes mecànics. Diem que el motor del cotxe és un component perquè es pot construir de forma independent, és fàcil d'ancorar en el xassís i connectar a la resta de components com ara el motor d'arrancada, els mecanismes de transmissió... i, a més, quan sigui necessari es podrà substituir per un altre motor equivalent.

Agafem, però, el sistema elèctric (cables, llums, bateria, etc.). Podem parlar en aquest cas de component? La resposta és **no**, perquè és impossible assemblar tot el sistema elèctric de cop, ni substituir-lo completament. Una altra cosa seria si parléssim de la bateria, d'un far, del quadre de fusibles etc.; això sí que serien components. El mateix raonament podem aplicar a la carrosseria: no és un component, però en canvi sí que ho és una porta (amb tot el seu mecanisme intern), o el capó, o la xapa lateral davantera esquerra.

Per la banda contrària, tampoc podem considerar un component la pintura amb la qual pintarem el capó o el cargol amb el qual ancorarem el motor al xassís. La seva funció, tot i que important, no és prou rellevant en el conjunt del producte com per considerar-lo component. Són elements massa simples i genèrics. Si haguéssim de construir un cotxe partint d'elements tan simples, l'esforç que hauríem de realitzar i la probabilitat d'equivocar-nos durant el muntatge serien molt elevades.

Granularitat

El terme granularitat prové de l'argot fotogràfic. En fotografia es parla de gra fi quan els punts que componen una fotografia són de dimensions molt petites i es necessita un nombre molt gran de punts per compondre la fotografia. Per contra, es parla de gra gros quan la dimensió dels punts és gran i, per tant, el nombre de punts que compondran la mateixa fotografia serà molt més reduït. El gra fi s'associa amb granularitat baixa i el gra gros,amb granularitat alta.

Direm que els elements simples tenen un grau de **granularitat baixa** perquè la seva funcionalitat és reduïda i es necessiten molts elements per aconseguir completar tota la funcionalitat. A mida que la complexitat dels elements creixi, direm que augmenta el grau de granularitat. En aquest sentit, podem considerar que els productes acabats tenen la màxima granularitat. Els components solen tenir una **granularitat intermèdia**, malgrat que hi pot haver també components grans.

Les aplicacions informàtiques segueixen la mateixa lògica. Si es tracta d'aplicacions orientades a objecte, els objectes del model i les utilitats més bàsiques i genèriques del llenguatge representarien els elements de granularitat baixa, mentre que el grau de granularitat dels components seria més elevat i fins i tot gran, com en el cas dels *plugins*.

És important trobar un cert equilibri entre la mida de l'aplicació i la dels components. Si el nombre de components d'una aplicació és mol reduït, la complexitat de cada una d'elles serà probablement alta i, per tant, resultaran costosos de fabricar. Per contra, si el nombre creix, la complexitat de cada component baixarà però s'incrementarà la dificultat del procés d'assemblatge.

És evident que les mateixes tècniques de descomposició que apliquem a les aplicacions o als productes mecànics podrien aplicar-se als components quan aquests tinguin un grau de granularitat elevat. És a dir, els components més complexos poden estar compostos, a la vegada, per altres components més petits.

1.3.4 Descomposició dels components en classes

És important adonar-se que malgrat la granularitat dels components pugui ser alta, la seva funcionalitat haurà de romandre descomposta en diverses classes, evitant la creació de megaclasses que dificultarien la implementació i el manteniment del component.

Per assegurar una descomposició adient, la funcionalitat dels components grans pot dividir-se en diverses interfícies, les quals agrupen les operacions de forma coherent seguint uns rols perfectament identificats. Així, per exemple, els connectors JDBC estan dividits en vàries interfícies de *rols perfectament identificats*. La `Connection` agrupa les operacions que permeten gestionar la connexió i desconnexió a un SGBD, l'`Statement` que gestiona l'escriptura de sentències SQL, i la seva execució o la `ResultSet`, que permet recuperar les dades obtingudes de l'execució d'una consulta.

Depenent del disseny del component, les diferents interfícies es poden anar instanciant de forma encadenada, com passa als connectors JDBC (el *Driver* instancia objectes `Connection`, que generen objectes `Statement` o derivats i a partir dels quals s'obtindran instàncies de `ResultSet`) o bé, com ja s'ha vist, disposar d'una interfície o classe `Factory` que permeti la instanciació de qualsevol de les interfícies que composin el component.

Malgrat tot, a vegades, el disseny del component pot aconsellar de reunir tota la funcionalitat en un única interfície. Això, però, no hauria d'evitar la descomposició del component en diverses classes. És clar que la interfície haurà d'estar implementada per una classe, però si la granularitat del component és elevada i correm el risc de codificar una megaclasse, caldrà delegar la funcionalitat en classes internes. Així, la classe implementadora de la interfície serviria només de punt d'entrada al component, la qual mantindria actives totes les instàncies de les classes que suportin realment la funcionalitat del component, i a cada invocació d'alguna operació delegaria a la instància corresponent per dur a terme el càlcul demanat.

La interfície que agrupa tota la funcionalitat del component sol anomenar-se `Façana`, i la classe que la implementa també, perquè actuarien com la cara visible del component amagant les classes internes que realment encapsularien la funcionalitat del component. El principal objectiu d'aquesta tècnica és facilitar l'ús del component amagant la complexitat estructural i oferint només una visió funcional. Als programadors que usin aquest tipus de component no els cal conèixer l'estructura de classes que componguin el component, ni el paper de cada una d'elles. Només cal que coneguin quines operacions se li poden demanar al component i quina és la seva sintaxi.

1.3.5 Components i reutilització

Malgrat que un dels objectius principals dels components és la reutilització, crear components reutilitzables no és una tasca fàcil. La reutilització implica un alt nivell d'abstracció, la conseqüència de la qual acostuma a ser la creació de components molt genèrics o de components més complexos del que podrien requerir les aplicacions en què es vulguin utilitzar.

Els components reutilitzables genèrics solen acollir un conjunt d'operacions que poden usar-se en diversitat de situacions. Per exemple, diem que són components reutilitzables genèrics determinats tipus de dades (i les operacions associades) que permeten manipular la informació d'una determinada manera. Ens referim, per exemple, al tipus de dades (i operacions) que la majoria de llenguatges moderns disposen per tractar col·leccions d'informació (*arrays*, *strings*, piles, cues, llistes ordenades, arbres, taules de dispersió, etc.).

Quan parlem de components reutilitzables la complexitat dels quals s'ha hagut d'incrementar, volem fer referència a aquells components als quals s'hi ha hagut d'afegir tipus de dades, classes o operacions extres per assegurar la intolerabilitat

i permetre la seva utilització en múltiples situacions. Entre aquest tipus de components reutilitzables trobarem els components d'accés a dades com ara JDBC o les interfícies d'usuari com ara `Awt` o `Swing` del llenguatge Java. Fixeu-vos que aquests components generen elements un amica artificials, per exemple els tipus `Layout` o els tipus `DataModel`, que faciliten l'adaptació del component gràfic a qualsevol dispositiu o per a qualsevol tipus de dada.

Com es pot deduir, els llenguatges de programació incorporen una gran quantitat de components reutilitzables, els quals solen utilitzar-se de base per construir components de programari més específics.

Cal adornar-se que a mida que incrementem el grau d'especialització del component, aquests són més difícilment reutilitzables. Tot i això, cal destacar l'enorme esforç que la indústria està fent per generar components específics reutilitzables. Són els anomenats components verticals o de domini. Entre d'altres, podem situar, per exemple, els components especialitzats en geolocalització o en interpretació d'imatges. Es tracta de solucions específiques que s'embolcallen de forma que puguin ser usades en diverses situacions. Així, per exemple, un component de geolocalització podria fer-se servir per desenvolupar una aplicació de navegació de mapes de carreteres o en un programa de gestió d'una flota de transports, en una aplicació de realitat virtual, etc.

Sigui com sigui, es tracta de components que per la seva flexibilitat solen requerir una fase d'adaptació a les aplicacions de què hauran de formar part i a aquells components amb què hauran d'interaccionar.

Anomenem també components de domini a les entitats del model de l'aplicació, atès que es tracta de components totalment especialitzats i contextualitzats a l'aplicació. Es tracta de components difícilment reutilitzables; tot i això, si es construeixen amb una perspectiva reutilitzadora es trobaran punts en comú amb altres aplicacions de l'empresa.

1.3.6 Adaptació dels components

Quan construïm una aplicació basada en components reutilitzables resulta realment difícil no haver de realitzar cap canvi. D'una banda, caldrà fer adaptacions específiques per especialitzar els components al context concret de l'aplicació. De l'altra, quan sigui necessari que diversos components dissenyats de forma independent interaccionin entre ells, caldrà ajustar les dades d'entrada i sortida per tal de fer-les compatibles.

Especialització incremental

El procés d'especialització té com a objectiu la construcció de components molt més adaptats al context i les necessitats de l'aplicació que haguem d'implementar. Normalment es parteix de components genèrics la funcionalitat dels quals no

representa en si mateixa una solució, sinó que caldrà construir noves operacions a partir de les que aquests ofereixin. Per exemple, els connectors JDBC són una solució massa genèrica per usar-los sense adaptar.

La quantitat de codi que cal escriure cada cop que fem servir un connector JDBC per fer una inserció o modificació de dades o per obtenir algunes de les dades emmagatzemades al'SGBD és considerable, però sovint, les dades a intercanviar entre l'aplicació i l'SGBD solen estar força clares, de manera que no resulta gaire difícil de pensar en un component específic que defineixi les principals operacions d'intercanvi d'informació (per exemple, la inserció, modificació o obtenció d'una entitat determinada de l'aplicació).

Patró de disseny

Entenem per *patró de disseny* la descripció que explica la solució a un problema plantejat i que pot extrapolar-se per solucionar altres problemes semblants. Els patrons no s'apliquen només als dissenys de programari, sinó també a qualsevol branca de l'enginyeria i l'arquitectura. Per exemple, la descripció de com cal actuar per poder aixecar un edifici sense que acabi ensorrant-se segur que es va tenir en compte per fer la reforma de la plaça de braus de les Arenes de Barcelona. Aquestes indicacions constitueixen un patró de construcció per reformar edificis als quals cal manipular els fonaments, o desplaçar totalment o parcial l'estructura arquitectònica. Quan la descripció fa referència a la forma com cal dissenyar i implementar en tal o en tal altre programa informàtic, parlem de patrons de disseny de programari.

De vegades, durant el procés d'adaptació ens pot convenir fer una especialització incremental. És a dir, en comptes de passar directament del component més genèric al més específic podem anar construint nous components cada cop més especialitzats fins obtenir un component perfectament adaptat a l'aplicació que calgui implementar. Aquest tipus de disseny permet crear nous components reutilitzables més específics que els components de partida, però encara amb cert grau de generalització o flexibilització que permetrà la reutilització en altres aplicacions que segueixin un patró de disseny semblant.

L'especialització incremental és molt adequada per crear aplicacions que es vulguin desenvolupar seguint un cicle de vida en espiral o iteratiu, ja que a cada volta es pot avaluar la possibilitat de crear un nou embolcall més especialitzat o bé dividir alguna especialització ja realitzada en dos components per tal de poder fer servir-ne un com a base d'altres desenvolupaments.

Marcs de treball

Els marcs de treball o *frameworks* de programari són components d'alt nivell que responen a dos objectius. D'una banda, actuen de connectors entre components més simples, i de l'altra aporten una solució específica a un problema determinat que pot ser reutilitzada en diverses aplicacions. L'exemple més proper de marc de treball el tenim amb JPA. Es tracta d'un marc de treball reutilitzable en múltiples aplicacions que ofereix una solució d'alt nivell al problema de la persistència d'objectes d'una aplicació en un sistema de base de dades relacional.

Generalment, els marcs de treball requereixen poca adaptació, però per contra l'aplicació que l'utilitzi ha d'adaptar-se a la solució proposada. Per exemple, si

una aplicació decideix fer servir JPA, la base de dades on es faci la persistència haurà de ser relacional. A més, hauran d'especificar-se les entitats usant el sistema de mapatge definit per l'estàndard.

Els marcs de treball solen fer-se servir molt en aplicacions basades en prototips i també en cicles de vida incrementals, perquè permeten un desenvolupament molt ràpid. Obtenim resultats espectaculars en molt poc temps.

Adaptació de la informació intercanviada

Un dels problemes als quals ens haurem d'enfrontar quan implementem una aplicació composta de diversos components que interactuïn entre ells serà la compatibilitat dels tipus de dades d'entrada i sortida de les operacions definides en els components.

Una de les solucions més comunes és la creaciód'un adaptador per a cada component amb les mateixes operacions que l'original, de manera que l'adaptador actuï de recol·lector de la informació d'entrada i sortida facilitant els canvis de tipus necessaris per complir amb els requisits del component. D'aquesta manera es pot definir un conjunt de tipus de dades estàndard de l'aplicació (en el cas dels components de persistència aquests tipus serien les entitats definides en el model de l'aplicació) com a moneda d'intercanvi per a cada component.

Pel que fa als components de persistència, si el model és estable i únic, la solució plantejada ja és prou bona. Ara bé, de vegades trobem aplicacions (per exemple, les distribuïdes) en què les classes del costat client podrien ser diferents de les del costat servidor, tot i implementar una mateixa entitat. En aquestes ocasions caldrà treballar fent servir interfícies.

L'ús d'interfícies en l'especificació del model dóna molta més llibertat i flexibilitat a l'hora d'enfrontar-se a modificacions, però per contra incrementa la complexitat del disseny i la implementació.

Lògica d'un model

Entenem per lògica o negoci d'un model les operacions i procediments que es poden realitzar sobre les seves dades. Hi ha models amb lògiques molt simples, bàsicament operacions d'assignació i obtenció de la informació continguda a l'estat de les seves instàncies (mètodes accessors), però també hi ha models rics en altres procediments de control, interacció o càlcul. En aquests casos, diem que el model posseeix una lògica complexa.

Existeix encara una tercera solució aplicada sobretot en aquelles aplicacions que hagin de treballar amb models que tinguin una lògica complexa. No sempre pot interessar (per raons de seguretat, per evitar errades o simplement per simplificar la implementació) el trasllat de les instàncies del model, amb tota la seva lògica complexa, als diferents components. En aquest casos podem utilitzar objectes de transferència d'informació. Cada component treballaria amb les seves pròpies classes i instancies de manera que els objectes de transferència permetessin la sincronització de la informació comuna.

Es tractaria d'objectes que internament contenen tota la informació que el model original pugui necessitar, però, per exemple, sense la lògica complexa del model, només les dades.

Lògicament, aquest tipus de solució necessitarà d'algun procés que permeti copiar les dades des del model de l'aplicació als objectes de transferència i a l'inrevés. Podem crear un component encarregat de realitzar aquesta transferència de dades o bé podem implementar un mètode específic d'intercanvi d'informació en cada classe que representi una entitat.

1.4 Creació de components amb Java

Malgrat que la construcció de components no hauria de dependre del llenguatge utilitzat, el cert és que alguns llenguatges com Java defineixen un model estàndard per crear components de qualitat adaptats a les característiques del llenguatge.

1.4.1 JavaBeans

Sota el nom de **JavaBeans**, l'empresa Sun Microsystems ha plasmat la seva idea de com s'implementen els components de programari. De fet, un `JavaBean` no és res més que una classe de Java normal, la qual ha de complir un seguit de convencions: per exemple,ha de tenir un constructor per defecte, no ha de tenir accés públic als seus atributs sinó que, en cas que es puguin manipular, caldrà fer-ho sempre mitjançant els accessors d'escriptura i lectura.

Aquests simples requisits, en combinació amb la potencialitat del llenguatge , aporten el conjunt de característiques que permeten definir un model de component estàndard per a Java. Recordeu que el llenguatge Java, quan instancia qualsevol objecte, carrega en memòria també molta informació referent a la classe (metainformació). La convenció que els `JavaBeans` disposin sempre d'un constructor per defecte permet crear instàncies d'objectes a partir d'aquesta metainformació, invocant el mètode `newInstance`.

```
1 Class.forName(nomClasse).newInstance();
```

L'ús d'accessors per manipular l'estat de les instàncies permet controlar de forma independent els accessos de lectura i els d'escriptura. No passaria així si l'accés a l'atribut fos públic. Quan l'accés als atributs no és directe l'especificació dels `JavaBeans` els anomena propietats.

A més, l'ús d'accessors també permet afegir validacions, controls o processos associats a un dels accessors. Aquesta és una manera senzilla d'afegir als components la capacitat de gestionar esdeveniments o bé de fixar limitacions a l'hora de manipular l'estat dels objectes .

La metainformació que Java manté durant l'execució de les seves aplicacions permet fer execucions de mètodes sense que el programador hagi de conèixer prèviament a quina classe pertanyen. És el que es coneix com a *processos d'introspecció o de reflexió*. Això possibilita que biblioteques com JAXB o JPA puguin obtenir i manipular l'estat de qualsevol objecte sense gaire intervenció del programador.

En resum, direm que els principals aspectes que els `JavaBeans` suporten són els següents:

- Manipulació de l'estat de les instàncies dels `JavaBeans` a través de propietats, sobre les quals es podran descriure limitacions d'accés, de validació o de control en general.
- Gestió d'esdeveniments lligada als canvis que es vagin produint en les instàncies dels `JavaBeans` durant les execucions.
- Processos d'introspecció que possibiliten la creació, la personalització de noves instàncies dels `JavaBeans` i la interacció amb altres components o eines d'una forma totalment dinàmica (en temps d'execució).
- Capacitat de persistència. És a dir, ha de ser possible emmagatzemar l'estat del component en un suport d'emmagatzematge i poder-lo recuperar sense perdre informació durant el procés. Cal dir que és necessari que les classes dels `JavaBeans` que requereixin persistència implementin la interfície `Serializable`.

Malgrat que hi ha la tendència a definir els `JavaBeans` únicament com a components reutilitzables de la **Interfície Gràfica d'Usuari**, aquesta és una visió esbiaixada i restrictiva. Segurament, la raó d'aquesta confusió ha estat el fet que els components gràfics són components horitzontals molt estesos, altament reutilitzables i fàcils d'integrar en qualsevol aplicació, però la resta de components en Java també haurien de seguir les convencions descrites pels `JavaBeans`.

Una altra confusió força comuna és la que limita el desenvolupament de JavaBeans no gràfics a les aplicacions distribuïdes. Concretament sota la forma d'*Entreprise JavaBeans* (EJB). La dificultat a l'hora de desenvolupar aplicacions distribuïdes ha fet que Java realitzés una gran esforç en especificar components que les suportessin i va batejar aquestes especificacions com a EJB. Els components de les primeres versions tenien consideracions molt específiques, i eren tan poc flexibles que el seu ús es confinava exclusivament a les aplicacions distribuïdes, però la darrera versió (coneguda com a EJB3) ha aconseguit trencar aquest vincle incorporant components usats en qualsevol tipus d'aplicació però adaptant-los també a les consideracions distribuïdes.

És tracta de components que s'han desenvolupat amb independència dels EJB, però l'èxit que han aconseguit ha obligat a contemplar-los dins l'especificació. Ens referim, entre d'altres, als components d'accés a dades, anomenats també DAO (*Data Acces Object*), els quals són el principal objecte d'estudi d'aquest mòdul.

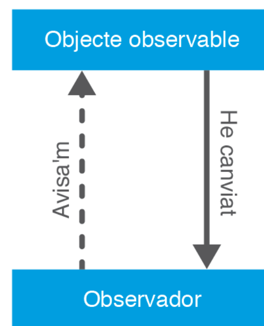
1.4.2 Esdeveniments

Entenem per esdeveniment digital qualsevol canvi que succeeixi durant l'execució d'un programa. La gestió d'esdeveniments intenta aprofitar aquest canvis per invocar procediments específics.

Des de la versió 1.1 del JDK, Java disposa de forma estàndard de classes i interfícies que permeten gestionar la captació d'esdeveniments en el moment que es produeixin per tal de poder associar processos condicionats a determinats esdeveniments.

La gestió d'esdeveniments es basa en un patró de disseny de programari anomenat Observer (figura 1.1). Aquest patró defineix un escenari amb dos elements,l'objecte que cal observar i l'observador. L'objecte a observar també s'anomena font d'esdeveniments perquè és allà on es produeixen els canvis i s'originen els esdeveniments.

Figura 1.1. Esquema de com funciona el patró Observer



Per poder observar, l'observador haurà d'indicar a l'objecte observable que l'avisi quan hi hagi un canvi. L'observador restarà passiu fins que sigui avisat, moment en què activarà els processos que calguin.

A la pràctica, aquest patró necessita un tercer element, l'esdeveniment. Es tracta de l'element encarregat de notificar als observadors quin canvi s'ha produït.

Vegem ara com podem implementar amb Java un sistema d'aquestes característiques. El JDK disposa de la classe `java.util.EventObject`. Cal considerar-la com la classe base de la qual en derivarà qualsevol tipus d'esdeveniment. Es tracta d'una classe que espera rebre en el seu constructor la font de dades de l'esdeveniment, i d'aquesta manera l'observador podrà analitzar els canvis soferts i actuar conseqüentment.

Si a cada notificació necessitem enviar a més de la font d'esdeveniments altres dades complementàries, podem implementar classes derivades d'`EventObject` que rebin les dades necessàries durant la instanciació.

La font d'esdeveniments, o classe d'objectes observables, serà una classe normal de Java amb la particularitat que haurà de tenir una operació per associar observadors i una altra per eliminar-los.

Necessitarem també definir una interfície amb un únic mètode que usarem per realitzar les notificacions. Qualsevol classe que hagi de fer d'observador de la font d'esdeveniments haurà d'implementar la interfície per aconseguir la compatibilitat. Per poder definir una jerarquia diversa, Java disposa d'una interfície sense mètodes anomenada `java.util.EventListener`. Malgrat que no és obligatori, és aconsellable que tots els observadors pertanyin a la mateixa jerarquia i, per tant,en derivin.

Veiem un exemple senzill. Disposarem d'una classe amb un comptador i un mètode anomenat `incrementaValor` que afegirà una unitat al valor del comptador cada cop que s'invoqui. A més, el comptador s'instanciarà amb un valor màxim a partir del qual deixarà de comptar. El comptador acceptarà observadors que implementin la interfície `ControlValorListener`. Aquesta interfície declara un mètode anomenat `fontIncrementada` que rep un esdeveniment de tipus `ValorIncrementatEvent`.

