POO i gestors de base de dades

Joan Arnedo Moreno

Índex

Introducció **5**

Resultats d'aprenentatge **7**

1 Introducció al diagrama estàtic UML **9**
1.1 Esquema general de resolució de problemes mitjançant orientació a objectes . . . 10
1.2 Definició de classes mitjançant UML . . . 11
1.2.1 Definició d'atributs . . . 12
1.2.2 Definició de mètodes . . . 14
1.3 Relacions entre classes . . . 15
1.3.1 Associacions . . . 15
1.3.2 Agregacions . . . 19
1.3.3 Composicions . . . 19
1.3.4 Classes associatives . . . 20
1.3.5 Associacions reflexives . . . 22
1.3.6 Herència . . . 23
1.4 Exemples de diagrames estàtics . . . 23
1.4.1 Una agenda . . . 24
1.4.2 Un reproductor multimèdia . . . 25
1.4.3 Una aplicació de gestió . . . 26
1.5 Els diagrama estàtic i els mapes d'objectes . . . 27

2 Aplicacions amb BD no orientades a objectes **29**
2.1 Traducció del Model a una BD relacional . . . 30
2.2 El llenguatge SQL . . . 33
2.2.1 Tipus de dades . . . 34
2.2.2 Gestió de taules . . . 38
2.2.3 Consulta de dades . . . 40
2.2.4 Manipulació de dades . . . 51
2.3 JDBC . . . 54
2.3.1 Càrrega del controlador . . . 55
2.3.2 Establiment de la connexió . . . 56
2.3.3 Execució de sentències SQL . . . 58
2.3.4 Tancament de la connexió . . . 65
2.3.5 Exemple d'aplicació JDBC: El gestor d'encàrrecs . . . 66
2.4 Seguretat en l'accés a la BD . . . 69

3 Aplicacions amb BD orientades a objectes **73**
3.1 Els llenguatges ODL i OQL . . . 73
3.1.1 El llenguatge ODL . . . 74
3.1.2 El llenguatge OQL . . . 75
3.2 La llibreria db4O . . . 76
3.2.1 Obertura de la BDOO . . . 77
3.2.2 Emmagatzematge de nous objectes . . . 79

3.2.3 Cerca d'objectes 82
3.2.4 Actualitzacions d'objectes 86
3.2.5 Esborrat d'objectes 88
3.3 JDO (Java Data Objects) 89
3.3.1 Instanciació d'un objecte PersistenceManager 91
3.3.2 Interacció amb la BD 91

Introducció

El conjunt d'objectes existents a memòria en un moment donat de l'execució del programa indiquen l'estat intern de l'aplicació. Aquest va canviant d'acord a les interaccions de l'usuari. En tractar-se d'objectes emmagatzemats a la memòria de l'ordinador, quan l'aplicació deixa d'executar-se, totes les dades associades desapareixen per sempre, no hi ha persistència. A més a més, aquestes dades només són accessibles per l'aplicació que ha instanciat els objectes.

Sovint, és necessari assolir la persistència de dades, de manera que sigui possible conservar certa informació entre diferents execucions de l'aplicació, o compartir dades entre diferents aplicacions. La manera més simple de fer-ho és desant totes les dades dins d'un fitxer, però aquesta solució és poc eficient per a aplicacions complexes, o simplement no és factible si cal compartir informació entre aplicacions que no s'executen sobre el mateix ordinador. Les aplicacions modernes normalment s'inclinen per usar alternatives millors, com les bases de dades.

En aquesta unitat se segueix amb l'estudi de les classes accessibles a través de l'API del Java, ja que, igual que existeix un conjunt de classes accessibles mitjançant l'API del Java per gestionar errors o interfícies d'usuari, també hi ha una part que gestiona l'accés a bases de dades i que, per tant, val la pena estudiar amb més detall. Ara bé, per a alguns casos de bases de dades l'API no és suficient i no disposa de cap classe que resolgui la tasca que cal dur a terme. Quan això succeeix, cal anar més enllà i estudiar com usar llibreries creades per altres programadors diferents dels creadors del Java i incorporar-les als vostres programes.

Abans de poder establir com fer millor ús d'una base de dades, però, cal tenir ben clar de quina manera s'estructurarà tota la informació dins el vostre programa. Quins objectes hi ha i quines dades contenen. Per assolir aquesta fita, una eina molt útil és el llenguatge UML, que permet representar el disseny d'aplicacions orientades a objectes de manera relativament intuïtiva.

L'apartat "Introducció al diagrama estàtic UML" dóna una visió sobre com dur a terme el disseny d'aplicacions utilitzant aquest llenguatge d'especificació d'aplicacions orientades a objectes. D'aquesta manera, abans de ni tan sols començar a escriure codi font, el programador ja pot definir clarament sobre quina informació ha de tractar l'aplicació i com s'estructura a memòria, de la mateixa manera que un arquitecte no comença a construir un gratacels sense haver fet uns plànols abans.

L'apartat "Aplicacions amb BD no orientades a objectes" presenta els aspectes bàsics per poder assolir la persistència de les dades mitjançant el mecanisme més popular per desplegar aplicacions a gran escala: una base de dades relacionals. Aquest sistema es basa en taules i no està vinculat explícitament a les aplicacions

orientades a objectes, sinó a qualsevol tipus d'aplicació. Per tant, per fer-ne ús cal un pas previ de traducció del model orientat a objectes a un model relacional.

L'apartat "Aplicacions amb BD orientades a objectes" introdueix un nou tipus de bases de dades, relativament recents, on la informació emmagatzemada sí que s'estructura d'una manera semblant a com es fa amb els objectes a memòria dins una aplicació orientada a objectes. Per tant, això permet estalviar-se el pas de la traducció prèvia i accedir a les seves dades d'una manera més natural.

Un aspecte molt important al llarg de tota la unitat és ser conscient de la impossibilitat de descriure fins la darrera funcionalitat de totes les classes implicades tant en una interfície gràfica com en la gestió de l'entrada/sortida per assolir persistència. Les llibreries de Java són molt extenses i complexes. Per tant, és inevitable haver d'acudir a la documentació oficial (l'anomenada API de Java) per poder estudiar amb detall quins mètodes ofereix cada classe i per a què serveixen.

Resultats d'aprenentatge

En finalitzar aquesta unitat l'alumne/a:

1. Gestiona informació emmagatzemada en bases de dades relacionals mantenint la integritat i consistència de les dades.

- Identifica les característiques i mètodes d'accés a sistemes gestors de bases de dades relacionals.
- Programa connexions amb bases de dades.
- Escriu codi per emmagatzemar informació en bases de dades.
- Crea programes per recuperar i mostrar informació emmagatzemada en bases de dades.
- Efectua esborrats i modificacions sobre la informació emmagatzemada.
- Crea aplicacions que executin consultes sobre bases de dades.
- Crea aplicacions per a possibilitar la gestió d'informació present en bases de dades relacionals.

2. Gestiona informació emmagatzemada en bases de dades objecte- relacionals mantenint la integritat i consistència de les dades.

- Identifica les característiques de les bases de dades objecte-relacionals.
- Analitza la seva aplicació en el desenvolupament d'aplicacions mitjançant llenguatges orientats a objectes.
- Classifica i analitza els diferents mètodes que suporten els sistemes gestors de bases de dades per a la gdestió de la informació emmagatzemada de forma objecte-relacional.
- Programa aplicacions que emmagatzemen objectes en bases de dades objecte-relacionals.
- Realitza programes per recuperar, actualitzar i eliminar objectes de les bases de dades objecte-relacionals.
- Realitza programes per emmagatzemar i gestionar tipus de dades estructurades, compostos i relacionats.

3. Utilitza bases de dades orientades a objectes, analitzant les seves característiques i aplicant tècniques per mantenir la persistència de la informació

- Identifica les característiques de les bases de dades orientades a objectes.

- Analitza la seva aplicació en el desenvolupament d'aplicacions mitjançant llenguatges orientats a objectes.
- Defineix les estructures de dades necessàries per a l'emmagatzematge d'objectes.
- Classifica i analitza els diferents mètodes suporten els sistemes gestors per a la gestió de la informació emmagatzemada.
- Programa aplicacions que emmagatzemin objectes en les bases de dades orientades a objectes.
- Realitza programes per recuperar, actualitzar i eliminar objectes de les bases de dades orientades a objectes.
- Realitza programes per emmagatzemar i gestionar tipus de dades estructurades, compostos i relacionats.

1. Introducció al diagrama estàtic UML

A l'hora de resoldre un problema, no sol ser una bona idea començar a anar per feina sense aturar-se un moment a pensar què es vol fer i, en el cas de problemes complexos, cal idear sobre el paper un petit esquema o croquis de què cal fer. Llavors, sobre el paper, es pot reflexionar si la solució sembla tenir sentit i es pot anar refinant, ja sigui un mateix o amb l'ajut d'altres entesos. Fins que no s'arriba a una solució satisfactòria, no es comença la feina. En el cas d'un problema d'una disciplina complexa, evidentment, cal anar una mica més enllà i no n'hi ha prou només amb un croquis *ad hoc*. Els plànols d'un arquitecte per fer un gratacels no són simples esbossos. Cal un mecanisme formal per descriure el problema i la solució, de manera que tota persona implicada pugui plasmar les seves idees de manera que tothom l'entengui. El desenvolupament d'aplicacions no és una excepció.

Un dels avantatges més importants de la metodologia de l'orientació a objectes, i el principal motiu que l'ha fet rellevant, és que ha permès establir un llenguatge unificat per representar dissenys: el llenguatge UML (*unified modelling language*). D'aquesta manera, el desenvolupament del programari es pot equiparar a la resta de disciplines tècniques, en les quals és possible generar un esquema d'allò que es vol crear, que pot ser interpretat per qui s'encarregarà de fabricar-ho, sense necessitat que el constructor hagi format part del procés de disseny. A més a més, també és possible que altres experts externs avaluïn l'esquema abans de la fabricació, de manera que es puguin detectar errades abans de consumir cap recurs. Aquesta possibilitat és molt valuosa quan es tracta de crear sistemes complexos.

En el cas concret del desenvolupament del programari, quan parlem de constructor ens referim a un programador.

L'**UML** és un llenguatge estàndard que permet especificar amb notació gràfica programari orientat a objectes.

Els orígens de l'UML es remunten a l'any 1994, quan Grady Booch i Jim Rumbaugh van començar a unificar diferents tècniques de modelització orientada a objectes, en un intent de trobar una solució satisfactòria als problemes que els dissenyadors tenien a l'hora d'especificar el programari. Al llarg d'aquest procés, s'hi va unir Ivar Jacobson. Cadascú va aportar el propi mètode: el mètode Boosch, OMT (*object modelling technique*, 'tècnica de modelització d'objectes') i OOSE (*object oriented software engineering*, 'enginyeria de programari orientada a objectes').

L'UML no està limitat al programari, sinó que també permet especificar altres sistemes com, per exemple, models de negoci.

Els motius principals que els van dur a unir esforços van ser tres:

- Evitar que cada mètode evolucionés independentment, cosa que portaria a una situació amb molts mètodes heterogenis que seria perjudicial per als dissenyadors.

- Establir una notació única que aportés certa estabilitat al camp de l'orientació a objectes. Això era molt important per tal que el diferent programari de suport a la metodologia fos compatible entre si.
- Trobar millores mitjançant la col·laboració entre ells, partint de l'experiència individual en la creació de cada mètode.

L'OMG és una organització sense ànim de lucre amb l'objectiu d'impulsar les tecnologies basades en l'orientació a objectes.

L'any 1996 van fer la seva proposta d'UML 0.9, subjecta a l'escrutini i els comentaris de la comunitat, la qual es va convertir en objectiu estratègic de diverses organitzacions, com Digital, Hewlett-Packard, Oracle o Microsoft. Els esforços d'aquestes organitzacions van ser canalitzats per l'Object Management Grup (OMG, Grup de Gestió d'Objectes), que va donar lloc a l'UML 1.0. En els anys següents, l'UML ha seguit evolucionant i se n'han fet noves versions, que es continuen utilitzant dins el procés de disseny d'aplicacions.

1.1 Esquema general de resolució de problemes mitjançant orientació a objectes

A nivell general, els passos ordenats per resoldre un problema usant orientació a objectes es poden resumir en:

1. Plantejar l'escenari descriptivament, amb llenguatge humà. Com més detallada sigui la descripció, més fàcil serà la feina.
2. Localitzar, dins la descripció de l'escenari, els elements que es consideren més importants: els que realment interactuen amb vista a resoldre el problema. Aquests seran els objectes del programa. Normalment, solen ser substantius dins la descripció.
3. Considerar quins elements són d'una certa complexitat. Redefinir-los com a agrupacions d'objectes més simples. Una bona estratègia és partir del fet que tot l'escenari en si és un objecte complex (igual que una màquina també és un objecte complex) i anar-lo descomponent en parts més petites.
4. Agrupar els diferents objectes segons el tipus: quins objectes veiem que tenen propietats o comportaments idèntics. Cada tipus d'objecte serà una classe que caldrà especificar.
5. Identificar i enumerar les característiques dels objectes de cada classe: quines en són les propietats (els atributs) i el comportament (les operacions que ofereixen). N'hi ha prou amb una llista general, escrita en llenguatge humà però suficientment entenedora.
6. Establir les relacions que hi ha entre els objectes de les diferents classes a partir del paper que interpreten en el problema general. Els objectes no es generen en un buit, sinó que estan relacionats entre si, de la mateixa manera que les peces d'una màquina o els elements d'un edifici no floten en l'aire, sinó que estan connectats per formar un tot. De la mateixa manera,

un cop identificats els objectes que conformen el problema a resoldre (el programa que es vol fer, en aquest cas), cal identificar com es relacionen entre ells. Normalment, aquesta mena d'enllaços es pot identificar com "un objecte d'aquest tipus conté objectes d'aquesta altra classe" o "una instància d'aquesta classe gestiona instàncies d'aquesta altra classe". A mode d'ajut, es pot generar un **mapa d'objectes**.

7. Per a cada classe, especificar-ne formalment els atributs i les operacions, extrets a partir de la llista de propietats dels seus objectes dels punts 5 i 6. Normalment, especificar-ne els atributs és un procés més immediat que l'especificació de les operacions.

UML ofereix una notació formal per poder plasmar tots els elements resultants de cadascun dels passos: classes, atributs, mètodes, relacions entre objectes... El resultat final d'aquest procés seguint la notació UML és un esquema anomenat *diagrama estàtic UML.*

> Un **diagrama estàtic UML** mostra el conjunt de classes i objectes importants que formen part d'un sistema, juntament amb les relacions existents entre aquestes classes i objectes. Mostra d'una manera estàtica l'estructura d'informació del sistema i la visibilitat que té cadascuna de les classes, donada per les seves relacions amb les altres en el model.

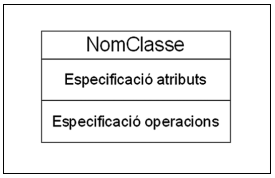
Mitjançant UML també és possible definir formalment molts altres aspectes del comportament de l'aplicació (comportament dinàmic, relacions entre crides a mètodes...), però aquí ens centrarem en el diagrama estàtic, que és la base de tot.

1.2 Definició de classes mitjançant UML

La peça fonamental d'un diagrama estàtic són les classes, ja que, de fet, el codi d'un programa orientat a objectes es compon de classes, on es defineixen els atributs i els mètodes de què disposaran les seves instàncies (els objectes) quan el programa s'executi. Per tant, abans de poder crear un diagrama complex, el primer pas és identificar-les i saber com representar-les en UML.

En UML, una classe es representa en format complet mitjançant una caixa dividida horitzontalment en tres parts. La part superior compleix exactament la mateixa funció i té el mateix format que en el format simplificat, i és on consta el nom de la classe. En la part del mig es defineixen els atributs que tindran les seves instàncies. Finalment, en la part inferior, es defineixen les operacions que es poden cridar sobre qualsevol de les seves instàncies. L'aspecte és el que es mostra en l'exemple de la figura 1.1.

FIGURA 1.1. Notació formal d'una classe en UML.



1.2.1 Definició d'atributs

En definir una classe, els atributs s'especifiquen segons la sintaxi següent. El camp de valor inicial es correspon al valor que pren l'atribut en el moment d'instanciar un objecte d'aquesta classe. Concretar-lo en l'especificació dels atributs és opcional. Com veieu, el format és semblant a la declaració d'una variable qualsevol en Java.

```
1 visibilitat nomAtribut: tipus [= valor inicial]
```

Nomenclatura

Per a atributs s'usen paraules concatenades, en què la primera inicial està amb minúscula i la resta amb majúscula. Per exemple: `elMeuAtribut`.

Els dos aspectes més importants de la definició d'un atribut en representar una classe en UML són la seva visibilitat i el seu tipus de dades.

Pel que fa a la seva visibilitat, aquesta indica si l'atribut és accessible directament des d'altres classes. Hi ha diferents tipus de visibilitat, si bé es destacaran els dos més utilitzats: la visibilitat pública i la privada. Tot i que és possible triar entre diverses, normalment, els atributs es defineixen amb visibilitat privada.

L'UML no indica explícitament quin és el significat real de cada tipus de visibilitat, i deixa aquesta tasca a cada llenguatge de programació. El motiu és que aquest terme es refereix a l'accessibilitat a un objecte en l'àmbit del codi. Tot i així, es descriurà quina sol ser la seva interpretació en la majoria de llenguatges de programació orientats a objectes. Cada tipus de visibilitat s'identifica en la definició de l'atribut amb un símbol especial:

- Un **atribut públic** s'identifica amb el símbol "+". En aquest cas, si una instància `a:` està enllaçada amb una instància `b:`, `a:` pot accedir lliurement als valors emmagatzemats en els atributs de `b:`.
- Un **atribut privat** s'identifica amb el símbol "-". No es pot accedir a aquest atribut des d'altres objectes, independentment del fet que existeixi un enllaç o no. A efectes pràctics, és com si no existís fora de l'especificació de la classe i, en conseqüència, només es pot utilitzar en les operacions dins la mateixa classe en què s'ha definit.

Quant al tipus de dades, donat que UML és un mecanisme de notació genèric, no vinculat a cap llenguatge específic de programació, els tipus de dades que es poden usar no són exactament els mateixos que en Java, si es vol ser estricte. Se sol usar el llenguatge natural, en anglès, per indicar el tipus, enlloc d'una paraula clau concreta depenent del llenguatge. Tot seguit s'enumeren els més típics si es fa en català:

TAULA 1.1. Tipus bàsics dels atributs.

Tipus	Significat	Exemple
Enter	Un nombre sense decimals	1, 56, 128, 15487
Real	Un nombre amb decimals	1,34, 3,2415, 267,14, 41,0
Caràcter	Una lletra	A, a, b, g, -, ?, ç
Booleà	Cert/fals	Cert, fals
Byte	Un byte	0x30, 0xA2, 0xFF
Matriu de... (...[ ])	Un conjunt d'elements... [1, 2, 3], [a, b, c, f, g], [1,2, 3,0]	

En el darrer cas, els tipus múltiples es poden especificar de dues maneres diferents, segons la interpretació que es vol expressar:

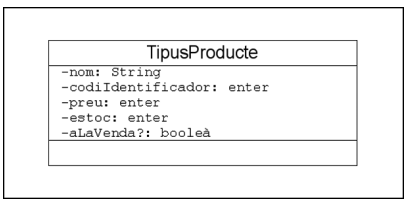
- enter[5] indica que hi ha exactament cinc enters.
- enter[0..5] indica que hi pot haver entre zero i cinc enters.

De totes maneres, també es pot considerar que ja hi ha predefinits un conjunt de tipus de propòsit general, que pràcticament tots els llenguatges suporten d'una manera o d'una altra:

- `String`, per especificar tipus de dades que corresponen a cadenes de caràcters, així s'evita haver d'operar amb caràcters.
- `List<nomTipus>`, per especificar seqüències d'elements de tipus "nomTipus", sense cap fita predeterminada. En el camp "nomTipus" s'indica el tipus d'element que conté la llista, per exemple:List<enter>, List<Cita>, List<String> ...

Així, doncs, els atributs d'una classe anomenada `TipusProducte` es poden definir tal com mostra la figura 1.2.

FIGURA 1.2. Definició d'atributs de la classe TipusProducte



Els atributs de classe són especialment útils per definir constants.

A part dels atributs que defineixen les propietats de cada instància d'una classe, hi ha un tipus especial d'atributs, anomenats **atributs de classe** o estàtics. A l'hora de definir-los, es diferencien amb un subratllat i, si són constants, normalment estan escrits amb el nom en majúscules. A part d'això, la sintaxi és idèntica als atributs genèrics.

```
1 visibilitat nomAtributClasse: tipus [= valor inicial]
```

Per exemple:

```
+PI: real
```

1.2.2 Definició de mètodes

La convenció de nomenclatura d'una operació és idèntica a la dels atributs.

Dins la definició d'una classe, les operacions disponibles s'especifiquen en UML de la manera següent:

```
visibilitat nomOperació (llistaParàmetres): tipusRetorn
```

El camp "llistaParàmetres" té el format següent:

```
nomParàmetre1: tipus, ... , nomParàmetreN: tipus
```

Recordeu que el "+" abans del nom indica que les operacions tenen visibilitat pública.

Totes les explicacions donades per als tipus o la visibilitat en el cas dels atributs també són aplicables a les operacions. En aquest cas, normalment les operacions solen ser públiques, i es deixen com a privades operacions que es consideren auxiliars, o que simplement faciliten la comprensió de les tasques que duran a terme les instàncies de les classes, però que no es poden invocar des de cap altra part del futur codi.

Alguns exemples d'especificacions d'operacions poden ser:

```
+afegirMèdia (m: Mèdia)
+ajustarVolum (v: enter)
+pausa/reanuda ()
+mèdiaSegüent(): Mèdia
```

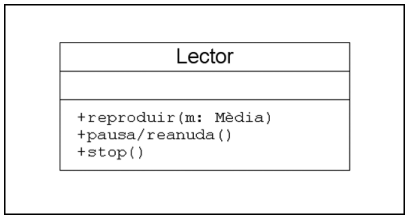
Addicionalment, hi ha un conjunt d'operacions que no sempre cal especificar, ja que se suposen en dissenyar una classe: les operacions accessores. S'acostumen a considerar operacions o mètodes accessors els que donen accés de lectura o d'escriptura als atributs d'una classe. La nomenclatura estàndard per a l'accessor d'escriptura (per modificar el valor de l'atribut) i de lectura (per consultar-lo) és, respectivament:

- setNomAtribut (valor: tipus).
- getNomAtribut(): tipus.

Per tant, en l'especificació d'una classe no cal explicitar tots els accessors entre les seves operacions. Per a cada atribut especificat ja es dóna per entès que sempre hi ha les operacions *set* i *get* associades, a menys que es digui el contrari.

La figura 1.3 mostra una representació UML de les operacions d'una classe anomenada `Lector`.

FIGURA 1.3. Especificació d'operacions de la classe Lector.



1.3 Relacions entre classes

Un cop identificades les classes dels objectes que componen el problema a resoldre, el pas següent és establir quines relacions hi ha entre elles i representar-les dins el diagrama estàtic UML. De fet, les relacions són el punt especialment rellevant d'un diagrama estàtic UML, ja que, en cas contrari, les classes queden representades en un buit, disperses. Només es disposa d'una llista de classes i prou, sense cap idea de com col·laboren entre elles o quin sentit tenen dins l'aplicació futura. En el diagrama estàtic, cada relació indica que hi ha una connexió entre els objectes d'una classe i els d'una altra. Fent un símil amb una màquina, l'existència d'una relació és equivalent al fet que dues de les seves peces estiguin connectades entre elles.

1.3.1 Associacions

El tipus de relació més freqüent és l'associació.

> Es considera que hi ha una **associació** entre dues classes quan es vol indicar que els objectes d'una classe poden cridar operacions sobre els objectes d'una altra.

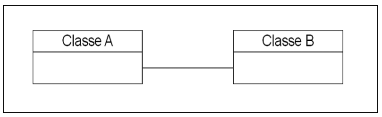
Per tant, perquè dos objectes puguin interactuar hi ha d'haver una associació entre les seves classes al diagrama estàtic UML, de manera que es considera que estan **enllaçats**. En cas contrari, la crida d'operacions no és possible. Quan per mitjà d'un enllaç un objecte **objecteA:** crida una operació sobre un objecte **objecteB:**, les transformacions fetes per l'operació únicament afectaran l'objecte **objecteB:**. No n'afectaran cap de la resta d'instàncies que hi hagi en aquell moment que pertanyin a la mateixa classe que l'**objecteB:**.

Eines CASE

Normalment, els diagrames estàtics UML es generen mitjançant l'ajut d'eines CASE (*computer aided software engineering*, ,enginyeria de programari assistida per ordinador').

El diagrama estàtic UML estarà format en la seva totalitat per la representació gràfica de totes les classes identificades i les relacions que hi ha entre totes elles. En la figura 1.4 es representen les relacions amb una línia que uneix les classes implicades.

FIGURA 1.4. Associació entre classes.



Tot i que les associacions indiquen que hi ha enllaços entre objectes, es representen gràficament en el diagrama respecte a les seves classes. Quan dues classes es representen relacionades, les seves instàncies poden estar enllaçades en algun moment de l'execució de l'aplicació.

Els enllaços són dinàmics. Poden variar al llarg de l'execució de l'aplicació.

Donada una associació, hi ha un conjunt de descriptors addicionals que es poden especificar:

- En el centre, entre les dues classes implicades, s'especifica el **nom de l'associació**. Aquest nom indica què representa l'associació a nivell conceptual.
- En cada extrem de l'associació s'especifica quines són les funcions de les classes implicades. El nom indica, de manera entenedora, el paper que tenen els objectes de cada classe en la relació.
- Amb una fletxa se n'especifica la **navegabilitat**. Partint del nom de l'associació i les funcions de les classes, s'ha de poder establir quina és la classe origen i quina la destinació. A efectes pràctics, la navegabilitat indica el sentit de les interaccions entre objectes: a quina classe pertanyen els objectes que poden cridar operacions i a quina els objectes que reben aquestes crides.
- Juntament amb la funció, s'especifica la **cardinalitat**, o multiplicitat, de la relació per a cada extrem. La cardinalitat especifica amb quantes instàncies d'una de les classes pot estar enllaçada una instància de l'altra classe.

Si cal, res no impedeix que una associació sigui **navegable** en ambdós sentits, tot i que no és el cas més freqüent. En aquest cas, no cal posar cap fletxa.

La **cardinalitat** defineix quantes instàncies diferents d'una classe ClasseA es poden associar amb una instància de la classe ClasseB en un moment determinat de l'execució del programa.

Les cardinalitats més usades solen ser les 1, 1..* i *.

El nombre d'instàncies per a cada cas s'indica amb una llista de nombres enters, ubicada en l'extrem oposat de l'associació. Per exemple, la cardinalitat que indica quantes instàncies de la ClasseB pot enllaçar una instància de la ClasseA s'ubica en l'extrem de l'associació en què està representada la classe ClasseB. En cas que es vulgui indicar un nombre indeterminat, sense fita superior en el nombre d'enllaços, s'utilitza el símbol *. Per establir rangs de valors possibles, s'usen les fites inferior i superior separades amb tres punts.

Diferents cardinalitats i el seu significat

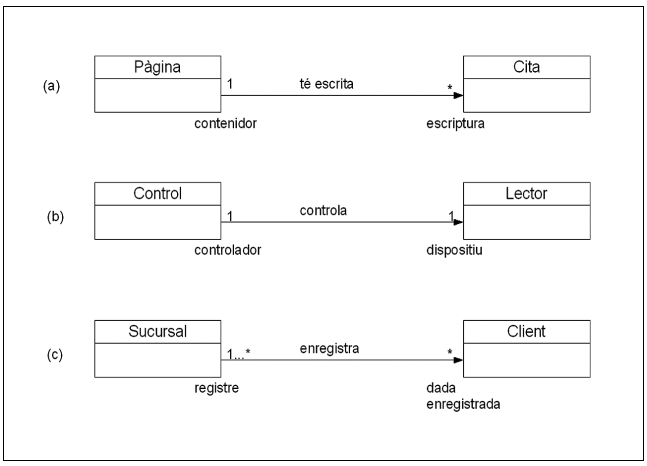
- 1: Només un enllaç.
- 0...1: Cap o un enllaç.
- 2,4: Dos o quatre enllaços.
- 1...5: Entre un i cinc enllaços (un, dos, tres, quatre o cinc).
- 1...*: Entre un i un nombre indeterminat. És a dir, més d'un.
- *: Un nombre indeterminat. És equivalent a 0...*.

Perquè el programa es consideri correcte a l'hora d'implementar el disseny, s'ha de fer el necessari perquè les condicions que expressen les cardinalitats sempre siguin certes. Si una cardinalitat és, per exemple, 1, això vol dir que tot objecte d'aquesta classe sempre està enllaçat amb un, i només un, objecte de l'altra classe. Mai no hi pot haver dins el programa en execució un cas en què no es compleixi aquesta condició.

De tots aquests descriptors, només la navegabilitat i la cardinalitat són imprescindibles (i de les dues, la cardinalitat és la més important), ja que la decisió que es prengui en aquests aspectes dins l'etapa de disseny tindrà implicacions directes sobre la implementació. Les funcions i el nom són opcionals, si bé molt útils amb vista a la comprensió del diagrama estàtic i com a mètode de reflexió per al dissenyador.

En la figura 1.5 es mostra una representació completa d'una associació, amb tota la informació que cal especificar.

FIGURA 1.5. Representació gràfica d'una associació entre classes.



A partir del que expressa la figura, aquestes són algunes de les conclusions a les quals arribaria un observador aliè al procés de disseny:

- **A partir d'(a)**. Una instància qualsevol de la classe `Pàgina` pot enllaçar fins a un nombre indeterminat d'objectes diferents de la classe `Cita`. Això inclou no tenir-ne cap (una pàgina en blanc). Així, doncs, hi ha pàgines que tenen tres cites, d'altres deu, d'altres cap, etc.

- **A partir d'(a)**. Recíprocament, donat un objecte qualsevol de la classe `Cita`, només estarà enllaçat a un únic objecte `Pàgina`. Per tant, no es pot tenir una mateixa cita en dues pàgines diferents (però sí tenir dues citacions diferents i de contingut idèntic, amb els mateixos valors per als atributs, cadascuna a una pàgina diferent). Tampoc no hi pot haver citacions que, tot i ser en l'aplicació, no estiguin escrites a cap pàgina.

- **A partir de (b)**. Un objecte de la classe Control sempre té un objecte de la classe Lector enllaçat. Per tant, un tauler de control sempre controla un únic lector de mèdia. No es pot donar el cas que un tauler de control no controli cap lector. La inversa també es certa; tot lector és controlat per algun tauler de control.

- **A partir de (b)**. Donada la navegabilitat especificada, s'està dient que el tauler de control pot cridar operacions sobre el lector, però no a l'inrevés. Això té sentit, ja que és el tauler de control qui controla el lector.

- **A partir de (c)**. Donat un client, aquest pot estar enregistrat en més d'una sucursal, però almenys sempre ho estarà en una. Mai no es pot donar el cas d'un client donat d'alta en el sistema però que no estigui enregistrat en cap sucursal.

Cada objecte és únic i té la seva pròpia identitat, independentment dels valors dels seus atributs.

Tot i que no és habitual, res no impedeix que dues classes tinguin més d'una associació entre elles. Normalment, aquest cas s'aplica quan es vol fer una diferenciació explícita de diferents tipus de relació o de funcions entre instàncies de cada classe. En la figura 1.6 es mostra un exemple en què es pot aplicar aquest principi.

FIGURA 1.6. Múltiples associacions entre dues classes.

Atès que un transportista pot adoptar diferents papers en la seva relació amb una sucursal, això es pot representar especificant que hi ha diferents tipus d'associació entre sucursals i transportistes.

Tot i la versatilitat dels diagrames estàtics UML, aquests no sempre són capaços de reflectir totes les restriccions necessàries en els enllaços entre objectes de diferents classes. Les associacions indiquen els possibles enllaços, però no especifiquen condicions concretes per a la seva presència. En aquests casos s'utilitza una **restricció textual**. Aquesta no és més que una frase addicional al peu del diagrama, escrita en llenguatge humà, en què s'explica breument en què consisteix aquesta restricció.

L'*object constraint language* és un mecanisme formal per expressar restriccions textuals sense usar el llenguatge humà.

Els transportistes

Amb les associacions representades en la figura 1.6 és possible expressar que sempre hi ha un únic transportista de reserva i diversos de servei. Malauradament, és impossible

indicar que un transportista concret, per exemple el transportista2:, no pot ser alhora de reserva i de servei.

Per resoldre aquest problema cal afegir una restricció textual al peu del diagrama que expliqui la condició en la qual es pot complir l'associació "de reserva": un transportista de reserva no pot estar de servei i viceversa.

1.3.2 Agregacions

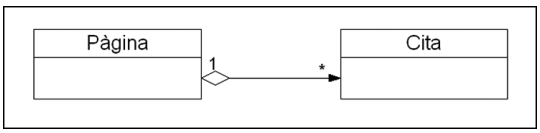
Hi ha un tipus d'associació especial mitjançant la qual el dissenyador vol donar un sentit més específic a l'enllaç, com ara que els objectes de certa classe formen part dels objectes d'una altra. Aquest subtipus d'associació s'anomena *agregació*.

Una **agregació** és un tipus d'associació especial que especifica explícitament que hi ha una relació de tot-part entre els objectes de l'agregat (el tot) i el component (la part).

En el diagrama estàtic UML, això es representa gràficament afegint un rombe blanc en l'extrem de l'associació on hi ha la classe que representa el tot. Com que amb aquest símbol ja es diu quina és la relació entre els objectes d'ambdues classes, es poden ometre el nom i la funció en els descriptors de l'associació.

Això es mostra en la figura 1.7, en la qual es refina l'associació PàginaCita que s'ha mostrat en els exemples anteriors. Una pàgina conté citacions escrites al seu interior i es pot considerar que el que s'ha escrit en una pàgina és part d'aquesta. Per tant, aquest cas és aplicable.

FIGURA 1.7. Agregació Pàgina-Cita



De totes maneres, val la pena esmentar que la definició d'aquest tipus d'associació no és gaire estricta, de manera que en la literatura hi ha diverses interpretacions de què vol dir realment *conté* o és *part de*. En darrera instància, és una interpretació personal del dissenyador considerar si un tipus d'objecte és realment part d'un altre.

1.3.3 Composicions

Un altre subtipus d'associació amb una semàntica especial són les anomenades *composicions*. Amb aquestes associacions també s'expressa el concepte de *és part de*, però anant més enllà: es considera que la classe composta no té sentit

Qualsevol associació en què el dissenyador posaria un verb de l'estil *té*, *conté*, *és part de*, etc. és una agregació o una composició.

sense els seus components. En contraposició, en una agregació, l'agregat sí que té sentit sense cap dels seus components.

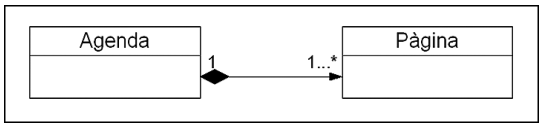
> Una **composició** és una forma d'agregació que requereix que els objectes components només pertanyin a un únic objecte agregat i que, a més a més, aquest darrer no té sentit que existeixi si no n'existeixen els components.

En l'agregació , segons la mateixa definició, la cardinalitat en la banda de l'agregat sempre ha de ser 1.

Cal anar amb compte, ja que de vegades és fàcil confondre agregació i composició.

Aquest tipus d'associació es representa de manera idèntica a una agregació, només que en aquest cas el rombe és de color negre. La figura 1.8 en mostra un exemple amb l'associació Agenda-Pàgina.

FIGURA 1.8. Agregació Agenda-Pàgina Agenda-Pàgina.



Un altre exemple immediat de composició són les associacions que sortirien de en el reproductor multimèdia.

En aquest cas, en usar una composició per representar aquesta associació, el dissenyador expressa que no té sentit una agenda sense pàgines. De fet, conceptualment, el propi conjunt de pàgines és l'agenda en si. Tampoc no pot ser que una mateixa pàgina (es parla de la mateixa pàgina física, no d'una rèplica) sigui de més d'una agenda alhora. En canvi, sí que té sentit una pàgina en blanc sense cap cita, motiu pel qual el cas Pàgina-Cita és una agregació però no una composició.

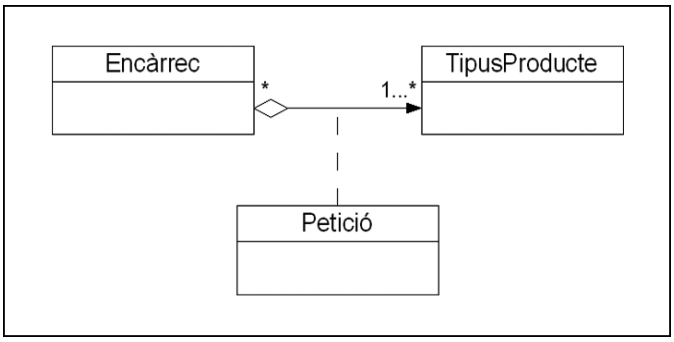
1.3.4 Classes associatives

Hi ha circumstàncies que fan que el dissenyador consideri necessari afegir propietats addicionals a una associació, el valor de les quals pot variar segons quines siguin les instàncies enllaçades. En definitiva, el dissenyador vol especificar atributs en una associació. Amb aquesta finalitat s'usen les *classes associatives*.

> Les **classes associatives** representen associacions que es poden considerar classes.

Una classe associativa es representa amb el mateix format que una classe: és una nova classe que cal especificar dins la descomposició del problema. Com es mostra en la figura 1.9, una línia discontinua uneix la nova classe amb l'associació que representa.

FIGURA 1.9. Exemple de classe associativa



En la classe associativa Petició hi hauria les propietats vinculades a la petició d'un tipus de producte concret dins un encàrrec (per exemple, el nombre de productes que cal enviar o el seu color). Cap d'aquestes propietats no correspon a l'encàrrec ni al tipus de producte. Tot això es plasmarà en forma d'atributs dins d'aquesta classe.

Totes les classes associatives que apareguin en el diagrama estàtic UML s'hauran d'especificar completament, tant pel que fa als atributs com a les operacions. Tot i representar una associació i no un element identificable dins del problema, a efectes pràctics són una classe més, com qualsevol altra.

Una particularitat de les classes associatives és que es poden representar amb classes i associacions normals. Aquest fet és important, ja que és l'única manera de representar-les en un mapa d'objectes i la majoria de llenguatges de programació no suporten les associacions amb aquest tipus de classes vinculades.

Java no suporta directament classes associatives.

Seguint el sentit de la navegabilitat (classe origen - classe destinació) el desenvolupament és el següent:

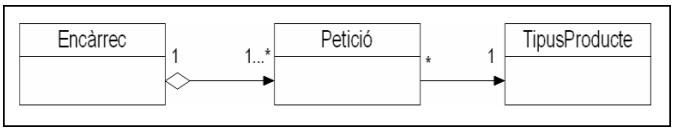
1. Eliminar l'associació original.
2. Generar una associació de la classe origen a la classe associativa. El tipus de la nova associació i la navegabilitat són idèntics a l'original.
3. Generar una altra associació de la classe associativa a la destinació. La navegabilitat és de classe associativa a destinació.

Ara la classe associativa ja és una classe normal dins el diagrama estàtic UML, però atès que ara hi ha dues associacions, cal adaptar-hi les cardinalitats:

- La cardinalitat en els extrems oposats a l'antiga classe associativa sempre és 1.
- La cardinalitat en l'extrem oposat de la classe origen, en què ara hi ha la classe associativa, és la que hi havia respecte a l'antiga classe destinació.
- La cardinalitat en l'extrem oposat de la classe destinació, en què ara hi ha la classe associativa, és la que hi havia respecte a l'antiga classe origen.

El resultat d'aplicar aquesta traducció es mostra en la figura 1.10.

FIGURA 1.10. Traducció de classe associativa.



De fet, depenent de les interpretacions que ha fet el dissenyador respecte a la descripció del problema, es pot arribar a aquest diagrama directament des del pas d'identificació de classes. De totes maneres, en fer un diagrama estàtic UML, és recomanable usar classes associatives sempre que apliqui i només fer la traducció en el moment d'implementar.

1.3.5 Associacions reflexives

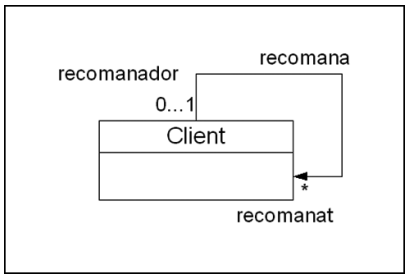
Atès que una associació indica enllaços entre instàncies d'una classe, res no impedeix que un objecte estigui enllaçat amb objectes del mateix tipus. Quan això passa, es representa mitjançant una associació reflexiva.

Per la seva definició, no es pot aplicar una associació reflexiva a una composició.

Una **associació reflexiva** és aquella en què la classe origen i la destinació són la mateixa.

Un exemple d'aquest cas es mostra en la figura 1.11, en què els clients de l'aplicació de gestió recomanen altres clients. Es tracta d'una associació reflexiva, ja que tant qui recomana com qui és recomanat, un client, pertanyen a la mateixa classe.

FIGURA 1.11. Associació reflexiva



Un dels moments en què el dissenyador ha d'anar amb més compte en aquest tipus d'associacions és quan n'especifica la cardinalitat. La cardinalitat a l'origen de la relació sempre ha de preveure el cas 0. En cas contrari, és impossible generar un mapa d'objectes que la satisfaci, ja que s'hi genera un bucle infinit. Això s'ha de tenir en compte, independentment del fet que, conceptualment, afegir-hi cardinalitat 0 tingui sentit o no.

Exemple de bucle infinit: un arbre genealògic

Un exemple d'aquesta problema és l'ús d'una associació reflexiva entre objectes d'una classe `Persona` per crear un arbre genealògic. Una persona sempre ha nascut a partir de dos pares (pare i mare), que a la vegada també són persones. Per tant, és lògic, i correcte

des del punt de vista conceptual, que la cardinalitat en origen sigui 2, i en destinació, * (una persona pot tenir un nombre indeterminat de fills).