L'esdeveniment s'instanciarà passant-li la font de dades i un valor enter que representi les unitats incrementades des de la darrera notificació. El codi serà semblant a aquest:

```
public class ValorIncrementatEvent extends EventObject{
    private int increment=0;

    public ValorIncrementatEvent(Comptador o, int increment) {
        super(o);
        this.increment = increment;
    }

    @Override
    public Comptador getSource(){
        return (Comptador) super.getSource();
    }

    public int getIncrement() {
        return increment;
    }
}
```

La interfície dels observadors:

```
public interface ControlValorListener extends EventListener{
    public void fontIncrementada(ValorIncrementatEvent event);
}
```

La classe `Comptador` que representarà la font d'esdeveniments:

```
public class Comptador {
    //Llista d'observadors
    private ArrayList<ControlValorListener>
                  listeners = new ArrayList();
    //valor màxim del comptador
    private int valorMaxim= 1000;
    //valor del comptador
    private int valor=0;

```

```
10      //Constructor per defecte
11      public Comptador() {
12      }
13
14      /**
15       * Constructor al qual se li passa el valor màxim
16       * del comptador.
17       * @param valorMaxim és el valor màxim del comptador
18       */
19      public Comptador(int valorMaxim){
20          this.valorMaxim= valorMaxim;
21      }
22
23      /**
24       * Incrementa el valor en una unitat i cada 5
25       * unitats notifica l'increment als observadors.
26       */
27      public void incrementaValor(){
28          if(valor<valorMaxim){
29              valor++;
30              if(valor%5==0){
31                  notificaIncrement();
32              }
33          }
34      }
35
36      /**
37       * Obté el valor màxim del comptador
38       * @return el valor màxim del comptador
39       */
40      public int getValorMaxim() {
41          return valorMaxim;
42      }
43
44      /**
45       * Obté el valor del comptador
46       * @return valor del comptador
47       */
48      public int getValor() {
49          return valor;
50      }
51
52      /**
53       * Afegeix un observador per ser notificat
54       * @param ctrl observador a afegir
55       */
56      public void afegirObservador(ControlValorListener ctrl){
57          listeners.add(ctrl);
58      }
59
60      /**
61       * Elimina l'observador passat per paràmetre de la llista
62       * d'observadors a notificar.
63       * @param ctrl  observador a eliminar
64       */
65      public void eliminarObservador(ControlValorListener ctrl){
66          listeners.remove(ctrl);
67      }
68
69      // notificació dels canvis a tots els observadors
70      // assignats
71      private void notificaIncrement(){
72          ValorIncrementatEvent event =
73                  new ValorIncrementatEvent(this, 5);
74          for(ControlValorListener x: listeners){
75              x.fontIncrementada(event);
76          }
77      }
78  }
```

Per poder fer una prova crearem dos observadors que calcularan el percentatge que el valor del comptador representa respecte del valor màxim. A més, un d'ells només realitzarà el càlcul si l'increment acumulat supera les 20 unitats:

```
public class ObservadorIncrements1
            implements ControlValorListener{
  private double percentatge;

  @Override
  public void fontIncrementada(
                ValorIncrementatEvent event) {
     percentatge = event.getSource().getValor()*100.0
                     /event.getSource().getValorMaxim();
  }

  public double getPercentatge() {
     return percentatge;
  }

  @Override
  public String toString(){
     return (percentatge + "%");
  }
}

public class ObservadorIncrements2
                 implements ControlValorListener{
    private double percentatge;
    private int increment=0;

    @Override
    public void fontIncrementada(
                    ValorIncrementatEvent event){
        increment+=event.getIncrement();
        if(increment>=20){
            percentatge = event.getSource().getValor() *100.0
                /event.getSource().getValorMaxim();
            increment=0;
        }
    }

    public double getPercentatge() {
        return percentatge;
    }

    @Override
    public String toString(){
        return (percentatge + "%");
    }
}
```

Farem una prova per veure com funciona codificant el següent algoritme:

```
public static void main(String[] args) {
    Comptador font = new Comptador(2000);
    ObservadorIncrements1 obs1 = new ObservadorIncrements1();
    ObservadorIncrements2 obs2 = new ObservadorIncrements2();

    //assigna els observadors
    font.afegirObservador(obs1);
    font.afegirObservador(obs2);

    //incrementem fins arribar al valor màxim mostrant
    // els resultats dels comptadors i dels observadors
    while(font.getValor()<font.getValorMaxim()){
        font.incrementaValor();

        System.out.println("Valor: " + font.getValor());
```

```
16          System.out.println("Percent 1: " + obs1.getPercentatge());
17          System.out.println("Percent 2: " + obs2.getPercentatge());
18      }
19  }
```

La gestió d'esdeveniments és molt útil per treballar amb components, ja que permet que aquests interactuïn sense necessitat d'haver de saber durant la codificació quins seran exactament els components que interactuaran.

La gestió d'esdeveniments permet una assignació dinàmica. Fixeu-vos que des de la perspectiva de la font de dades no es necessari saber quina classe farà d'observador. N'hi ha prou que implementi la interfície esperada.

1.4.3 Propietats

Els `JavaBeans` identifiquen les propietats amb un nom i representen aquells atributs de l'estat que poden tenir efectes a considerar en l'aspecte o en la conducta de les seves instàncies.

No existeix cap element físic a les classes dels `JavaBeans` que identifiqui les propietats. De fet, es reconeixen perquè coincideixen amb aquells atributs que disposen d'accessors de lectura o escriptura. Els accessors se solen definir anteposat els prefixos *get* o *set* al nom de l'atribut.

Aquesta convenció és molt útil per fer servir en entorns que treballin amb llenguatges de tipus script com els entorns Web, ja que coneixent el nom de la propietat es dedueix el nom dels accessors, els quals es poden invocar usant introspecció.

L'ús d'accessors permet controlar l'àmbit d'accés amb independència de si es tracta d'un accés de lectura o d'un accés d'escriptura. A més, fent servir accessors és fàcil crear una gestió d'esdeveniments per controlar els canvis a les propietats.

Abans d'entrar en detall en la gestió de canvis de les propietats fent servir esdeveniments, analitzem algunes singularitats de les propietats segons els tipus de valor que emmagatzemin o segons si es tracta de valors simples o indexats (llistes i vectors de dades).

Com ja s'ha comentat, els noms dels accessors d'atributs amb valors simples es generaran anteposat els prefixos *get* o *set* al nom de l'atribut. El primer representa el mètode de lectura del valor de l'atribut:

```
1  tipusAtribut get<NomAtribut>()
```

I el segon, el mètode d'escriptura de l'atribut:

```
1  void set<NomAtribut>(tipusAtribut valor)
```

Imaginem que volem crear un atribut de tipus `Integer` anomenat `duresaAigua`. Les definicions bàsiques dels accessors d'aquest atribut serien:

```
Integer getDuresaAigua(){
   return this.duresaAigua;
}

void setDuresaAigua(Integer valor){
   this.duresaAigua=valor;
}
```

Ara bé, si l'atribut és específicament de tipus booleà, la convenció Java indica que el seu accessor de lectura pot anteposar la partícula *is* en comptes de *get*. Així, un atribut de tipus booleà anomenat *empty* generaria els següents accessors:

```
Boolean isEmpty(){
   return this.empty;
}

void setEmpty(Boolean valor){
   this.empty=valor;
}
```

Tot i que la forma clàssica seria també correcta:

```
Boolean getEmpty(){
   return this.empty;
}
```

La convenció permet també una generació específica d'accessors d'atributs *multivalents* (*arrays* o llistes), a més a més de les formes clàssiques, per tal de suportar l'accés a cada un dels elements emmagatzemat a l'atribut *multivalent*.

Imaginem que disposem d'un atribut de tipus *array* anomenat `temperaturesMensuals` que contédotze registres `Doubles` amb les temperatures mitjanes de cada mes. Imaginem, també, que disposem d'un segon atribut (`precipitacions`) que mantingui associades a les diferents viles i ciutats de Catalunya la seva precipitació anual.

A banda de les formes clàssiques:

```
Double[] getTemperaturesMensuals(){
   return this.temperaturesMensuals ;
}

void setTemperaturesMensuals(Double[] vector){
   this.temperaturesMensuals=vector;
}

Map<String, Integer> getPrecipitacions(){
   return this.precipitacions;
}

void setPrecipitacions(Map<String, Integer> map){
   this.precipitacions=valor;
}
```

Podem fer coexistir accessos que permetin l'accés a les dades simples:

```
Double getTemperaturesMensuals(int index){
   return this.temperaturesMensuals[index] ;
}
```

```
4
5  void setTemperaturesMensuals(int index, Double vector){
6     this.temperaturesMensuals[index]=vector;
7  }
8
9  Integer getPrecipitacions(String ciutat){
10    return this.precipitacions.get(ciutat);
11 }
12
13 void setPrecipitacionsString ciutat, Integer valor){
14    this.precipitacions.put(ciutat, valor);
15 }
```

L'accés indexat és realment important definir-lo quan necessitem accedir individualment a les dades de la col·lecció, ja que normalitza els diferents tipus d'accés.

Cal entendre que el concepte de propietat és quelcom ampli i abstracte, i no només fa referència als valors dels atributs. Podem també definir propietats calculades sense necessitat que existeixi una correspondència directa amb cap atribut.

Imaginem, per exemple, que tenim una classe que enregistri els clients allotjats en un hotel. Suposem que contingui almenys dos atributs, un amb la capacitat màxima que l'hotel és capaç de suportar i l'altre amb el nombre total de clients que té en cada moment l'hotel. Seria possible definir la propietat `taxaOcupacio`de la següent manera:

```
1 Double getTaxaOcupacio(){
2     return clientsAllotjats * 100.0 / capacitat;
3 }
```

De la mateixa manera, podem fer servir els mètodes d'escriptura per controlar les assignacions i assegurar que aquestes siguin sempre coherents. Per exemple, si sabem que el sou d'un treballador no podrà ser mai negatiu, podem afegir un control que asseguri que això no passarà mai:

```
1 void setSou(Double sou){
2     if(sou>0){
3        this.sou=sou;
4     }else{
5        ferQuelcom();
6     }
7 }
```

Caldria matisar, però, aquesta possibilitat. Sovint, a l'hora de programar caiem en la temptació de voler-ho controlar tot. La pràctica demostra que aquesta és una tasca impossible i que sovint no aporta una major qualitat però si un cost més elevat. Cal anar amb cura a l'hora de decidir què cal controlar i què no. De fet, és preferible un component que tingui una bona documentació que clarifiqui exactament l'escenari en què s'ha de moure per tal que les seves operacions tinguin sempre èxit i siguin eficients, que no pas un component que controli tota la casuística però que disposi d'una documentació pobra. Fins i tot, tenint una documentació adequada, l'excessiu control dificulta també la reutilització a causa de l'encariment del producte i la sobredimensió que pugui agafar el component.

Hem vist el patró "Observador" en el punt "Esdeveniments" d'aquest mateix apartat.

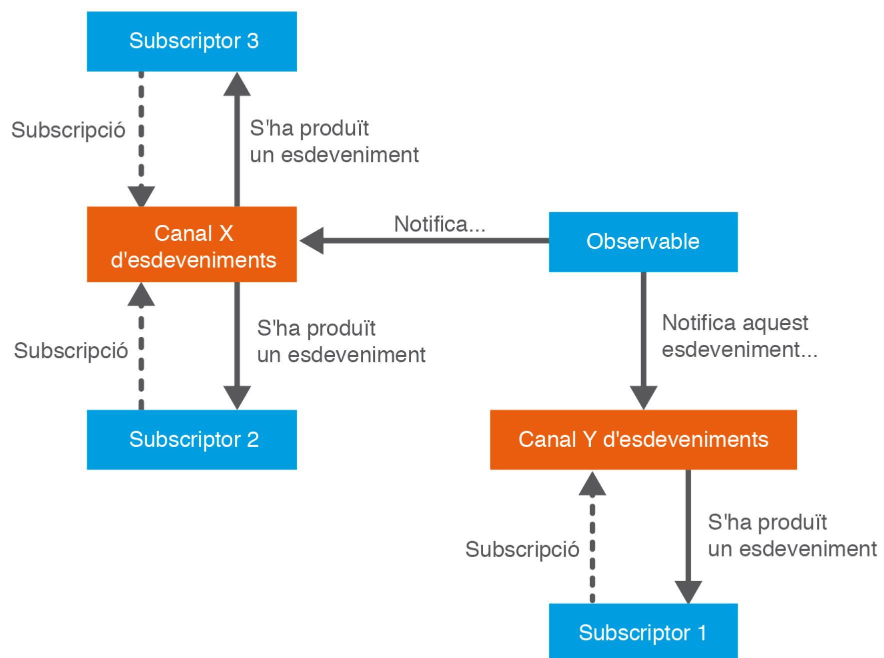
Finalment, també és important constatar l'ús dels mètodes accessors per poder definir un sistema de gestió d'esdeveniments lligats als canvis de valor de les

propietats. Es pot fer servir el patró `Observador`, però el llenguatge Java disposa en la seva versió estàndard d'un seguit d'eines que faciliten la construcció de sistemes gestors d'esdeveniments en `JavaBeans`.

Les propietats com a origen d'esdeveniments

En comptes d'aplicar estrictament el patró Observador, s'aplica una variant anomenada `Publicació-subscripció` que facilita la gestió de múltiples esdeveniments i de múltiples observadors dins una mateixa classe (figura 1.2). El patró Observador es modifica lleugerament afegint un quart element, el canal d'esdeveniments. Així, l'objecte observable pren el paper de publicador d'esdeveniments. Els observadors prenen el paper de subscriptors, els quals se subscriuen a les notificacions que arriben exclusivament de determinats esdeveniments i no d'altres (canals d'esdeveniments).

Figura 1.2. Esquema de com funciona el patró publicació-subscripció



Els canals d'esdeveniments juguen un paper clau de classificació dels subscriptors. D'aquesta manera s'evita haver d'informar sempre a tots els subscriptors, limitant les notificacions als subscriptors associats exclusivament a un canal determinat.

La classificació en canals respon bàsicament al tipus de paper que els subscriptors jugaran respecte el component observat. `JavaBeans` contempla dos tipus de notificacions, les purament informatives, que tenen com a missió exclusiva informar del canvi d'una propietat, i les de control, que a més d'informar permeten decidir si el canvi és coherent i per tant factible, o per contra és incoherent i no es podrà realitzar.

D'altra banda, si el `JavaBean` té definides moltes propietats, ens pot interessar disposar almenys d'un canal per a cada propietat. Cal tenir en compte, però, que aquesta no és una condició obligatòria, ja que en cas que el component tingui poques propietats podria resultat més senzill disposar d'un únic canal i fer una selecció durant el propi llançament de l'esdeveniment que permeti decidir, en funció de la propietat afectada, quins processos cal activar.

Tipus de canals contemplats en les biblioteques estàndards dels JavaBeans

Java contempla de forma estàndard dos tipus de canals lligats als `JavaBeans`. Cada canal té associat un tipus d'interfície específica que hauran d'implementar els subscriptors.

Un dels canals està implementat per la classe `PropertyChangeSupport`. L'objectiu d'aquesta classe és notificar als subscriptors que hi ha hagut un canvi en alguna de les propietats per tal que puguin decidir les accions a emprendre. La notificació és purament informativa, i les accions empreses pels subscriptors no afectaran l'assignació del nou valor.

Els subscriptors associats a aquest canal han d'implementar la interfície `PropertyChangeListener`, la qual disposa d'un únic mètode anomenat `propertyChange`, el qual rep un esdeveniment de tipus `PropertyChangeEvent`.

L'altre tipus de canal es troba implementat per la classe `VetoableChangeSupport`. Aquest és un canal pensat per donar suport a subscriptors que puguin vetar (d'aquí el seu nom) l'assignació del nou valor, impedint que aquesta es porti a terme.

Els subscriptors associats a aquest canal han d'implementar la interfície `VetoableChangeListener`, que disposa d'un mètode anomenat `vetoableChange` que rep un esdeveniment de tipus `PropertyChangeEvent`, però a diferència de l'anterior està previst que aquest mètode llanci una excepció de tipus `PropertyVetoException` en cas de no acceptar el canvi de valor de la propietat.

Anem a veure ara com es relacionen cada una d'aquestes classes i interfícies. Comencem pel conjunt `PropertyChangeSupport`, `PropertyChangeListener` i `PropertyChangeEvent`.

El constructor del canal ha de rebre per paràmetre una instància de la font d'esdeveniments. Com que normalment els canals els instanciarà una font de d'esdeveniments, caldrà instanciar aquest tipus de canal fent una crida similar a aquesta:

```
1 propertySupport = new PropertyChangeSupport(this);
```

Els canals `PropertyChangeSupport` disposen d'un mètode per associar subscriptors. El mètode s'anomena `addPropertyChangeListener` i segueix la sintaxi següent:

```
void addPropertyChangeListener(PropertyChangeListener listener)
```

També disposa d'un mètode per desvincular-los un cop associats. La sintaxi és semblant:

```
void removePropertyChangeListener(PropertyChangeListener listener)
```

El mètode anomenat `firePropertyChange` permetrà a aquest canal indicar que s'ha produït un canvi en alguna de les propietats del `JavaBean`. La invocació d'aquest mètode forçarà la creació d'un esdeveniment de tipus `PropertyChangeEvent` amb les dades passades per paràmetre, i seguidament s'invocarà el mètode `propertyChange` de cada subscriptor per notificar-los el canvi. No cal que el programador enviï les notificacions.

```
propertySupport.firePropertyChange(nomPropietat,
                  oldValue, newValue);
```

Vegem-ho amb un exemple. Imaginem que treballem amb un `JavaBean` que representi un empleat d'una companyia i que de moment només ens cal notificar els canvis de les propietats dels empleats actius per refrescar les finestres on es visualitzin les seves dades per tal de donar una visió coherent en totes les finestres obertes. Suposem que la classe disposa només de tres propietats.En primer lloc el NIF, de només lectura, que s'assignarà en el moment de la instanciació i en què no es permetrà fer-hi canvis durant l'execució. Això significa que no caldrà fer notificacions degudes a aquesta propietat. A banda del NIF, l'empleat mantindrà també el nom i el sou. Ambdues són propietats susceptibles de ser canviades. En tractar-se de només dues propietats decidim crear un únic canal i un únic subscriptor que rebrà totes les notificacions.

La classe `Empleat`

```
public class Empleat implements Serializable {
    public static final String PROP_NIF = "nif";
    public static final String PROP_NOM = "nom";
    public static final String PROP_SOU_BASE = "souBase";
    private String nif;
    private String nom;
    private double souBase;
    transient private PropertyChangeSupport propertySupport;

    protected Empleat() {
        propertySupport = new PropertyChangeSupport(this);
    }

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

    /**
     * @return the nif
     */
    public String getNif() {
        return nif;
    }

    /**
     * @return the nom
```

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

    public void setNom(String value) {
        String oldValue = getNom();
        nom = value;
        propertySupport.firePropertyChange(PROP_NOM, oldValue, getNom());
    }

    /**
     * @return the souBase
     */
    public double getSouBase() {
        return souBase;
    }

    /**
     * @param souBase the souBase to set
     */
    public void setSouBase(double souBase){
        double oldValue = getSouBase();
        this.souBase = souBase;
        propertySupport.firePropertyChange(PROP_SOU_BASE,
                           oldValue, souBase);
    }

    public void addPropertyChangeListener(
                    PropertyChangeListener listener) {
        propertySupport.addPropertyChangeListener(listener);
    }

    public void removePropertyChangeListener(
                    PropertyChangeListener listener) {
        propertySupport.removePropertyChangeListener(listener);
    }
}
```

La classe que implementi `PropertyChangeListener` haurà de codificar el mètode `propertyChange`, el qual rebrà un esdeveniment de tipus `PropertyChangeEvent`. La sintaxi del mètode serà la següent:

```
public void propertyChange(PropertyChangeEvent pce)
```

Per poder extreure informació de l'esdeveniment disposem bàsicament de tres mètodes: `getPropertyName`, que obté el nom de la propietat on s'ha produït el canvi; `getOldValue`, la invocació del qual permetrà conèixer quin valor tenia la propietat abans del canvi, i `getNewValue`, que permetrà saber el valor de la propietat un cop produït el canvi.

Imaginem que volem crear un subscriptor sensible als canvis de les propietats *nom* i *sou base*. En aquest cas, les accions realitzades consisteixen a mostrar els canvis via consola, però seria possible realitzar qualsevol altra acció.

```
public class CanviaNomOSouListener implements
                    PropertyChangeListener {

    @Override
    public void propertyChange(PropertyChangeEvent pce) {
        if(pce.getPropertyName().equals(Empleat.PROP_NOM)){
            System.out.print("El nom ha canviat de: ");
            System.out.print(pce.getOldValue());
```

```
            System.out.print(" a ");
            System.out.println(pce.getNewValue());
        }else if (pce.getPropertyName().equals(
                        Empleat.PROP_SOU_BASE)){
            System.out.print("El sou base ha canviat de: ");
            System.out.print(pce.getOldValue());
            System.out.print(" a ");
            System.out.println(pce.getNewValue());
        }
    }
}
```

Fixeu-vos que en aquest cas fem servir un únic subscriptor per controlar dues propietats. Usem una instrucció alternativa per decidir l'acció en funció de la propietat modificada.

Tot i això, seria possible declarar diversos canals pel control de cada una de les propietats. Imaginem que la classe `Empleat` tingués una altra propietat anomenada `triennis` per emmagatzemar el nombre de triennis d'antiguitat del treballador. En comptes de modificar el subscriptor, podem decidir afegir un nou canal específic per controlar la propietat `triennis`.

```
public class Empleat implements Serializable {
    ...
    public static final String PROP_TRIENNIS = "triennis";
    ...
    private int triennis=0;
    ...
    transient private PropertyChangeSupport propertyTriennisSupport;

    protected Empleat() {
       propertySupport = new   PropertyChangeSupport(this);
       propertyTriennisSupport = new
                    PropertyChangeSupport(this);
    }
    ...

    /**
     * @param triennis the triennis to set
     */
    public void setTriennis(int triennis) {
        int oldValue = getTriennis();
        this.triennis = triennis;
        propertyTriennisSupport.firePropertyChange(PROP_TRIENNIS,

            oldValue,
                                triennis);
    }

    public void addPropertyTrienisChangeListener(
                     CanviaTriennisListener listener){
        propertyTriennisSupport
                .addPropertyChangeListener(listener);
    }

    public void removeTrienisPropertyChangeListener(
                     CanviaTriennisListener listener){
        propertyTriennisSupport
            .removePropertyChangeListener(listener);
    }
```

Així, la funció de control de canvis de la propietat se simplifica, ja que només afecta una única propietat.