Malauradament, si es fa així, és impossible generar un mapa d'objectes que compleixi aquesta cardinalitat. Per a cada objecte :Persona hi ha d'haver dos enllaços provinents d'altres objectes:Persona, i això és impossible tret que hi hagi cicles (cosa que no pot ser en un arbre genealògic). L'única manera d'evitar-ho és establir que la cardinalitat pot ser 0 o 2.

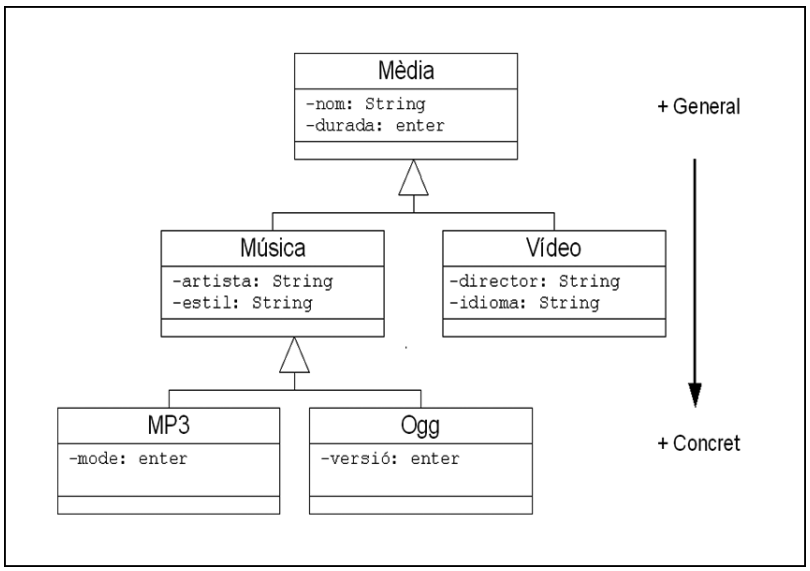
El cas 0 serà una excepció per als objectes :Persona inicials dins el sistema (per als quals no s'enregistrarà quins van ser els pares).

1.3.6 Herència

Finalment, dins un diagrama estàtic UML també s'estableixen les relacions d'herència entre els objectes de diferents classes. Això permet definir relacions d'especialització/generalització, on la classe a la part superior de la relació representa un concepte més general que el de la part inferior, més específic. Aquest tipus de relació normalment també indica que la classe més específica és idèntica a una altra, excepte en alguns petits aspectes on, o bé és diferent, o bé se suposen propietats addicionals.

Per exemple, diferents tipus de fitxers de mèdia es poden vincular entre si d'acord a una relació d'herència tal com mostra la figura 1.12. A la dreta es veu quin concepte es considera més general i quin més específic.

FIGURA 1.12. Herència dins un diagrama estàtic UML



1.4 Exemples de diagrames estàtics

Una vegada s'ha efectuat el procés de descomposició de programes es pot fer un possible diagrama estàtic UML. En cada cas, el diagrama segueix cadascuna de

les composicions proposades del problema. Per fer més entenedora la lectura dels diagrames, només s'inclou el nom de les relacions menys evidents.

Per a cada cas, també es representa un possible mapa d'objectes per a un moment determinat de l'execució de l'aplicació. És interessant veure com cada associació al diagrama estàtic es tradueix en un cert nombre d'enllaços entre objectes, sempre respectant la cardinalitat especificada en l'associació.

1.4.1 Una agenda

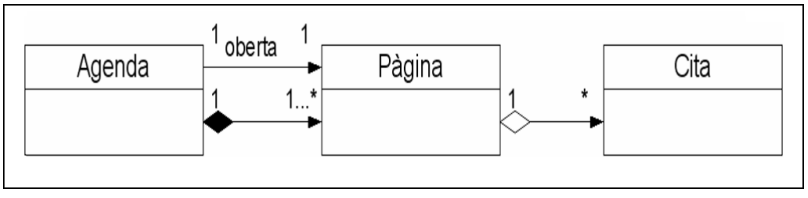
El primer exemple és molt senzill, amb molts pocs elements i funcionalitats i amb un paral·lelisme clar amb el món real per facilitar-ne la comprensió. Es vol dissenyar una agenda que permeti consultar les dates d'un calendari per a un any concret i apuntar cites a unes hores concretes. En aquest cas, és possible fer un cert paral·lelisme amb el món real, ja que el concepte d'agenda hi existeix. Es pot pensar en l'agenda com un llibre en què es van passant pàgines endavant o endarrere, cadascuna de les quals correspon a un dia. En cada pàgina es poden escriure cites establertes per a unes hores d'inici i de finalització determinades. Aquesta descripció en llenguatge humà seria la que s'ha descrit en el pas 1 de l'esquema d'aplicació de l'orientació a objectes.

Una bona manera de descompondre un problema en objectes és partir de l'element més general i, a partir d'aquí, anar extraient els elements més senzills. Així, doncs, en aquest cas es pot partir d'un objecte agenda, i deduir els elements que el componen. Una agenda es compon de les pàgines, les quals contenen cites.

Per saber exactament el tipus de relacions entre classes, cal establir la relació de dependència entre elles. Per això, també va bé fer-se una idea de si una situació té sentit o no si estiguéssiu tractant amb un objecte del món real. Així doncs, una agenda està formada per pàgines, i no té sentit una agenda sense pàgines, pel que s'està parlant d'una composició. D'altra banda, cal saber la pàgina oberta en tot moment, per la qual cosa cal una altra relació. Aquesta pot ser una simple associació, ja que no compleix cap de les característiques dels altres tipus de relacions. Finalment, les pàgines contenen cites, però sí que té sentit que una pàgina, en un moment donat, encara no tingui cap cita, per la qual cosa es tracta d'una agregació.

El diagrama estàtic UML de l'agenda es mostra en la figura 1.13.

FIGURA 1.13. Diagrama estàtic UML de l'agenda.



La instància d'Agenda pot tenir dos tipus de relacions amb les pàgines. D'una banda, la de composició: entre totes les pàgines formen l'agenda. A més a més, una agenda també sap per quina pàgina està oberta, que seria la pàgina que es pot llegir en aquest moment.

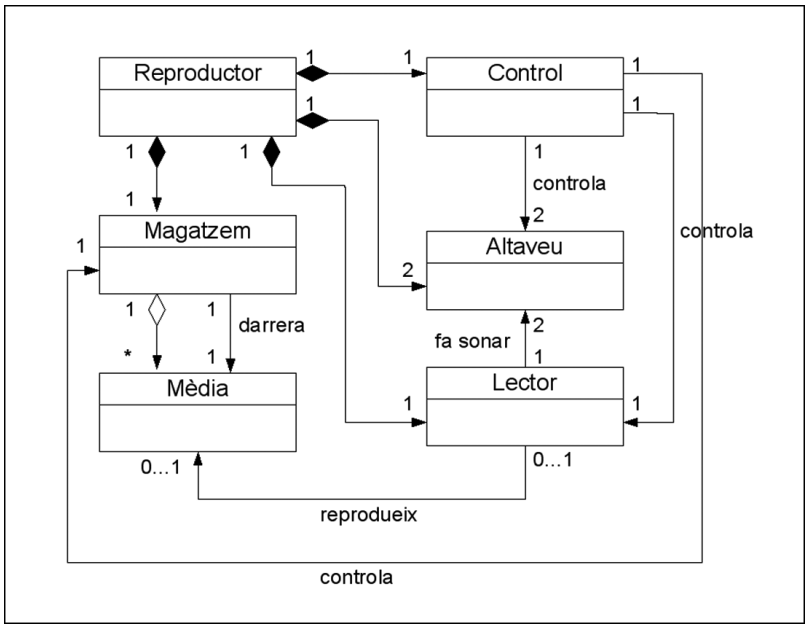
1.4.2 Un reproductor multimèdia

En aquest exemple es presenta la descomposició d'un sistema de reproducció multimèdia (música, vídeos...). La motivació pot ser crear una aplicació senzilla per a l'ordinador o generar el sistema de control d'un reproductor portàtil (un dispositiu físic). En aquest cas, es parteix de la identificació de classes següent: `Reproductor`, `Mèdia`, `Control`, `Lector`, `Magatzem` i `Altaveu`.

El diagrama estàtic UML del reproductor es pot veure en la figura 1.14. En fer-lo, s'ha decidit que el reproductor té dos altaveus, de manera que es pot controlar el mode mono o estèreo. També cal tenir en compte els punts següents:

- La cardinalitat de la relació `Mèdia-Lector` especificada en l'extrem de la classe `Mèdia` (a l'esquerra) indica que hi pot haver moments en què el lector no reprodueix cap cançó (cas 0 de la cardinalitat: el lector reprodueix 0 cançons).
- La cardinalitat en l'altre extrem expressa que, per a qualsevol cançó, o bé s'està reproduint en el lector (cas 1) o no (cas 0).
- El magatzem pot ser buit (cas 0 de la cardinalitat "*" en la relació `Magatzem - Mèdia`).
- S'ha decidit que la relació entre `Magatzem` i `Mèdia` és una agregació, però també es pot interpretar com una associació normal si no es considera que les dades emmagatzemades, les cançons, són part del magatzem i tenen entitat pròpia.
- La relació "darrera" entre `Magatzem` i `Mèdia` expressa quina és la darrera cançó a què s'ha accedit en el magatzem, per fer-ne accés seqüencial.

FIGURA 1.14. Diagrama estàtic UML del reproductor.

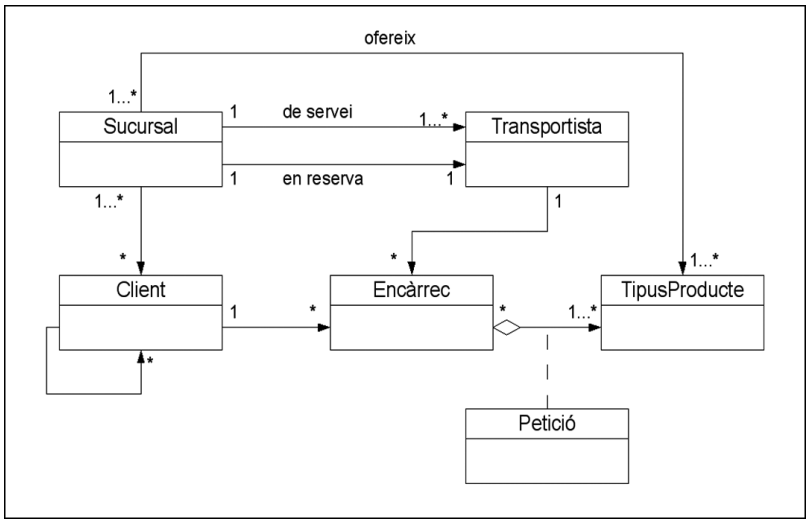


1.4.3 Una aplicació de gestió

Suposeu que una empresa vol crear una aplicació que gestioni el transport d'encàrrecs d'una sucursal d'una franquícia. Cada sucursal té un conjunt de transportistes assignats, el nombre dels quals pot variar segons la grandària de la sucursal. Cada dia hi ha un transportista que no treballa, però es considera que està en reserva. Cada un disposa del seu propi vehicle, identificat per un número de llicència. Quan un client vol fer un encàrrec, se n'enregistren les dades personals i especifica les condicions de lliurament: dia i hora, adreça... En l'encàrrec fa constar la llista de productes que vol que li serveixin. Tan bon punt es genera un encàrrec, automàticament, ja s'assigna algun transportista perquè el serveixi. Mai no hi ha encàrrecs sense transportista assignat. Els clients també tenen l'opció de recomanar amics seus perquè s'hi apuntin com a clients. Aquest fet es té en compte de cara a algunes promocions o descomptes especials.

En aquest cas, es partirà de la descomposició en les següents classes: `Encàrrec`, `Sucursal`, `Transportista`, `Client`, `TipusProducte`. El diagrama estàtic UML de l'aplicació de gestió és el següent, representat en la figura 1.15.

FIGURA 1.15. Diagrama estàtic UML de l'aplicació de gestió



1.5 Els diagrama estàtic i els mapes d'objectes

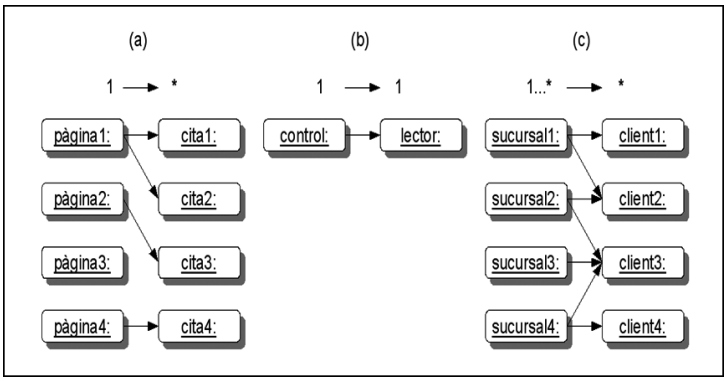
Una eina útil per reflexionar sobre si una cardinalitat representa allò que el dissenyador vol és crear mapes d'objectes. Es tracta d'esquemes que representen els diferents estats possibles de l'aplicació.

> En un **mapa d'objectes** es mostren tots els objectes instanciats i els enllaços que hi ha entre ells en un moment determinat de l'execució, l'aplicació d'acord amb el que s'ha representat en el diagrama estàtic UML.

Els mapes d'objectes només són una eina de suport, i no s'utilitzen com a mecanisme formal per representar el disseny. En el diagrama estàtic UML ja hi ha tota la informació necessària.

La figura 1.16 representa un seguit de mapes d'objectes, un per cada cas, amb diferents objectes enllaçats correctament segons la cardinalitat especificada en l'associació. Els enllaços es representen amb fletxes segons la navegabilitat de les associacions. El cas (a) correspon a l'associació PàginaCita, el cas (b) a la Control-Lector i el (c) a la Sucursal-Client. Damunt de cada mapa representat hi ha un recordatori de la cardinalitat de l'associació.

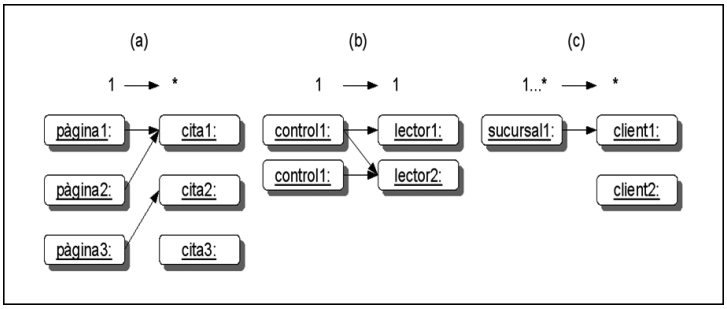
FIGURA 1.16. Exemple de mapes d'objectes



La figura 1.17 mostra alguns casos d'estats que es considerarien incorrectes segons les cardinalitats especificades en les associacions:

- **El cas (a)** és incorrecte, ja que una cita sempre ha d'estar escrita en alguna pàgina (cardinalitat esquerra = 1), i la instància cita3: no ho compleix, en no estar enllaçada a cap objecte :Pàgina. Només seria correcte si la cardinalitat fos, per exemple, 0..1.
- **El cas (b)** és incorrecte, ja que un tauler de control només controla un únic lector enllaçat (cardinalitat dreta = 1), i la instància control1: no ho compleix, en estar enllaçada a dues instàncies, lector1: ilector2:. A més a més, un lector només pot estar enllaçat a un únic tauler de control (cardinalitat esquerra = 1), i la instància lector2: tampoc no ho compleix.
- **El cas (c)** és incorrecte, ja que tot client ha d'estar enregistrat en alguna sucursal, i la instància client2: no està enllaçada amb cap.

FIGURA 1.17. Exemples d'enllaços incorrectes



2. Aplicacions amb BD no orientades a objectes

La persistència mitjançant fitxers és més que suficient si el tractament que es vol fer de les dades és totalment seqüencial: l'aplicació llegeix el fitxer en la seva totalitat i a partir del seu contingut genera el conjunt d'objectes que necessita per dur a terme la seva tasca. Per exemple, un editor estàndard de textos o gràfic. Però en cas que es vulgui fer un accés aleatori a diferents parts del fitxer, o quan el nombre de dades és realment molt gran, tant que carregar tot el fitxer superaria la memòria de l'ordinador, la utilitat d'aquest sistema cau en picat.

Una altra restricció molt important és que usar fitxers deixa de funcionar, o si més no, la seva gestió es fa massa complicada perquè realment valgui la pena, quan diverses aplicacions volen accedir concurrentment a les dades emmagatzemades. És per aquest motiu que la majoria d'aplicacions que necessiten gestionar una gran quantitat d'informació o poden existir accés concurrent utilitzen una base de dades per aconseguir la persistència.

Les files d'una BD relacional també s'anomenen tuples.

Una **base de dades (BD)** és un mecanisme per emmagatzemar informació de manera que sigui fàcil i eficient de recuperar. En la seva accepció més simple, la de base de dades relacional, aquesta pren la forma d'un seguit de taules formades per files i columnes.

Quan parlen de BD, ens referim sempre a una BD relacional.

Al contrari del que passava amb la persistència mitjançant fitxers o la seriació d'objectes, quan s'utilitza persistència mitjançant una BD, no es tracta de recuperar immediatament tots els objectes emmagatzemats i instanciar-los a memòria. Justament, una BD s'utilitza especialment quan hi ha massa objectes o diferents equips han d'accedir a les mateixes dades, pel que fer-ho no té sentit, ja que no soluciona el problema. La part del Model que es vol que sigui persistent sempre és en la BD, i quan s'instanciï un objecte, normalment serà per encapsular un conjunt d'informació recuperada de la BD per poder operar amb les seves dades de manera temporal.

BD

A part de les BD relacionals, també hi ha les orientades a objectes i les jeràrquiques, que estructuren les dades de manera diferent, en lloc de només taules.

També és important mantenir la consistència entre els objectes a memòria i la seva representació en la BD, si en algun moment es dóna el cas que durant l'execució de l'aplicació hi ha aquesta duplicitat. Si el valor d'algun atribut de la instància a memòria varia, tard o d'hora aquest nou valor s'ha de veure reflectit en la seva representació en la BD. !

La manera en què realment s'emmagatzema i s'accedeix a tota aquesta informació depèn de l'anomenat *sistema gestor de base de dades*, o *SGBD (database management system*, o DBMS, en anglès). L'aspecte que cal destacar d'aquest sistema és que és totalment transparent al desenvolupador d'una aplicació que accedeix a la BD. Ell s'encarrega de resoldre tots els aspectes vinculats a la integritat com, per exemple, que no es repeteixin claus primàries, o l'accés concurrent a les dades.

Les capacitats de cada SGBD i com funcionen internament depèn totalment de cada fabricant. De fet, força aspectes del mateix accés al sistema varien segons el fabricant, pel que només es pot fer una descripció genèrica en els aspectes més senzills. Pels aspectes vinculats al tipus de SGBD concret, aquest text es basa en la distribució de codi obert Apache Derby, desenvolupada totalment en Java.

2.1 Traducció del Model a una BD relacional

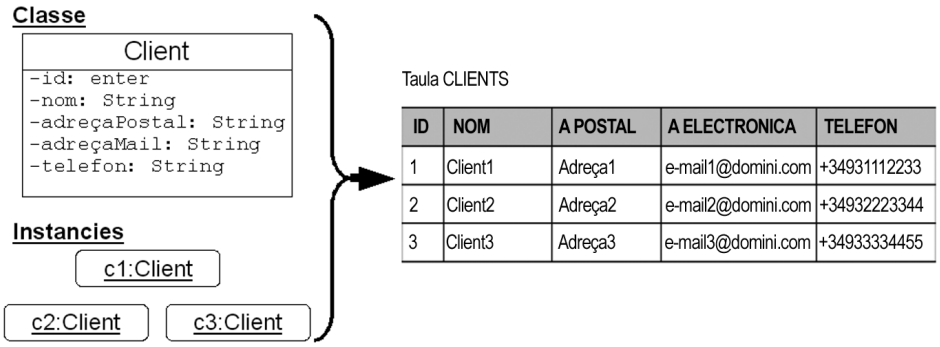
El fet que una BD estructuri la informació en forma de taules implica un procés de transformació des de la manera en què els objectes s'estructuren a memòria, el Model, a com s'organitzen en una BD:

- Cada classe instanciable del Model es materialitza en una taula en la BD.
- Cada atribut definit en una classe es materialitza en la BD com una columna dins la seva taula.
- La persistència de cada objecte del Model es materialitza en la BD en una fila dins la taula que correspon a la seva classe. En cada cel·la de la taula s'emmagatzema el valor que cada objecte en concret té assignat a l'atribut.

Hi ha una nomenclatura formal per representar gràficament BD. S'anomena model entitat-relació.

La figura 2.1 mostra un exemple senzill de traducció de classe a taula d'una BD. Cada fila correspon a la persistència de tres instàncies diferents de la classe Client.

FIGURA 2.1. Exemple de traducció de model a taula d'una base de dades.



En la BD hi ha tantes taules com tipus d'objectes es vol emmagatzemar. Quan es vol recuperar un objecte emmagatzemat, simplement se cerca en la taula corresponent per obtenir els valors dels seus atributs, de manera que si cal es pugui instanciar.

Per cercar un **objecte** concret en una taula, és molt important que sempre hi hagi algun atribut que sigui únic per a cada instància, de manera que no hi hagi cap ambigüitat. La columna que l'emmagatzema és el que es coneix com la **clau primària** d'una taula.

La necessitat de la clau primària està justificada pel fet que, en usar taules per emmagatzemar els objectes, aquests deixen de tenir referències que els identifiquin de manera única i mitjançant les quals s'hi pugui accedir. Per tant, cal afegir algun identificador únic que faci el mateix servei que una referència quan el Model es troba en la memòria. Normalment, s'escull un identificador de tipus enter, ja que ocupa poc espai i és fàcil de comparar.

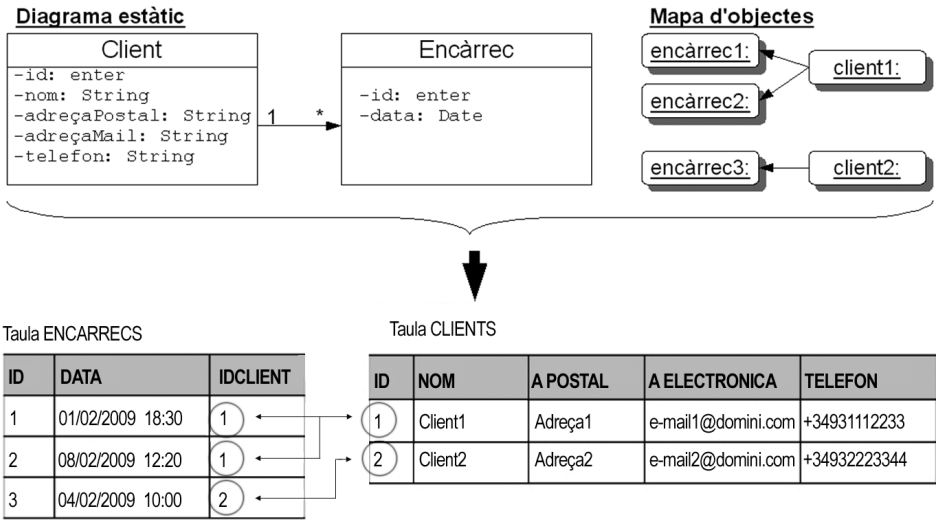
La desaparició de les referències en traspassar un model orientat a objectes a taules dins una BD també té un efecte especialment important en les associacions entre classes: els atributs que conformen llistes d'altres objectes. La traducció d'una associació varia segons la seva cardinalitat, però per tots els casos cal incloure els identificadors únics, les claus primàries, d'elements d'una taula en altres taules. Aquests identificadors que delimiten elements d'una altra taula s'anomenen **claus foranes**.

Quan la cardinalitat en un dels extrems de l'associació a traduir és unitària, 1 o 0..1, llavors, donades dues taules A i B, que representen dues classes relacionades per una associació en el diagrama estàtic UML, es faria el següent:

1. Escollir la taula associada a la classe oposada a la que té cardinalitat unitària. Suposem que aquesta taula és A.
2. Afegir a la taula A una nova columna, en què s'emmagatzemen claus primàries de la taula B. Mitjançant aquesta nova columna és com es referencien a partir d'ara els elements de la taula A.
3. No cal fer res en la taula B.
4. Si els dos extrems de la relació són unitaris, es pot escollir indistintament qualsevol taula per incloure la clau primària de l'altra.

La figura 2.2 mostra un exemple que aclareix molt millor aquest procés. En aquesta figura, d'acord amb els passos indicats, la taula ENCARRECS correspondria a la taula A de la descripció i CLIENTS, en tenir la classe que representa cardinalitat unitària a l'associació, la taula B.

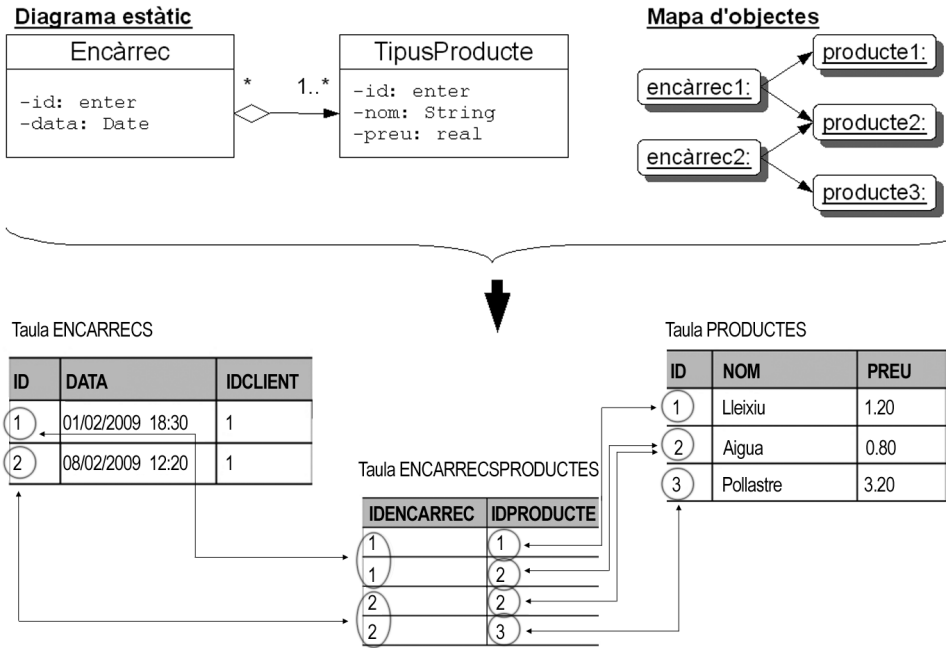
FIGURA 2.2. Tnaslació de les relacions entre objectes als camps d'una taula.



Per enumerar els encàrrecs d'un client determinat, només cal saber-ne l'identificador, i llavors llistar totes les files de la taula d'encàrrecs que tenen aquest identificador en la columna IDCLIENT. En aquest exemple, IDCLIENT és una clau forana a la taula ENCARRECS.

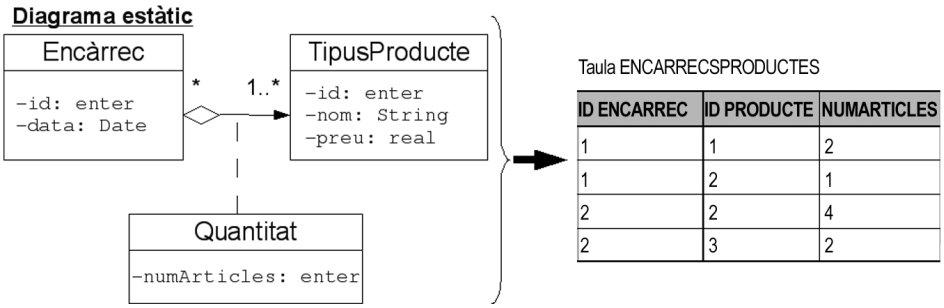
El cas d'una associació en què els dos extrems tenen cardinalitat múltiple, 1..* o *, és una mica més aparatós, ja que requereix la intervenció d'una nova taula dedicada exclusivament a enumerar totes les relacions entre elements de les dues taules associades usant-ne les claus primàries. Cada fila representa una associació entre dos elements. La figura 2.3 aclareix aquest cas:

FIGURA 2.3. Translació de cardinalitats múltiples a camps d'una taula.



Aquest sistema també permet traduir fàcilment a taula una classe associativa, ja que en el fons no és més que un conjunt d'atributs vinculat a una associació. Per tant, com es pot veure en la figura 2.4, el mecanisme és el mateix, però afegint noves columnes d'acord amb els atributs definits en la classe associativa.

FIGURA 2.4. Translació de classes associatives a camps d'una taula.



Finalment, queda exposar el cas especial de les relacions d'herència. Malauradament, l'herència és una propietat exclusiva de l'orientació a objectes, pel que no es pot traduir totalment a un model relacional. Només es pot intentar simular de la millor manera possible. Per fer-ho, hi ha dues possibilitats.

D'una banda, es pot crear una taula diferent per cada classe instanciable, de manera que els objectes s'inclouran en la taula que correspongui d'acord amb el seu tipus exacte (la subclasse més concreta). Això té l'inconvenient que en les taules relatives a classes més concretes hi haurà columnes que ja apareixen en les taules de classes més generals. A més a més, quan calgui fer operacions en les taules d'un tipus d'objecte concret que estigui en les zones més generals de la jerarquia, cal mirar cada taula.

D'altra banda, en lloc de crear diverses taules, se'n pot crear una de sola amb tots els objectes a dins. Les columnes que es definiran correspondran a l'agregació de tots els possibles atributs dins la jerarquia d'herència. Això estalvia tenir columnes repetides, però implica que hi haurà elements amb valors nuls per a certes columnes (ja que pel seu subtipus no els correspon tenir cap valor). Per tant, cal tenir-ho ben present en fer consultes i modificacions a la taula.

En qualsevol cas, sempre que es recuperin dades de les taules de la BD i es vulgui instanciar un objecte que forma part d'una jerarquia d'herència, el desenvolupador haurà d'establir alguna manera d'identificar a quina subclasse concreta pertany cada fila d'una taula.

2.2 El llenguatge SQL

Tot i les particularitats de cada fabricant, hi ha un llenguatge genèric que permet efectuar les accions més bàsiques amb una BD: el llenguatge SQL.

L'**SQL** (*structured query language*, llenguatge de consultes estructurades) és un llenguatge estàndard per a l'accés a BD relacionals, de manera que és possible operar-hi: consultar les dades emmagatzemades, modificar-les...

SQL

Si bé aquest llenguatge és un estàndard, cal tenir en compte que cada fabricant disposa de les seves extensions propietàries, totalment incompatibles entre elles.

Una sentència SQL no és més que una cadena de text, amb una sintaxi concreta, que indica un seguit d'ordres a la BD. En aquest sentit, no és gaire diferent d'un petit bocí de codi d'un programa. Aquesta sintaxi SQL és molt extensa i es poden escriure llibres sencers descrivint-ne totes les funcionalitats. En aquest apartat només es fa un breu incís sobre quines són les operacions mínimes indispensables i la seva sintaxi més bàsica. El suficient per entendre com es poden desenvolupar aplicacions en Java que utilitzin accés a una BD. Les operacions que permet SQL es poden dividir en dos tipus. D'una banda, les que componen el llenguatge de definició de dades (*data definition language*, o DDL), que serveixen per afegir informació a la BD, principalment per gestionar taules. D'altra banda, hi ha les que formen part del llenguatge de manipulació de dades (*data manipulation language*, o DML), que permeten llegir o modificar el contingut de la BD.

De moment veurem quina mena de sentències SQL accepten les BD, i després ja veurem com és possible agafar el text d'aquestes sentències i executar-les sobre la BD mitjançant codi Java.

2.2.1 Tipus de dades

Abans de poder crear cap taula, cal decidir quin és el tipus de les dades que conté a cadascuna de les seves cel·les (igual que cal triar els tipus de dades dels atributs a les classes). Els tipus de dades disponibles i quines són les seves paraules clau són específics a SQL, i per tant, no es pot reusar la nomenclatura de Java o UML. A més a més, cal tenir en compte que cada fabricant de BD defineix els seus propis tipus, per la qual cosa la llista pot variar segons el sistema utilitzat. Cal consultar la documentació de la BD per saber quins hi ha disponibles. De totes formes, tot seguit es llisten els que normalment podreu usar quan treballeu amb els mecanismes que proporciona Java per accedir a BD.

TAULA 2.1. Tipus de dades SQL.

Tipus	Descripció	Tipus Java semblant
CHAR(mida)	Cadena de text de mida fixa.	`String`
VARCHAR(mida)	Cadena de text de mida variable.	`String`
LONG VARCHAR(mida)	Cadena de text arbitràriament llarga, de mida variable.	`String`
BINARY	Bloc binari curt de mida fixa.	`byte[]`
VARBINARY	Bloc binari curt de mida variable.	`byte[]`
LONG BINARY	Bloc binari llarg de mida variable.	`byte[]`
BIT	Un bit, que pot ser només 0 o 1.	`boolean`
TINYINT	Enter de 8 bits, entre -128 i 127.	
SMALLINT	Enter de 16 bits, entre -32768 i 32767.	`short`
INTEGER	Enter de 32 bits, entre -2147483648 i 2147483647.	`int`
BIGINT	Enter de 64 bits.	`long`
REAL	Real de simple precisió.	`float`
DOUBLE	Real de doble precisió.	`double`
DATE	Una data consistent en dia, mes i any.	`java.sql.Date`
TIME	Temps consistent en hora, minuts i segons.	`java.sql.Time`
TIMESTAMP	Combinació de DATE i TIME, i, a més a més, el seu equivalent en nanosegons.	`java.sql.Date`

Vegem amb una mica més de detall algunes de les característiques més rellevants de cada tipus de dades, que caldrà tenir en compte en programar en Java aplicacions que gestionen dades dins una BD usant SQL.

Tipus relatius a cadenes de text

El tipus `VARCHAR` sol ser el més típicament usat per desar cadenes de text a la BD, i està suportat de totes les principals bases de dades. En definir-lo, cal usar un paràmetre que especifica la longitud màxima de la cadena. Per tant, `VARCHAR(25)` defineix una cadena la longitud de la qual pot ser de fins a 25 caràcters. Ara bé, cal tenir present que les BD solen tenir un límit en la mida màxima que poden suportar, que sol rondar els 254 caràcters. Tot i definir aquesta mida màxima, sempre que es desa un valor a una cel·la d'aquest tipus, quan posteriorment es consulta aquest valor, la cadena de text retornada té com a mida el seu valor original al ser desada. No s'"omple" artificialment amb espais o altres caràcters fins arribar al màxim.

En contrast, les cadenes de longitud fixa, usant el tipus CHAR, sí que sempre mantenen la seva longitud indicada en ser introduïdes, i posteriorment consultades, a la BD, i per tant, si el nombre de caràcters és inferior al valor especificat, la resta de posicions s'omplen amb espais en blanc fins arribar a la mida definida. La seva mida màxima també sol ser de 254 caràcters. Per tant, donades les definicions `VARCHAR(10)` i `CHAR(10)`, si desem el text "Hola" a la BD, en consultar posteriorment el valor, en el primer cas s'obtindrà "Hola", mentre que en el segon "Hola ". Això és un fet que cal tenir molt en compte.

Finalment, el tipus `LONGVARCHAR`, en canvi, ve justificat perquè totes les principals bases de dades han de suportar algun tipus de gran cadena de gran longitud. Per "gran longitud" s'està referint a mides de fins un gigabyte.

Tot i aquestes particularitats, en general, a un programador en Java no li cal distingir entre els tres tipus de dades a l'hora de gestionar les dades, totes elles es poden expressar dins el codi del programa com un tipus `String` o directament a un *array* de `char` (`char[]`).

A SQL els **literals de cadenes de text** s'especifiquen entre cometes simples, '...'

Tipus relatius a cadenes binàries

Els tres tipus vinculats a valors binaris segueixen una filosofia molt semblant a les cadenes de text, al menys en el que es refereix al seu aspecte de cadena de mida fixa o variable. Per tant, una dada del tipus `BINARY` es farceix amb valors addicionals fins fer-la arribar a la mida establerta. Ara bé, desafortunadament, l'ús d'aquests tipus de binaris no ha estat estandarditzada i el seu suport varia considerablement entre les principals bases de dades. Per tant, és imprescindible mirar-ne la documentació, ja que el que s'aplica per a un cas pot no ser cert per a un altre. Fins i tot, pot ser que una BD no implementi alguns d'aquests tipus.

De manera semblant a les cadenes de text, els tipus `BINARY` i `VARBINARY` solen estar limitats a 254 bytes, mentre que `LONGVARBINARY` es pot usar per a mides

arbitràriament grans, normalment fins a un gigabyte de dades. En qualsevol cas, les dades relatives a aquests tipus es poden gestionar dins d'un programa en Java usant *arrays* de `byte` (`byte[]`).

Finalment, el tipus `BIT` és especial, ja que gestiona un únic valor binari, 0 o 1. Per aquest motiu, tot i no ser estrictament el mateix, el tipus de dades recomanat per gestionar-lo dins un programa en Java és el `boolean`. Es pot fer que `true` equivalgui a 1 i `false` a 0.

Tipus relatius a valors numèrics

Tots els tipus associats a valors numèrics, ja siguin enters o reals, es comporten de manera molt similar, ja que existeix una correspondència pràcticament immediata entre el tipus de dades SQL i els tipus de dades que proporciona Java. Per tant, saber com gestionar dades d'aquests tipus en programes que accedeixen a una BD no és gaire difícil. Només cal usar el tipus equivalent i ja està, sense cap complicació extra.

L'únic cas on cal anar amb compte és per al tipus `TINYINT`, de 8 bits, ja que el tipus de valors enters més petit en Java, el `short`, és de 16 bits. De cara a consultar valors des de la base de dades no hi ha cap problema, però en emmagatzemar-ne, cal ser conscients que si el valor enter a desar és superior al rang possible (-128 i 127), el valor que acabarà a la BD no serà correcte. De fet, a nivell general, com passa en fer conversions entre tipus numèrics en Java, si en consultar un valor de la BD s'usa una variable d'un tipus amb major capacitat, mai no hi haurà problemes, per la qual cosa res no impedeix usar un `double` Java per consultar valors de tipus `REAL`.

Tipus relatius a dates

Finalment, entre els tipus bàsics de SQL es troben uns d'especials que serveixen per gestionar valors de dates. Si bé, dins una aplicació Java genèrica, l'ús de dates no ha de ser necessàriament molt usat, a les BD molt sovint sí que cal desar dates, per la qual cosa tenir un tipus especial només per a això està bastant justificat i facilita molt la feina (més que haver de desar per separat cada part o usar cadenes de text amb cert format).

El tipus `DATE` representa una data que consta de dia, mes i any, mentre que el tipus `TIME` indica només hores, minuts i segons. Com ja passava amb altres tipus de dades, aquests s'implementen per només un subconjunt de les bases de dades principals. Algunes bases de dades ofereixen alternatives de tipus SQL que serveixen per al mateix propòsit. El tipus `TIMESTAMP` és una combinació dels dos, però està suportat per un nombre molt petit de bases de dades i per tant no és gaire recomanable dependre'n molt. Aquest darrer tipus, junt amb les dades relatives a data i temps, també desa un camp auxiliar on es compta fins a una precisió de nanosegons.

Vigileu especialment si useu els mecanismes de correcció automàtica del Netbeans.

Java proporciona tres classes que serveixen específicament per tractar aquests tipus de dades SQL (i només per al cas d'SQL): `java.sql.Date`, `java.sql.Time` i `java.sql.Timestamp`, ja que les seves classes per gestionar dates a nivell genèric no encaixen exactament en els seus camps amb aquests tipus. Per tant, alerta, cal anar amb molt de compte amb la classe que s'usa per gestionar dades d'aquests tipus als vostres programes, ja que, per exemple, en Java hi ha dues classes anomenades `Date`. Una al *package* `java.util` i una altra al `java.sql`.

Cal dir que, tot i que aquestes tres classes, de fet, hereten de `java.util.Date`, i, per tant, en el pitjor cas sempre és possible fer conversions entre elles d'alguna manera (consultant les dades dins l'objecte i adaptant-les al nou format), sempre és preferible usar la classe vinculada exclusivament a SQL.

Dominis

A més dels tipus de dades predefinits, existeix la possibilitat de treballar amb dominis definits per l'usuari, on s'especifiquen un conjunt de valors concrets possibles. Per exemple, suposeu que voleu definir un camp on els únics valors possibles són un conjunt de ciutats concret i no voleu definir el tipus com un VARCHAR, on es podria escriure qualsevol cosa. En aquest cas, usar un domini seria la solució ideal. Per fer-ho cal usar la sentència **CREATE DOMAIN** sobre la base de dades.

```
1 CREATE DOMAIN nomDomini AS tipusDades
2 CONSTRAINT nomRestricció
3 CHECK (condicions)
```

Els valors "nomDomini" i "nomRestricció" els trieu vosaltres (un cop definits, es poden usar en altres comandes). El "tipus de dades" correspon al tipus genèric al qual pertanyen els valors que es volen concretar. Per exemple, per al cas de noms de ciutats, podria ser VARCHAR (25), si cap nom supera els 25 caràcters. El paràmetre de "condicions" indica què ha de complir un valor dins aquest domini per ser considerat vàlid. SQL contempla una sintaxi per establir comparacions i avaluar diferents condicions lògiques complexes (`AND`, `OR`...). De totes maneres, normalment per a aquest cas el que se sol fer és indicar directament una llista de valors, mitjançant la sentència **VALUE IN**, on s'inclou entre parèntesis els valors acceptats. Per exemple:

```
1 CREATE DOMAIN ciutats AS VARCHAR(25)
2 CONSTRAINT llistaCiutats
3 CHECK (VALUE IN ('Barcelona', 'Badalona', 'Mansou', 'Alella'))
```

Quan es defineix un domini, aquest es pot usar a nivell de sintaxi de les sentències de gestió de taules exactament igual que qualsevol altre tipus SQL. Per tant, donat el domini de ciutats, es pot usar "ciutats" com si fos un tipus de dades.