```java
public class CanviaTriennisListener implements
                    PropertyChangeListener{

    @Override
    public void propertyChange(PropertyChangeEvent pce) {
        System.out.print("Els triennis han canviat de ");
        System.out.print(pce.getOldValue());
        System.out.print(" a ");
        System.out.println(pce.getNewValue());
    }
}
```

Anem finalment a ampliar l'exemple per veure com es pot fer servir el tipus de canal `VetoableChangeSupport`. Com ja s'ha dit, es tracta d'un tipus de canal especial que permet decidir si és o no possible realitzar la modificació d'una propietat determinada. El conjunt de classes i interfícies implicades són: `VetoableChangeSupport`, `VetoableChangeListener`, de nou `PropertyChangeEvent` i l'excepció anomenada `PropertyVetoException`.

La implementació d'aquest tipus de canal és molt similar ala que ja hem estudiat, excepte que cal tenir en compte que és el llançament de l'excepció allò que impedirà realitzar l'assignació. Això significa que la notificació al subscriptor s'haurà de realitzar abans de fer realment el canvi de valor. Anem a veure-ho.

Imaginem que a la classe `Empleat`, a més de tenir les propietats esmentades, disposem també d'un complement salarial, el qual haurà de complir sempre la premissa de no ser superior a la meitat del sou del treballador. En aquest cas, podem implementar un canal de tipus `VetoableChangeSupport` per fer controlar aquesta restricció.

El control dels complements el realitzarem implementant una classe com la que segueix:

```java
public class ControlComplementsListener implements
                    VetoableChangeListener{
    private Empleat empleat;

    public ControlComplementsListener(Empleat empleat) {
        this.empleat = empleat;
    }

    @Override
    public void vetoableChange(PropertyChangeEvent pce)
                    throws PropertyVetoException {
        Double limit = empleat.getSouBase()/2;
        Double nouValor = (Double) pce.getNewValue();
        if(nouValor>limit){
        //Si supera el límit permès llancem una excepció
            throw new PropertyVetoException(
                "El valor dels complements ("
                        + pce.getNewValue()
                + ") supera el límit permès ("
                        + limit + ")", pce);
        }
    }
}
```

La implementació del mètode `setComplement` de la classe `Empleat` es definirà de la següent manera:

```
public void setComplements(double complements)
                  throws PropertyVetoException {
    double oldValue = getComplements();
    vetoableComplementsSupport.fireVetoableChange(PROP_COMPLEMENTS,
                                                  oldValue,
                                  complements);
    this.complements = complements;
    propertySupport.firePropertyChange(PROP_COMPLEMENTS,
                                          oldValue, complements);
}
```

Fixeu-vos en la situació de la invocació del mètode `fireVetoableChange` de la instància `vetoableComplementsSupport` abans de fer l'assignació del nou valor a l'atribut `complements`; així, en cas de llançar-se l'excepció `PropertyVetoException`, s'aconseguiria evitar la modificació.

Tractament d'esdeveniments. Una forma flexible i dinàmica de vincular components

És important ressaltar que els esdeveniments poden permetre la creació de vincles dinàmics entre components d'una forma molt flexible i ràpida. Els esdeveniments es fan servir en pràcticament totes les aplicacions per enllaçar els components de la interfície d'usuari amb els components del model de dades de l'aplicació. Totes les interfícies gràfiques estan preparades per gestionar els esdeveniments que l'usuari provoca amb les seves accions, de manera que la modificació d'un camp de text o la selecció d'una opció en una *combo-box*, o en un conjunt de *radio-button*, es notifiqui a d'altres components per tal que puguin plasmar els canvis originats per l'usuari i engegar els processos requerits.

La gestió d'esdeveniments, però, pot fer-se servir també en altres situacions de dependència, en les quals un component necessiti detectar els canvis provocats en les propietats d'un altres.

Per exemple, és una tècnica útil per definir relacions febles o que puguin anar canviant al llarg del temps, entre classes d'un mateix model de dades. Imaginem, per exemple, un model de dades amb la classe `Empleat` associada a una determinada categoria professional. Els empleats es trobaran associats a diferents projectes segons la seva categoria professional. Podem definir un conjunt de processos per tal que cada cop que un empleat canviï de categoria professional, es comprovi a quants projectes es troba vinculat el treballador i generi un avís a cada un dels caps de projecte per tal que li busquin un substitut. La invocació dels processos es pot vehicular d'una manera molt flexible a partir de la notificació de l'esdeveniment generat per un canvi de categoria.

Trobem exemples també entre components de diversos mòduls sense una vinculació directa. Imaginem que seguint amb el mateix exemple de gestió d'empleats, l'empresa decideix crear una portal d'informació per als seus empleats. Aquests disposaran d'una agenda on es reflectiran els projectes en què treballin, les tasques associades a cada projecte, les fites, etc. És evident que el model de dades de l'agenda serà diferent del de la gestió d'empleats, però ambdós components tindran nexes comuns que els vincularan. Es poden fer servir esdeveniments per

generar automàticament els registres del calendari de l'agenda de cada treballador cada cop que un empleat sigui vinculat a un projecte o cada cop que es modifiqui, s'afegeixi o s'anul·li qualsevol tasca dels projectes on es trobin vinculats els empleats.

La gestió d'esdeveniments també permet flexibilitzar l'adaptació entre components. En aquest sentit, Java disposa de les anomenades *inner class* i *anonymous inner class* per fer adaptacions concretes i molt lligades a determinades classes.

L'ús de classes incrustades permet evitar una excessiva proliferació de classes sense un objectiu funcional des del punt de vista de l'aplicació. A més, si es tracta de classes incrustades anònimes (*anonymous inner class*), l'adaptació pot ajustar-se a cada circumstància i cada línia de codi.

Per exemple, imaginem que necessitem notificar els canvis de determinades propietats de la classe `Empleat` d'una forma específica quan ens trobem realitzant la modificació en un mòdul concret i no en altres. Suposem que la interfície gràfica controli totes les finestres obertes amb les dades d'una mateix empleat i que les finestres disposin d'un mètode anomenat `refrescar`, el qual se li pot passar l'empleat a refrescar. Es tracta d'una notificació específica del mòdul. Cada cop que instanciem un empleat en el mòdul de la interfície gràfica de l'usuari farem el següent:

```
1  empleat = new Empleat("11111111A");
2
3  //Afegim una instància anònima de PropertyChangeListener
4  empleat.addPropertyChangeListener(new PropertyChangeListener(){
5      @Override
6      public void propertyChange(PropertyChangeEvent pce) {
7        FinestraEmpleat[] finestres =
8                 totesLesFinestresObertes(empleat);
9        for(FinestraEmpleat finestra: finestres){
10          finestra.actualitza(empleat);
11       }
12     }
13 });
```

1.5 Components per a la persistència

En aquest aparat distingirem els diferents tipus de components implicats en la persistència de les dades, molts dels quals s'han vist a les altres unitats ("Persistència en fitxers", "Persistència en BDR-BDOR-BDOO" i "Persistència en BD natives XML"). Aquí veurem també com fer-los treballar conjuntament per tal de treure'n el màxim profit.

Les dades de les aplicacions s'encapsulen en les classes del model. Les classes importants del model s'anomenen també **entitats**. Generalment, les entitats es codifiquen com a components independents que en Java s'acostumen a conèixer com a *JavaBeans d'Entitats*.

En estreta relació amb els *JavaBeans d'Entitats*, les aplicacions també necessiten implementar altres components que permetin gestionar les instàncies de les enti-

tats que l'aplicació vagi requerint a cada execució. Són components encarregats de la creació de les instàncies (ja sigui a partir de dades emmagatzemades prèviament o no) i del seu emmagatzematge, assegurant que l'estat de les entitats persisteixi d'una execució a una altra. Generalment els coneixem com a JavaBeans d'accés a dades, objectes d'accés a dades o simplement components d'accés a dades.

Normalment, els components d'accés a dades gestionaran les instàncies d'una única entitat, però depenent de la complexitat del model i el grau de vinculació entre entitats, alguns components poden assumir la gestió de diverses entitats.

Per tal de facilitar el desplegament i la integració amb altres parts de l'aplicació, els components anteriors acostumen a agrupar-se en components més grans a mode de marc de persistència específic de l'aplicació o component d'accés a partir del qual es pot accedir a la resta.

1.5.1 Esdeveniments associats als components d'accés a dades

Els components d'accés a dades tenen una especial rellevància dins l'aplicació perquè totes les instàncies del model passen en els moments clau per les seves mans. Cal entendre per moments clau els moments en què l'estat d'alguna instància es fa persistent o bé els moments en què les dades persistents es reflecteixen a l'estat d'alguna instància existent o acabada de crear.

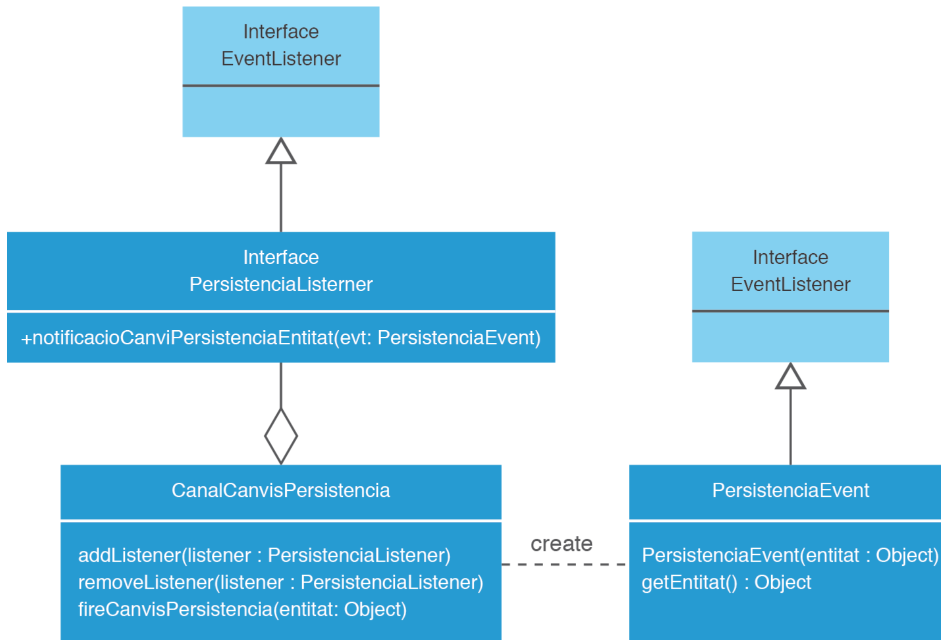
En aquest sentit, pot ser molt interessant disposar de canals de subscripció associats als esdeveniments clau d'aquest tipus de components. D'entrada, distingirem almenys tres tipus d'esdeveniments: la inserció d'una entitat a la base de dades, la seva actualització i la càrrega de les dades emmagatzemades en alguna instància.

La llista indicada podria ser ampliada si fos necessari per a l'aplicació. Per exemple, ens podria interessar distingir entre el procés de creació d'una entitat nova del procés de refresc del d'una entitat emmagatzemada prèviament. O potser ens podria interessar llançar un esdeveniment just abans d'iniciar l'emmagatzematge i un altre just després d'haver-lo acabat.

Implementació d'un gestor d'esdeveniments per a components d'accés a dades

A continuació mostrarem una forma senzilla d'implementar qualsevol dels gestors d'esdeveniments esmentats. Aplicarem el disseny del patró de Publicació-Subscripció que ja heu vist amb anterioritat. Com a exemple, implementarem la gestió d'esdeveniments de càrrega d'instàncies a partir de les dades emmagatzemades. Crearem el canal, l'esdeveniment i l'observador amb els noms que podeu veure a la figura 1.3.

Figura 1.3. Diagrama de classe del gestor d'esdeveniments de la càrrega d'entitats des d'un component de persistència



La interfície `PersistenciaListener` declara un únic mètode `notificacioCanviPersistenciaEntitat`. Cal recordar que l'ús de la interfície assegura la compatibilitat del sistema amb qualsevol classe que la implementi.

```
public interface PersistenciaListener extends EventListener {
   void notificacioCanviPersistenciaEntitat(PersistenciaEvent evt);
}
```

La classe `CanalCanvisPersistencia` farà el paper d'agregadora de les instàncies de `PersistenciaListener` per tal d'invocar-los el mètode `notificacioCanviPersistenciaEntitat` cada cop que s'executi `fireCanvisPersistencia`, la qual cosa s'hauria d'esdevenir només quan es produeixi el canvi que es desitja controlar.

```
public class CanalCanvisPersistencia {
   List<PersistenciaListener> listeners =
            new ArrayList<PersistenciaListener>();

   public void addListener(PersistenciaListener listener) {
       listeners.add(listener);
   }

   public void removeListener(PersistenciaListener listener){
       listeners.remove(listener);
   }

   public void fireCanvisPersistencia(Object entitat){
       if(entitat==null){
           return;
       }
       PersistenciaEvent evt = new PersistenciaEvent(entitat);
       for (PersistenciaListener listener: listeners) {
           listener.notificacioCanviPersistenciaEntitat(evt);
       }
    }
}
```

La invocació del mètode `fireCanvisPersistencia` es realitzarà en el component de persistència de l'aplicació quan s'hagi produït algun canvi que es desitgi controlar.

La implementació de l'esdeveniment `PersistenciaEvent` és força simple, ja que només emmagatzemarà la instància de l'entitat carregada.

```
public class PersistenciaEvent extends EventObject{

    public PersistenciaEvent(Object entitat) {
        super(entitat);
    }

    public Object getEntitat(){
        return getSource();
    }
}
```

Fixeu-vos que la instància s'emmagatzema a l'atribut *source* de la classe estàndard `EventObject`, però per facilitar l'aprenentatge s'afegeix una propietat de només lectura usant el mètode `getEntitat`, que retorna l'entitat afectada invocant `getSource`.

Gestió dels esdeveniments usant JPA

En cas d'implementar els components de persistència fent servir JPA, la gestió d'esdeveniments s'implementa d'una forma totalment diferent a l'explicada. De fet, JPA aprofita les `Anotation` per definir quines classes i quins mètodes caldrà invocar en produir-se un determinat canvi durant els processos depersistència de les entitats.

JPA defineix les anotacions `PrePersist` i `PostPersist` per marcar, respectivament, quins mètodes s'invocaran just en el moment en què una entitat estigui a punt de fer-se persistent (inserció a les taules de la base de dades) o immediatament després d'haver-se fet persistent.

També defineix les anotacions `PreRemove` i `PostRemove` per poder marcar les invocacions a realitzar abans i després d'haver eliminat de la base de dades una instància emmagatzemada.

`PreUpdate` i `PostUpdate` són les anotacions que permeten executar mètodes, just abans i just després d'una actualització de la base de dades.

Finalment, `PostLoad` marcarà quins mètodes caldrà executar just després de carregar una instància amb les dades emmagatzemades al'SGBD.

En general, les anotacions que invoquin mètodes en el moment abans de produir-se un canvi es podran fer servir com a validadors dels canvis, llançant una excepció derivada de `RuntimeException` per impedir el canvi.

Els mètodes anotats tal com s'ha indicat poden formar part de les entitats o bé d'altres classes independents que prendran el paper de subscriptors (listeners) d'alguna entitat. En el cas que els mètodes formin part de la pròpia entitat,

només caldrà anotar correctament els mètodes. No serà necessària cap altra adaptació. Si volguéssim implementar quelcom semblant al control de la classe `Empleat`, implementant les notificacions i validacions en mètodes propis de la classe `Empleat`, hauríem de fer el següent:

```
@Entity
public class Empleat implements Serializable {
    @Id
    private String nif;
    private String nom;
    private double souBase;
    private double complements;

    ...

    @PostPerssit
    @PostUpdate
    private void canvisEnviatsAlSGBD() {
            System.out.println("S'ha enviat a l'SGBD: ");
            System.out.print("\tnom: ");
            System.out.println(nom);
            System.out.print("\tsou: ");
            System.out.println(souBase);
        }
    }

    @PrePersist
    @PreUpdate
    public void validaDades() throws ComplementExceditException {
        Double limit = getSouBase()/2;
        if(complements>limit){
    //Si supera el límit permès llancem una excepció
            throw new ComplementExceditException(
                "El valor dels complements ("
                        + complements
                + ") supera el límit permès ("
                       + limit + ")");
        }
    }
}
```

Tot i l'aparent sofisticació, aquest sistema presenta el problema que es barregen els procediments propis de les entitats d'aquells que només tinguin com a objectiu la notificació o la validació de determinats canvis.

Per solucionar aquest problema, JPA permet agrupar els mètodes a invocar en classes independents. Per poder fer efectives les invocacions quan es produeixin (o estiguin a punt de produir-se) els canvis, caldrà associar les classes a l'entitat que es desitja controlar fent servir l'anotació `EntityListeners`. Fent servir aquesta modalitat, caldria implementar la classe `Empleat` tal com segueix:

```
@Entity
@EntityListeners({NotificadorEmpleat.class, ValidadorEmpleat.class})
public class Empleat implements Serializable {
    @Id
    private String nif;
    private String nom;
    private double souBase;
    private double complements;
    ...
}
```

Òbviament, caldrà implementar les classes indicades a l'anotació:

```
public class NotificadorEmpleat {
    @PostPerssit
    @PostUpdate
    private void canvisEnviatsAlSGBD(Empleat emp) {
            System.out.println("S'ha enviat a l'SGBD: ");
            System.out.print("\tnom: ");
            System.out.println(emp.getNom());
            System.out.print("\tsou: ");
            System.out.println(emp.getSouBase());
        }
    }
}

public class ValidadorEmpleat {
    @PrePersist
    @PreUpdate
    public void validaDades(Empleat emp)
               throws ComplementExceditException {
        Double limit = emp.getSouBase()/2;
        if(emp.getComplements()>limit){
        //Si supera el límit permès llancem una excepció
            throw new ComplementExceditException(
                "El valor dels complements ("
                        + emp.getComplements()
               + ") supera el límit permès ("
                        + limit + ")");
        }
    }
}
```

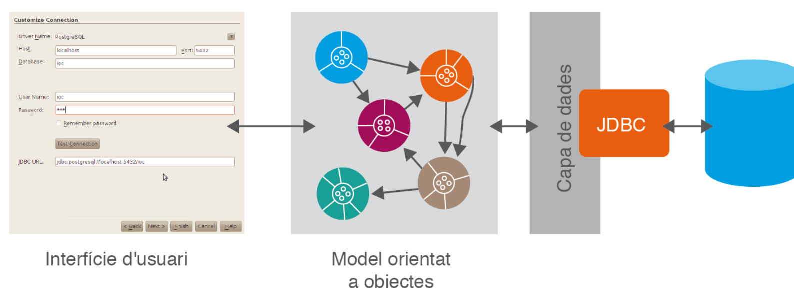
2. Exemples d'implementació de components de persistència

Els components de persistència que implementarem treballaran sobre JDBC i contra un SGBD PostgreSQL. De nou, per qüestions de dimensió, resulta impossible d'implementar un component de persistència per a cada un dels sistemes que s'han estudiat (persistència en fitxers binaris o xml, persistència en SGBD relacionals usant eines de mapatge, en bases de dades orientades a objecte o gestors de bases de dades xml). Sortosament, l'extrapolació a partir d'un d'ells no és difícil amb els coneixements que ja heu adquirit.

D'aquesta manera el nucli de l'aplicació (sovint anomenat també lògica) treballa exclusivament amb instàncies d'entitats, no pas amb dades simples. És el que també s'anomena *model orientat a objectes*. La persistència, però, no forma part del model, ja que és una necessitat del sistema informàtic per tal de conservar les dades malgrat que finalitzi l'execució de l'aplicació. Normalment les aplicacions disposen d'un conjunt de classes encarregades específicament de la persistència, i alliberen els models d'objectes d'aquesta tasca, de manera que aquests només es responsabilitzaran de la lògica de l'aplicació.

És a dir, les entitats o objectes del model no acostumen a connectar amb l'SGBD ni encarregar-se de la seva persistència, sinó que aquestes tasques es deleguen a unes classes específiques que solem qualificar de classes de la capa de dades o de persistència, com podeu veure en la figura 2.1. En aquesta, Els objectes no connecten directament amb la font, sinó que ho fan a través d'un intermediari, el component de persistència o d'accés a dades, que s'encarregarà de crear i posar en circulació totes les instàncies del model necessàries per fer-la funcionar. També s'encarrega de mantenir la coherència entre el model i l'SGBD a mida que l'execució vagi avançant.

FIGURA 2.1. Arquitectura d'una aplicació orientada a l'objecte amb accés a una font de dades



Quan dissenyem una aplicació cal també que definim quines operacions usarem per dur a terme el procés de persistència de les seves entitats. Moltes d'aquestes operacions tindran una sintaxi molt semblant, amb l'única diferència de l'entitat afectada per l'operació. Ens referim a funcions com inserir una instància,

modificar les dades persistents a partir d'una instància, eliminar una entitat de l'SGDB, etc. En canvi, d'altres seran operacions molt adaptades a l'entitat. Podem necessitar trobar els clients assignats a una determinada zona, o bé aquells que hagin comprat un determinat article o els que pertanyin a tal o tal altre sector. Podem necessitar conèixer quants productes diferents comercialitzem d'un mateix article, o quins productes estan a punt d'esgotar-se, o un gran ventall de diverses formes de recuperació de les entitats específiques.

Sigui com sigui, les operacions relatives a la persistència solen tenir una sintaxi força estable, ja que depenen molt del model. Per això és força comú, encara que no imprescindible, gestionar el component de persistència amb interfícies Java que declarin la sintaxi de les operacions a més de les classes que les implementin. D'aquesta manera, resultarà força fàcil intercanviar les classes per tal d'adaptar-les a les diverses fonts d'emmagatzematge. És a dir, si usem interfícies serà més fàcil canviar la implementació, de manera que en comptes de suportar JDBC, suporti JPA o bé alguna base de dades orientada a objectes, o bé XML, etc. L'ús d'interfícies permet independitzar la lògica de l'aplicació del sistema de persistència finalment utilitzat.

Tenint en compte aquesta premissa, dissenyarem el nostre component usant interfícies. Cal procurar també que la sintaxi de les operacions sigui independent del sistema de persistència usat. Per això evitarem incloure paràmetres o retorns específics d'algun dels sistemes en aquelles operacions publiques que hagin de ser usades per altres components externs o per la pròpia aplicació.

2.1 Especificació de les interfícies d'accés al component de persistència

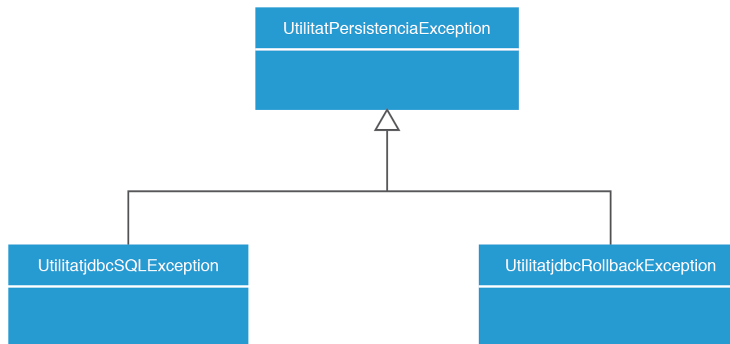
Les interfícies d'accés al component han de permetre obrir i tancar la connexió amb el sistema d'emmagatzematge. A més, per aconseguir obrir la connexió acostuma a ser necessària alguna configuració que identifiqui la font de dades. Cal tenir en compte, però, que cada sistema de persistència utilitza diverses formes de configuració. Per tal d'aconseguir quelcom de genèric declararem un mètode anomenat iniciar, sense paràmetres ni retorn, la invocació del qual activarà algun mecanisme de configuració de la connexió.