Si mai es vol eliminar la definició d'un domini a la nostra BD, es pot usar la sentència **DROP DOMAIN**.

```
1 DROP DOMAIN nomDomini (RESTRICT|CASCADE)
```

Aquesta necessita un paràmetre, que pot ser **RESTRICT** o **CASCADE**. En el primer cas, el domini només s'esborrarà realment si no s'usa enlloc de la BD (a cap taula). En el segon cas, s'esborrarà encara que estigui en ús, però allà on es doni aquest cas, es reemplaça automàticament pel tipus de dades associat al domini. Per exemple, si s'elimina el domini "ciutats", a totes les taules on s'usin les columnes passaran a ser del tipus `VARCHAR(25)` automàticament, ja que no poden quedar simplement sense cap tipus definit.

2.2.2 Gestió de taules

Per poder accedir a qualsevol dada dins els vostres programes, evidentment, abans de res aquestes dades han d'existir dins taules a la BD relacional. Normalment, les taules es creen *a priori*, i ja existeixen abans que la vostra aplicació hi interactuï. Per fer-ho, existeixen diferents programes que, mitjançant una interfície gràfica, permeten editar les propietats de les taules de les quals es vol disposar (noms de files, columnes...). Els IDE complexos com Netbeans incorporen ja una funcionalitat a aquest efecte, per tal de facilitar la feina.

De totes maneres, hi pot haver casos on pot ser necessari crear taules. El cas més immediat seria si precisament el que es vol és inicialitzar el sistema gestor de bases de dades, però fer-ho manualment és molt complicat o costós, o depèn de molts paràmetres que poden variar segons el cas. En aquest cas, res millor que un programa per automatitzar la feina. Per tant, és interessant donar una ullada a les sentències SQL per crear, esborrar i modificar les diferents taules que componen una BD.

Creació de taules

Per crear una taula s'usa la sentència **CREATE TABLE**, d'acord amb una llista de noms de columna i el tipus de dades que conté. Per a cada parell de noms de columna i tipus de dades es poden incloure un seguit de paràmetres per especificar informació addicional, com quina correspon a la clau primària, quin és el tipus de dades que es pot emmagatzemar, la seva mida màxima...

```
1 CREATE TABLE nomTaula
2 (nomColumna1 tipus [valorDefecte] [restriccions],
3 nomColumna2 tipus [valorDefecte] [restriccions],
4 ...)
```

Els camps de valor per defecte i de restriccions són opcionals.

Per al primer cas, es poden indicar valors per defecte a una cel·la d'aquella columna. Això és útil, per exemple, quan, en emmagatzemar informació dins una BD, trobeu que alguns dels valors dins una cel·la encara no se saben. Se sabrà més endavant, però ara mateix el que no voleu és deixar d'emmagatzemar la informació que ja sabeu. Que el procés d'afegir una fila a una taula no sigui tot o res. Per dur a terme això, normalment el que es fa és disposar d'algun valor especial, o valor per defecte, amb el qual s'indica que en aquella cel·la no hi ha un valor vàlid encara.

Per fer-ho, s'usa el paràmetre `DEFAULT valor`. El valor pot ser un literal que indica exactament el valor per defecte, o bé es poden usar algunes paraules clau especials d'SQL:

Taula 2.2. Paraules clau especials d'SQL.

Valor	Descripció
NULL	Valor especial nul (marca d'invàlid). Semblant a una referència a `null`.
USER	El nom de l'usuari de la BD que ha executat la sentència.
CURRENT_DATE	La data actual, només per a camps de tipus DATE.
CURRENT_TIME	L'hora actual, només per a camps de tipus TIME.

En relació amb les restriccions, n'hi ha unes quantes de disponibles, per la qual cosa ens centrarem en les més importants. Per una banda, hi ha les restriccions de columna, que indiquen condicions que ha de complir qualsevol accés a la BD per ser considerat vàlid. Si no es compleix la condició, la BD retorna un error. D'aquesta manera podeu garantir automàticament que les dades que es desen mantenen una coherència sense haver de fer-ho amb codi dins el vostre programa. Entre les més típiques es troben:

Taula 2.3. Restriccions típiques d'SQL.

Restricció	Descripció
PRIMARY KEY	La columna desa la clau primària de cada fila (els valors no poden ser repetits ni nuls).
NOT NULL	No s'admeten valors nuls.

Per exemple, si es vol fer una taula de clients on el seu identificador serà la clau primària de cada client, es faria:

```
1 CREATE TABLE CLIENTS
2 (ID INTEGER PRIMARY KEY,
3 NOM VARCHAR(30),
4 APOSTAL VARCHAR(50),
5 AELECTRONICA VARCHAR(25),
6 TELEFON VARCHAR(15))
```

Amb la qual cosa es crea la taula (buida, de moment):

Taula 2.4. Taula CLIENTS inicial

ID	NOM	APOSTAL	AELECTRONICA	TELEFON

Modificació de taules

Un cop una taula ja ha estat creada, és possible fer-hi modificacions, sense haver d'esborrar-la i crear-ne una de nova, usant la sentència **ALTER TABLE**.

```
1 ALTER TABLE nomTaula
2 acció
```

Les accions possibles més rellevants **ADD**, **ALTER** i **DROP**, per afegir, modificar o eliminar una columna dins la taula. Cada acció usa un seguit de paràmetres semblants a quan es crea una taula des de zero, només que en aquest cas es tracta d'una definició *a posteriori*. La sinxati d'aquestes accions és:

```
ADD nomColumna tipus [valorDefecte] [restriccions]
MODIFY nomColumna tipus [valorDefecte] [restriccions]
DROP nomColumna (RESTRICT|CASCADE)
```

En el cas de DROP, no cal definir res, ja que el que s'està fent és eliminar. Ara bé, com que les dades de la columna poden estar referenciades dins altres taules de la BD, cal preveure aquesta situació. Indicant el paràmetre RESTRICT, l'operació no es durà a terme si es dóna aquest cas. Amb el paràmetre CASCADE, tot el que referenciï aquesta columna també s'eliminarà.

Per exemple, si es vol afegir una columna amb el nombre de vegades que un client ha comprat a l'establiment, es pot fer:

```
ALTER TABLE CLIENTS
ADD NCOMANDES INTEGER NOT NULL
```

Amb això, la taula CLIENTS passa a ser la següent:

Taula 2.5. Taula CLIENTS actualitzada

ID	NOM	APOSTAL	AELECTRONICA	TELEFON	NCOMANDES

A part, també és possible canviar el nom d'una taula amb la sentència:

```
ALTER TABLE nomTaula
RENAME TO nouNom
```

Esborrat de taules

En qualsevol moment també és possible eliminar totalment una taula de la BD a partir del seu nom. La sintaxi és la següent:

```
DROP TABLE taula (RESTRICT|CASCADE)
```

Els paràmetres RESTRICT i CASCADE tenen el mateix significat que en modificar una columna (de fet, esborrar una taula és com esborrar totes les seves columnes). Un cop eliminada, tota la informació que conté es perd, per tant, cal anar amb molt de compte en executar aquesta sentència.

2.2.3 Consulta de dades

La tasca més freqüent, la que segurament s'executarà més vegades en un programa que utilitza una BD, és de ben segur la cerca d'informació entre les dades que té

emmagatzemades. Per fer-ho, es disposa de la sentència **SELECT**, que permet recuperar informació de la BD d'acord amb algun criteri. Atès que una cerca ha de ser capaç de poder englobar molts criteris, de manera que es trobi exactament la dada que es necessita, aquesta sentència pot incorporar opcions ben complexes. Per això, partirem de la seva sintaxi bàsica i després veurem com es pot anar refinant la cerca, ampliant-la.

Per anar seguint les diferents instruccions, partirem d'una taula anomenada CLIENTS, on es desen un seguit de clients amb certa informació. Sobre ella es veuran diferents exemples per a cada tipus de sentència:

Taula 2.6. Taula CLIENTS

ID	NOM	APOSTAL	AELECTRONICA	TELÈFON	NCOMANDES
1	Client1	Adreça1	e-mail1@domini.com	+34931112233	4
2	Client2	Adreça2	e-mail2@domini.com	+34932223344	1
3	Client3	Adreça3	e-mail3@domini.com	+34933334455	10
4	Client4	Adreça3	e-mail4@domini.com	+34933335566	7

Per començar, la seva sintaxi base, que permet una cerca molt general, és:

```
SELECT nomNolumna1 [AS nouNom], nomColumna2 [AS nouNom]...
FROM nomTaula
```

En executar-la, la BD retorna les columnes demanades de la taula indicada. La part `As nouNom`, que es pot posar darrere el nom de cada columna, és opcional i permet reanomenar la columna tal com es mostra en el resultat, enlloc d'usar el nom original que hi ha a la taula.

El format de la resposta, com s'engloben les dades consultades, ja depèn del mecanisme usat per executar la sentència. Per exemple, si s'ha usat una aplicació gràfica, les columnes es poden mostrar en una nova taula reduïda, només amb les columnes desitjades. Si és un programa per línia de comandes, potser s'imprimeixen per pantalla com un text. O si es tracta d'un accés mitjançant codi en Java, com no podria ser d'altra manera, estaran englobades dins un objecte. De moment, però, no cal donar més voltes a aquest fet. N'hi ha prou a saber que, d'una manera o una altra, existirà un mecanisme per poder accedir al resultat sempre que s'executi aquesta sentència. Per ara, es visualitzarà la resposta d'una sentència **SELECT** com una nova taula.

Per tant, donades les següents sentències, si es volen enumerar només els noms i les adreces dels clients de la taula original, es faria així:

Sentència:

```
SELECT NOM, APOSTAL AS ADREÇA
FROM CLIENTS
```

Resultat:

TAULA 2.7. Resultat de la comanda SELECT

NOM	ADREÇA
Client1	Adreça1
Client2	Adreça2
Client3	Adreça3
Client4	Adreça3

Si, en lloc d'una llista de noms de columnes, s'usa un **asterisc**, *, es retornen tots els valors de la taula.

Sentència:

```
1 SELECT *
2 FROM CLIENTS
```

Resultat:

TAULA 2.8. Resultat de la comanda SELECT

ID	NOM	APOSTAL	AELECTRONICA	TELÈFON	NCOMANDES
1	Client1	Adreça1	e-mail1@domini.com	+34931112233	4
2	Client2	Adreça2	e-mail2@domini.com	+34932223344	1
3	Client3	Adreça3	e-mail3@domini.com	+34933334455	10
4	Client4	Adreça3	e-mail4@domini.com	+34933335566	7

També val la pena comentar que, en cas de taules on pot haver-hi valors repetits a les columnes, si es vol que en el resultat d'una consulta descarti repeticions, es pot usar `SELECT DISTINCT`, en lloc de només `SELECT`. L'exemple següent té en compte el fet que els clients 3 i 4 viuen al mateix lloc.

```
1 SELECT DISCTINCT APOSTAL
2 FROM CLIENTS
```

Resultat:

TAULA 2.9. Resultat de la comanda SELECT

APOSTAL
Adreça1
Adreça2
Adreça3

Aplicació de funcions

Ara bé, sovint, el que es vol no és pas simplement llistar tota la informació d'una columna, sinó només aquelles dades que compleixen certes condicions. Atès que les sentències de consulta són les més executades, el que no té sentit, o seria molt ineficient, és que només es pugui consultar un seguit de columnes senceres i després sigui tasca vostra haver de processar-les usant codi per refinar la cerca. Per exemple, cercar quin és el darrer client que s'ha donat d'alta, o si existeix algun client amb una adreça de correu electrònic concreta. Així doncs, en realitat, mitjançant la sentència `SELECT` és possible delegar aquesta tasca a la BD, que, en definitiva, és qui emmagatzema i gestiona tota la informació, de manera que tot plegat resulti més eficient.

Les funcions acceptades són semblants a les que existeixen als programes típics de full de càlcul.

Per una banda, SQL conté un seguit de funcions predefinides que es poden aplicar sobre les columnes, de manera que es demana a la BD que treballi, no sobre els valors de la columna, sinó sobre el resultat d'aplicar aquesta funció. Entre les més destacades:

Taula 2.10. Funcions SQL

Funció	Descripció
MAX(nomColumna)	Retorna la fila amb el valor màxim en aquesta columna.
MIN(nomColumna)	Retorna la fila amb el valor mínim en aquesta columna.
SUM(nomColumna)	Retorna la suma de totes les files.
AVG(nomColumna)	Retorna el valor mitjà de totes files.
COUNT(nomColumna)	Retorna el nombre de files.
FIRST(nomColumna)	Retorna només la primera fila.
LAST(nomColumna)	Retorna només la darrera fila.
UCASE(nomColumna)	Transforma els valors de les files a tot majúscules.
LCASE(nomColumna)	Transforma els valors de les files a tot minúscules.
LEN(nomColumna)	Retorna la longitud dels valors a les files.
ROUND(nomColumna)	Arrodoneix els valors.

Evidentment, les funcions només es poden aplicar sobre aquelles columnes de tipus de dades on tingui sentit. Per exemple, només es pot fer la suma dels valors d'una columna on es desin valors numèrics.

Cerques mitjançant condicions

Ara bé, sovint, en usar aquestes funcions, la semàntica de les dades consultades canvia. Per exemple, si es vol comptar el nombre d'elements que té la taula, i s'usa la funció `COUNT`, ara el valor retornat ja no es correspon estrictament al valor de cap columna en concret. No és ni un ID, ni un Nom... Per això, la sentència `SELECT` permet demanar a la BD que, quan retorni el resultat, reanomeni la columna transformada amb un altre nom. Això pot ser útil per fer més llegible

el resultat, o, en definitiva, el codi d'un programa que la usi. Aquesta és la utilitat de l'apartat opcional `AS nouNom`.

Per exemple, si es vol veure quants clients hi ha a la taula CLIENTS, es podria fer la consulta següent:

Sentència:

```
1 SELECT COUNT(NOM) AS NCLIENTS
2 FROM CLIENTS
```

Resultat:

TAULA 2.11. Resultat d'executar SELECT

NCLIENTS
4

Si bé la sentència `SELECT`, tal com s'ha vist fins ara, és suficient per recuperar dades de manera molt bàsica, és possible fer cerques molt més potents mitjançant l'addició del paràmetre opcional `WHERE condició` al final. Aquesta condició pot combinar diferents operadors, o fins i tot crides a les funcions SQL, de manera que arribi a ser tan complexa com es vulgui, sempre i quan s'avaluï a cert o fals, de manera molt semblant a les condicions d'una sentència condicional o iterativa d'un programa en Java (`if`, `while`...). En cas de combinar diferents condicions simples per fer-ne una de més complexa, aquestes es poden anar agrupant usant parèntesi, com al Java.

Els operadors que accepta SQL són els següents. Aneu amb compte, ja que alguns són diferents que en Java. Per exemple, la igualtat no és un doble igual (==), sinó només un (=).

TAULA 2.12. Operadors SQL

Operador	Descripció
=	Igual
<	Menor
<=	Menor o igual
>	Major
>=	Major o igual
<>	Diferent
NOT	Negació lògica
AND	Conjunció lògica
OR	Disjunció lògica

Recordeu que els literals de cadenes de text en SQL s'escriuen entre cometes simples, i no dobles cometes com al Java.

Per exemple, si es vol enumerar només els clients que viuen a l'adreça "Adreça3", es podria fer la consulta que hi ha a continuació. En aquesta, és la pròpia BD qui s'encarrega de fer el processament i la discriminació dels elements, no cal que ho fem nosaltres després al codi del nostre programa, la qual cosa no només és molt més eficient des del punt de vista d'execució del programa, sinó que és més ràpid de codificar i fa el codi més simple.

Sentència:

```
SELECT *
FROM CLIENTS
WHERE APOSTAL='Adreça3'
```

Resultat:

TAULA 2.13. Resultat d'executar SELECT

ID	NOM	APOSTAL	AELECTRONICA	TELÈFON	NCOMANDES
3	Client3	Adreça3	e-mail3@domini.com	+34933334455	10
4	Client4	Adreça3	e-mail4@domini.com	+34933335566	7

Ara bé, com s'ha dit, res no impedeix fer condicions complexes on s'usen també les funcions exposades amb anterioritat. Per exemple, si es vol cercar tots els noms dels clients que han fet un conjunt de compres inferior a la mitjana (i, ja de pas, els etiquetem com a "pitjors clients"), es podria fer la consulta que ve a continuació. El valor mitjà de comandes és 5.5, per la qual cosa, el resultat estableix els clients que tenen menys comandes que aquest valor.

Sentència:

```
SELECT NOM AS PITJORCLIENT
FROM CLIENTS
WHERE NCOMANDES < AVG(NCOMANDES)
```

Resultat:

TAULA 2.14. Resultat d'executar SELECT

PITJORCLIENT
Client1
Client2

Altres cerques mitjançant WHERE

Finalment, després d'un apartat `WHERE` també es poden afegir un seguit de paraules clau que permeten anar més enllà dels típics operadors lògics o matemàtics, de manera que encara es poden fer cerques més concretes, o d'acord a criteris encara més específics, però freqüents en el context d'una cerca. Aquestes opcions es poden combinar amb altres condicions per fer cerques encara més complexes. Tot seguit s'expliquen algunes de les més destacades.

L'opció **IN**, o **NOT IN**, permet indicar una llista de valors, de manera que la condició es considera certa si el valor d'una fila a la columna corresponent és (o no) entre algun dels valors de la llista. Per exemple, si es volguessin consultar els noms i correus electrònics dels clients que viuen a l'adreça postal 2 o 3 (fixeu-vos que hi ha dos clients que viuen a l'Adreça 3), es faria així:

Sentència:

```
SELECT NOM, AELECTRONICA
FROM CLIENTS
WHERE APOSTAL IN ('Adreça2', 'Adreça3')
```

Resultat:

TAULA 2.15. Resultat d'executar SELECT

NOM	AELECTRONICA
Client2	e-mail2@domini.com
Client3	e-mail3@domini.com
Client4	e-mail4@domini.com

L'opció **NULL**, o **NOT NULL**, permet filtrar una consulta d'acord a les files que en alguna columna tenen un valor NULL, o no. Per exemple, si es vol obtenir una llisa de telèfons vàlids, partint de la suposició que es permet donar d'alta clients encara que no se sàpiga el seu telèfon, es pot fer d'aquesta manera:

Sentència:

```
SELECT NOM, TELÈFON AS CLIENTSATRUCAR
FROM CLIENTS
WHERE TELÈFON NOT NULL
```

Resultat:

TAULA 2.16. Resultat d'executar SELECT

NOM	CLIENTSATRUCAR
Client1	+34931112233
Client2	+34932223344
Client3	+34933334455
Client4	+34933335566

L'opció **BETWEEN valorInicial AND valorFinal** permet simplificar la creació de condicions on es vol veure si un valor es troba dins un rang. Per exemple, si es vol trobar el nom dels clients que han fet entre 3 i 8 compres, es faria amb aquesta sentència:

Sentència:

```
SELECT NOM
FROM CLIENTS
WHERE NCOMANDES BETWEEN 3 AND 8
```

Resultat:

TAULA 2.17. Resultat d'executar SELECT

NOM
Client1
Client4

Finalment, l'opció **LIKE patró** permet establir si els valors compleixen un patró concret, en lloc d'haver de comparar una igualtat estricta. Això és molt útil, ja que sovint, en fer cerques sense saber exactament el contingut de les dades disponibles, el que es vol és trobar elements que compleixin de manera aproximada certes condicions, ja que *a priori* és impossible saber exactament què es pot cercar realment. El patró es representa com un literal de cadena de text, amb l'opció de posar un subratllat, '_', per indicar qualsevol lletra, i un percentatge, '%', per indicar una seqüència de 0 o més lletres qualssevol.

Per exemple, suposem que es vol cercar el nom dels clients amb un número de telèfon on, en alguna part, hi ha dos nombres 4 consecutius. La consulta seria la següent:

Sentència:

```
SELECT NOM, NTELÈFON
FROM CLIENTS
WHERE NTELÈFON LIKE '%44%'
```

Resultat:

TAULA 2.18. Resultat d'executar SELECT

NOM	TELÈFON
Client2	+34932223344
Client3	+34933334455

Ordenació dels resultats

Igual que la BD ja és capaç de dur a terme cerques d'acord a certs criteris usant les comandes adients d'SQL, de manera que no cal consultar totes les dades i després processar-les dins els vostres programes, també és possible establir criteris d'ordenació, de manera que el resultat ja està ordenat automàticament i no cal que ho feu vosaltres *a posteriori*.

Per assolir-ho, es pot usar el paràmetre **ORDER BY nomColumna1, nomColumna2,...** al final de la sentència `SELECT`. Aquest paràmetre ordena les files del resultat d'acord als valors de diferents columnes, per ordre d'importància. En el cas de valors de tipus cadena de text, s'ordenen els elements alfabèticament.

Per defecte, l'ordenació és ascendent, del valor més petit al més gran, però si es desitja es pot fer el cas invers usant la paraula clau `DESC` immediatament després del nom d'una columna. Per exemple, per mostrar els clients ordenats per nombre de compres, de manera que primer es mostra el que n'ha fet més, s'usaria la sentència:

Sentència:

```
SELECT *
FROM CLIENTS
ORDER BY NCOMANDES DESC
```

Resultat:

TAULA 2.19. Resultat d'executar SELECT

ID	NOM	APOSTAL	AELECTRONICA	TELÈFON	NCOMANDES
3	Client3	Adreça3	email3@domini.com	+34933334455	10
4	Client4	Adreça3	email4@domini.com	+34933335566	7
1	Client1	Adreça1	email1@domini.com	+34931112233	4
2	Client2	Adreça2	email2@domini.com	+34932223344	1

Combinació de taules

De vegades, és necessari combinar la informació de diverses taules diferents per obtenir un únic resultat, sovint quan les condicions del paràmetre `WHERE` depenen de valors desats a altres taules. SQL permet fer operacions a partir de les dades de més d'una taula simplement especificant a l'apartat `FROM` que la informació a tractar prové de més d'una taula enlloc d'una, enumerades com una llista separada per comes: `FROM nomTaula1, nomTaula2...`. En principi, es poden combinar tantes taules com es vulgui, però ens centrarem en el cas de només dues taules.