```
/**
* Assigna els valors de configuració de la connexió a partir
* d'un sistema d'inicialització concretat a cada instància
* d'aquesta interfície.
* @throws UtilitatPersistenciaException. Si la configuració
* falla es llançarà una excepció.
*/
void iniciar() throws UtilitatPersistenciaException;
```

Seguint el mateix raonament, les excepcions llançades des d'una interfície d'accés a un component de persistència hauran de ser també excepcions genèriques per tal que puguin suportar qualsevol de les possibles errades amb independència del sistema de persistència escollit (figura 2.2). Per aconseguir-ho farem servir

una jerarquia d'excepcions. A l'arrel de la jerarquia disposarem de la classe més genèrica. Usarem sempre aquesta classe en les declaracions dels mètodes de les interfícies externes. En el nostre cas, l'excepció situada a l'arrel de la jerarquia s'anomena `UtilitatPersistenciaException`.

Figura 2.2. Jerarquia d'excepcions pròpia per facilitar la captura i el tractament de les excepcions degudes a alguna errada durant la connexió a qualsevol sistema de persistència



La jerarquia només s'estén per a excepcions específiques JDBC, però quan es desenvolupi per a altres sistemes caldrà ampliar-la amb altres excepcions específiques.

Usarem el sistema de documentació de Java per descriure amb cura la funcionalitat del mètode. A més, cal indicar, quan sigui possible, quines condicions poden provocar error i si es coneixen quines altres provocarien un mal funcionament del mètode.

De forma similar, especificarem els mètodes obrir i tancar:

```
/**
 * Obté una connexió a partir de les dades configurades en
 * invocar el mètode iniciar. Un cop obtinguda
 * la connexió, aquesta romandrà oberta i es podrà reciclar
 * fins que no s'invoqui el mètode tancar. Abans
 * d'invocar aquest mètode, cal haver invocat el mètode
 * iniciar. En cas que s'invoqui obrir sense cap
 * invocació prèvia del mètode, iniciar produirà un error que
 * es materialitzarà en el llançament de
 * UtilitatPersistenciaException.
 * El mètode iniciar només s'ha d'invocar una única vegada.
 * Un cop invocat, si les dades de configuració són correctes
 * i el sistema persistent es troba accessible, no hauria de
 * produir-se cap error.
 * @throws UtilitatPersistenciaException. Es produirà un
 * llançament d'aquesta excepció en cas que s'intenti invocar
 * aquest mètode sense haver invocat mai el mètode
 * <i>iniciar</i>. També es produirà error en cas que el
 * sistema de persistència no es trobi accessible o en el
 * cas que la configuració assignada durant la invocació del
 * mètode iniciar no sigui correcta.
 */
void obrir() throws UtilitatPersistenciaException;

/**
 * Tanca la connexió al sistema de persistència oberta per la
 * invocació del mètode <i>obrir</i>. En cas que la connexió
 * no es trobi oberta o que el sistema de persistència no
 * sigui accessible es produirà un error.
 * En cas d'error, aquest serà enregistrat en un fitxer, però
 * no es llançarà cap excepció. S'intenta així evitar una
 * excessiva imbricació de sentències try-catch sense perdre
```

```
33  * cap informació dels errors produïts.
34  */
35 void tancar();
```

Fixeu-vos que aquestes tres funcions serien comunes a qualsevol aplicació, ja que al marge del model de dades o del sistema de persistència utilitzat sempre serà necessari configurar, connectar amb el sistema persistent i en acabat, realitzar la desconnexió.

Podeu disposar del codi sencer del'exemple implementat a l'annex "Codi dels exemples" que il·lustren la unitat "Components d'accés a dades".

Per tal de facilitar la reutilització del codi separarem aquestes funcions de les més específiques segregant-les en una interfície genèrica i realitzant la implementació en classes també genèriques. Anomenarem la interfície `GestorPersistencia`. Qualsevol altra interfície específica de cada model de dades haurà d'estendre aquesta.

Els mètodes més específics dependran directament del model de dades. Per cada entitat necessitarem probablement un mètode per inserir les instàncies al sistema de persistència, un altre per actualitzar-ne les dades (modificar el sistema persistent) quan les instàncies emmagatzemades canviïn. Un tercer mètode hauria de permetre l'eliminació de les instàncies de la font de dades permanent. També seria necessari algun mètode per recuperar una instància de la font de dades i algun per recuperar el conjunt d'instàncies emmagatzemades segons el tipus i d'altres característiques.

2.1.1 Organització de les interfícies d'accés al component

Abans de començar a definir els mètodes més específics d'accés al component de persistència caldrà prendre algunes decisions sobre l'estructura d'interfícies que es vol que el component tingui, ja que en funció de la decisió la sintaxi del mètode es veurà alterada.

La complexitat del model de dades requerirà, molt probablement, que quan codifiquem els mètodes d'accés els implementem en diverses classes per tal d'evitar la creació d'una megaclasse que s'encarregui de tota la funcionalitat. Ara bé, aquesta subdivisió, tan necessària per organitzar el codi, no té per què reflectir-se també en les interfícies. De fet, una solució força comuna és la d'agrupar tota la funcionalitat d'accés al component, en una única interfície (malgrat que la implementació es realitzi en múltiples classes). El que s'aconsegueix d'aquesta manera és simular una *façana* d'accés, la qual amagarà les classes i interfícies internes utilitzades en el component.

Per usar el component de persistència no serà necessari conèixer l'estructura interior, només caldrà estar familiaritzat amb les classes i interfícies que traspassin el component i els mètodes declarats per suportar la funcionalitat. És el que normalment en el món del disseny de programari es coneix com a patró de tipus *façana*.

Els avantatges que suposa són evidents, i sobretot són relatius a la corba d'aprenentatge dels implementadors que hagin d'usar el component. Tot i això, si el nombre de classes que representen entitats arriba a ser molt gran i la quantitat de mètodes d'accés creix desmesuradament, caldria plantejar-se dividir la façana en diverses parts.

Quan el nombre de mètodes és elevat, en aquest tipus de component s'hi afegeix un altre problema. No es tracta d'un problema fonamental, però si que pot afectar la corba d'aprenentatge, la productivitat i la taxa d'error dels programadors. Ens referim als noms que es donen als mètodes. Sempre que sigui possible cal fer servir mètodes sobrecarregats per tal de reduir el nombre de noms. Tot i això, de vegades la sobrecàrrega no sempre permet establir les diferències necessàries entre mètodes i no queda altre remei que manipular el nom dels mètodes amb la conseqüent dificultat que això suposa per als programadors.

Una possible alternativa al disseny anterior consisteix a crear components de persistència especialitzats per a cada una de les entitats. D'aquesta manera evitem conflictes de noms i podem crear una sintaxi més o menys estàndard per a la majoria d'operacions d'accés, les quals serien sobreescrites per cada una de les classes encarregades de la persistència d'una de les entitats de l'aplicació.

Les classes paramètriques permeten declarar i codificar mètodes genèrics sense necessitat de definir els tipus de dades que intervindran en la sintaxi.

Cal tenir en compte que aquesta no és una solució fàcil, ja que d'una banda caldrà evitar la proliferació de connexions per a cada un dels minicomponents de persistència, i de l'altra, per aconseguir l'estandardització de les principals operacions caldrà definir de forma paramètrica les classes i interfícies superiors de la jerarquia. A més, com que hem decidit treballar amb interfícies necessitarem un objecte *instanciador* o *fàbrica* de les instàncies dels components.

Per tal que us pugueu fer a la idea de com cal implementar ambdós dissenys, implementarem les dues solucions en el nostre component. Així, crearem un estil façana per gestionar la persistència de les entitats relacionades amb els comercials i els clients. Per afinitat, també inclourem la persistència dels proveïdors i les entitats que hi estan relacionades, com ara les zones o el sectors.

Per a la resta d'entitats usarem components de persistència específics per a cada una d'elles.

En l'apartat "Implementació dels components de persistència basats en JDBC" hi trobareu més detalls sobre les classes que els implementaran, aquí només remarcarem l'ús d'una interfície encarregada d'instanciar els components.

2.1.2 Estratègia façana en el disseny de components de persistència

Sota el nom de `GestorPersistenciaEmpresesIPersonal` declararem la interfície façana amb tota la funcionalitat que calgui per gestionar la persistència de les classes `Pais`, `Poblacio`, `Zona`, `Sector`, `Client`, `Comercial` i `Proveidor`. Es tracta d'una interfície derivada de `GestorPersistencia`, i per tant inclou també per herència els tres mètodes d'aquesta (`iniciar`, `obrir` i `tancar`).

```
1 public interface GestorPersistenciaEmpresesIPersonal
2                                     extends GestorPersistencia {
```

A continuació mostrarem alguns dels mètodes més significatius per a la declaració de la interfície, acompanyant cada un d'ells de la descripció de la seva funcionalitat, de les condicions d'error i, si s'escau, de les condicions necessàries per a la correcta invocació del mètode.

Començarem pel mètode `inserir`. Com que el paràmetre del mètode és la pròpia entitat en cada cas, aplicarem sobrecàrrega. És a dir, tots els mètodes s'anomenaran inserir, i es distingiran entre ells per la tipologia del paràmetre que representarà l'entitat a inserir. Vegem l'entitat `Pais`:

```
1  /**
2   * S'insereix al sistema persistent el país passat per
3   * paràmetre. Per poder inserir l'entitat amb èxit, cal que
4   * no existeixi a la font de dades cap altra entitat
5   * emmagatzemada amb el mateix identificador. Quan això passi
6   * es produirà un error i es llançarà una excepció de tipus
7   * UtilitatPersistenciaException.
8   * @param pais és el país a inserir.
9   * @throws UtilitatPersistenciaException. Es produirà el
10  * llançament d'aquesta excepció quan el sistema persistent
11  * no sigui accessible, quan no hi hagi una connexió oberta o
12  * quan s'intenti emmagatzemar un país amb el mateix nom que
13  * un altre que ja estigui emmagatzemat.
14  */
15 public void inserir(Pais pais) throws UtilitatPersistenciaException;
```

Gràcies a la sobrecàrrega, per a la resta d'entitats es declararà de forma molt semblant, només caldrà matisar aquells aspectes característics de cada entitat. Així, per exemple, per a l'entitat `Client` que disposa del codi de client o identificador de l'entitat, indicarem que el valor d'aquest se substituirà sempre per un valor únic i diferent a qualsevol altre client ja emmagatzemat.

```
1  /**
2   * S'insereix al sistema persistent el client passat per
3   * paràmetre. Abans de fer efectiva la inserció s'assignarà
4   * de forma automàtica un valor únic per a l'atribut id (codi
5   * de client) d'aquesta entitat, amb independència del valor
6   * que l'atribut tingui abans de la invocació d'aquest
7   * mètode.
8   * El client no s'inserirà amb èxit si el valor del seu nif
9   * coincideix amb el d'algun altre client ja emmagatzemat.
10  * Quan això passi es produirà un error i es llançarà una
11  * excepció de tipus UtilitatPersistenciaException.
12  * @param pais és el país a inserir.
13  * @throws UtilitatPersistenciaException. Es produirà el
14  * llançament d'aquesta excepció quan el sistema persistent
15  * no sigui accessible, quan no hi hagi una connexió oberta o
16  * quan s'intenti emmagatzemar un país amb el mateix nom que
17  * un altre que ja estigui emmagatzemat.
18  */
19 public void inserir(Client entitat)
20             throws UtilitatPersistenciaException;
```

El mètode `modificar`, que actualitzarà la font de dades amb l'estat d'una instància d'alguna de les entitats, presenta el mateix estil, ja que també podem aplicar sobrecàrrega del mètode. Mirem com queda el mètode per a les entitats zona, per exemple:

```
1  /**
2   * Actualitza el sistema persistent amb les dades incloses a
3   * l'estat de l'entitat que es passa per paràmetre.
```

```
4   * L'entitat ha de ser persistent abans de l'actualització.
5   * En cas contrari es produiria un error de tipus
6   * UtilitatPersistenciaException.
7   * @param zona des d'on fer l'actualització.
8   * @throws UtilitatPersistenciaException Es produirà el
9   * llançament d'aquesta excepció quan el sistema persistent
10  * no sigui accessible, quan no hi hagi una connexió oberta o
11  * quan s'intenti actualitzar una zona que no sigui
12  * persistent (és a dir, que no es trobi emmagatzemada en
13  * sistema persistent).
14  */
15 public void modificar(Zona zona)
16             throws UtilitatPersistenciaException;
```

El mètode `eliminar` pot tractar-se també declarant paràmetres sobrecarregats, si fem servir les pròpies instàncies de les entitats per indicar al sistema persistent quins registres haurà d'eliminar per aconseguir que deixi de ser persistent. Fixem-nos ara en el mètode `eliminar` de l'entitat `Comercial`.

```
1  /**
2   * Fa que el comercial que es passa per paràmetre deixi de
3   * ser persistent. Si no existeix cap entitat persistent amb
4   * el mateix identificador que la instància del paràmetre, es
5   * produirà un error i es llançarà una excepció de tipus
6   * UtilitatPersistenciaException.
7   * @param comercial candidat a deixar de ser persistent.
8   * @throws UtilitatPersistenciaException. Es produirà el
9   * llançament d'aquesta excepció quan el sistema persistent
10  * no sigui accessible, quan no hi hagi una connexió oberta o
11  * quan s'intenti eliminar un comercial que no sigui
12  * persistent (és a dir, que no es trobi prèviament
13  * emmagatzemat).
14  */
15 public void  eliminar(Comercial comercial)
16                         throws UtilitatPersistenciaException;
```

Si en comptes de fer servir la instància volguéssim aprofitar el valor de l'identificador de l'entitat per determinar l'objecte a eliminar, no podrem pas sobrecarregar els paràmetres, ja que hi pot haver més d'una entitat fent servir el mateix tipus d'identificador. Malgrat que hi ha diverses maneres de solucionar aquest problema, la forma més senzilla passa per assignar un nom diferent a cada una de les entitats a eliminar. Així, el mètode per eliminar poblacions tindrà la forma següent:

```
1  /**
2   * Fa que la població identificada amb els paràmetres
3   * corresponents al nom de la població i el del seu país,
4   * deixi de ser persistent. Si no existeix cap entitat
5   * persistent identificada amb els paràmetres, es produirà un
6   * error i es llançarà una excepció de tipus
7   * UtilitatPersistenciaException.
8   * @param nomPoblacio és el nom amb què s'identifica la
9   * població.
10  * @param nomPais és el nom que identifica el país on es
11  * troba la població.
12  * @throws UtilitatPersistenciaException. Es produirà el
13  * llançament d'aquesta excepció quan el sistema persistent
14  * no sigui accessible, quan no hi hagi una connexió oberta o
15  * quan no existeixi cap població emmagatzemada identificada
16  * amb el valor dels paràmetres.
17  */
18 public void eliminarPoblacio(String nomPoblacio, String nomPais) throws
       UtilitatPersistenciaException;
```

En canvi, a l'eliminació d'un proveïdor s'haurà d'especificar el següent:

```
/**
 * Fa que el proveïdor identificat amb el codi de proveïdor
 * que es passa per paràmetre deixi de ser persistent. Si no
 * existeix cap entitat emmagatzemada, identificada amb el
 * valor del paràmetre, es produirà un error i es llançarà
 * una excepció de tipus UtilitatPersistenciaException.
 * @param id és el codi del proveïdor que identifica el
 * candidat a deixar de ser persistent.
 * @throws UtilitatPersistenciaException. Es produirà el
 * llançament d'aquesta excepció quan el sistema persistent
 * no sigui accessible, quan no hi hagi una connexió oberta o
 * quan no existeixi cap proveïdor emmagatzemat,
 * identificat amb el valor del paràmetre.
 */
public void eliminarProveidor(int id)
                        throws UtilitatPersistenciaException;
```

L'obtenció d'una instància a partir del valor que la identifica tampoc admet sobrecàrrega. Actuarem de la mateixa manera:

```
/**
 * Obté una instància del sector emmagatzemat i identificat
 * amb l'id que es passa per paràmetre.
 * @param id és el valor que identifica l'entitat que es
 * desitja recuperar.
 * @return Instància de l'entitat recuperada amb les dades
 * emmagatzemades.
 * @throws UtilitatPersistenciaException. Es produirà el
 * llançament d'aquesta excepció quan el sistema persistent
 * no sigui accessible, quan no hi hagi una connexió oberta o
 * quan no existeixi cap sector emmagatzemat,
 * identificat amb el valor del paràmetre.
 */
public Sector obtenirSector(String id)
            throws UtilitatPersistenciaException;
```

Si ens interessa poder refrescar l'estat de les entitats amb les dades emmagatzemades, per si s'han emmagatzemat modificacions durant la vida d'una instància, haurem de definir els mètodes pertinents. Els anomenarem `refrescar`. I com que reben per paràmetre la instància a refrescar, sí que serà possible aplicar sobrecàrrega.

```
/**
 * Refresca els atributs de l'entitat passada per paràmetre
 * amb les dades emmagatzemades de forma persistent.
 * @param entitat és el comercial a refrescar
 * @return el comercial refrescat.
 * @throws UtilitatPersistenciaException. Es produirà el
 * llançament d'aquesta excepció quan el sistema persistent
 * no sigui accessible, quan no hi hagi una connexió oberta o
 * quan no existeixi cap comercial emmagatzemat,
 * identificat amb el mateix valor que la instància del
 * paràmetre.
 */
 public Comercial refrescar(Comercial entitat)
                       throws UtilitatPersistenciaException;
```

També ens serà d'utilitat un mètode que ens indiqui si una entitat es troba ja emmagatzemada o, en cas contrari, encara no és persistent. La sintaxi del mètode amb aquesta funcionalitat serà aquesta:

```
/**
 * Indica si l'objecte passat per paràmetre és persistent. La
 * comprovació de l'existència es basa exclusivament amb el
 * valor del nif del comercial. La resta d'atributs no són
 * rellevants per a la comprovació.
 * @param entitat és el comercial a comprovar.
 * @return cert si l'entitat a comprovar és persistent. Fals
 * en cas contrari.
 * @throws UtilitatPersistenciaException. Es produirà el
 * llançament d'aquesta excepció quan el sistema persistent
 * no sigui accessible, quan no hi hagi una connexió oberta o
 * quan no existeixi cap comercial emmagatzemat,
 * identificat amb el mateix valor que la instància del
 * paràmetre.
 */
public boolean esPersistent(Comercial entitat)
                        throws UtilitatPersistenciaException;
```

Depenent dels requeriments de l'aplicació indicats durant la fase inicial del seu disseny, caldrà definir diversos mètodes de recuperació de les entitats. Per exemple, ens pot interessar cercar un client per `nif` en comptes d'obtenir-lo per codi de client, o bé ens pot interessar trobar els clients d'una zona determinada o que pertanyin a un sector concret. Vegem-ho:

```
/**
 * Obté una instància persistent de client, identificada amb
 * el pNif que es passa per paràmetre.
 * @param pNif és el nif que identifica l'entitat que es
 * desitja recuperar.
 * @return Instància de l'entitat recuperada amb les dades
 * emmagatzemades.
 * @throws UtilitatPersistenciaException. Es produirà el
 * llançament d'aquesta excepció quan el sistema persistent
 * no sigui accessible, quan no hi hagi una connexió oberta o
 * quan no existeixi cap client emmagatzemat, amb el nif
 * indicat per al paràmetre.
 */
public Client obtenirClient(String pNif)
                        throws UtilitatPersistenciaException;

/**
 * Obté una llista de tots aquells clients emmagatzemats que
 * pertanyen a la zona passada per paràmetre.
 * @param zona d'on obtenir els clients.
 * @return llista de clients persistents que pertanyen a la
 * zona. En cas que no hi hagi cap client que pertanyi a la
 * zona es retornarà una llista buida.
 * @throws UtilitatPersistenciaException. Es produirà el
 * llançament d'aquesta excepció quan el sistema persistent
 * no sigui accessible o quan no hi hagi una connexió oberta.
 */
public List<Client> obtenirClients(Zona zona)
                       throws UtilitatPersistenciaException;

/**
 * Obté una llista de tots aquells clients emmagatzemats que
 * treballin al sector productiu passat per paràmetre.
 * @param sector d'on obtenir els clients.
 * @return llista de clients persistents que treballen al
 * sector productiu que es passa per paràmetre. En cas
 * que no hi hagi cap client del sector especificat es
 * retornarà una llista buida.
 * @throws UtilitatPersistenciaException. Es produirà el
 * llançament d'aquesta excepció quan el sistema persistent
 * no sigui accessible o quan no hi hagi una connexió oberta.
 */
```

```
43 public List<Client> obtenirClients(Sector sector)
44                         throws UtilitatPersistenciaException;
```

Finalment, volem també fer èmfasi en la importància de no crear instàncies d'entitats de forma massa lleugera, per tal de facilitar la sincronització. Per això també atorgarem al gestor de persistència la funció d'instanciador d'entitats. Per a cada constructor de cada entitat crearem un mètode que instanciï les entitats d'acord amb els paràmetres requerits per a cada constructor, i a la vegada les insereixin al sistema de persistència abans de retornar-les. Per exemple:

```
1  /**
2   * Crea una instància nova de zona i la inicialitza amb
3   * els paràmetres corresponents al nom i la descripció de la
4   * zona. La instància s'emmagatzemarà de forma persistent
5   * abans de ser retornada. En cas que ja existís una zona
6   * emmagatzemada amb el mateix nom, es llançaria una excepció
7   * indicant que l'entitat que s'ha intentat crear no es pot
8   * inserir perquè ja existeix una altra zona amb el mateix
9   * identificador.
10  * @param id és el nom de la zona que cal instanciar.
11  * @param descripcio és la descripció de la zona que cal instanciar.
12  * @return instància de zona creada i emmagatzemada.
13  * @throws UtilitatPersistenciaException. Es produirà el
14  * llançament d'aquesta excepció quan el sistema persistent
15  * no sigui accessible, quan no hi hagi una connexió oberta o
16  * quan s'intenti instanciar una zona amb el mateix
17  * identificador que una altra que ja estigui emmagatzemada.
18  */
19 public Zona novaZona(String id, String descripcio)
20                         throws UtilitatPersistenciaException;
```

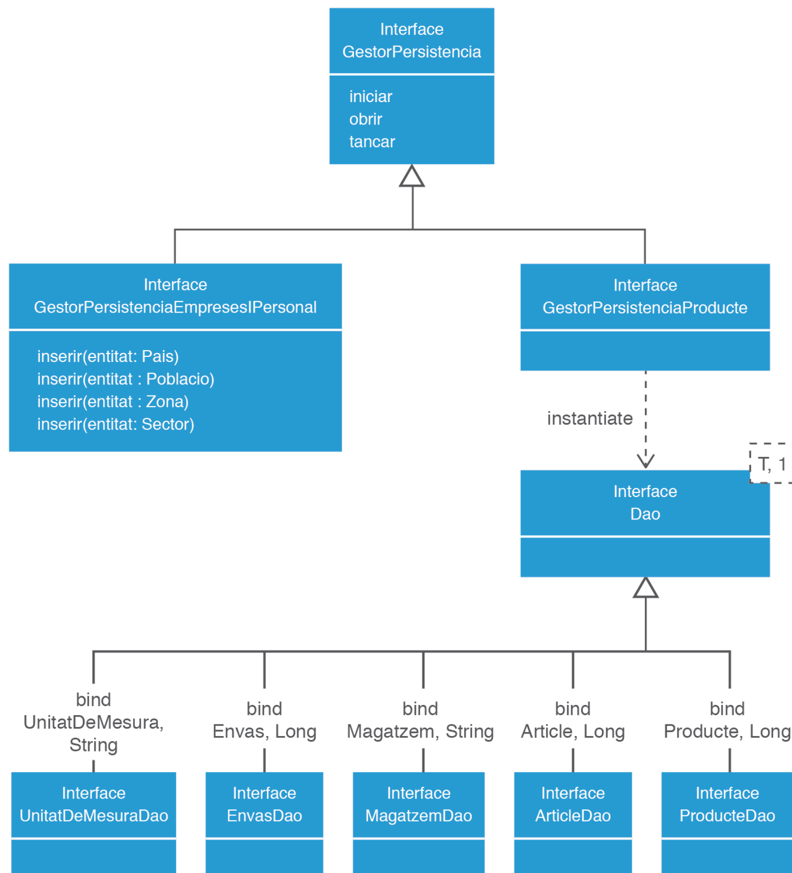
Per coherència també crearem els mateixos mètodes que permetin instanciar de la mateixa manera les entitats, però sense realitzar la inserció per quan ens calgui crear entitats de forma temporal i no emmagatzemar-les. Identificarem aquests mètodes perquè afegirem al no el sufix *temporal*. Vegem el mateix exemple d'instanciació d'una zona:

```
1 /**
2  * Crea i retorna una instància nova de zona i la inicialitza
3  * amb els paràmetres corresponents al nom i la descripció.
4  * @param id és el nom de la zona que cal instanciar.
5  * @param descripció és la descripció de la zona que cal instanciar.
6  * @return instància de zona creada.
7  */
8 public Zona novaZonaTemporal(String id, String descripcio);
```

2.1.3 Estratègia fàbrica en el disseny de components de persistència

Aquí plantejarem com desenvolupar un conjunt d'interfícies per suportar la gestió de la persistència. Inicialment crearem una interfície paramètrica que reculli la funcionalitat general de qualsevol entitat, i seguidament crearem també, per a cada entitat, una d'interfície amb la funcionalitat específica requerida per a cada entitat (figura 2.3).

Figura 2.3. Visió global de les interfícies definides al component de persistència de l'aplicació



Per tal d'evitar la proliferació de connexions obertes (una per a cada entitat), eximirem aquests components de les tasques de connectar i desconnectar amb la font de dades. Això ho farà l'encarregat d'instanciar cada component de persistència, el qual implementarà la interfície `GestorPersistenciaProducte` que declara els mètodes adequats per realitzar les instanciacions.

Interfície Dao

Es tracta de la interfície que agrupa els mètodes de gestió de la persistència comuns a qualsevol entitat. La interfície es troba parametritzada per aconseguir adequar-se a cada entitat concreta. Accepta dos paràmetres: el tipus d'entitat (T) i el tipus d'identificador de l'entitat (I).