Quan s'usa aquest mecanisme, per a cada nom de columna usada a qualsevol part de la sentència cal especificar a quina taula exactament s'està referint, no n'hi ha prou de posar només el nom de la columna. Això evita conflictes en duplicitats de noms entre taules diferents. Per fer-ho, només cal usar la sintaxi `nomTaula.nomColumna`, de manera molt semblant a l'accés a elements públics dins objectes en Java. Aquesta nomenclatura cal usar-la tant quan s'enumeren les columnes després del `SELECT` com dins la condició al `WHERE`, si n'hi ha.

Si els noms de les taules són llargues, pot passar que el text de la sentència sigui una mica farragós. Per això, en declarar les taules on fer la consulta, és possible assignar àlies abreujats, que poden ser usats a la resta de la sentència. L'àlies s'escriu directament després del nom de la taula.

```
1 SELECT àlies.nomColumna1, àlies,nomColumna2, ...
2 FROM nomTaula1 àlies1, nomTaula2 àlies2
3 WHERE condició
```

Per veure-ho més clar, utilitzem un exemple. Suposem que existeix una nova taula PREMIS on s'enumeren premis de vals de descompte segons el nombre de compres acumulades d'un client. Pot ser interessant combinar la taula de clients amb aquesta per saber si a algun client li correspon alguna oferta.

Taula 2.20. Resultat d'executar SELECT

NCOMANDES	VAL
5	Un val de descompte del 5% a la propera compra.
10	Un val per a una tovallola de platja de regal.
15	Un val per a un 2 per un en qualsevol producte.
20	Dos vals de descompte de 10 euros en comandes diferents.

Si es vol processar un enviament de vals de premi a les diferents adreces de clients que s'ho han guanyat, caldrà combinar les dues taules: saber quins premis mereix cada client i saber a quin nom i a quina adreça cal enviar el premi.

Sentència:

```
SELECT c.NOM, c.APOSTAL, p.VAL
FROM CLIENTS c, PREMIS p
WHERE (c.NCOMANDES >= p.NCOMANDES)
```

Resultat:

Taula 2.21. Resultat d'executar SELECT

NOM	APOSTAL	VAL
Client3	Adreça3	Un val de descompte del 5% a la propera compra.
Client3	Adreça3	Un val per a una tovallola de platja de regal.
Client4	Adreça3	Un val de descompte del 5% a la propera compra.

Un fet molt important és veure com en el resultat s'han combinat dades de totes dues taules. Ara hi ha columnes tant de la taula CLIENTS com de la taula PREMIS, barrejades. Atès que el client 3 es mereix dos premis, ara apareix dues vegades repetit, una per cada cop que el seu nombre de compres compleix la condició establerta (major o igual que) amb qualsevol de les files a la columna NCOMANDES de la taula de PREMIS. Com que succeeix dues vegades, hi ha dues aparicions, i s'associa cada columna VAL a cada aparició. Bàsicament, la comparació es fa a nivell de totes les files d'una taula contra totes les files de l'altra, una a una.

En versions posteriors d'SQL, la sintaxi per dur a terme aquesta tasca s'ha modificat, de manera que per fer la mateixa tasca es pot usar un format de sentència diferent, mitjançant el paràmetre **JOIN ... ON ...**. Amb els dos es poden obtenir els mateixos resultats, però internament la consulta feta amb JOIN ... ON ... és molt més ràpida.

```
SELECT àlies.nomColumna1, àlies,nomColumna2, ...
FROM nomTaula1 àlies1
JOIN nomTaula2 àlies2
ON condicióOn
WHERE condicióWhere
```

En aquest cas, la segona taula, que es pot considerar com a auxiliar, ja que és la que conté informació extra sobre la qual fer comparacions o extreure dades, s'identifica explícitament. La condició a la part `ON` indica també explícitament quina és la condició sota la qual es vinculen les dues taules. En aquesta sintaxi, pot ser que no calgui afegir el `WHERE` si amb la condició `ON` és suficient per vincular les dades entre taules.

Per exemple, la sentència per llistar els premis que cal enviar es podria adaptar a aquest altre format de la manera que es veu a continuació, amb idèntics resultats. Cal comptar que en aquesta consulta, la condició que vincula les dues taules és la relació entre les columnes NCOMANDES d'amdues, per la qual cosa, aquesta és la que correspon a l'`ON`:

Sentència:

```
SELECT c.NOM, c.APOSTAL, p.VAL
FROM CLIENTS c
JOIN PREMIS p
ON (c.NCOMANDES >= p.NCOMANDES)
```

Un fet molt curiós de la unió de taules és que, de fet, res no impedeix operar amb la mateixa taula alhora, usant el mateix nom de taula al `FROM` i al `JOIN`. Tot i que pot semblar un cas molt estrany, suposeu la següent consulta: llistar tots els clients que han fet menys comandes que el client 4. Amb la unió de taules aquesta consulta és possible, ja que el que cal fer és creuar totes les dades de la taula CLIENTS amb les del client 4 (que també és a la mateixa taula). Per tant, es pot fer:

Sentència:

```
SELECT c.NOM
FROM CLIENTS c1
JOIN CLIENTS c2
ON (c1.NCOMANDES < c2.NCOMANDES)
WHERE c2.NOM='Client4'
```

Resultat:

TAULA 2.22. Resultat d'executar SELECT

NOM
Client1
Client2

Per veure més clar què ha succeït, podeu dividir la sentència en dues parts. Per una banda, el `WHERE` indica que es treballa amb una versió de la taula CLIENTS, "c2", on només hi ha els clients amb nom "Client4" (una única fila). Per altra banda, hi ha la taula "c1", que és la versió completa (4 files). Llavors, mitjançant l'`ON`, es fa la combinació de "c1" i "c2", comparant totes les files d'una contra les de l'altra i obtenint només els casos on el valor de les comandes de "c1" és menor que el de "c2". En total, 4 comparacions, i dues són certes, per als clients 1 i 2.

2.2.4 Manipulació de dades

L'objectiu final de disposar de taules correctament creades és, en definitiva, poder desar-hi informació per consultar-la o modificar-la posteriorment. Sense dades no té sentit poder fer consultes. Si bé res no impedeix que una BD tingui un conjunt de dades estàtic que mai no varia en el temps, i per tant ja n'hi ha prou d'afegir-les tot just després de crear la taula, el més normal, en una aplicació, és que les dades d'una taula puguin anar variant al llarg del temps, a mesura que es va afegint o eliminant informació.

Per dur a terme aquesta tasca, SQL disposa del conjunt de sentències corresponent que es poden executar sobre la BD. Tot seguit, veurem les més importants amb més detall.

Inserció de dades

Mitjançant la sentència **INSERT INTO** es pot afegir una nova línia a una taula concreta. La sintaxi base és:

```
1 INSERT INTO nomTaula
2 VALUES (valor1, valor2,...)
```

Els termes "valor1", "valor2" es corresponen als valors de cadascuna de les columnes de la taula en qüestió, enumerats exactament en el mateix ordre en què s'han enumerat les pròpies columnes en la creació de la taula i usant el mateix tipus de dades, de manera semblant, per exemple, a com s'especifica una llista de paràmetres en la crida d'un mètode (en el mateix ordre que en la seva definició). Evidentment, el nombre de valors també ha de concordar exactament amb el nombre de columnes. Si no es compleix alguna d'aquestes condicions, hi haurà un error.

Els valors individuals s'expressen com literals, tot i que també es pot usar la paraula clau `NULL` per indicar que es vol emmagatzemar un valor nul, o `DEFAULT` si volem que la BD emmagatzemi un valor per defecte. Ara bé, per poder fer això, cal que no s'entri en conflicte amb els paràmetres usats en definir la columna durant la creació de la taula. Per tant, per al primer cas, no es poden usar valors nuls si s'ha usat el paràmetre `NOT NULL` o `PRIMARY KEY`, i per al segon cas, cal que hi hagi realment un valor per defecte definit per a la columna. En cas contrari, no es procedirà a la inserció de les dades.

Per exemple, si sobre la taula CLIENTS s'executés la següent comanda d'inserció, llavors passaria a tenir la informació següent:

Sentència:

```
1 INSERT INTO CLIENTS VALUES (5,'Client5', 'Adreça5','e_mail5@domini.com','
      +34933336677', 3)
```

Taula resultant:

TAULA 2.23. Resultat d'executar INSERT

ID	NOM	APOSTAL	AELECTRONICA	TELÈFON	NCOMANDES
1	Client1	Adreça1	e-mail1@domini.com	+34931112233	4
2	Client2	Adreça2	e-mail2@domini.com	+34932223344	1
3	Client3	Adreça3	e-mail3@domini.com	+34933334455	10
4	Client4	Adreça3	e-mail4@domini.com	+34933335566	7
5	Client5	Adreça5	e-mail5@domini.com	+34933336677	3

Eliminació de dades

Si mai es vol esborrar una fila, o un conjunt de files, cal usar la sentència **DELETE**. Aquesta usa el paràmetre `WHERE`, de manera idèntica a quan es fan consultes, però ara com a criteri d'esborrat. Per tant, aquelles files que en una sentència `SELECT` serien el resultat, en una sentència `DELETE` són les esborrades. Un cop més, la condició pot ser tan complexa com es desitgi i, fins i tot, es poden usar funcions SQL (esborrar la fila amb el valor més baix, la primera fila).

```
1 DELETE FROM nomTaula
2 WHERE condició
```

Per exemple, per eliminar de la taula original (amb 4 clients) tots els que viuen a l'adreça 3, es faria així:

Sentència:

```
1 DELETE FROM CLIENTS
2 WHERE APOSTAL='Adreça3'
```

Taula resultant:

TAULA 2.24. Resultat d'executar DELETE

ID	NOM	APOSTAL	AELECTRONICA	TELÈFON	NCOMANDES
1	Client1	Adreça1	email1@domini.com	+34931112233	4
2	Client2	Adreça2	email2@domini.com	+34932223344	1

Modificació de dades

Un cop es disposa ja de dades dins la taula, també es poden modificar directament, sense haver d'esborrar-les i afegir-les de nou. La sentència **UPDATE** permet canviar el valor d'una cel·la. La sintaxi és:

```
UPDATE taula
SET nomColumna1=valor1, nomColumna2=valor2,...
WHERE condició
```

> Per seleccionar només una fila, pot resultar molt útil usar la **clau primària** de la taula dins la condició.

Els paràmetres després de SET indiquen quins són els nous valors per a un conjunt de columnes, que poden ser totes o només una part, de manera que se seleccionen exactament les cel·les a canviar. La condició dins el WHERE serveix, novament, per delimitar quines files cal considerar de cara a l'actualització. Si més d'una fila compleix la condició, s'actualitzen totes amb els valors indicats. Per tant, cal anar amb una mica de compte en aquest cas i, si volem modificar només una cel·la, assegurar-se que la condició delimita exactament una única fila.

Per exemple, si volem canviar l'adreça del client 4 per una de nova, es podria fer així:

Sentència:

```
UPDATE CLIENTS
SET APOSTAL='novaAdreça'
WHERE ID='4'
```

Taula Resultant:

Taula 2.25. Resultat d'executar UPDATE

ID	NOM	APOSTAL	AELECTRONICA	TELÈFON	NCOMANDES
1	Client1	Adreça1	email1@domini.com	+34931112233	4
2	Client2	Adreça2	email2@domini.com	+34932223344	1
3	Client3	Adreça3	email3@domini.com	+34933334455	10
4	Client4	novaAdreça	email4@domini.com	+34933335566	7

Com en el cas de la inserció, es poden usar els valors especials DEFAULT o NULL. També és possible usar expressions com a nou valor d'actualització, en el cas de dades numèriques: sumes, restes... Per exemple, si des de l'adreça 3 s'han fet 5 comandes noves, es podria fer amb aquesta sentència:

Sentència:

```
UPDATE CLIENTS
SET NCOMANDES=NCOMANDES + 5
WHERE APOSTAL='Adreça3'
```

Taula Resultant:

TAULA 2.26. Resultat d'executar UPDATE

ID	NOM	APOSTAL	AELECTRONICA	TELÈFON	NCOMANDES
1	Client1	Adreça1	email1@domini.com	+34931112233	4
2	Client2	Adreça2	email2@domini.com	+34932223344	1
3	Client3	Adreça3	email3@domini.com	+34933334455	15
4	Client4	Adreça3	email4@domini.com	+34933335566	12

2.3 JDBC

Java proporciona l'API Java Database Connectivity (connectivitat Java a bases de dades) com a mecanisme per poder generar i invocar sentències SQL sobre un BD relacional mitjançant codi en programes Java. La seva particularitat és que, en contrast amb altres sistemes existents, ofereix una interfície comuna per a l'accés a qualsevol tipus de BD, independentment del fabricant. Per al desenvolupador, la BD real que hi ha al darrere és totalment transparent i obvia la necessitat d'efectuar cap mena de configuració en la màquina on s'executa l'aplicació que accedeix a les dades. Aquesta biblioteca es troba principalment en els paquets `java.sql` i `javax.sql`.

El controlador de BD Microsoft és `sun.jdbc.obdc.JdbcOdbcDriver`.

Partint de la suposició que ja hi ha una BD correctament configurada i a la qual volem accedir des del codi d'un programa Java, el resum de passos que cal fer dins l'aplicació és:

1. Importar correctament els *packages* corresponents.
2. Carregar el controlador (*driver*) per a l'accés a la BD. Aquest depèn de la BD a accedir.
3. Establir la connexió a la BD.
4. A partir d'aquí, ja es poden executar sentències SQL en la BD i processar les respostes.
5. Quan ja no es vol treballar més amb la BD, cal tancar la connexió.

JDBC, igual que moltes altres API en Java, està dissenyat amb la simplicitat en el pensament i intenta que l'ordre de les operacions que ha de fer l'operador sigui genèric i, fins a cert punt, lògic. Igual que per llegir dades d'un fitxer el que cal fer és dir quina és la seva ubicació, obrir-lo, llegir o escriure les dades i tancar-lo, en aquest cas la idea és similar. Simplement, "llegir-lo o escriure'l" vol dir invocar una sentència SQL, enlloc de posicionar un apuntador. Tot i així, cal tenir un cert domini d'SQL per poder fer correctament aquesta feina, és clar.

Tot seguit, es veurà amb més detall els passos més importants (es dóna per feta la importació correcta de *packages*), però només a títol d'introducció, per fer-se una idea del significat de cada pas. Es mostra un fragment de codi que consulta tots els clients d'una BD i en mostra el nom i l'adreça postal per pantalla. En aquest cas, totes les classes implicades en l'accés a una BD pertanyen al paquet `java.sql`.

```
//Importar classes
import java.sql.*;
...

//Carregar el controlador per la BD Apache Derby
Class.forName("org.apache.derby.jdbc.ClientDriver");

//Establir la connexió
String urlBaseDades = "jdbc:derby://localhost:1527/GestioEncarrecs";
String usuari = "administrador";
String contrasenya = "pswdificil";
Connection c = DriverManager.getConnection(urlBaseDades , usuari, contrasenya);

//Enviar una sentència SQL per recuperar els clients
Statement cerca = c.createStatement();
ResultSet r = cerca.executeQuery("SELECT * FROM CLIENTS");
while (r.next()) {
  System.out.println("Nom: " + r.getString("NOM") +  ", Adreça: " + r.getString
      ("APOSTAL"));
}

//Tancar la connexió
c.close();
```

2.3.1 Càrrega del controlador

Per permetre la independència de la plataforma, JDBC proporciona un gestor de controladors que gestiona dinàmicament tots els aspectes específics de l'accés a un tipus de BD concret. JDBC és qui s'encarrega de transformar totes les crides genèriques als accessos corresponents d'acord als mecanismes específics de la BD amb què s'interactua. Com a desenvolupadors, us podeu desentendre, fins a cert punt, de tots aquests detalls. Per tant, si es disposa de quatre tipus diferents de bases de dades, de diferents fabricants, per connectar-se caldrà disposar de quatre controladors diferents.

Els controladors sempre prenen la forma d'una classe Java, identificada de manera absoluta amb el nom complet del paquet que la conté i el nom de la classe en si. El seu registre de controladors es fa automàticament quan es carrega la classe a memòria, cosa que es fa amb la crida `Class.forName(nomControlador)`. Des del punt de vista del programador, no cal fer res més. Per exemple, per a l'accés a una BD Apache Derby, cal carregar el seu controlador específic, amb la crida següent:

```
Class.forName("org.apache.derby.jdbc.ClientDriver");
```

Atès que aquest controlador només serveix per a aquest tipus de BD, si s'intenta usar per connectar-se a una BD de qualsevol altre tipus (Oracle, PostgreSQL...), el programa no funcionarà.

El CLASSPATH indica on es troben les classes accessibles quan s'executa un programa Java.

Cal tenir en compte que el fet que des del punt de vista del desenvolupador la càrrega del controlador sigui senzilla no vol dir que el procés de disposar i configurar correctament un controlador sigui immediata. La classe que conté el controlador del SGBD serà diferent per a cada cas, per la qual cosa cal consultar la documentació del fabricant del programari per esbrinar quin és el nom de la classe en qüestió i saber com instal·lar-la al sistema. Tot i que per carregar el controlador no cal usar cap instrucció `import`, sí que cal que aquesta classe estigui inclosa en el `CLASSPATH` perquè es pugui localitzar correctament.

En usar **JDBC**, la instrucció "Class.forName" és l'única que canvia, d'acord al tipus concret de BD al qual s'accedeix. La resta d'instruccions del programa serà exactament igual, sempre que no s'usin extensions propietàries de l'SQL.

2.3.2 Establiment de la connexió

Un cop s'ha carregat correctament el controlador, l'aplicació es pot connectar remotament a algun servidor que conté la BD mitjançant la crida al mètode estàtic `getConnection` de la classe `DriverManager` que es descriu a continuació. Aquesta classe ofereix els serveis bàsics de gestió de controladors JDBC i és el punt d'accés a ells des dels vostres programes.

La definició d'aquest mètode és:

```
1 Connection getConnection(String url, String user, String psw) throws
      SQLException
```

Un URL és el que s'escriu en la barra d'adreces d'un navegador web per accedir a una pàgina web concreta.

El paràmetre "url" és una cadena de text amb l'identificador de la ubicació de la BD. Quan es configura una BD, aquesta normalment ens informa de quin és aquest identificador, que dependrà en part de la màquina on s'ha instal·lat.

Un **URL** (*uniform resource locator*, localitzador uniforme de recurs) és un apuntador a algun recurs o servei disponible a internet. Aquest recurs pot ser quelcom tan simple com un fitxer, o elements més complexos com objectes, un servidor web o, és clar, una BD accessible remotament.

Adreça IP

Es tracta d'un identificador únic per a cada màquina connectada a internet. Consta de 4 bytes i normalment es representa amb cada byte en notació decimal, separats per punts. Per exemple: 192.168.0.34

A nivell general, normalment, un URL es caracteritza perquè està dividida en fragments prou significatius:

- El protocol que cal usar per accedir al recurs. O sigui, quin serà el format de les dades que arribaran al servei, de manera que les pugui interpretar correctament.
- El nom de la màquina, o la seva l'adreça IP, de manera que s'identifica a l'equip on es troba disponible el recurs o el servei.

- El port del servei, un identificador únic assignat a tots els serveis que s'executen en la màquina, de manera que és possible dirigir peticions a un servei concret donada una màquina.
- Finalment, especifica el nom del recurs pròpiament dins la màquina.

Així, doncs, un URL és, per exemple:

```
http://ioc.xtec.cat:80/educacio/ioc-estudis
```

Indica que, mitjançant el protocol anomenat HTTP, el que usa la web, cal accedir al port 80, on se sol executar el servidor web, i obtenir el recurs "/educacio/ioc-estudis", que és una pàgina web.

En el cas d'un URL per accedir a una BD mitjançant JDBC, té la particularitat que la part del protocol es divideix en dos: el protocol principal, que és "jdbc" i el subprotocol. Sense entrar en detall, es tracta d'un identificador que especifica el nom del controlador o mecanisme de connectivitat a la BD que cal usar. Per exemple, per a un servei de BD Apache Derby, el subprotocol és "derby". Cal mirar la documentació de la BD. La resta de la URL és idèntica a qualsevol altra: nom de màquina, port on s'executa el servei de la BD i nom de la BD concreta a què es vol accedir.

Per accedir a la BD anomenada "gestioEncarrecs" a un servei de BD de tipus Apache Derby que s'executa al port 1527 de la màquina amb adreça IP 192.168.2.1, la URL seria la següent:

```
jdbc:derby://192.168.2.1:1527/GestioEncarrecs
```

L'identificador `localhost` es pot usar si la màquina a la qual s'accedeix és la mateixa on s'executa l'aplicació.

Sempre que es desplega una BD, normalment es poden configurar comptes d'usuari, protegides amb contrasenya, de manera que no es permet que qualsevol aplicació aliena pugui llegir alegrement les dades contingudes. Per tant, perquè el programa pugui accedir a la BD, si està correctament protegida, també caldrà disposar d'un nom d'usuari i d'una contrasenya correctes, que és el que representen els paràmetres "user" i "psw". Si no hi ha cap compte d'usuari o contrasenya habilitat, cosa poc recomanable, es pot posar un text buit ("") als dos paràmetres.

Si la connexió remota s'estableix de manera correcta (la BD realment existeix, el servei està en marxa, correctament configurat i no hi ha cap problema en la xarxa), es retorna una instància de la classe `Connection`, a partir de la qual es pot interactuar per dur a terme qualsevol acció amb la BD. Per tant, el codi per connectar-se a la BD amb la URL anterior, si s'ha configurat amb un compte d'usuari "administrador", amb la contrasenya "pswdificil", seria:

```
String urlBaseDades = "jdbc:derby://localhost:1527/GestioEncarrecs";
String usuari = "administrador";
String contrasenya = "pswdificil";
Connection c = DriverManager.getConnection(urlBaseDades , usuari, contrasenya);
```

Val la pena comentar que, tot i que a l'exemple els paràmetres de la crida estan definits com a cadenes de text al mateix codi de l'aplicació, en una aplicació real

aquesta seria una solució poc encertada, ja que si mai canvia la ubicació de la BD (la màquina o el port) o el seu nom, caldria modificar el programa i tornar a compilar el programa. En la realitat, aquesta informació hauria de ser configurable, de manera que es pugui canviar sense haver de tocar el codi del programa. Per exemple, en un fitxer de configuració o com a paràmetre d'execució.

2.3.3 Execució de sentències SQL

Totes les interaccions amb la BD pràcticament sempre s'efectuen mitjançant l'enviament i l'execució de sentències SQL i el processament de les respostes. Les sentències SQL prenen la forma de simples cadenes de text que podeu generar al vostre gust (`CREATE TABLE...`, `INSERT...`, `SELECT.....`), sempre que sigui amb la sintaxi correcta. En aquest aspecte, res no varia.

Un cop es disposa de la cadena de text amb la sentència que es vol invocar a la BD, el mecanisme ofert per JDBC per enviar-les a través d'una connexió establerta és mitjançant l'ús de la classe `Statement` (sentència). Si es tracta d'una operació de consulta de dades, JDBC processa la resposta de la BD i la presenta dins el programa com un objecte del tipus `ResultSet`, del qual podem extreure la informació associada a partir dels mètodes que proporciona aquesta classe. Ambdues es troben al *package* `java.sql`.

La classe Statement

Les instàncies de la classe `Statement` només es poden instanciar mitjançant la crida al mètode `createStatement`, de la classe `Connection`. No es pot invocar directament el constructor usant el `new` (de fet, aquesta classe és, en realitat, una interfície Java).

És possible assignar valors per defecte usant el mètode sense paràmetres `createStatement()`.

```
1 Statement createStatement(int tipus, int concurrencia)
2 throws SQLException
```

Els paràmetres d'entrada especifiquen quines seran les propietats de les respostes de l'execució de la sentència, només per al cas d'executar consultes (`SELECT`). Dins la classe `ResultSet` és on es poden trobar definides el seguit de constants estàtiques que s'accepten com a paràmetre. Entre les més destacades es troben:

TAULA 2.27. Tipus de paràmetres d'entrada

Paràmetre	Constant	Descripció
tipus	`ResultSet.TYPE_FORWARD_ONLY`	La resposta només es pot navegar unidireccionalment, només endavant. És el que normalment s'usa.
tipus	`ResultSet.TYPE_SCROLL_INSENSITIVE`	La resposta només es pot navegar endavant i endarrere.
tipus	`ResultSet.TYPE_SCROLL_SENSITIVE`	Com l'anterior i, a més a més, si hi ha canvis a la mateixa BD mentre s'usa la resposta, aquests es reflecteixen a les dades que conté.
concurrència	`ResultSet.CONCUR_READ_ONLY`	Les dades contingudes a la resposta no es poden modificar (mode de lectura, només).
concurrència	`ResultSet.CONCUR_UPDATABLE`	Les dades contingudes a la resposta es poden modificar (mode lectura-escriptura).

Els mètodes sobrecarregats permeten recuperar valors per número de columna, segons l'ordre que ocupi en la taula...

Cal tenir en compte, però, que no tots els controladors permeten totes aquestes opcions.

Un cop s'ha inicialitzat correctament l'objecte i ja es disposa de la cadena de text amb la sentència SQL, aquesta sentència es pot enviar a la BD invocant el mètode corresponent. En el cas d'una consulta de dades (`SELECT`), cal emprar el mètode `executeQuery`, mentre que en qualsevol altre cas (`INSERT`, `UPDATE`, `DELETE`), quan es fan modificacions al contingut de la mateixa BD, cal usar el mètode `executeUpdate`. Ambdós estan sobrecarregats, de manera que permeten concretar certes particularitats de l'execució de la sentència. La crida més senzilla és la que té com a paràmetre, simplement, la cadena de text amb la sentència SQL a enviar.

```
1 ResultSet executeQuery(String sql) throws SQLException
2 int executeUpdate(String sql) throws SQLException
```

Si no es produeix cap excepció, llavors això voldrà dir que la sentència s'ha executat correctament. Fixeu-vos que només en el primer cas, quan es fa una consulta, realment es retorna un conjunt de dades. El valor de retorn d'una modificació sol ser 0, o el nombre de files que han estat manipulades per la sentència.

Per exemple, per consultar tots els clients que viuen a l'adreça 3 d'una taula, es faria:

```
1 Connection c = ...
2
3 Statement s = c.createStatement(ResultSet.TYPE_FORWARD_ONLY, ResultSet.
      CONCUR_READ_ONLY);
4 String sentencia= "SELECT * FROM CLIENTS WHERE APOSTAL='Adreça3'";
5 ResultSet res = c.executeQuery(sentencia);
```

D'altra banda, si es vol afegir un nou client, llavors caldria fer:

```
Connection c = ...

Statement s = c.createStatement(ResultSet.TYPE_FORWARD_ONLY, ResultSet.
    CONCUR_READ_ONLY);
String sentencia= "INSERT INTO CLIENTS VALUES (5,'Client5',
                  'Adreça5','e_mail5@domini.com','+34933336677', 3)";
int res = c.executeUpdate(sentencia);
```

Estrictament, la dificultat en l'execució de la sentència mitjançant la classe `Statement` recau íntegrament en la vostra habilitat per crear sentències SQL sintàcticament i semànticament correctes. Com passava en crear la connexió, recordeu que les sentències no han de ser necessàriament cadenes de text estàtiques definides dins el codi (segurament, poques vegades ho siguin), també poden dependre d'altres variables o d'altres paràmetres. Per exemple:

```
private static int LAST_ID = 0;

private void crearClient(String nom, String ad, String, String ml, String tlf,
    int nc)
throws SQLException {

  Connection c = ...
  ...

  Statement s = c.createStatement(ResultSet.TYPE_FORWARD_ONLY, ResultSet.
      CONCUR_READ_ONLY);
  LAST_ID++;
  String sentencia= "INSERT INTO CLIENTS VALUES (" + LAST_ID + ",'" +
                    nom + "','" +
                    ad + "','" +
                    ml + "','" +
                    tlf + "'," +
                    nc + ")";
  int res = c.executeUpdate(sentencia);
  ...
}
```

La classe PreparedStatement

Mitjançant la classe `Statement` es pot executar qualsevol sentència SQL sense límit. Ara bé, molt sovint, en un programa que accedeix a una BDD, l'usuari no disposarà de la possibilitat de fer qualsevol consulta que es pugui imaginar, sinó que es trobarà limitat a les funcionalitats que proporciona la seva interfície d'usuari. Així doncs, per exemple, en un programa que gestiona encàrrecs de clients usant una interfície gràfica, hi haurà un seguit de menús o botons que enumeraran un seguit d'operacions finites i molt concretes que es poden dur a terme: llistar noms de clients ordenats alfabèticament, cercar els encàrrecs pendents d'un client... Mitjançant sentències SQL es poden fer consultes ben complicades, però el programa ja té dins el seu codi un conjunt que són les úniques que s'usaran mai. L'usuari normalment no haurà de treballar directament amb el llenguatge SQL (SELECT, INSERT...), només indicar certs paràmetres molt concrets de la consulta (el nom del client a cercar, les dades d'un client que es volen llistar).

La classe `PreparedStatement` és una subclasse de `Statement` que permet executar sentències parametritzades, de manera que facilita aquest tipus de compor-

tament dins un programa. La seva particularitat principal és l'eficiència superior en relació amb `Statement`, ja que en el moment de definir-la es precompila, de manera que estalvia feina a la BDD. Per tant, és preferible usar-la per a sentències senzilles que depenen de paràmetres molt concrets aportats per l'usuari, i que cal usar repetides vegades, només canviant aquests paràmetres, al llarg de l'execució del programa.

Per instanciar un objecte `PreparedStatement`, també cal cridar un mètode estàtic de la classe `Connection` (n'hi ha diversos, en estar sobrecarregat). Ara bé, en aquest cas, hi ha un paràmetre addicional, que és la sentència SQL parametritzada. En aquesta sentència, el text no està complet, sinó que s'escriu un interrogant ('?') a cada lloc on es vol ubicar un paràmetre. Per exemple, suposem que un programa vol usar aquesta classe per dur a terme la funció de cercar les dades d'un client donat el seu nom.

```
1 Connection c =...
2
3 //Ara el text de la sentència es posa en crear-la, no en executar-la
4 String sentencia= "SELECT * FROM CLIENTS WHERE NOM=?";
5 PreparedStatement s = c.preparedStatement(sentencia);
```

Alerta, els valors dels índexs dels paràmetres van de 1..N, no de 0..N com en altres classes que gestionen llistes.

Atès que la sentència parametritzada no és vàlida en si mateixa, abans de poder-la executar cal assignar valors als seus paràmetres, un per a cada interrogant dins el seu text. Internament, l'objecte `PreparedStatement` organitza els paràmetres ordenant-los amb un índex de 1 a N, de manera que el primer '?' és el paràmetre 1, el segon és el 2... Mitjançant un seguit de mètodes, permet assignar valors a cadascun dels índexs. Hi ha un mètode per a cada tipus de dades. Cal invocar-ne tants com paràmetres (nombre d'interrogants) hi ha al text de la sentència.

- `setString(int parameterIndex, String x)`
- `setInt(int parameterIndex, int x)`
- `setDouble(int parameterIndex, double x)`
- etc.

Evidentment, cal ser molt acurat i usar el mètode que correspon al tipus de dades d'acord a la definició de la BDD, de manera que la sentència final sigui correcta. Per exemple, si s'està fent una cerca per nom de client, en l'exemple anterior caldrà usar el mètode `setString` per assignar correctament el paràmetre, ja que la columna NOM és de tipus VARCHAR (una cadena de text). Un cop l'objecte està correctament inicialitzat amb els valors dels paràmetres adients, de manera que cap queda sense un valor assignat, es pot executar usant `executeQuery` o `executeUpdate`, com amb `Statement`. Aquest cop, però no cal definir cap sentència en cridar aquests mètodes.

```
1 //Es pregunta el nom a la interfície d'usuari
2 String nom = ...
3
4 s.setString(1, nom);
5 //Si "nom" és "Client1", ara la sentència equival a:
6 //"SELECT * FROM CLIENTS WHERE NOM='Client1'"
7
8 ResultSet res = s.executeQuery();
```

La classe ResultSet

La instància de `ResultSet` retornada per una crida `executeQuery` conté la llista de files resultant de la consulta a la BD. Si la consulta no ha obtingut cap resultat, la instància de `ResultSet` estarà buida, però mai no es donarà el cas que retorni una referència a `null`. Si hi ha cap error (la sentència SQL executada no era correcta), es produirà una excepció.

`ResultSet` ofereix els mètodes necessaris per navegar per la llista i accedir als valors emmagatzemats. Aquesta navegació sempre és fila per fila, però el mode d'accés variarà segons els paràmetres emprats en instanciar l'objecte `Statement` associat: unidireccional, de manera semblant a com ho faria un `Iterator`, o bidireccional. En qualsevol dels casos, l'objecte `ResultSet` disposa d'un apuntador intern on recorda quina és la posició actual. Mitjançant la invocació de certs mètodes (`next()` o `previous()`), es pot fer avançar o recular l'apuntador. Ambdós s'avaluen a `true` si la nova posició de l'apuntador després de desplaçar-se conté una fila vàlida, o `false` si ja ens hem passat de la llista, tant sigui per l'inici com o pel final.

En el cas d'un `ResultSet` unidireccional, un cop s'arriba al final, ja no es pot accedir més a les dades. Cal tornar a executar una consulta.

Aquest sistema té la particularitat que, a l'inici de tot, l'apuntador està una posició per endavant de la primera fila, de manera que la primera crida a `next()` sempre es posiciona a la primera fila. Per tant, si s'intenta obtenir cap dada des del `ResultSet`, només obtenir-la, sense posicionar-se almenys en alguna fila correcta, hi haurà un error.

Un cop posicionats en la fila que es vol llegir, és possible consultar el valor de les cel·les per a cada columna definida en la taula, usant el nom de la columna. Per fer-ho, cal cridar el mètode `getXXX` adequat segons el tipus de dades de la columna a consultar, usant com a paràmetre el nom de la mateixa columna. En escollir quin mètode usar per obtenir les dades d'una columna concreta, cal tenir en compte les equivalències entre tipus de dades SQL i Java. Per exemple, si la dada a la BD és de tipus `INTEGER`, no podeu usar un mètode per obtenir una cadena de text (`getString`), i viceversa. Si el valor emmagatzemat en la columna no es correspon amb el tipus de dades del mètode cridat, es produirà un error. Dins el `ResultSet` totes les dades ja s'hauran convertit del tipus original SQL a la BD a tipus tractables en el codi Java, per la qual cosa no cal fer cap conversió concreta, només ser conscients de les equivalències.

Existeix un mètode per a cada tipus de dades, per exemple:

- `String getString (String nomColumna)`
- `int getInt (String nomColumna)`

- `short getShort (String nomColumna)`
- `double getDouble (String nomColumna)`
- `java.sql.Date getDate(String nomColumna)`
- etc.

Cal anar amb compte, però, amb la possibilitat que el valor que hi ha a la cel·la de la BD sigui `NULL` i, per tant, invàlid. En el cas dels mètodes que retornen objectes (`String`, `Date`), aquests senzillament retornaran una referència a `null`, en aquest cas. Ara bé, en el cas de mètodes que retornen un tipus primtiu, el resultat serà 0 (o `false` per als booleans). En aquest cas, com que no és possible a simple vista saber si el 0 en qüestió és perquè el valor és realment un 0 o `NULL`, cal usar el mètode auxiliar `wasNull()`, immediatament després de la crida a `getXXX`, el qual ens dirà si el valor retornat a la darrera consulta era un valor `NULL` a la BD o no.

Si es vol llistar per pantalla la llista dels noms de tots els clients de la BD, es pot fer així:

```
1  Connection c = ...
2
3  Statement s = c.createStatement(ResultSet.TYPE_FORWARD_ONLY, ResultSet.
       CONCUR_READ_ONLY);
4  String sentencia= "SELECT NOM FROM CLIENTS";
5  ResultSet res = c.executeQuery(sentencia);
6
7  while (res.next() ) {
8    String nom = res.getString("NOM");
9    if (!res.wasNull())
10     System.out.println(nom);
11 }
```

Aquest codi faria exactament el mateix si, en lloc de consultar només els noms, a la sentència SQL es consultessin totes les dades (`SELECT *`), o qualsevol subconjunt de dades que inclogués el nom (`SELECT NOM, APOSTAL...`). Atès que la crida a `getString` especifica la columna que vol extreure del resultat, s'obtindrien sempre les mateixes dades per pantalla. Ara bé, recordeu que, si es vol fer un codi eficient i no carregar la BD innecessàriament, cal escriure sentències SQL que es limitin a consultar estrictament les dades que necessiteu i cap altra.

Finalment, val la pena comentar que encara que un `ResultSet` només sigui navegable unidireccionalment, per exemple perquè no queda més remei ja que la BDD no suporta l'execució de consultes que retornin navegació bidireccional, això no vol dir que us hàgiu de resignar i haver de recórrer diverses vegades les dades per fer diferents operacions amb les mateixes dades dins un programa. Recordeu que res no impedeix agafar els valors i desar-los en estructures Java que sí que permetin gestionar la informació de manera més còmoda, com ara Lists, Sets o Maps, i, a partir d'aquí, treballar amb elles.

```
Connection c = ...

Statement s = c.createStatement(ResultSet.TYPE_FORWARD_ONLY, ResultSet.
    CONCUR_READ_ONLY);
String sentencia= "SELECT NOM FROM CLIENTS";
ResultSet res = c.executeQuery(sentencia);

List<String> llistaNoms = new ArrayList<String>();

while (res.next() ) {
  String nom = res.getString("NOM");
  if (!res.wasNull())
    llistaNoms. add(nom);
}

//A "llistaNoms" tenim tots els noms resultants de la consulta
```

Modificacions a les dades via ResultSet

Si a l'hora d'instanciar l'objecte `Statement` s'ha definit que el `ResultSet` retornat és de lectura-escriptura (paràmetre tipus `ResultSet.CONCUR_UPDATABLE`), llavors també és possible modificar el contingut de les files que conté, de manera que la manipulació es trasllada a la BD. És una manera alternativa de fer actualitzacions a la BD des d'una perspectiva més d'orientació a objectes, manipulant instàncies, en lloc d'usar sentències de text SQL.

Aquest sistema és possible ja que la generació d'un `ResultSet` no es basa a executar la resposta a la BD, obtenir absolutament totes les dades corresponents, encapsular-les, retornar-les al codi Java via JDBC, i oblidar-nos de la BD. En realitat, internament, un `ResultSet` és una associació viva amb la BD, que va obtenint les dades a poc a poc des de la BD a través de la connexió, a mesura que es van consultant les files de la resposta. Això permet que, igual que es pot llegir, també es pugui escriure a través d'aquesta associació. O que, si hi ha canvis a la BD, surtin immediatament també al `ResultSet` sense haver d'esperar a fer una altra consulta (si s'ha usat el paràmetre `ResultSet.TYPE_SCROLL_SENSITIVE`).

Com amb una simple consulta, cal posicionar-se a la fila a modificar. En aquest cas, els mètodes d'escriptura s'anomenen `updateXXX`, de manera anàloga als de lectura. En termes generals, tot el que s'aplica als mètodes de lectura, com el tractament de tipus de dades Java-SQL, s'aplica a aquests.

- `void updateString (String nomColumna, String s)`
- `void updateInt (String nomColumna, int i)`
- `short updateShort (String nomColumna, short s)`
- `double updateDouble (String nomColumna, double d)`
- `void updateDate(String nomColumna, java.sql.Date d)`
- ...

No obstant això, si es vol escriure un valor NULL, el que cal usar és el mètode updateNull(String nomColumna), independentment del tipus de dades desat a la columna.

Ara bé, un cop feta la modificació al ResultSet, en ser un conjunt de dades locals a l'aplicació, cal avisar-lo que forci una actualització sobre la BD. Això es fa amb el mètode updateRow(). Aquest mètode actualitza la informació només de la fila on es troba actualment l'apuntador i cap altra.

Per exemple, el codi següent passa a majúscules els noms de tots els clients de la BD, aprofitant els mètodes que ofereix Java per manipular cadenes de text. Fixeu-vos que ara el segon paràmetre de creació de l'Statement és ResultSet.CONCUR_UPDATABLE.

```
Connection c = ...

Statement s = c.createStatement(ResultSet.TYPE_FORWARD_ONLY, ResultSet.
    CONCUR_UPDATABLE);
String sentencia= "SELECT NOM FROM CLIENTS";
ResultSet res = c.executeQuery(sentencia);

while (res.next() ) {
  String nom = res.getString("NOM");
  String nouNom = nom.toUpperCase();
  res.updateString("NOM", nouNom);
  res.updateRow();
}
```

Un cop més, cal ser curós de no consultar més dades a la BD de les imprescindibles per fer la tasca encomanada (no fer SELECT * si realment no hem de treballar amb totes les dades al final).

A part de modificar dades, també és possible afegir i eliminar files. Per al primer cas, s'usa primer el mètode moveToInsertRow(), per indicar que la posició actual es considera ara una fila nova, una successió de crides a updateXXX per posar les dades a cada columna, i finalment el mètode insertRow(). Per eliminar la fila actual, només cal invocar el mètode deleteRow().

```
res.moveToInsertRow();
  res.updateInteger("ID", 5);
  res.updateString("NOM", "Client5");
  res.updateString("APOSTAL","Adreça5");
  res.updateString("AELECTRONICA","e_mail5@domini.com");
  res.updateString("TELÈFON","+34933336677");
  res.updateInteger("NCOMANDES",3);
  res.inserRow();
```

2.3.4 Tancament de la connexió

El manteniment d'una connexió oberta a una BD és una tasca costosa per al sistema, tant pel vostre programa com pel servei de BD a l'altre extrem de la connexió; per tant, quan ja no s'ha d'usar més, cal tancar-la, de la mateixa manera que cal tancar un flux després d'haver-ne llegit tota la informació. El tancament s'efectua, simplement, cridant sobre la connexió (l'objecte Connection) el mètode close:

```
Connection c = ...
...
c.close();
```

En qualsevol cas, si un se n'oblida, la connexió no es manté oberta indefinidament, ja que quan el recol·lector de memòria de Java elimini l'objecte `Connection` generat també s'encarregarà de tancar la connexió. De totes maneres, aquest mecanisme és només un últim recurs que proporciona Java en cas que us hàgiu despistat. Cal que sempre tanqueu vosaltres la connexió al codi.