```
public interface Dao<T, I> {
```

Implementarem els mètodes de gestió comuns tenint en compte els paràmetres.

```
/*
     * Crea una instància nova de l'entitat referenciada per
     * aquest objecte DAO, la qual no és emmagatzemada de
     * forma persistent sinó que es crea amb l'objectiu de
     * realitzar alguna gestió temporal.
     * @return instància de l'entitat referenciada per
     * aquest objecte DAO.
     */
    public T novaInstanciaTemporal();
```

```
10
11     /*
12      * Crea una instància nova de l’entitat referenciada per
13      * aquest objecte DAO, la qual s’emmagatzemarà de forma
14      * persistent abans de ser retornada. Aquest mètode
15      * només és adequat en cas que l’entitat tingui un
16      * identificador generat de forma automàtica, ja que en
17      * cas contrari la inserció generarà un error.
18      * @return instància de l’entitat referenciada per aquest
19      * objecte DAO.
20      * @throws UtilitatPersistenciaException. Es llança per
21      * qualsevol error provinent del sistema de persistència.
22      */
23     public T novaInstancia() throws UtilitatPersistenciaException;
24
25     /*
26      * Crea una instància nova de l’entitat referenciada per
27      * aquest objecte DAO i s’inicialitza amb l’identificador
28      * passat per paràmetre. La instància no s’emmagatzemarà,
29      * per tant s’hauria de destinar a realitzar alguna
30      * gestió temporal.
31      * @param identificador de la instància (clau primària).
32      * @return instància de l’entitat referenciada per aquest
33      * objecte DAO.
34      */
35     public T novaInstanciaTemporal(I id);
36
37     /*
38      * Crea una instància nova de l’entitat referenciada per
39      * aquest objecte DAO i s’inicialitza amb l’identificador
40      * passat per paràmetre. La instància s’emmagatzemarà de
41      * forma persistent abans de ser retornada. En cas que ja
42      * existís una entitat persistent amb el mateix
43      * identificador es llançaria una excepció indicant que
44      * l’entitat que s’ha intentat crear no es pot inserir
45      * perquè el seu identificador ja està utilitzat.
46      * @param identificador de la instància (clau primària).
47      * @return instància de l’entitat referenciada per aquest
48      * objecte DAO.
49      * @throws UtilitatPersistenciaException. Es llança per
50      * qualsevol error provinent del sistema de persistència.
51      */
52     public T novaInstancia(I id)
53             throws UtilitatPersistenciaException;
54
55     /*
56      * Refresca els atributs de l’entitat passada per
57      * paràmetre amb les dades emmagatzemades.
58      * @param entitat a refrescar
59      * @return L’entitat refrescada.
60      * @throws UtilitatPersistenciaException. Es llança per
61      * qualsevol error provinent del sistema de persistència.
62      */
63     public T refrescar(T entitat)
64                 throws UtilitatPersistenciaException;
65
66     /*
67      * Emmagatzema les dades contingudes als atributs de
68      * l’entitat passada per paràmetre. Si l’entitat no
69      * fos persistent en el moment de la invocació, es
70      * farà una inserció. En canvi si l’entitat ja fos
71      * persistent es realitzarà una actualització de les
72      * dades emmagatzemades.
73      * @param entitat a emmagatzemar
74      * @throws UtilitatPersistenciaException. Es llança per
75      * qualsevol error provinent del sistema de persistència.
76      */
77     public void emmagatzemarDades(T entitat)
78                 throws UtilitatPersistenciaException;
79
```

```
    /*
     * Indica si l'objecte passat per paràmetre és
     * persistent. La comprovació de l'existència es basa
     * exclusivament en el valor dels atributs
     * identificadors de la instància.
     * @param entitat a comprovar.
     * @return cert si l'entitat a comprovar és persistent.
     * Fals en cas contrari.
     * @throws UtilitatPersistenciaException. Es llança per
     * qualsevol error provinent del sistema de persistència.
     */
    public boolean esPersistent(T entitat)
                throws UtilitatPersistenciaException;

    /*
     * Actualitza les dades emmagatzemades amb les
     * contingudes a l'estat de la instància passada per
     * paràmetre. L'entitat ha d'haver estat emmagatzemada
     * amb anterioritat. Contràriament es produirà un error.
     * @param entitat des d'on fer l'actualització.
     * @throws UtilitatPersistenciaException. Es llança per
     * qualsevol error provinent del sistema de persistència.
     */
    public void modificar(T entitat)
                throws UtilitatPersistenciaException;

    /*
     * S'emmagatzema per primer cop l'entitat passada per
     * paràmetre. És a dir, que per poder—la inserir amb èxit
     * cal que no existeixi a la base de dades cap altra
     * entitat emmagatzemada amb el mateix identificador.
     * Quan això passi es produirà un error i es llançarà
     * una excepció de tipus UtilitatPersistenciaException.
     * @param entitat a inserir.
     * @throws UtilitatPersistenciaException. Es llança per
     * qualsevol error provinent del sistema de persistència.
     */
    public void inserir(T entitat)
                throws UtilitatPersistenciaException;

    /*
     * Fa que l'entitat identificada amb la clau que es
     * passa per paràmetre deixi de ser persistent. Si no
     * existeix cap entitat persistent identificada amb la
     * clau, es llançarà una excepció de tipus
     * UtilitatPersistenciaException.
     * @param clau que identifica l'entitat candidata a
     * deixar de ser persistent.
     * @throws UtilitatPersistenciaException. Es llança per
     * qualsevol error provinent del sistema de persistència.
     */
    public void eliminarPerClau(I clau)
                throws UtilitatPersistenciaException;

    /*
     * Fa que l'entitat passada per paràmetre deixi de ser
     * persistent. Si no existeix cap entitat persistent
     * identificada amb el mateix valor que la instància passada
     * per paràmetre, es produirà un error i es llançarà
     * una  excepció de tipus UtilitatPersistenciaException.
     * @param entitat és l'entitat candidata a deixar de ser
     * persistent.
     * @throws UtilitatPersistenciaException. Es llança per
     * qualsevol error provinent del sistema de persistència.
     */
    public void eliminar(T entitat)
                throws UtilitatPersistenciaException;

    /*
     * Obté una instància persistent (emmagatzemada
```

```
150      * prèviament) identificada amb la clau que es passa per
151      * paràmetre.
152      * @param clau que identifica l'entitat que es desitja
153      * recuperar.
154      * @return Instància de l'entitat recuperada a partir de
155      * les dades emmagatzemades.
156      * @throws UtilitatPersistenciaException. Es llança per
157      * qualsevol error provinent del sistema de persistència.
158      */
159     public T obtenirInstancia(I clau)
160                 throws UtilitatPersistenciaException;
161 
162     /*
163      * Obté una llista de totes les entitats emmagatzemades
164      * del tipus referenciat per aquest DAO.
165      * @return llista d'entitats persistents del tipus
166      * referenciat per aquest DAO.
167      * @throws UtilitatPersistenciaException. Es llança per
168      * qualsevol error provinent del sistema de persistència.
169      */
170     public List<T> obtenirTot()
171                 throws UtilitatPersistenciaException;
172 }
```

Interfícies específiques d'accés a dades

La concreció dels paràmetres de la interfície es realitzarà a la declaració de les interfícies específiques. L'objectiu d'aquestes interfícies, a més de concretar els tipus paramètrics de `Dao`, consisteix també a agrupar la funcionalitat específica de cada entitat. Vegem, per exemple, la interfície `MagatzemDao`.

```
1  public interface MagatzemDao extends Dao<Magatzem, String> {
2 
3      /*
4       * Crea una instància nova de l'entitat referenciada per
5       * aquest objecte DAO i s'inicialitza amb l'identificador
6       * passat per paràmetre. La instància no s'emmagatzemarà,
7       * per tant s'hauria de destinar a realitzar alguna
8       * gestió temporal.
9       * @param id és l'identificador de la instància (clau
10      * primària).
11      * @param desc és la descripció associada al magatzem.
12      * @return instància de l'entitat referenciada per aquest
13      * objecte DAO.
14      */
15     Magatzem novaInstanciaTemporal(String id, String desc);
16 
17     /*
18      * Crea una instància nova de l'entitat referenciada per
19      * aquest objecte DAO i s'inicialitza amb l'identificador
20      * i la descripció que es passen per paràmetre. La
21      * instància s'emmagatzemarà de forma persistent abans
22      * de ser retornada. En cas que ja existís una entitat
23      * persistent amb el mateix identificador es llançaria
24      * una excepció indicant que l'entitat que s'ha intentat
25      * crear no es pot inserir perquè el seu identificador
26      * ja està utilitzat.
27      * @param id és l'identificador de la instància (clau
28      * primària).
29      * @param desc és la descripció associada al magatzem.
30      * @return instància de l'entitat referenciada per aquest
31      * objecte DAO.
32      * @throws UtilitatPersistenciaException. Es llança per
33      * qualsevol error provinent del sistema de persistència.
34      */
```

```
    Magatzem novaInstancia(String id, String desc)
                            throws UtilitatJdbcSQLException;

    /*
     * Elimina els registres d'estoc associats al magatzem
     * identificat per la cadena passada per paràmetre.
     * @param idMagatzem és la clau que identifica el
     * magatzem.
     * @throws UtilitatPersistenciaException. Es llança per
     * qualsevol error provinent del sistema de persistència.
     */
    void eliminarEstoc(final String idMagatzem)
                            throws UtilitatJdbcSQLException;

    /*
     * Obté una llista de tots els productes en estoc en el
     * magatzem que es passa per paràmetre.
     * @param entitat és el magatzem del que es desitja
     * obtenir la llista de productes en estoc.
     * @return Llista de productes en estoc.
     * @throws UtilitatPersistenciaException. Es llança per
     * qualsevol error provinent del sistema de persistència.
     */
    List<ProducteEnEstoc> getEstoc(final Magatzem entitat)
                            throws UtilitatJdbcSQLException;

    /*
     * Modifica l'estoc del magatzem identificat per la
     * cadena del primer paràmetre d'acord amb els valors
     * continguts a l'array que es passa en el segon paràmetre.
     * @param idMagatzem és l'identificador del magatzem a
     * modificar.
     * @param productesEnEstoc és una vector de tipus
     * ProdcuteEstoc
     * contenint els estocs dels producte indicats.
     * @throws UtilitatPersistenciaException. Es llança per
     * qualsevol error provinent del sistema de persistència.
     */
    void modificarEstoc(String idMagatzem,
            ProducteEnEstoc[] productesEnEstoc)
                                throws UtilitatJdbcSQLException;
}
```

La resta d'interfícies específiques s'implementaran de forma molt semblant.

Interfície instanciadora

Per tal de facilitar la implementació dels components específics d'accés a dades usarem un component responsable de crear les instàncies que també s'encarregui de gestionar la connexió i la desconnexió al SGBD. La interfície que representi el component instanciador derivarà de `GestorPersistencia` doncs aquesta comtempla ja la funcionalitat de connexió a la font de dades.

D'altra banda, la interfície del component instanciador haurà de declarar mètodes per obtenir els diversos components `Dao` que es desitgi controlar. Per exemple, la interfície que anomenarem `GestorPersistenciaProducte` controlarà la instanciació de tots aquells components específics (derivats de `Dao`) relacionats amb els productes.

```
public interface GestorPersistenciaProducte
                extends GestorPersistencia {
    /*
     * Crea i retorna una instància de tipus ArticleDao.
```

```
5          * La instància disposarà de connexió activa.
6          * @return una instància de tipus ArticleDao.
7          */
8         ArticleDao crearArticleDao();
9
10        /*
11         * Crea i retorna una instància de tipus EnvasDao.
12         * La instància disposarà de connexió activa.
13         * @return una instància de tipus EnvasDao.
14         */
15        EnvasDao crearEnvasDao();
16
17        /*
18         * Crea i retorna una instància de tipus MagatzemDao.
19         * La instància disposarà de connexió activa.
20         * @return una instància de tipus MagatzemDao.
21         */
22        MagatzemDao crearMagatzemDao();
23
24        /*
25         * Crea i retorna una instància de tipus ProducteDao.
26         * La instància disposarà de connexió activa.
27         * @return una instància de tipus ProducteDao.
28         */
29        ProducteDao crearProducteDao();
30
31        /*
32         * Crea i retorna una instància de tipus
33         * UnitatDeMesuraDao. La instància disposarà de connexió
34         * activa.
35         * @return una instància de tipus UnitatDeMesuraDao.
36         */
37        UnitatDeMesuraDao crearUnitatDeMesuraDao();
38    }
```

2.2 Implementació dels components de persistència basats en JDBC

En aquest apartat, a més d'implementar els components de persistència de l'aplicació, mostrarem també diverses formes de facilitar la programació dels processos comuns a qualsevol component JDBC.

Un cop definides les interfícies amb els mètodes que configuraran els components de persistència caldrà implementar les classes. És aquí on haurem de decidir quin sistema persistent utilitzarem, ja que en funció del sistema escollit la implementació es farà d'una o altra manera.

S'ha escollit la implementació de la persistència via JDBC perquè dóna prou joc per arribar a copsar d'una forma força completa la implementació de la persistència.

La implementació d'aquest sistema sol requerir una gran quantitat de codi, ja que es tracta d'un component que treballa a baix nivell i necessita de moltes adaptacions.

2.2.1 Processos comuns d'accés a dades

Començarem creant un conjunt d'utilitats bàsiques i genèriques que ens ajudaran en el desenvolupament posterior. Agruparem la funcionalitat en una classe ano-

menada UtilitatJDBC, una classe genèrica que permet gestionar les principals accions a realitzar amb una connexió.

Tots els mètodes d'aquesta classe són estàtics, perquè es tracta d'una utilitat purament funcional i que no necessita estat. Disposa d'utilitats específiques per realitzar el tancament d'objectes Statement,ResultSet o Connection silenciant l'error però enregistrant-lo en un fitxer a traves del sistema Logger.

```
public static void tancarConnexio(Connection con){
    try {
        if(con!=null && !con.isClosed()){
            con.close();
        }
    } catch (SQLException ex) {
        Logger.getLogger(UtilitatJDBC.class.getName())
                .log(Level.SEVERE, null, ex);
    }
}

public static void tancaStatement(Statement stm, ResultSet rs){
    try {
        if(rs!=null && !rs.isClosed()){
            rs.close();
        }
    } catch (SQLException ex) {
        Logger.getLogger(EstructuraComandes.class.getName())
                .log(Level.SEVERE, null, ex);
    }
    try {
        if(stm!=null && !stm.isClosed()){
            stm.close();
        }
    } catch (SQLException ex) {
        Logger.getLogger(EstructuraComandes.class.getName())
                .log(Level.SEVERE, null, ex);
    }
}

public static void tancaStatement(Statement stm){
    tancaStatement(stm, null);
}

public static void tancaResultSet(ResultSet rs){
    tancaStatement(null, rs);
}
```

El tancament del ResultSet acostuma a demanar-se de forma simultània al tancament de l'Statement que l'ha generat, per això s'ha implementat un mètode que permet tancar ambdós a l'hora. Malgrat tot, com es veu en els dos darrers mètodes, el tancament també es pot fer per separat.

Per tal de portar un registre de tots els errors que es puguin produir durant la connexió s'han implementat dos mètodes que enregistraran l'error via Logger i encapsularan l'excepció dins una jerarquia pròpia de tres classes: UtilitatJdbcException, la més genèrica; UtilitatJdbcSQLException, que farem servir exclusivament per encapsular errors específics de tipus SQL, i UtilitatJdbcRollBackException, que representarà aquelles excepcions que s'hagin produït executant un rollback. Així es facilitarà la captura i el posterior tractament dels errors.

```
public static void onError(Exception ex)
                throws UtilitatJdbcException{
```

```
        Logger.getLogger(EstructuraComandes.class.getName()).log(
                Level.SEVERE, null, ex);
        throw new UtilitatJdbcException(ex.getMessage(), ex);
    }

    public static void onError(SQLException ex)
                        throws UtilitatJdbcSQLException{
        Logger.getLogger(EstructuraComandes.class.getName()).log(
                Level.SEVERE, null, ex);
        throw new UtilitatJdbcSQLException(ex.getMessage(), ex);
    }
```

Per poder activar la connexió caldrà conèixer la *url*, l'*usuari* i la *contrasenya* del'SGBD. El mètode `obrir` rebrà per paràmetre les dades de la connexió i la retornarà activa.

```
public static Connection obrir(String driver, String url,
                String user, String password)                    throws
                    UtilitatJdbcException{
    Connection con = null;
    try {
        Class.forName(driver);
        con = DriverManager.getConnection(url, user, password);
    } catch (SQLException ex) {
        onError(ex);
    } catch (ClassNotFoundException ex) {
        onError(ex);
    }
    return con;
}
```

El mètode `desfer` és una utilitat que ens ajudarà a fer el seguiment dels errors. Imaginem que durant l'execució d'una sentència SQL es produeix un error i en capturar l'excepció iniciem un crida a `rollback` per desfer tots els canvis produïts durant la transacció actual, però en fer la petició de `rollback` es produeix de nou un error. El llançament del segon error emmascararà el primer, el qual es perdrà si no fem res. Si silenciem el segon, el primer no es perdria, però perdríem la possibilitat d'analitzar el segon. El mètode desfer que proposem rep per paràmetre l'excepció que ha provocat la petició de `rollback`. Si l'acció es produeix amb èxit es retorna el control sense gestionar l'excepció perquè es procedeixi a actuar per defecte. Ara bé, si l'acció no té èxit ambdues excepcions, la passada per paràmetre (error originari) i l'acabada de llançar s'encapsularan dins una excepció de tipus `UtilitatJdbcRollbackException`, la qual serà llançada de manera que la seva captura permeti analitzar ambdues excepcions contingudes.

```
public static void desfer(Connection con, SQLException sqlEx)
                                throws UtilitatJdbcRollbackException{
    try {
       con.rollback();
    } catch (SQLException ex) {
       Logger.getLogger(EstructuraBDComandes.class.getName())
                .log(Level.SEVERE, null, ex);
       throw new UtilitatJdbcRollbackException(ex, sqlEx);
    }
}
```

El mètode `desfer` servirà específicament per desfer instruccions en aquells mètodes que executin diverses sentències d'una mateixa transacció.

Per tal de facilitar l'automatització de les crides SQL implementarem dos mètodes a la classe`UtilitatJDBC` que ens ajudaran a executar sentències SQL passades en format de cadenes i col·leccions de cadenes.

```
public static void executar(Connection con, String sentenciaSql)
                                       throws UtilitatJdbcException{
    //permet tornar l'autocommit a la seva forma original
    boolean autocommit=true;
    Statement stm = null;
    try {
        autocommit=con.getAutoCommit(); //salvem l'original
        con.setAutoCommit(true);  //assegurem el mode autocomit
        stm = con.createStatement();  //creem l'Statement
        stm.executeUpdate(sentenciaSql); //executem la sentència
        con.setAutoCommit(autocommit);  //restaurem de nou
    } catch (SQLException ex) {
        onError(ex);    //tractament de l'error
    }finally{
        tancaStatement(stm);  //tancament controlat
    }
}
```

En el mètode anterior, en tractar-se de només una sentència, se suposarà que es vol executar aïllada i, per tant, s'activarà el mode `autocommit`. Volem tornar a deixar la connexió amb el mateix mode amb què ha entrat. Per això farem servir la variable `autocommit`.

La versió que treballa amb una col·lecció de cadenes és idèntica a l'anterior, amb la diferència que cal controlar el procés `commit` de forma manual per tal de tractar totes les sentències de la col·lecció com una única instrucció unitària.

```
public static void executar(Connection con, String[] sentenciesSql)
                                       throws UtilitatJdbcException{
    boolean autocommit=true;
    Statement stm = null;
    try {
        autocommit = con.getAutoCommit();
        con.setAutoCommit(false);
        stm = con.createStatement();

        //bucle per executar totes les sentències
        for(String sent: sentenciesSql){
            stm.executeUpdate(sent); //executem les sentències
        }
        con.commit();  // tot ha anat bé i validem les accions
    con.setAutoCommit(autocommit);
    } catch (SQLException ex) {
       //Hi ha hagut un error. Cal desfer—ho tot.
     //Passem per paràmetre l'error obtingut per no perdre'l
     //en cas d'error durant el rollback
        desfer(con, ex);
        onError(ex); //Tractament de l'error
    }finally{
        tancaStatement(stm); //tancament controlat
    }
}
```

Les sentències a executar es passen per paràmetre contingudes dins d'un *array*. El mètode `desfer` es crida dins d'un bloc `catch` que captura l'error passant-lo com a paràmetre de `desfer`, com ja s'ha comentat.

Ja disposem de suficients mètodes per començar a treballar en força situacions.

Començarem, doncs, a implementar la interfície creant una classe que gestioni la connexió a la base de dades via JDBC. La classe codificarà els mètodes obrir i tancar, però no iniciar. Això ho farem així per tal d'independitzar el sistema de configuració de la connexió. Anomenarem a la classe `AbstractGestorJDBC`. Gràcies a les utilitats que hem implementat, la codificació d'aquesta classe resultarà molt senzilla.

```
public abstract class AbstractGestorJDBC
                implements GestorPersistencia {
    protected String driver;
    protected String user;
    protected String password;
    protected String url;
    protected Connection con;

    @Override
    public void obrir() throws UtilitatPersistenciaException{
        con = UtilitatJDBC.obrir(driver, url, user, password);
    }

    @Override
    public void tancar(){
        UtilitatJDBC.tancarConnexio(getConnexio());
    }

    /*
     * Obté el driver JDBC
     * @return el driver
     */
    public String getDriver() {
        return driver;
    }

    /*
     * * Obté l'usuari per realitzar la connexió a l'SGDB
     * @return l'usuari
     */
    public String getUser() {
        return user;
    }

    /*
     * Obté la contrasenya per realitzar la connexió
     * @return la contrasenya
     */
    public String getPassword() {
        return password;
    }

    /*
     * Obté la URL de l'SGBD per realitzar la connexió
     * @return la url
     */
    public String getUrl() {
        return url;
    }

    /*
     * Obté la connexió JDBC
     * @return la connexió
     */
    public Connection getConnexio() {
        return con;
    }
}
```

Fixeu-vos que els atributs d'`AbstactGestorJDBC` són exclusivament de lectura.

La raó és perquè de l'escriptura se n'encarregarà la classe que implementi el mètode iniciar. Per tal que sigui possible la manipulació dels atributs per una classe derivada, cal declarar-los *protected*.

De la configuració se n'encarregarà la classe `ComandesJDBC`. Farem servir un fitxer de *Properties* per aconseguir les dades necessàries per connectar amb la base de dades correcta, però hauríem pogut optar per altres solucions. Fent que aquesta classe tingui per objectiu només la configuració del sistema, facilitarem el manteniment posterior si algun dia es decideix canviar.

Els fitxers de *Properties* són fàcils de fer servir i molt útils per la seva versatilitat, ja que es tracta de fitxers de text on podem emmagatzemar un conjunt de propietats identificades per un nom al qual se li associa un valor. En tractar-se d'un fitxer de text, l'edició i modificació resulta molt senzilla. Concretament, el nostre fitxer contindrà el següent contingut:

```
1 driver=org.postgresql.Driver
2 user=jdbc
3 password=jdbc
4 url=jdbc:postgresql://localhost:5432/ioc_comandes_jdbc
```

Una manera de despreocupar-nos de la configuració és fer coincidir el nom del fitxer de *Properties* amb el de la classe, de manera que cada classe (que ho necessiti) disposi del seu fitxer de configuració. Situarem el fitxer de propietats a la carpeta *cfg*, ubicada directament al directori on s'executi l'aplicació.

```
1  public class ComandesJDBC extends AbstractGestorJDBC{
2  
3      @Override
4      public void iniciar() throws UtilitatPersistenciaException {
5          try {
6              String nomFitxer = "cfg"+File.separatorChar
7                                 + ComandesJDBC.class.getSimpleName()
8                                 + ".properties";
9  
10             Properties properties = new Properties();
11             properties.load(new FileReader(nomFitxer));
12 
13             driver = properties.getProperty("driver");
14             user = properties.getProperty("user");
15             password= properties.getProperty("password");
16             url = properties.getProperty("url");
17         } catch (IOException ex) {
18             UtilitatJDBC.onError(ex);
19         }
20     }
21 }
```

2.2.2 Creació de taules en el'SGBD

Sempre que ens plantegem implementar una aplicació JDBC cal preveure que en instal·lar-la caldrà crear els elements de la base de dades enl'SGBD on desitgem ubicar la persistència (taules, índex, restriccions, usuaris, permisos, etc.). Anem, en primer lloc, a implementar una classe encarregada de la instal·lació i modificació de l'estructura de dades enl'SGDB i la importació inicial de dades.

La base de dades la crearem manualment des de l'administrador del'SGBD. Suposarem que ja n'existeix una anomenada *ioc_comandes_jdbc*. L'aplicació que implementarem s'haurà de connectar a la citada base de dades abans de procedir a la instal·lació.

Farem servir una tècnica senzilla que permet adaptar l'aplicació als requeriments específics de qualsevol SGBD. Consisteix a fer servir *scripts* SQL emmagatzemats en fitxers de text, els quals llegirem des de la nostra aplicació Java d'instal·lació i executarem via JDBC.

Els fitxers *script* contenen sentències SQL separades entre elles per un punt i coma. La nostra aplicació convertirà el contingut dels scripts en un `String` i l'executarà fent servir un `Statement`.

En tractar-se de fitxers de text, és molt senzill substituir l'*script* (adaptat a PostgreSQL) per un altre de sintaxi Oracle o MySQL sense necessitat d'haver de tocar codi ni compilar-lo. Fins i tot, en cas que fos necessari, seria possible, durant el procés d'instal·lació, realitzar modificacions en els *scripts* que quedarien actius de forma immediata.

Per tal d'aconseguir un sistema altament configurable, també farem servir aquí un fitxer de propietats que indiqui a l'aplicació el nom dels fitxers de text amb els *scripts* de la instal·lació adequats al'SGBD que haguem de fer servir.

La classe encarregada de la instal·lació l'anomenarem `EstructuraBDComandes`. Aquesta tindrà associat un fitxer de configuració anomenat `EstructuraBDComandes.properties` situat a la carpeta *cfg* ubicada en el directori on s'hagi d'executar l'aplicació.

La classe disposarà, a més del procés d'instal·lació, d'altres processos que automatitzin la modificació de l'estructura de dades, per quan sigui necessari incorporar noves versions de l'aplicació que gestionin l'eliminació de la mateixa quan es decideixi desinstal·lar-la o que ajudin a incorporar dades de forma massiva provinents d'alguna exportació.

Cada un d'aquest processos tindrà un fitxer *script* associat que es reconeixerà a partir del fitxer de propietats `EstructuraBDComandes.properties`.

Els processos d'instal·lació, modificació o desinstal·lació s'executen en comptades ocasions, per això implementarem una aplicació independent de la qual se n'encarregui de l'explotació de les dades. `EstructuraBDComandes` disposarà únicament dels quatre procediments esmentats (creació, eliminació i modificació de l'estructura de dades, així com la introducció massiva de dades), i es podrà executar des de consola.

Podeu trobar els fitxers de configuració i els scripts SQL consultant l'annex d'aquesta unitat anomenat "Codi dels exemples" que il·lustren la unitat "Components d'accés a dades". Tant els fitxers de configuració com els scripts SQL es troben a la carpeta `cfg`.

Mostrem ara el contingut d'`EstructuraBDComandes.properties`. Es tracta, com es pot veure, d'una relació dels quatre fitxers script (`.sql`), més el fitxer de configuració del sistema d'enregistrament (*Logger*) identificat per la propietat *fitxer_logger_cfg*.

```
fitxer_logger_cfg=cfg/logger.properties
fitxer_creacio_estructura=cfg/crear_ioc_comandes.sql
```

```
fitxer_eliminacio_estructura=cfg/eliminar_ioc_comandes.sql
fitxer_importacio_dades=cfg/importarDades_ioc_comandes.sql
fitxer_modificacio_estructura=cfg/modificar_ioc_comandes.sql
```

La classe `EstructuraDBComandes` estendrà `ComandesJDBC` per tal de disposar de la funcionalitat de connexió a la base de dades i poder executar els *scripts*. Els noms de cada un dels quatre fitxers *scripts* es mantindran en quatre atributs de tipus `String`, com es mostra a continuació:

```
public class EstructuraBDComandes {
    String sqlDeCreacio;
    String sqlDEliminacio;
    String sqlDeModificacio;
    String sqlDImportacio;
    ComandesJDBC fontDeDades = new ComandesJDBC();

    ...
}
```

La classe sobreescriurà el mètode d'inicialització (`iniciar`) per tal d'incorporar la seva configuració específica. Es configuraran tres aspectes de l'aplicació: el sistema d'enregistrament (*logger*), l'assignació dels fitxers *script*, i la configuració i connexió del sistema JDBC a partir de la classe `ComandesJDBC`.

```
public void iniciar() throws UtilitatJdbcException{
    super.iniciar();  //configurem la connexió JDBC
    try {
        //nom del fitxer de propietats per configurar aquesta classe
        String nomFitxer = "cfg"+File.separatorChar
                        + EstructuraBDComandes.class.getSimpleName()
                        + ".properties";

        //Llegeix i carrega les propietats
        Properties properties = new Properties();
        properties.load(new FileReader(nomFitxer));

        //configura el sistema d'enregistrament (Logger)
        LogManager.getLogManager().readConfiguration(
                        new FileInputStream(
                properties.getProperty(
                            "fitxer_logger_cfg")));

        //assigna els scripts
        sqlDeCreacio = properties.getProperty(
                                "fitxer_creacio_estructura");
        sqlDEliminacio = properties.getProperty(
                                "fitxer_eliminacio_estructura");
        sqlDeModificacio = properties.getProperty(
                                "fitxer_modificacio_estructura");
        sqlDImportacio = properties.getProperty(
                                "fitxer_importacio_dades");
    } catch (IOException ex) {
        ComandesJDBC.onError(ex);
    }
}
```

Els quatre processos tindran la mateixa estructura, llegiran el fitxer específic i executaran el seu contingut via JDBC. Per evitar la repetició de codi es farà servir la funcionalitat d'`UtilitatJDBC` anomenada `executa`.

També s'ha implementat el mètode `llegirFitxerSQLIExecutarLo`, que mostrem al final, el qual fa la lectura d'un fitxer *script*, el converteix a `String` i l'e-

xecuta. Així, els quatre mètodes que implementin els quatre processos esmentats quedaran reduïts a una crida de `llegirFitxerSQLIExecutarLo`, seleccionant l'*script* corresponent.

```
public void crearEstructura() throws UtilitatJdbcException{
    llegirFitxerSQLIExecutarLo(sqlDeCreacio);
}

public void eliminarEstructura() throws UtilitatJdbcException{
    llegirFitxerSQLIExecutarLo(sqlDEliminacio);
}

public void importarDades() throws UtilitatJdbcException{
    llegirFitxerSQLIExecutarLo(sqlDImportacio);
}

public void modificarEstructura() throws UtilitatJdbcException{
    llegirFitxerSQLIExecutarLo(sqlDeModificacio);
}

private void llegirFitxerSQLIExecutarLo( String script)
                                      throws UtilitatJdbcException {
    FileReader reader = null;
    StringBuilder stringBuilder = new StringBuilder();
    int charsLlegits=0;
    char[] buffer = new char[512];
    try{
        //lectura del fitxer d'instruccions SQL
        reader = new FileReader(script);
        while(charsLlegits!=-1){
            stringBuilder.append(buffer, 0, charsLlegits);
            charsLlegits=reader.read(buffer);
        }
   //Execució fent servir UtilitatJDBC
        UtilitatJDBC.executar(getConnexio(),
                                    stringBuilder.toString());
    } catch (IOException ex) {
        UtilitatJDBC.onError(ex);  //tractament de l'error
    }
}
```

La classe disposarà d'un mètode per discriminar el procés a executar. La discriminació es farà analitzant el nom de la comanda que es passarà per paràmetre. Noteu que fem servir `startsWith`, de manera que s'acceptaran comandes com `crea`, `creació`, `crear`, `insta`, `instal·lació`, `install`, `instal·lar`, `elim`, `eliminar`, `elimina`, `eliminació`, etc.

```
public void executaComanda(String comanda) throws UtilitatJdbcException{
    if(comanda.toLowerCase().startsWith("crea")
            || comanda.toLowerCase().startsWith("insta")){
        crearEstructura();
    }else if(comanda.toLowerCase().startsWith("elim")){
        eliminarEstructura();
    }else if(comanda.toLowerCase().startsWith("modif")){
        crearEstructura();
    }else if(comanda.toLowerCase().startsWith("import")){
        importarDades();
    }else{
        System.out.println("Argument " + comanda + " desconnegut");
    }
}
```

La comanda s'enviarà com a argument del *main*. De fet, farem que el *main* accepti múltiples arguments que s'executin un darrere l'altre. Així, si féssim la següent crida des del sistema operatiu:

```
java ioc.dam.m6.exemples.comandes.controlsgbd.EstructuraDBComandes crea import
```

crearia les taules en primer lloc i després afegiria dades des d'un *script*.

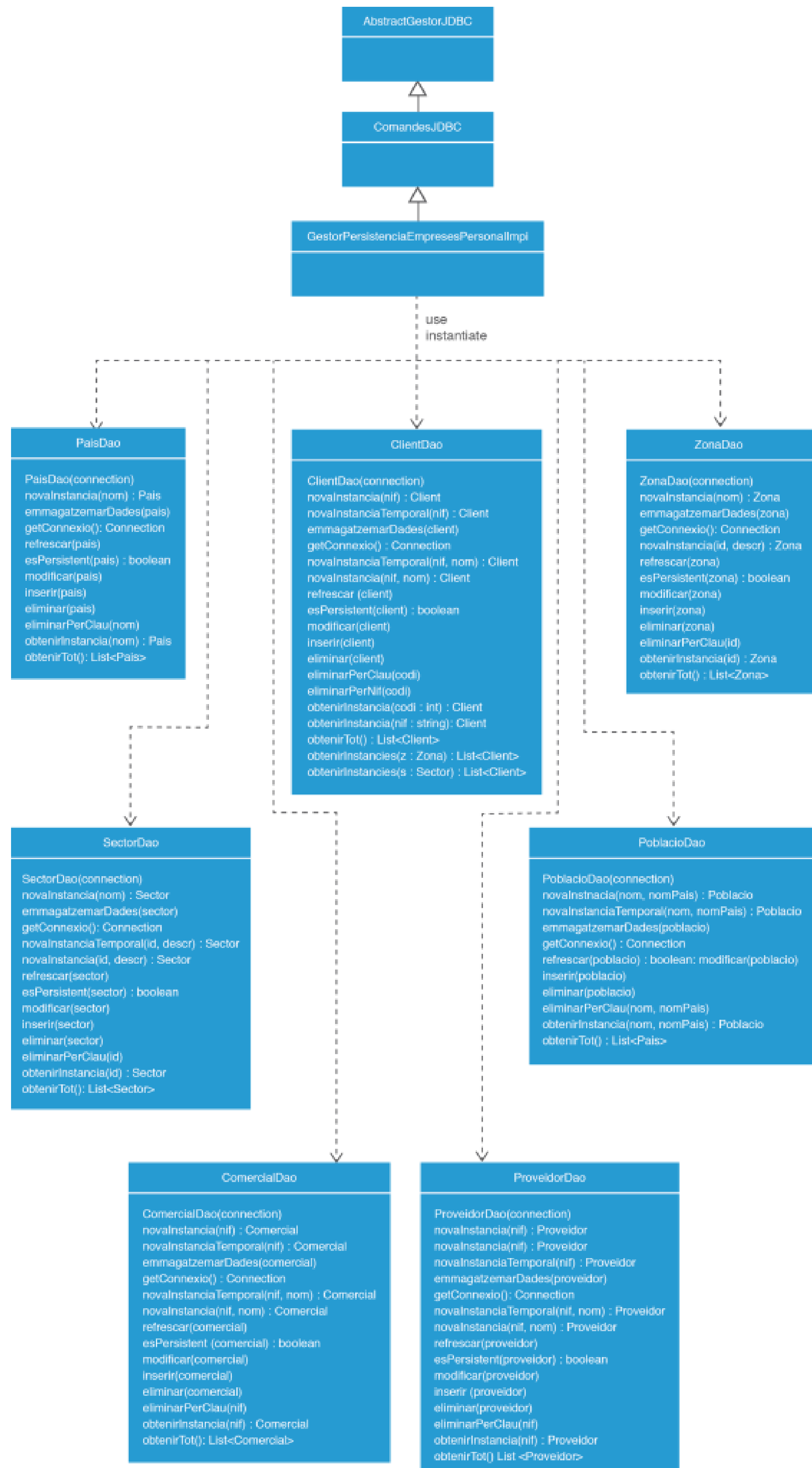
La implementació del mètode *main* quedarà tal com segueix:

```
public static void main(String[] args) {
      EstructuraBDComandes estructuraComandes =
                        new EstructuraBDComandes();
    try {
          estructuraComandes.iniciar();
            estructuraComandes.obrir();

            if(args.length==0){
                estructuraComandes.executaComanda("crea");
            }else{
                for(String comanda: args){
                    estructuraComandes.executaComanda(comanda);
                }
            }
        } catch (UtilitatPersistenciaException ex) {
            Logger.getLogger(EstructuraBDComandes.class.getName())
                        .log(Level.SEVERE, null, ex);
        }finally{
            estructuraComandes.tancar();
        }
}
```

2.2.3 Implementació de la interfície amb rol de façana

Recordem que la interfície `GestorPersistenciaEmpresesIPersonal` tenia el paper de façana en un dels components de la nostra aplicació. A l'hora d'implementar-la cal que ens plantegem si és factible implementar-la en una única classe, o bé si és preferible dividir-la en diverses. Generalment, la dimensió de les interfícies façana aconsellarà que realitzem la codificació fent servir diverses classes. Els components de persistència són fàcilment divisibles, mentre que cada entitat del model pot induir una classe d'accés a les dades del seu estat. Crearem, doncs, les classes `PaisDao`, `PoblacioDao`, `ZonaDao`, `SectorDao`, `ComercialDao`, `ClientDao` i `ProveidorDao`, que encapsularan els algoritmes pertinents per gestionar la seva persistència i que respondran als mètodes específics declarats a`GestorPersistenciaEmpresesIPersonal`. A la figura 2.4 podeu veure les diferents classes amb la declaració dels mètodes que els pertoca.

Figura 2.4. Classes que componen el component GestorPersistenciaEmpresesIPersonal



Noteu també que cada una de les classes d'accés a dades no implementa cap interfície específica. Això és així perquè es tracta de classes in-

ternes. El programador que l'usi no necessitarà saber de la seva existència, ja que la interacció la realitzarà sempre contra la *façana* (la interfícieGestorPersistenciaEmpresesIPersonal).

Per poder disposar d'un únic accés al component (*façana*) caldrà crear una classe que disposi de tota la funcionalitat de la interfície. La classe instanciarà tots i cada un dels components de persistència específics. La implementació de cada un dels mètodes consistirà només en una única crida al mètode corresponent d'una dels components instanciats.

La implementació de la interfície *façana* la realitzarà la classe GestorPersistenciaEmpresesIPersonalImpl. Cal adonar-se que els components específics d'accés a dades no hereten de ComandesJDBC, i per tant no gestionaran directament la connexió a la base de dades, sinó que serà GestorPersistenciaEmpresesIPersonalImpl l'encarregada de la connexió. D'aquesta manera, evitarem mantenir obertes set connexions. De fet, el gestor mantindrà oberta només una connexió que compartirà amb tots els components específics en el moment de la instanciació.

La classeGestorPersistenciaEmpresesIPersonalImpl derivarà de ComandesJDBC per disposar de la configuració adequada l'SGDB de l'aplicació. Serà necessari declarar un atribut per cada component d'accés a dades.

```
1  public class GestorPersistenciaEmpresesIPersonalImpl
2                      extends ComandesJDBC
3                      implements GestorPersistenciaEmpresesIPersonal {
4      private PaisDao paisDao = null;
5      private PoblacioDao poblacioDao = null;
6      private ComercialDao comercialDao = null;
7      private ZonaDao zonaDao = null;
8      private SectorDao sectorDao = null;
9      private ClientDao clientDao = null;
10     private ProveidorDao proveidorDao = null;
```

La instanciació dels components específics caldrà realitzar-la després d'haver creat la connexió, ja que aquesta es comparteix amb els components DAO passant-la per paràmetre en el moment de la instanciació. Així doncs, sobreescriurem el mètode obrir de ComandesJDBC per tal de fer les instanciacions immediatament després d'obtenir la connexió.

```
1  @Override
2      public void obrir() throws UtilitatPersistenciaException {
3          super.obrir(); //obtenim la connexió
4
5          paisDao = new PaisDao(getConnexio());
6          poblacioDao = new PoblacioDao(getConnexio());
7          comercialDao = new ComercialDao(getConnexio());
8          zonaDao = new ZonaDao(getConnexio());
9      }
```

La resta de mètodes tindran només la crida corresponent.

```
1  @Override
2      public void inserir(Pais pais)
3              throws UtilitatPersistenciaException {
4          paisDao.inserir(pais);
5      }
```

```
    @Override
    public void eliminar(Pais pais)
                throws UtilitatPersistenciaException {
        paisDao.eliminar(pais);
    }

    ...

    @Override
    public void modificar(Comercial entitat)
                      throws UtilitatJdbcSQLException {
        comercialDao.modificar(entitat);
    }

    @Override
    public Comercial obtenirComercial(String clau)
                      throws UtilitatJdbcSQLException {
        return comercialDao.obtenirInstancia(clau);
    }

    ...
}
```

L'accés a dades recau exclusivament sobre els components específics, que seran els que acabin executant les sentències SQL i realitzant l'intercanvi de dades. Per exemple, la implementació de la inserció d'una població a la classe `PoblacioDao` podem implementar-la com segueix:

```
public void inserir(Poblacio valor) throws UtilitatJdbcException{
    StringBuilder sql = new StringBuilder();

    sql.append("INSERT INTO POBLACIO (nom, nom_pais) ");
    sql.append("VALUES('");
    sql.append(valor.getNom());
    sql.append("', '");
    sql.append(valor.getNomPais());
    sql.append("')");

    UtilitatJDBC.executar(getConnexio(), sql.toString());
}
```

La classe `StringBuilder` ens ajudarà a concatenar de forma eficient el text de la sentència SQL. D'aquesta manera podem aprofitar les utilitats de la classe `UtilitatJDBC`, ja que cal recordar que no disposem de cap utilitat que ens faciliti l'execució de sentències parametritzades. La raó és molt senzilla: a les sentències paramètriques, a més d'instanciar-se i executar-se, cal assignar-hi valors als paràmetres, i aquest no és un procés genèric, sinó que depèn exclusivament de la sentència, del nombre de paràmetres que aquesta tingui, del tipus de dada a assignar-hi, etc. Serà necessari implementar-lo en el mateix component específic.

Quelcom de semblant passa amb les consultes, ja que la recollida de les dades depèn també de la sentència i no es pot generalitzar. Així doncs, de moment, quan necessitem fer servir sentències parametritzades o sentències que retornin dades (consultes), no podrem aprofitar les utilitats de la classe `UtilitatJDBC`, sinó que caldrà codificar tota la implementació en el mètode del component específic. Vegem un parell d'exemples de la classe `PaisDao` i `PablacioDao`, respectivament:

```
public Pais obtenirInstancia(String nom) throws UtilitatJdbcSQLException{
    Pais ret = null;
    ResultSet rs = null;
    PreparedStatement pstm=null;
    try {
        pstm = getConnexio()
              .prepareStatement("SELECT nom FROM pais WHERE nom=?");
        pstm.setString(1, nom);
        rs = pstm.executeQuery();
        if(rs.next()){  //Només esperem un únic país.
            ret = new Pais(rs.getString(1));
        }
    } catch (SQLException ex) {
        onError(ex);
    }finally{
        tancaStatement(pstm, rs);
    }
    return ret;
}

public Poblacio obtenirInstancia(String nom, String nomPais)
                                           throws UtilitatJdbcSQLException{
    Poblacio ret = null;
    ResultSet rs = null;
    PreparedStatement pstm=null;
    StringBuilder sql = new StringBuilder();
    sql.append("SELECT nom, nom_pais FROM poblacio ");
    sql.append("WHERE nom=? and nom_pais=?");
    try {
        pstm = getConnexio().prepareStatement(sql.toString());
        pstm.setString(1, nom);
        pstm.setString(2, nomPais);
        rs = pstm.executeQuery();
        if(rs.next()){
            ret = new Poblacio(rs.getString“(”nom),
                           new Pais(rs.getString“(”nom_pais)));
        }
    } catch (SQLException ex) {
        onError(ex);
    }finally{
        tancaStatement(pstm, rs);
    }
    return ret;
}
```

Quan les sentències SQL que no siguin consultes siguin prou senzilles, podrem fer servir la classe `StringBuilder` per concatenar un conjunt de sentències parcials i valors de l'entitat, per tal d'aprofitar les utilitats implementades. Ara bé, si les sentències són consultes o bé presenten certa complexitat de manera que la concatenació ens dificulta l'obtenció d'una visió global, ens resultarà impossible la utilització d'`UtilitatJDBC`.

Difícilment, doncs, ens lliurarem d'escriure una gran quantitat de mètodes amb moltes línies similars (creació dels `Statement` execució, control dels errors, tancaments, etc.) .

Implementació d'un component específic

A continuació descriurem la major part de la implementació del component específic de la persistència dels comercials. Es tracta d'un component amb certa complexitat per tal que copseu les principals dificultats que s'acostumen a trobar en aquest tipus d'implementació.

El gestor encarregat de la persistència dels comercials serà `ComercialDao`:

```
public class ComercialDao{
    ...
}
```

El principal problema per implementar la persistència dels comercials és l'adaptació que cal fer amb la classe `Adreça` i les col·leccions de correus electrònics i telèfons associades a cada comercial.

El tractament que cal donar a l'adreça dels comercials és molt senzill, ja que cal actuar com si aquesta formés part de la mateixa classe `Comercial`. En canvi, pel que fa a les dues col·leccions de les formes de contacte, en no tractar-se pròpiament d'entitats sinó de dades totalment dependents, la responsabilitat de la seva persistència recau totalment sobre la classe propietària, la classe `Comercial` en aquest cas.

Aquesta dependència la trobem ja reflectida en les sentències SQL de creació de les taules *comercial_correuselectronics* i *comercial_telefons*. Com podeu veure, s'ha especificat a la relació forana l'opció *ON DELETE CASCADE*. D'aquesta manera assegurarem des del'SGBD que l'eliminació de qualsevol comercial també implicarà l'eliminació dels seus correus i telèfons. Així ens alliberarem d'anar comprovant l'existència o no d'elements d'informació de contacte i la seva eliminació prèvia abans de començar a eliminar un comercial.

```
CREATE TABLE comercial_correuselectronics(
  comercial_nif VARCHAR(15) NOT NULL,
  correuelectronic VARCHAR(255),
  ordre_correu INTEGER NOT NULL,
  CONSTRAINT comercial_correuselectronics_pkey
      PRIMARY KEY (comercial_nif, ordre_correu),
  CONSTRAINT fk5d6be8c0acb9bc93 FOREIGN KEY (comercial_nif)
      REFERENCES comercial (nif) MATCH SIMPLE
      ON UPDATE CASCADE ON DELETE CASCADE
);
```

Gràcies a això, la sentència d'eliminació és igual de simple que si es tractés d'una taula única.

```
public void  eliminar(Comercial comercial) throws UtilitatJdbcException{
    eliminarComercial(comercial.getNif());
}

public void eliminarComercial(String clau) throws UtilitatJdbcException{
    StringBuilder sql = new StringBuilder();

    sql.append("DELETE FROM COMERCIAL WHERE NIF='");
    sql.append(clau);
    sql.append("'");

    executar(sql.toString());
}
```

La inserció, en canvi, precisarà d'un tractament explícit que tingui en compte que els elements de cada col·lecció s'emmagatzemen en taules diferents. Usarem tres sentències d'inserció parametritzades, malgrat que constituiran només una transacció, de manera que s'emmagatzemi tot o res.

```
public void inserir(Comercial entitat) throws UtilitatJdbcSQLException{
    int param=0;
    boolean autocommit=true;
    PreparedStatement comStm = null;
    PreparedStatement correuStm = null;
    PreparedStatement telefonStm = null;
    StringBuilder comStr = new StringBuilder();
    comStr.append("INSERT INTO comercial(nif,nom,via,codipostal, ");
    comStr.append("poblacio, pais, telefonprincipal, zona_id) ");
    comStr.append("VALUES(?, ?, ?, ?, ?, ?, ?, ?)");
    StringBuilder correuStr = new StringBuilder();
    correuStr.append("INSERT INTO comercial_correuselectronics ");
    correuStr.append("(comercial_nif, ordre_correu,
                                    correuelectronic) ");
    correuStr.append("VALUES(?, ?, ?)");
    StringBuilder telefonStr = new StringBuilder();
    telefonStr.append("INSERT INTO comercial_telefons ");
    telefonStr.append("(comercial_nif, numero, tipus) ");
    telefonStr.append("VALUES(?, ?, ?)");
    try {
        autocommit=getConnexio().getAutoCommit();
        getConnexio().setAutoCommit(false);
        comStm = getConnexio().prepareStatement(comStr.toString());
```

Observeu, en la continuació del codi, que el tractament de l'adreça no presenta cap complicació, ja que s'acaba tractant com un camp més. També cal destacar l'ús de la variable `param` per assenyalar la posició del paràmetre en les sentències SQL d'una forma força dinàmica, coincidint amb la posició real que ocupa dins la seqüència d'instruccions. D'aquesta manera es faciliten les possibles modificacions posteriors, ja sigui eliminant o afegint nous paràmetres. Només caldrà vigilar que l'ordre dels paràmetres dins la seqüència SQL coincideixi sempre amb l'ordre d'assignació de cada valor movent la instrucció Java amunt o avall.

```
comStm.setString(++param, entitat.getNif());
        comStm.setString(++param, entitat.getNom());
        comStm.setString(++param, entitat.getAdreca().getVia());
        comStm.setString(++param,
                             entitat.getAdreca().getCodiPostal());
        comStm.setString(++param,
                                 entitat.getAdreca().getNomPoblacio());
        comStm.setString(++param, entitat.getAdreca().getNomPais());
        comStm.setString(++param,
                                entitat.getInformacioDeContacte()
                                            .getTelefonPrincipal());
```

Si és necessari, podrem assignar el valor *null* a un paràmetre invocant el mètode `setNull` del `PreparedStament`. En aquests casos, cal indicar de quin tipus seria el valor del camp on s'envia fent servir la constant de la classe `Types`.

```
if(entitat.getZona()==null){
            comStm.setNull(++param, java.sql.Types.VARCHAR);
        }else{
            comStm.setString(++param, entitat.getZona().getId());
        }
        comStm.executeUpdate();
```

La classe `informacióDeContacte` obté els telèfons i els correus en forma d'iterador. Fent servir iteradors s'aconsegueix independitzar la seqüència de valors de les col·leccions de les diferents formes d'accés segons els tipus de

col·leccions. Fixeu-vos que la iteració aconseguida a partir de la col·lecció de tipus Map és molt similar a l'aconseguida a partir de la del tipus List que conté els correus electrònics.

```
if(entitat.getInformacioDeContacte().hiHaTelefons()){
            telefonStm = getConnexio()
             .prepareStatement(telefonStr.toString());
            Iterator<Telefon> telfs =
                entitat.getInformacioDeContacte()
                                                    .getTelefons();
            while(telfs.hasNext()){
                Telefon telf = telfs.next();
                param=0;
                telefonStm.setString(++param, entitat.getNif());
                telefonStm.setString(++param, telf.getNumero());
                telefonStm.setString(++param, telf.getTipus());
                telefonStm.executeUpdate();
            }
        }
        if(entitat.getInformacioDeContacte()
                        .hiHaCorreusElectronics()){
            correuStm =getConnexio()
                 .prepareStatement(correuStr.toString());
            Iterator<String> correus = entitat
                        .getInformacioDeContacte()

                            .getCorreusElectronics();
            int ordre=0;
            while(correus.hasNext()){
                String correu = correus.next();
                param=0;
                correuStm.setString(++param, entitat.getNif());
                correuStm.setInt(++param, ordre++);
                correuStm.setString(++param, correu);
                correuStm.executeUpdate();
            }
        }
        getConnexio().commit();
        getConnexio().setAutoCommit(autocommit);
    } catch (SQLException ex) {
        desfer(ex);
        onError(ex);
    }finally{
        tancaStatement(comStm);
        tancaStatement(correuStm);
        tancaStatement(telefonStm);
    }
}
```

Quan haguem de modificar un comercial existent, haurem de tenir en compte que pot tenir emmagatzemat un nombre de telèfons o de correus diferent al que estiguem a punt de guardar. És a dir, no n'hi haurà prou d'inserir els elements de les col·leccions. Com que probablement es tractarà sempre de col·leccions petites, serà preferible eliminar prèviament tots els elements emmagatzemats amb anterioritat abans de procedir a la inserció. Òbviament, hi ha altres maneres de realitzar la modificació, però per col·leccions petites aquesta és prou encertada.

Necessitarem, doncs, cinc sentències: una per actualitzar el valor de la taula *Comercial* amb una sentència UPDATE, dues per eliminar els elements previs corresponents a cada una de les taules *comercial_telefons* i *comercial_correuselectronics*, i dues més per realitzar la inserció en aquestes mateixes taules.

```java
public void modificar(Comercial entitat) throws UtilitatJdbcSQLException{
   int param;
   boolean autocommit=true;
   PreparedStatement comStm = null;
   PreparedStatement insCorreuStm = null;
   PreparedStatement insTelefonStm = null;
   PreparedStatement delCorreuStm = null;
   PreparedStatement delTelefonStm = null;
   StringBuilder comStr = new StringBuilder();
   comStr.append("UPDATE comercial SET nom=?,via=?,codipostal=?,");
   comStr.append("poblacio=?,pais=?,telefonprincipal=?,zona_id=? ");
   comStr.append("WHERE NIF=?");
   StringBuilder delCorreuStr = new StringBuilder();
   delCorreuStr.append("DELETE FROM comercial_correuselectronics ");
   delCorreuStr.append("WHERE comercial_nif=?");
   StringBuilder insCorreuStr = new StringBuilder();
   insCorreuStr.append("INSERT INTO comercial_correuselectronics ");
   insCorreuStr.append("(comercial_nif,ordre_correu, ");
   insCorreuStr.append("correuelectronic) ");
   insCorreuStr.append("VALUES(?, ?, ?)");
   StringBuilder delTelefonStr = new StringBuilder();
   delTelefonStr.append("DELETE FROM comercial_telefons ");
   delTelefonStr.append("WHERE comercial_nif=? ");
   StringBuilder insTelefonStr = new StringBuilder();
   insTelefonStr.append("INSERT INTO comercial_telefons ");
   insTelefonStr.append("(comercial_nif, numero, tipus) ");
   insTelefonStr.append("VALUES(?, ?, ?)");
```

Inicialment, executarem l'actualització de la taula principal:

```java
try {
      param=0;
      autocommit=getConnexio().getAutoCommit();
      getConnexio().setAutoCommit(false);
      comStm = getConnexio().prepareStatement(comStr.toString());
      comStm.setString(++param, entitat.getNom());
      comStm.setString(++param, entitat.getAdreca().getVia());
      comStm.setString(++param,entitat.getAdreca().getCodiPostal());
      comStm.setString(++param, entitat.getAdreca()
                              .getNomPoblacio());
      comStm.setString(++param, entitat.getAdreca().getNomPais());
      comStm.setString(++param, entitat.getInformacioDeContacte()

          .getTelefonPrincipal());
      comStm.setString(++param, entitat.getZona().getId());
      comStm.setString(++param, entitat.getNif());
      comStm.executeUpdate();
```

L'eliminació dels telèfons prèviament emmagatzemats que pertanyin al comercial que s'està actualitzant:

```java
delTelefonStm = getConnexio()
            .prepareStatement(delTelefonStr.toString());
      delTelefonStm.setString(1, entitat.getNif());
      delTelefonStm.executeUpdate();
```

I en cas que la instància disposi de telèfons, s'inseriran un a un tal com es va fer durant la inserció:

```java
if(entitat.getInformacioDeContacte().hiHaTelefons()){
            insTelefonStm = getConnexio()
            .prepareStatement(insTelefonStr.toString());
            Iterator<Telefon> telfs = entitat
                        .getInformacioDeContacte()
                                                .getTelefons();
```

```
            while(telfs.hasNext()){
                Telefon telf = telfs.next();
                param=0;
                insTelefonStm.setString(++param, entitat.getNif());
                insTelefonStm.setString(++param, telf.getNumero());
                insTelefonStm.setString(++param, telf.getTipus());
                insTelefonStm.executeUpdate();
            }
      }
```

Seguidament s'eliminaran els correus:

```
delCorreuStm = getConnexio()
             .prepareStatement(delCorreuStr.toString());
      delCorreuStm.setString(1, entitat.getNif());
      delCorreuStm.executeUpdate();
```

I s'inseriran els nous continguts a la llista de correus de la instància.

```
if(entitat.getInformacioDeContacte()
                     .hiHaCorreusElectronics()){
            insCorreuStm = getConnexio()
            .prepareStatement(insCorreuStr.toString());
            Iterator<String> correus = entitat
                       .getInformacioDeContacte()

                            .getCorreusElectronics();
            int ordre=0;
            while(correus.hasNext()){
                String correu = correus.next();
                param=0;
                insCorreuStm.setString(++param, entitat.getNif());
                insCorreuStm.setInt(++param, ordre++);
                insCorreuStm.setString(++param, correu);
                insCorreuStm.executeUpdate();
            }
      }
```

Si tot ha anat bé, es validarà l'acció i es restaurarà el mode `autocommit` a la connexió.

```
getConnexio().commit();
      getConnexio().setAutoCommit(autocommit);
   } catch (SQLException ex) {
```

Però en cas d'anar malament, es demanarà l'anul·lació de les accions i es llançarà l'excepció encapsulada en una instància d'`UtilitatJdbcSQLException`.

```
desfer(ex);
        onError(ex);
   }finally{
```

Sigui com sigui, cal tancar tots els `Statements` que restin oberts:

```
tancaStatement(comStm);
        tancaStatement(delCorreuStm);
        tancaStatement(insCorreuStm);
        tancaStatement(delTelefonStm);
        tancaStatement(insTelefonStm);
   }
}
```

Per obtenir les dades des de l'SGBD i actualitzar els atributs de les instàncies *Comercial*, farem servir també tres consultes independents, una per taula. Malgrat que seria possible realitzar una única consulta, aquesta generaria tantes files com telèfons i correus hi hagués. Cada fila contindrà un nombre considerable de camps que s'aniran repetint en cada una d'elles, de manera que podria generar-se un trànsit excessiu de dades innecessàries. Per això s'ha optat per independitzar cada una de les consultes.

Cal remarcar que no és trivial prendre una decisió com aquestes, perquè hi pot haver moltes circumstàncies que facin variar l'eficiència aconseguida amb cada alternativa (el trànsit de la xarxa, l'SGBD usat, el tipus de Driver JDBC amb el que connectem, etc.). És per això que davant del dubte, si és possible, s'aconsella realitzar proves d'eficiència implementant diverses alternatives per poder seleccionar la més adient.

```
public Comercial refrescar(Comercial entitat)
                                throws UtilitatJdbcSQLException{
 int camp=0;
 PreparedStatement comQry=null;
 PreparedStatement correusQry=null;
 PreparedStatement telefonsQry=null;
 ResultSet comRs = null;
 ResultSet correusRs = null;
 ResultSet telefonsRs = null;
 StringBuilder comSql = new StringBuilder();
 comSql.append("SELECT c.nom, c.via, c.codipostal, ");
 comSql.append("c.poblacio, c.pais, ");
 comSql.append("c.telefonprincipal, c.zona_id, z.descripcio ");
 comSql.append("FROM comercial c  ");
 comSql.append("LEFT JOIN zona z ON c.zona_id=z.id WHERE nif=?");
 StringBuilder correusSql = new StringBuilder();
 correusSql.append("SELECT ordre_correu, correuelectronic ");
 correusSql.append("FROM comercial_correuselectronics ");
 correusSql.append("WHERE comercial_nif=? ");
 correusSql.append("ORDER BY ordre_correu");
 StringBuilder telefonsSql = new StringBuilder();
 telefonsSql.append("SELECT numero,tipus FROM comercial_telefons ");
 telefonsSql.append("WHERE comercial_nif=?");
```

Inicialment, executarem la consulta principal contra la taula *Comercial*, ja que en cas que el resultat de la consulta no tingués efecte (a causa de la no existència del comercial en l'SGBD) no caldria realitzar les altres.

```
try {
      comQry = getConnexio().prepareStatement(comSql.toString());
      comQry.setString(1, entitat.getNif());
      comRs = comQry.executeQuery();
```

Si la consulta té èxit començarem a assignar valor a la instància.

```
(comRs.getString(++camp));
         entitat.getAdreca().setVia(comRs.getString(++camp));
         entitat.getAdreca().setCodiPostal(comRs.getString(++camp));
         entitat.getAdreca()
    .setPoblacio(new Poblacio(comRs.getString(++camp),
                                new Pais(comRs.getString(++camp))));
         entitat.getInformacioDeContacte()
                      .setTelefonPrincipal(comRs.getString(++camp));
         entitat.setZona(new Zona(comRs.getString(++camp),
                                comRs.getString(++camp)));
```

Llencem la consulta per obtenir tots els correus del comercial.

```
correusQry = getConnexio()
                   .prepareStatement(correusSql.toString());
         correusQry.setString(1, entitat.getNif());
         correusRs = correusQry.executeQuery();
```

Abans de començar a afegir els correus a la llista, eliminem els que pogués tenir prèviament i comencem a afegir correus.

```
entitat.getInformacioDeContacte().buidaCorreus();
         while(correusRs.next()){
             camp=0;
             entitat.getInformacioDeContacte()
               .afegirCorreuElectronic(correusRs.getString(++camp));
         }
```

Realitzem la mateixa operació amb els telèfons.

```
telefonsQry = getConnexio()
            .prepareStatement(telefonsSql.toString());
         telefonsQry.setString(1, entitat.getNif());
         telefonsRs = telefonsQry.executeQuery();
         entitat.getInformacioDeContacte().buidaTelefons();
         while(telefonsRs.next()){
             camp=0;
             entitat.getInformacioDeContacte()
                     .assignarTelefon(telefonsRs.getString(++camp),
                                     telefonsRs.getString(++camp));
         }
      }
  } catch (SQLException ex) {
```

Si es produeix un error SQL es llançarà l'excepció capturada encapsulada en una instància d'`UtilitatJdbcSQLException`.

```
onError(ex);
  }finally{
```

Finalment, tancarem les sentències i resultats que encara restin oberts.

```
tancaStatement(comQry, comRs);
      tancaStatement(correusQry, correusRs);
      tancaStatement(telefonsQry, telefonsRs);
  }
  return entitat;
}
```

A l'hora de recuperar la llista de comercials realitzarem només una recuperació parcial de les seves dades. La raó és evitar el malbaratament de recursos innecessaris i aconseguir una consulta eficient.

Obtenint el *nif* i el *nom* n'hi haurà ben bé prou per treballar amb la llista. Com que el *nif* és la clau primària de la taula de comercials, en cas que es necessiti es podrà demanar un refresc de la instància que vulguem conèixer els detalls.

```
public List<Comercial> obtenirTot() throws UtilitatJdbcSQLException{
   List<Comercial> ret = new ArrayList<Comercial>();
   ResultSet rs = null;
   Statement stm=null;
```

```
    try {
        stm = getConnexio().createStatement();
        rs = stm.executeQuery("SELECT nif, nom FROM COMERCIAL");
        while(rs.next()){
           ret.add( new Comercial(rs.getString(1), rs.getString(2)));
        }
    } catch (SQLException ex) {
        onError(ex);
    }finally{
        tancaStatement(stm, rs);
    }
    return ret;
}
```

El mètode que permet comprovar si una instància ja és persistent o encara no serà el següent:

```
public boolean esPersistent(Comercial entitat) throws UtilitatJdbcSQLException{
    boolean ret = false;
    String sql = "SELECT count(*) FROM COMERCIAL WHERE NIF='"
                    + entitat.getNif() + "'";
    ResultSet rs = null;
    Statement stm=null;
    try {
        stm = getConnexio().createStatement();
        rs = stm.executeQuery(sql);
        rs.next();
        ret = 0<rs.getInt(1);
    } catch (SQLException ex) {
        onError(ex);
    }finally{
        tancaStatement(stm, rs);
    }
    return ret;
}
```

Finalment, per a l'obtenció d'un comercial a partir de la seva clau reutilitzarem el mètode `refrescar`. Caldrà assegurar que el comercial ja sigui persistent; si no, retornarem un valor *null*.

```
public Comercial obtenirInstancia(String clau)
                                    throws UtilitatJdbcSQLException{
    Comercial ret = new Comercial(clau);
    if(esPersistent(ret)){
        refrescar(ret);
    }else{
        ret = null;
    }
    return ret;
}
```

2.2.4 Implementació de la interfície amb rol instanciador

L'altra alternativa proposada consisteix en un gestor que serveix components específics de persistència, d'aquesta manera només instanciarem aquells que siguin necessaris. El gestor, a més, decidirà si crear només una o més connexions per compartir entre els components o quines dades podran compartir.

El gestor, que anomenarem `GestorPersistenciaProducteImpl`, implementarà la interfície `GestorPersistenciaProducte`. La classe obrirà una única connexió i la compartirà amb tots el components. Si es preveiés una gran utilització simultània de la connexió per diversos components crearíem més d'una connexió, però no és el cas.

En aquest cas, doncs, la implementació resultarà molt senzilla. La connexió es gestiona gràcies a l'herència de `ComandesJDBC`, i la instanciació de cada component consistirà només a crear l'objecte passant-li la connexió.

```
public class GestorPersistenciaProducteImpl
            extends ComandesJDBC
                             implements GestorPersistenciaProducte {

    @Override
    public ArticleDao crearArticleDao() {
        return new ArticleDaoImpl(getConnexio());
    }

    @Override
    public EnvasDao crearEnvasDao() {
        return new EnvasDaoImpl(getConnexio());
    }

    @Override
    public MagatzemDao crearMagatzemDao() {
        return new MagatzemDaoImpl(getConnexio());
    }

    @Override
    public ProducteDao crearProducteDao() {
        return new ProducteDaoImpl(getConnexio());
    }

    @Override
    public UnitatDeMesuraDao crearUnitatDeMesuraDao() {
        return new UnitatDeMesuraDaoImpl(getConnexio());
    }
}
```

El nucli de la persistència recaurà sobre els components específics. La codificació d'aquests components d'accés a dades pot ser molt feixuga, de manera que intentarem cercar alguna solució que permeti agilitzar la seva creació.

Millora de la classe UtilitatJdbc

A causa de la singularitat de cada sentència no és possible fer servir la classe `UtilitatJdbc` per automatitzar sentències del tipus `PreparedStatement` ni sentències del tipus consulta, és a dir, que retornin dades. La raó és perquè en executar els mètodes d'`UtilitatJdbc` perdem la informació de com assignar valors als paràmetres o com recuperar les dades retornades. Podem plantejar-nos passar per paràmetre aquesta informació, però cal tenir en compte que les sentències SQL poden arribar-se a complicar molt, la qual cosa pot repercutir sobre l'eficiència i sobre la claredat de les sentències. Hi ha, però, una altra solució: delegar l'assignació de paràmetres o l'obtenció de resultats al component que disposi de la informació passant les instàncies per paràmetre. Abans, però, caldrà definir un conjunt d'interfícies que permetin aquesta delegació.

En cas que es tracti d'una sentència d'actualització parametritzada, caldrà disposar d'un mètode per obtenir la sentència i d'un altre per fer l'assignació. Anomenarem `getStatement` al mètode per obtenir una cadena amb la sentència SQL parametritzada, i `setParameter` al mètode que manipularà la instància de `PreparedStatement` generada a partir de la sentència SQL assignant valors als paràmetres de la sentència SQL.

```
public interface JdbcPreparedDao{
    String getStatement();
    void setParameter(PreparedStatement pstm) throws SQLException;
}
```

En cas que la sentència retorni dades perquè es tracta d'una consulta, necessitarem a més un tercer mètode per instanciar un objecte a partir de les dades obtingudes i retornar-lo. El mètode s'anomenarà `writeObject`.

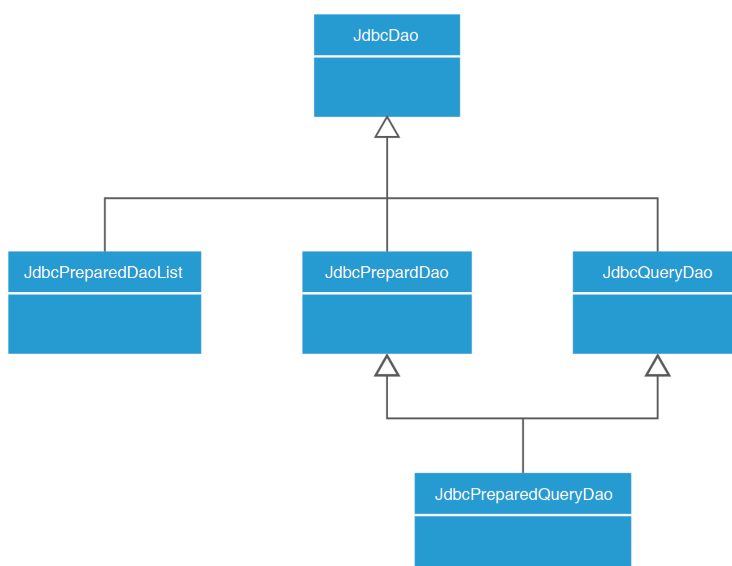
```
public interface JdbcPreparedQueryDao{
    String getStatement();
    void setParameter(PreparedStatement pstm) throws SQLException;
    Object writeObject(ResultSet rs) throws SQLException;
}
```

El mètode `writeObject` rebrà una instància de `ResultSet` amb les dades obtingudes després de l'execució de la sentència SQL.

En cas que la consulta no tingués paràmetres, només necessitarem els mètodes `getStatement` i `writeObject`.

Per tal de disposar de totes les varietats possibles organitzarem les interfícies com una jerarquia, de manera que cada una d'elles declari només un dels mètodes o bé una combinació d'aquests (vegeu figura 2.5).

Figura 2.5. Jerarquia d'interfícies per poder cobrir totes les possibilitats de delegació en l'assignació de paràmetres i obtenció de resultats



JdbcDao representa una sentència SQL i es pot usar quan es tracti de sentències no parametritzades. JdbcPreparedDao permet assignar valors als paràmetres d'una sentència SQL. La podeu usar per a sentències paramètriques de modificació. JdbcQueryDao declara la funcionalitat de rebre per paràmetre un `ResultSet`. Es tracta d'un mètode que servirà per omplir la instancia ResultSet amb els resultats d'una consulta. JdbcQueryDao només admet sentències de consulta SQL no parametritzades. La doble herència de la interfície `JdbcPreparedQueryDao` permet usar aquesta interfície per a sentències de consulta SQL paramètriques.

Cal fer esment especial de `JdbcPreparedDaoList`, una interfície per usar quan necessitem executar una mateixa sentència SQL diverses vegades fent servir en cada una d'elles dades diferents emmagatzemades en una llista o vector al qual s'hi pugui accedir de forma indexada.

```
1  public interface JdbcDao {
2      /*
3       * Obté la sentència SQL emmagatzemada en aquest objecte.
4       * @return una cadena amb la sentència SQL
5       */
6      String getStatement();
7  }
8
9  public interface JdbcPreparedDao extends JdbcDao{
10     /*
11      * Aquest mètode té per objectiu fer l'assignació de valors als
12      * paràmetres de l'objecte PreparedStatement. El paràmetre s'ha
13      * creat amb la cadena SQL que encapsula aquest objecte
14      * JdbcPreparedDao.
15      * @param pstm. Aquest paràmetre és un objecte de tipus
16      * PreparedStatement i s'ha creat amb la cadena SQL que
17      * encapsula aquest objecte JdbcPreparedDao.
18      * @throws SQLException
19      */
20     void setParameter(PreparedStatement pstm) throws SQLException;
21 }
22
23 public interface JdbcQueryDao extends JdbcDao{
24     /*
25      * Instancia un objecte amb les dades extretes del ResultSet
26      * passat per paràmetre, el qual conté els resultats de
27      * l'execució de la sentència SQL que aquest objecte
28      * JdbcQueryDao representa.
29      * @param rs és el ResultSet que conté els resultats de
30      * l'execució de la sentència SQL que aquest objecte
31      * JdbcQueryDao representa.
32      * @return La instància construïda.
33      * @throws SQLException
34      */
35     Object writeObject(ResultSet rs) throws SQLException;
36 }
37
38 public interface JdbcPreparedQueryDao
39             extends JdbcPreparedDao, JdbcQueryDao {
40 }
41
42 public interface JdbcPreparedDaoList extends JdbcDao{
43     /*
44      * Aquest mètode té per objectiu fer l'assignació de valors als
45      * paràmetres des d'una llista de dades indexada. El paràmetre
46      * pstm s'ha creat a partir de la cadena SQL i el valor id
47      * indica l'índex de la llista de dades que toca executar.
48      * Totes les execucions es realitzaran com una única transacció.
49      * @param id és l'índex corresponent a un conjunt de dades amb
50      * les quals assignar els paràmetres d'una de les execucions de la
51      * sentència SQL retornada per getStatemet().
```

```
52      * @param pstm és el PreparedStatement corresponent a la cadena
53      * SQL.
54      * @throws SQLException
55      */
56     void setParameter(int id, PreparedStatement pstm)
57                             throws SQLException;
58 
59     /*
60      * Obté el nombre de dades per a les quals caldrà executar la
61      * sentència SQL.
62      * @return enter nombre de dades per a les quals caldrà executar
63      * la sentència SQL.
64      */
65     int sizeList();
66 }
```

Gràcies a aquestes interfícies podem estendre les utilitats JDBC afegint noves utilitats. Ho farem estenent la classe `UtilitatJdbc`:

```
1 public class UtilitatJdbcPlus extends UtilitatJdbc{
```

S'hi afegeixen amb set mètodes diferents per executar sentències SQL contingudes en algun objecte de la jerarquia `JdbcDao`. Dos d'ells permetran executar sentències paramètriques encapsulades en objectes de tipus `JdbcPreparedDao` o bé `JdbcPreparedDaoList`. Vegem com haurem d'implementar el primer mètode:

```
1  public static void executar(Connection con,
2                              JdbcPreparedDao jdbcDao)
3                                      throws UtilitatJdbcSQLException{
4      boolean autocommit=true;
5      PreparedStatement stm = null;
6      try {
7          autocommit=con.getAutoCommit();
8          con.setAutoCommit(true);
9          stm = con.prepareStatement(jdbcDao.getStatement());
10         jdbcDao.setParameter(stm);
11         stm.executeUpdate();
12         con.setAutoCommit(autocommit);
13     } catch (SQLException ex) {
14         onError(ex);
15     }finally{
16         tancaStatement(stm);
17     }
18 }
```

Fixeu-vos com el paràmetre del tipus `JdbcPreparedDao` permet obtenir la sentència per crear una instància de `PreparedStatmenet`. Un cop instanciada la sentència, s'executarà l'assignació de paràmetres invocant el mètode `setParameter`.

Per tal de poder reunir en un vector diversos objectes `JdbcDao` amb independència del tipus, s'ha codificat un mètode que rep un vector, comprova el tipus d'objecte i executa la sentència de forma més adequada:

```
1  public static void executar(Connection con, JdbcDao[] jdbcDaos)
2                                      throws UtilitatJdbcSQLException{
3      boolean autocommit=true;
4      Statement stm = null;
5      try {
6          autocommit=con.getAutoCommit();
7          con.setAutoCommit(false);
8          for(JdbcDao dao: jdbcDaos){
9              if(dao instanceof JdbcPreparedDao){
```

```
                stm = con.prepareStatement(dao.getStatement());
                ((JdbcPreparedDao)dao).
            setParameter((PreparedStatement) stm);
                ((PreparedStatement)stm).executeUpdate();
            }else if(dao instanceof JdbcPreparedDaoList){
                stm = con.prepareStatement(dao.getStatement());
                for(int n=0;
                n<((JdbcPreparedDaoList)dao).sizeList();
                n++){
                    ((JdbcPreparedDaoList)dao)
                .setParameter(n, (PreparedStatement) stm);
                    ((PreparedStatement)stm).executeUpdate();
                }
            }else{
                stm = con.createStatement();
                stm.executeUpdate(dao.getStatement());
            }
        }
        con.commit();
        con.setAutoCommit(autocommit);
    } catch (SQLException ex) {
        desfer(con, ex);
        onError(ex);
    }finally{
        tancaStatement(stm);
    }
}
```

En cas que es tracti d'objectes `JdbcDao` que continguin consultes, cada tipus d'objecte (paramètric i no paramètric) podrà generar o bé un únic objecte (mètodes `obtenirObjecte`), o bé una llista d'objectes (mètodes `obtenirLlista`). Així, per exemple, les implementacions de les versions parametritzades es codifiquen:

```
public static Object obtenirObjecte(Connection con,
                            JdbcPreparedQueryDao jdbcDao)
                                    throws UtilitatJdbcSQLException{
    Object ret= null;
    boolean autocommit=true;
    PreparedStatement stm = null;
    ResultSet rs = null;
    try {
        autocommit=con.getAutoCommit();
        con.setAutoCommit(true);
        stm = con.prepareStatement(jdbcDao.getStatement());
        jdbcDao.setParameter(stm);
        rs = stm.executeQuery();
        if(rs.next()){
            ret=jdbcDao.writeObject(rs);
        }

        con.setAutoCommit(autocommit);
    } catch (SQLException ex) {
        onError(ex);
    }finally{
        tancaStatement(stm, rs);
    }
    return ret;
}
```

Fixeu-vos que, a més de crear la sentència, assignar-hi els paràmetres i executar-la, cal comprovar si hi ha alguna dada retornada, i en cas afirmatiu instanciar un objecte a partir del `ResultSet` obtingut invocant el mètode `writeObject`.

L'obtenció d'una llista presenta un codi molt similar, però en comptes de recuperar un únic objecte i retornar-lo es crea una llista on s'hi afegeixen tots el objectes creats a partir de les dades del ResultSet.

```
public static List obtenirLlista(Connection con,
                            JdbcPreparedQueryDao jdbcDao)
                                    throws UtilitatJdbcSQLException{
    List ret = new ArrayList();
    boolean autocommit=true;
    PreparedStatement stm = null;
    ResultSet rs = null;
    try {
        autocommit=con.getAutoCommit();
        con.setAutoCommit(true);
        stm = con.prepareStatement(jdbcDao.getStatement());
        jdbcDao.setParameter(stm);
        rs = stm.executeQuery();
        while(rs.next()){
            ret.add(jdbcDao.writeObject(rs));
        }

        con.setAutoCommit(autocommit);
    } catch (SQLException ex) {
        onError(ex);
    }finally{
        tancaStatement(stm, rs);
    }
    return ret;
}
```

Implementació dels components de persistència específics usant la classe UtilitatJdbc millorada

La manera de posar en pràctica les millores que acabem d'elaborar quan sigui necessari codificar un mètode que executi una sentència SQL de qualsevol tipus, consisteix a crear una instància de la jerarquia JdbcDao adequada al tipus de sentència que calgui executar. La instància haurà de mantenir en memòria la sentència SQL i haurà de resoldre específicament els mètodes d'assignació de valors i obtenció de dades. Amb la instància creada caldrà invocar el mètode executar, obtenirObjecte o obtenirLlista, segons sigui necessari.

Malgrat que es poden codificar classes específiques que implementin les interfícies, és molt còmode fer servir classes anònimes incrustades, ja que permet una total versatilitat a l'hora de la codificació. Vegem, per exemple, el mètode de la classe UnitatDeMesuraDaoImpl, que permet modificar les dades emmagatzemades al'SGDB a partir de l'estat d'una UnitatDeMesura.

```
@Override
public void modificar(final UnitatDeMesura entitat)
                throws UtilitatJdbcSQLException {

    JdbcPreparedDao jdbcDao = new JdbcPreparedDao() {
        @Override
        public String getStatement() {
          StringBuilder sql = new StringBuilder();
            sql.append("UPDATE unitat_de_mesura SET descripcio=? ");
            sql.append("WHERE simbol=?");
          return sql.toString();
        }

```

```
14          @Override
15          public void setParameter(PreparedStatement pstm)
16                              throws SQLException {
17              pstm.setString(1, entitat.getDescripcio());
18              pstm.setString(2, entitat.getSimbol());
19          }
20      };
21      UtilitatJdbcPlus.executar(con, jdbcDao);
22  }
```

Observeu com s'implementa una classe anònima en el mateix moment de la instanciació. Es tracta d'una forma molt còmoda d'estructurar el codi, ja que els mètodes de la interfície `JdbcDao` implementats es codifiquen dins el mateix mètode en què cal fer la invocació d'una execució o obtenció de dades mitjançant la `UtilitatJdbcPlus`. En tots els casos funciona de forma semblant, encara que la sentència sigui complexa. Vegem-ne un altre exemple, aquest cop de la classe `MagatzemDaoImpl`. Es tracta del mètode que obté la llista d'estoc d'un magatzem:

```
1   //CONSTANTS
2   private static final int POS_COL_M_ID=1;
3   private static final int POS_COL_M_DESC=2;
4   private static final int POS_COL_ME_QUAN=3;
5   ...
6   private static final int POS_COL_E_UNIT_ID=15;
7   private static final int POS_COL_EU_DESC=16;
8
9   ...
10
11  @Override
12  public List<ProducteEnEstoc> getEstoc(final Magatzem entitat)
13                  throws UtilitatJdbcSQLException{
14      List<ProducteEnEstoc> ret = new ArrayList<ProducteEnEstoc>();
15
16      JdbcPreparedQueryDao jdbcdao = new JdbcPreparedQueryDao() {
17          @Override
18          public String getStatement() {
19              StringBuilder sql = new StringBuilder();
20              sql.append("SELECT m.id, m.descripcio, me.quantitat, ");
                                        sql.append("me.ubicacio, ");
21              sql.append("me.producte_id, p.tipus_producte, p.preu,");
                                        sql.append("p.article_id, a.descripcio, "
                     );
22              sql.append("p.unitat_simbol, pu.descripcio, ");
23              sql.append("p.envas_id, e.quantitat, e.tipus, ");
                                       sql.append("e.unitat_simbol, eu.descripcio
                      ");
24              sql.append("FROM MAGATZEM m ");
25              sql.append("JOIN MAGATZEM_ESTOC me ");
26              sql.append("ON m.id=me.magatzem_id ");
27              sql.append("JOIN PRODUCTE p ON me.producte_id=p.id ");
28              sql.append("JOIN ARTICLE a ON p.article_id=a.id ");
29              sql.append("LEFT JOIN unitat_de_mesura pu ");
30              sql.append("ON p.unitat_simbol=pu.simbol ");
31              sql.append("LEFT JOIN ENVAS e ON p.envas_id=e.id ");
32              sql.append("LEFT JOIN unitat_de_mesura eu ");
33              sql.append("ON e.unitat_simbol=eu.simbol ");
34              sql.append("WHERE m.id=?");
35              return sql.toString();
36          }
37
38          @Override
39          public void setParameter(PreparedStatement pstm)
40                              throws SQLException {
41              pstm.setString(1, entitat.getId());
42          }
43
```

```
44          @Override
45          public Object writeObject(ResultSet rs) throws SQLException {
46              ProducteEnEstoc producteEnEstoc;
47              entitat.setDescripcio(rs.getString(POS_COL_M_DESC));
48              Producte producte = ProducteDaoImpl.crearProducte(
49                        rs.getInt(POS_COL_P_TIPUS_PROD));
50              producte.setId(rs.getLong(POS_COL_ME_PROD_ID));
51              producte.setArticle(new Article(
52                        rs.getLong(POS_COL_P_ARTICLE_ID),
53                                    rs.getString(POS_COL_A_DESC)));
54              producte.setPreu(rs.getDouble(POS_COL_P_PREU));
55              if(rs.getInt(POS_COL_P_TIPUS_PROD)==
56                  ProducteDaoImpl.TIPUS_PRODUCTE_A_GRANEL){
57                  ((ProducteAGranel)producte).setUnitat(
58                          new UnitatDeMesura(
59                      rs.getString(POS_COL_P_UNIT_ID),
60                      rs.getString(POS_COL_PU_DESC)));
61              }else if(rs.getInt(POS_COL_P_TIPUS_PROD)==
62                 ProducteDaoImpl.TIPUS_PRODUCTE_ENVASAT){
63                  ((ProducteEnvasat)producte).setEnvas(
64                    new Envas(rs.getLong(POS_COL_P_ENVAS_ID),
                                            rs.getString(POS_COL_E_TIPUS),
65                      rs.getDouble(POS_COL_E_QUANT),
66                      new UnitatDeMesura(
67            rs.getString(POS_COL_E_UNIT_ID),
68                        rs.getString(POS_COL_EU_DESC))));
69              }
70              Estoc estoc = new Estoc(rs.getString(POS_COL_ME_UBIC),
71                                      rs.getDouble(POS_COL_ME_QUAN));
72              producteEnEstoc = new ProducteEnEstoc(producte,
73                                  estoc, entitat);
74              return producteEnEstoc;
75          }
76      };
77
78      ret = UtilitatJdbcPlus.obtenirLlista(con, jdbcdao);
79      return ret;
80  }
```

Com es pot veure, l'ús d'aquestes utilitats permet centrar la codificació només en les coses específiques (la sentència SQL, l'assignació de paràmetres i la instanciació dels resultats), alliberant-nos de la tasca del tractament d'errors, de la creació de l'objecte `Statement` i de la seva execució.

Podeu trobar la resta de codi de les classes implementades a la biblioteca que se us adjunta amb aquest mòdul. Consulteu els annexos per poder-hi accedir. Cal tenir en compte que es tracta només d'exemples i que hi pot haver altres maneres d'aprofitar millor les utilitats esmentades.

2.3 Empaquetatge del component de persistència

A més de generar el fitxer JAR, ens interessarà també incloure dins el component els fitxers de configuració els *scripts* SQL de creació i modificació de taules i d'incorporació de dades, els fitxers de documentació JavaDoc i algun que altre fitxer de documentació amb informació sobre el component (instal·lació, configuració, etc.).

Per defecte, NetBeans situa el fitxer JAR en un directori del projecte anomenat *dist*. Es tracta d'un fitxer temporal que s'elimina i torna a generar cada cop que

compilem i construïm el projecte. És per això que no podem situar cap fitxer que no generi l'IDE perquè s'esborraria cada cop que féssim compilació i generació del component.

La solució passa per situar els fitxers extres en algun directori que no sigui *dist* i copiar-los cada cop que compilem i generem el component. La còpia es pot automatitzar modificant el fitxer de configuració del projecte. Es tracta d'un fitxer en format *Ant* al qual hi afegirem les accions extres de copiar els fitxers que ens interessi.

Abans d'explicar com afegir aquest accions, cal saber on situarem els nostres fitxers. Els fitxers de configuració, junt amb els *scripts*, els situarem en un directori anomenat *cfg* que pengi del mateix directori on s'executi l'aplicació. Per fer proves, si agafem les opcions per defecte de NetBeans hauríem de situar el directori de configuració a l'arrel de la carpeta *src* (on es troben els paquets de codi).

Es tracta d'una solució poc elegant, perquè es barregen el codi i els fitxers de configuració. Hi ha, però, una altra solució. NetBeans permet indicar per a cada projecte un directori d'execució. Editeu les propietats del projecte (aneu al menú *File* i escolliu l'opció *Project Properties*). Quan s'obri la finestra de propietats, cliqueu sobre la categoria *Run*. Us apareixerà un camp anomenat *Working Directory*. Usant el botó de navegació podreu assignar una carpeta diferent a l'arrel del codi font, com a directori d'execució de l'aplicació.

Suposem que heu assignat una carpeta anomenada *runFolder* ubicada al mateix directori del projecte. Allà hi hauríeu de tenir una carpeta anomenada *cfg* amb els fitxers indicats (*.properties* i *.sql*).

Imagineu també que sigui necessari crear un document per explicar com cal configurar el component. Situarem els documents informatius en un directori, al mateix nivell que la carpeta *runFolder* però sota el nom de *docuFolder*.

Ara caldrà automatitzar la còpia dels fitxers a la carpeta *dist,* que és on ubicarem el nostre component. Per configurar l'automatització clicarem sobre la pestanya *Files* de la finestra de l'esquerra per tal que ens apareguin tots els fitxers del projecte. Editarem el fitxer anomenat build.xml. Obtindrem un resultat semblant a:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  ...
3  <project name="ComponentsDePersistencia" default="default" basedir=".">
4      <description>
5         Builds, tests, and runs the project ComponentsDePersistencia.
6      </description>
7      <import file="nbproject/build-impl.xml"/>
8  
9      ...
10 </project>
```

Afegirem, després de l'etiqueta *import*, un nova tasca anomenada *-post-clean*. Es tracta d'una tasca reconeguda per NetBeans, la qual s'executarà sempre immediatament després d'esborrar el directori destí del fitxer JAR.

```
<?xml version="1.0" encoding="UTF-8"?>
...
<project name="ComponentsDePersistencia" default="default" basedir=".">
    <description>
       Builds, tests, and runs the project ComponentsDePersistencia.
    </description>
    <import file="nbproject/build-impl.xml"/>

    <target name="-post-clean">
        <copy todir="${dist.dir}/cfg" failonerror="false">
            <fileset dir="${work.dir}/cfg"/>
        </copy>
        <copy todir="${dist.dir}/doc" failonerror="false">
            <fileset dir="docuFolder"/>
        </copy>
    </target>
    ...
</project>
```

Ant disposa d'una acció representada per l'etiqueta *copy* que en invocar-se realitzarà una còpia al directori indicat per l'atribut *todir* del contingut indicat per l'etiqueta *fileset*. En el nostre cas, com que desitgem ubicar la documentació i la configuració en directoris diferents, definirem dues accions de còpia. NetBeans ha definit un conjunt de propietats (variables d'*Ant*) per emmagatzemar-hi el nom de determinats directoris. Així, per exemple, el directori de sortida on es genera el fitxer JAR es troba emmagatzemat a la propietat *dist.dir* . El directori de treball que hem definit més amunt es troba a *work.dir*. Per substituir les propietats pel seu valor cal tancar el nom d'aquest entre claus i anteposar-hi el símbol *$*. Així, l'acció:

```
<copy todir="${dist.dir}/cfg" failonerror="false">
            <fileset dir="${work.dir}/cfg"/>
        </copy>
```

significa que cal copiar tot el que hi hagi al directori *cfg* ubicat a la carpeta de treball (cadena continguda a *work.dir*), al directori *cfg* situat a la carpeta de destí del component *jar* (cadena continguda a *dist.dir*). L'atribut *failonerror* permet evitar que es produeixi un error durant la invocació de la tasca en cas que la còpia generi algun error.

L'automatització de la còpia permet despreocupar-nos de la compilació assegurant que sempre disposem de tots els fitxers necessaris a la carpeta on es genera el component apte per ser desplegat i integrat allà on calgui.

Per tal de completar tot el component cal també que generem el Javadoc. Podem fer la generació des del menú *Run* de NetBeans, seleccionant *Generate Javadoc*. La generació del Javadoc crearà els documents HTML amb la documentació de les classes del component i els situarà a la carpeta *javadoc* que ubicarà al directori de destinació del fitxer JAR.

Un cop creat el component, també podem generar un arxiu ZIP amb tots els continguts del component per tal que resulti més fàcil el seu desplegament.

Aquesta acció es pot realitzar manualment, però si es desitja*Ant* disposa també d'una acció per comprimir fitxers en format ZIP.

Podem fer servir la mateixa tasca que la usada per a la còpia; per assegurar que l'arxiu comprimit es crea també amb el *javadoc* inclòs, caldrà indicar que la tasca és dependent d'una altra tasca, la generació del *javadoc* (`-javadoc-build`).

```
<?xml version="1.0" encoding="UTF−8"?>
...
<project name="ComponentsDePersistencia" default="default" basedir=".">
    <description>
       Builds, tests, and runs the project ComponentsDePersistencia.
    </description>
    <import file="nbproject/build−impl.xml"/>

    <target depends="−javadoc−build" name="−post−clean">
        <copy todir="${dist.dir}/cfg" failonerror="false">
            <fileset dir="${work.dir}/cfg"/>
        </copy>
        <copy todir="${dist.dir}/doc" failonerror="false">
            <fileset dir="docuFolder"/>
        </copy>
        <zip destfile="${ant.project.name}.zip"
             basedir="${dist.dir}"
             update="true" />
    </target>
    ...
</project>
```

Concretament, aquesta modificació generarà un arxiu ZIP que s'anomenarà igual que el projecte, amb tot el contingut del directori destí (fitxer jar i directoris *javadoc*, *cfg* i *doc*).

2.4 Desplegament del component de persistència

Malgrat que el desplegament del component depèn en gran mesura de l'aplicació o component on calgui incorporar-lo, cal tenir previst el procés de desplegament descrivint quins elements incorporarà el componenti quins processos de configuració caldrà realitzar per integrar-lo.

Així, en el nostre cas caldria explicar que el paquet *zip* contindrà un fitxer *jar* que caldrà afegir com a biblioteca de l'aplicació, i una carpeta anomenada *cfg* on trobarem tots els recursos i fitxers de configuració. Concretament, cal modificar el fitxer *ComandesJDBC.properties*, per indicar l'usuari i la contrasenya d'accés al'SGBD, el nom de la classe del *driver* de la nostra biblioteca JDBC de connexió al'SGBD, així com la *url* de la base de dades amb la qual treballarem.

També serà necessari obtenir i afegir la biblioteca java del *driver* a l'aplicació i fer les instal·lacions oportunes (si cal) en el sistema operatiu.

Amb la descripció d'aquesta darrera fase podem donar per finalitzada la construcció del component de persistència.