Atès que el procés de creació d'una connexió també és costós, les aplicacions que fan ús de l'accés a una BD normalment obren una connexió en iniciar-se i no la tanquen fins tot just abans d'acabar-ne l'execució. De totes maneres, si es preveu que l'aplicació no farà ús de la connexió en un interval llarg de temps, val la pena tancar-la i tornar-la a obrir quan realment torni a fer falta, ja que si el servei de la BD té limitat el nombre de connexions, pot ser que el nostre programa estigui impedint que un altre s'hi connecti, tot i no estar accedint realment a la BD.

2.3.5 Exemple d'aplicació JDBC: El gestor d'encàrrecs

Tot just es mostra el codi d'un exemple d'aplicació que usa JDBC per gestionar informació a una BD. Concretament, aquest programa gestiona objectes de tipus `Client` de manera que les seves dades es troben a una BD a través de la qual es poden fer cerques o afegir-ne. Es composa de tres classes:

- `Client`, que defineix quina informació conté un client.
- `GestorBD`, que conté els mètodes associats a l'accés a la base de dades.
- `GestorEncarrecs`, que és la classe principal i ofereix la interfície d'usuari (per consola).

Abans de poder-la executar correctament, es clar, s'ha d'haver configurat correctament una BD d'acord al codi de connexió a `GestorBD`.

```
//Fitxer Client.java
public class Client {
  private int id;
  private String nom;
  private String apostal;
  private String aelectronica;
  private String telefon;

  public Client(int i, String n, String ap, String ae, String t){
    id = i;
    nom = n;
    apostal = ap;
    aelectronica = ae;
    telefon = t;
  }

  public int getId() { return id; }
```

```
  public String getNom() { return nom; }
  public String getAPostal() { return apostal; }
  public String getAElectronica() { return aelectronica; }
  public String getTelefon() { return telefon; }

  @Override
  public String toString() {
    return id + "\t" + nom + "\t" + apostal + "\t" + aelectronica + "\t" +
        telefon ;
  }
}
```

```
import java.util.*;
import java.sql.*;

//Aquesta classe fa el mecanisme de persistència independent de la GUI.
public class GestorBD {
  Connection conn;

  public GestorBD() throws Exception {
    Class.forName("org.apache.derby.jdbc.ClientDriver");
    conn = DriverManager.getConnection("jdbc:derby://localhost:1527/
        GestioEncarrecs", "administrador", "pswdificil");
  }

  public void tancar() throws Exception {
    conn.close();
  }

  public int obtenirNouIDClient() throws Exception {
    //Cercar ID maxim
    Statement cercaMaxId = conn.createStatement();
    ResultSet r = cercaMaxId.executeQuery("SELECT MAX(ID) FROM CLIENTS");
    if (r.next()) return (1 + r.getInt(1));
    else return 1;
  }

  public List<Client> cercarClient(String nom) throws Exception {
    Statement cerca = conn.createStatement();
    ResultSet r = cerca.executeQuery("SELECT * FROM CLIENTS WHERE NOM='" + nom
        +  "'");
    LinkedList<Client> llista = new LinkedList<Client>();
    while (r.next()) {
      llista.add(new Client(r.getInt("ID"),r.getString("NOM"),r.getString("
          APOSTAL"), r.getString("AELECTRONICA"),r.getString("TELEFON")));
    }
    return llista;
  }

  public void afegirClient(Client c) throws Exception {
    Statement update = conn.createStatement();
    String valors = c.getId() + ",'" + c.getNom() + "','" + c.getAPostal() +  "
        ','" + c.getAElectronica() + "','" + c.getTelefon() + "'";
    update.executeUpdate("INSERT INTO CLIENTS VALUES(" + valors + ")");
  }
}
```

```
//Fitxer GestorEncarrecs.java

import java.io.*;
import java.util.*;

//Classe Principal
public class GestorEncarrecs {

  GestorBD gestor;
  BufferedReader entrada;

```

```
12  public static void main(String[] args) throws Exception {
13    GestorEncarrecs gbd = new GestorEncarrecs();
14    gbd.start();
15  }
16
17  public GestorEncarrecs() throws Exception{
18    gestor = new GestorBD();
19    entrada = new BufferedReader(new InputStreamReader(System.in));
20  }
21
22  public void start() throws Exception {
23    int opcio;
24    while (0 != (opcio = menuPrincipal())) {
25      try {
26        switch (opcio) {
27          case 1:
28            cercarClient();
29            break;
30          case 2:
31            afegirClient();
32            break;
33          default: mostrarDades("Opció incorrecta\n");
34        }
35      } catch (Exception ex) {
36        mostrarDades("S'ha produït un error: " + ex + "\n");
37      }
38    }
39    gestor.tancar();
40  }
41
42  private int menuPrincipal() throws Exception {
43    String menu = "\nQuina acció vols realitzar?\n" + "[1] Cercar client\n"+  "
          [2] Afegir client\n" +  "[0] Sortir\n" + "Opció>";
44    String lin = entrarDades(menu);
45    try { int opcio = Integer.parseInt(lin); return opcio; }
46    catch (Exception ex) { return -1; }
47  }
48
49  //Amb els metodes entrarDades i mostrarDades, fem el codi independent
50  //de la interficie. Si mai es fan canvis, nomes cal canviar aquests
51  //dos metodes.
52
53  private String entrarDades(String pregunta) throws IOException {
54    mostrarDades(pregunta);
55    return entrarDades();
56  }
57
58  private String entrarDades() throws IOException {
59    String linia = entrada.readLine();
60    if ("".equals(linia)) return null;
61    return linia;
62  }
63
64  private void mostrarDades(String dades) throws IOException {
65    System.out.print(dades);
66  }
67
68  //Cercar un element d'acord al seu nom
69  private void cercarClient() throws Exception {
70    String nom = entrarDades("Introdueix el nom del client: "); if (null == nom
          ) return;
71    List<Client> llista = gestor.cercarClient(nom);
72    Iterator it = llista.iterator();
73    mostrarDades("Els clients trobats amb aquest nom son:\n
          ----------\n");
74    while (it.hasNext()) {
75      Client c = (Client)it.next();
76      mostrarDades(c.toString() + "\n");
77    }
78  }
```

```
79
80  //Afegeix un nou client
81  public void afegirClient() throws Exception {
82     mostrarDades("Introdueix les seguents dades del nou client (deixa en blanc
           per sortir).\n");
83     String nom = entrarDades("Nom: "); if (null == nom) return;
84     String apostal = entrarDades("Adreça postal: "); if (null == apostal)
           return;
85     String aelectronica = entrarDades("E-mail: "); if (null == aelectronica)
           return;
86     String telefon = entrarDades("Telefon: "); if (null == telefon) return;
87     int id = gestor.obtenirNouIDClient();
88     gestor.afegirClient(new Client(id,nom,apostal,aelectronica,telefon));
89     mostrarDades("Operació completada satisfactòriament.\n");
90   }
91 }
```

2.4 Seguretat en l'accés a la BD

Una BD que s'executa en una màquina és una porta d'accés a totes les dades emmagatzemades, les quals moltes vegades són confidencials (noms, adreces, etc.). Protegir aquestes dades, de manera que només les persones realment autoritzades les puguin llegir, és una tasca molt important, no tan sols de l'administrador del SGBD (per exemple, configurant un nom d'usuari i contrasenya prou segurs), sinó del mateix desenvolupador d'aplicacions. Si una aplicació que accedeix a una BD no es codifica correctament, pot succeir que persones no autoritzades poden arribar a accedir a les dades aprofitant males pràctiques en el codi. Per tant, quan es treballa amb una BD és molt important programar de manera segura.

> Mitjançant la disciplina de la **programació segura**, es pot donar un cert nivell de garantia que una aplicació es comportarà sempre de la manera esperada i un usuari amb males intencions no serà capaç de comprometre el sistema aprofitant errors en el codi.

Aquesta disciplina és molt extensa, però no es pot finalitzar aquest capítol sense mostrar algun exemple significatiu dels aspectes que cal tenir molt en compte en crear una aplicació que accedeix a una BD.

Un dels punts més importants en desenvolupar aplicacions d'aquest tipus és que, quan s'executen sentències SQL, en cap concepte aquestes han d'incloure directament cap tipus d'entrada de l'usuari. Totes les entrades s'han de validar prèviament i s'ha de comprovar si tenen el format esperat.

Suposem el fragment de codi següent, que permet fer cerques sobre la taula donats noms coneguts de clients:

```
...
String nom = entrarDades("Introdueix el nom del client");
Statement cerca = c.createStatement();
ResultSet r = cerca.executeQuery(
"SELECT * FROM CLIENTS WHERE NOM='" + nom +  "'");
System.out.println("Els clients trobats son:");
while (r.next()) {
...
```

Si l'usuari introdueix la cadena "Client1", s'executa la sentència SQL:

```
SELECT * FROM CLIENTS WHERE NOM='Client1'
```

De manera que, en fer l'accés a la BD, l'aplicació retorna una sortida de l'estil:

Els clients trobats són:

Taula 2.28.

Client1	Adreça1	email1@domini.com	+34931112233	4

Si hi ha més d'un client amb el nom "Client1", es llisten diverses files de la taula. Fins aquí, tot correcte. També, d'acord amb aquest fragment de codi, un usuari no pot accedir a les dades d'un client si no en coneix el nom, ni molt menys llistar tota la taula de cop. Sempre es podrien provar totes les combinacions possibles de noms, però intuïtivament ja es veu que aquesta via d'acció no és realment factible. Fins i tot en cas que es vagin provant noms a l'atzar, mai no es pot estar segur de tenir realment tot el contingut de tota la taula.

Ara bé, en aquest codi es comet el gegantí error d'utilitzar directament l'entrada de l'usuari per formar una sentència SQL de consulta (`SELECT`). Suposem que com a nom d'usuari s'introdueix la cadena de text següent: "Client1' OR '1'='1".

Atès que el codi traspassa directament l'entrada de l'usuari a la sentència SQL mitjançant una concatenació de cadenes de text, llavors la sentència que realment s'executaria seria:

```
SELECT * FROM CLIENTS WHERE NOM='Client1' OR '1'='1'
```

Ara l'expressió que s'executa és una mica estranya i inesperada, però és el resultat d'aplicar exactament el procés que s'ha programat. En aquest cas, com '1'='1'avalua cert i una expressió OR amb una de les seves entrades a cert sempre retorna també cert, això equival a fer:

```
SELECT * FROM CLIENTS
```

Per tant, s'acaba de llistar tota la taula de clients de manera inesperada a causa de la manca de comprovació del format de l'entrada de l'usuari abans de traspassar-la a una sentència SQL. Això és un error molt greu per part del desenvolupador. L'administrador del SGBD no hi pot fer res que protegeixi la màquina on es troba la BD contra aquest error, ja que es tracta d'un error de programació, no de configuració de la BD.

Malauradament, aquesta vulnerabilitat és molt comuna (entre les cinc primeres del rànquing mundial), especialment en aplicacions basades en formularis web. Es coneix com a **SQL-Injection**.

Una eina útil per fer comprovacions en el format de cadenes de text d'entrada són les expressions regulars.

Per evitar aquest problema, cal que el codi dels vostres programes processi qualsevol cadena de text que depèn d'una entrada de l'usuari abans de permetre que formi part del text d'una sentencia SQL. Per exemple, es pot veure si les dades introduïdes tenen el format esperat. No hi ha gaires telèfons o adreces de correu electrònic amb apòstrofs o símbols d'igualtat. Una altra opció és marcar els apòstrofs que hi ha dins una entrada de dades de l'usuari de manera que la BD no els interpreti com part de la sintaxi SQL, sinó com un caràcter qualsevol estrictament. Cada sistema de BD ofereix la seva manera d'escapar caràcters i diferenciar entre un caràcter especial d'SQL o un que són dades estrictament.

Una altra opció és usar sentències parametritzades, `PreparedStatement`, ja que aquestes són immunes a aquest atac.

3. Aplicacions amb BD orientades a objectes

Si bé les BD relacionals són les més populars i les que tenen més acceptació, la seva utilització dins una aplicació orientada a objectes implica un procés de traducció del diagrama UML original a un model relacional, totalment basat en taules. En aquesta traducció es perden moltes de les funcionalitats bàsiques de l'orientació a objectes, que s'han de simular d'alguna manera: referències a objectes, classes associatives, llistes d'objectes, herència, etc. Quan el diagrama és de certa complexitat, la traducció pot esdevenir molt complicada.

Per resoldre aquest problema hi ha les BD orientades a objectes (BDOO). Aquestes, en lloc d'organitzar les dades en taules, les organitzen exactament tal com ho fa un diagrama UML, mitjançant la definició del conjunt de classes i relacions entre elles. Per tant, no cal fer cap traducció.

Per evitar confusions, fem servir el terme *BDR* per referir-nos explícitament a una BD relacional i el terme *BDOO* per referir-nos a una BD orientada a objectes, de manera que ambdós quedin diferenciats.

El Java té una especificació per a BDOO anomenada JDO (*Java data objects*).

Actualment, l'aplicació de BDOO es limita a àmbits molt concrets, especialment els vinculats a àrees científiques. La seva implantació en aplicacions comercials d'àmbit general és molt baixa. Un dels problemes principals de les BDOO és que els fabricants tendeixen a crear solucions incompatibles, que no obeeixen cap especificació concreta. Al contrari que en el cas de les BDR, és molt possible que una aplicació client feta pel producte d'un fabricant concret no funcioni sobre una BDOO d'un altre fabricant. De fet, a les BDOO que suporten Java no s'accedeix mitjançant JDBC, ja que aquest mecanisme és específic per a BDR, sinó que normalment s'hi accedeix usant biblioteques específiques per a cada fabricant.

3.1 Els llenguatges ODL i OQL

De la mateixa manera que hi ha l'SQL com a llenguatge estàndard per accedir a una BDR independentment del fabricant, hi ha un llenguatge per accedir a les dades d'una BDOO: el llenguatge de consultes a objectes (*object query language*, OQL). Addicionalment, hi ha el llenguatge de descripció d'objectes (*object description language*, ODL), que serveix per especificar el format d'una BDOO: quina mena d'objectes pot contenir i les seves relacions. Malauradament, si bé aquests llenguatges estan especificats, i com ja s'ha dit, no es pot comptar amb el fet que qualsevol fabricant realment els suporti. De cap manera arriben al grau d'acceptació de l'SQL.

3.1.1 El llenguatge ODL

El llenguatge ODL s'utilitza per definir classes d'objectes persistents dins una BDOO, de manera que els seus objectes es puguin emmagatzemar. Dins la declaració de cada classe s'inclou:

- El nom de la classe.
- Declaracions opcionals de claus primàries.
- La declaració de l'extensió: el nom del conjunt d'instàncies existents.
- Declaracions d'elements: atributs, relacions o mètodes.

La sintaxi és la següent (entre claudàtors s'indiquen camps opcionals):

```
class nomClasse [(key nomAtribut)] {
attribute tipusAtribut nomAtribut;
...
relationship tipus<nomClasseDestinació> nomRelacio;
...
tipusRetorn nomMetode(params) [raises (tipusExcepcio)]
...
}
```

Com es pot apreciar, simplement és un canvi de sintaxi respecte al llenguatge Java pròpiament, però la majoria d'elements d'una classe són clarament identificables.

L'única diferència és la declaració explícita de les relacions en forma de la paraula clau relationship, en contrast en Java, que es tradueixen a atributs. Hi ha diferents tipus de relacions segons la cardinalitat que es vol expressar. De fet cadascun d'aquests tipus té una certa correspondència amb les classes que s'usen en Java per implementar relacions. Per a cardinalitat 1, és suficient de posar el nom de la classe destinació. En cas de cardinalitat múltiple, es pot triar entre diferents tipus:

- `<nomClasseDestinació>`, si la relació és només a un únic objecte.
- `Set<nomClasseDestinació>`, un conjunt no ordenat sense repeticions.
- `Bag<nomClasseDestinació>`, un conjunt no ordenat amb repeticions.
- `List<nomClasseDestinació>`, un conjunt ordenat amb repeticions, amb insercions eficients.
- `Array<nomClasseDestinació>`, un conjunt ordenat amb repeticions.

El seu significat és el mateix que el de les classes homònimes del Java (capacitat d'haver-hi repeticions d'elements, ordenades o no per índex, etc.). Normalment, la més usada és `Set<nomClasseDestinació>`.

Si retornem a un model orientat a objectes, amb referències, no cal l'atribut "id".

Tot seguit es mostra com es podria representar dues classes interdependents anomenades `Client` i `Encarrec`, que emmagatzemen dades a una aplicació de gestió de clients, mitjançant ODL.

```
class Client (key id) {
 attribute int id;
 attribute String nom;
 attribute String adreçaPostal;
 attribute String adreçaMail;
 attribute String telefon;

 relationship Set<Encarrec> encarrecs;

 String getId();
 ...
}

class Encarrec (key id)  {
 attribute int id;
 attribute Date data;
 ...
}
```

L'herència entre classes s'inicia en la seva declaració mitjançant la paraula clau `extends` seguit del nom de la superclasse:

```
class nomClasse extends nomSuperClasse {
  ...
}
```

3.1.2 El llenguatge OQL

El llenguatge OQL es limita a permetre consultes sobre una BDOO. El seu operador principal és `SELECT`, el qual té una gran similitud amb l'equivalent SQL. Tot i així, té algunes particularitats degudes a la manera com s'estructuren les dades mitjançant l'orientació a objectes (per exemple, no hi ha taules, és clar).

La sintaxi general és:

```
SELECT valor1,valor2,...
FROM llista de col.lecions i noms per membres típics
WHERE condició
```

Atès que ara ja no hi ha taules, cal tenir en compte dues coses. D'una banda, la llista de col·leccions especificada en l'apartat `FROM` correspon a alguna de les classes declarades. Juntament al nom d'aquesta classe s'especifica la variable que s'usarà en els termes `SELECT` i `WHERE` per referenciar valors. D'altra banda, els valors que es volen consultar o comparar són atributs de classes, pel que la manera de referir-s'hi és mitjançant la nomenclatura: `nomClasse.nomAtribut`. Aquesta possibilitat també es pot usar per indicar-ne les relacions.

Per exemple, a una aplicació de gestió de clients, si es volen consultar els clients de la BDOO d'acord amb la definició de les seves classes, es pot fer:

```
SELECT c.adreçaPostal, c.telefon
FROM Clients c
WHERE c.nom = "Client1"
```

Aquesta consulta retorna l'adreça postal i el telèfon del client amb nom "Client1".

També és possible accedir als encàrrecs per mitjà dels clients, seguint la seva relació:

```
SELECT e.data
FROM Clients c, c.encarrecs e
WHERE c.adreçaMail = "email1@domini.com"
```

Aquesta consulta retorna la data de tots els encàrrecs del client amb adreça de correu "email1@domini.com".

3.2 La llibreria db4O

Tot i els esforços per estandarditzar l'ús de les BDOO, no es pot dir que actualment hi hagi cap equivalent al llenguatge SQL. Tot i que sobre el paper hi ha l'ODL i l'OQL, a la pràctica ara mateix no hi ha cap llengua franca que es pugui garantir que està suportada, almenys en els seus aspectes fonamentals, per totes les bases de dades, tot i que després cada fabricant pugui afegir igualment les seves pròpies extensions propietàries. Per tant, cada tipus de BDOO ofereix el seu propi sistema per accedir als objectes emmagatzemats. Afortunadament, com aviat veureu, això no és gaire problemàtic, ja que l'avantatge d'usar una BDOO és poder crear codi on operar amb objectes persistents; és gairebé igual que treballar amb objectes a memòria, i, per tant, els mecanismes que ofereixen les diferents BDOO sovint són molt semblants a treballar amb objectes directament a memòria. El que varia són les llibreries de classes a usar, però no la idea general.

Les classes de db4o estan dins els *packages* `com.db4o...`

En aquest apartat es veurà un cas concret d'accés a una BDOO anomenada **db4o**, de lliure distribució, actualment amb versions per al Java i per a .NET. Evidentment, aquesta secció se centra exclusivament en la versió per al Java. No es troba en la distribució estàndard del Java, i, per tant, s'ha de descarregar i afegir als vostres projectes a part.

La llista de passos per interactuar amb aquesta BDOO és molt semblant a l'emprada mitjançant JDBC, si bé la manera com es fa amb codi Java és completament diferent en alguns aspectes.

Un cop s'han emmagatzemat objectes a una **BDOO db4o**, ja no es pot modificar la classe d'aquests objectes. Si es modifica (i, per tant, es torna a compilar el fitxer JAVA), totes les dades que hi ha dins deixen de tenir validesa. Si es vol usar la nova versió de la classe, cal tornar a generar el contingut de la BDOO des de zero.

3.2.1 Obertura de la BDOO

Normalment, els fitxers de db4o s'escriuen amb l'extensió ".db4o"

Una BDOO *db4o* no és més que un fitxer, si bé força complex en la seva estructura interna. Per tant, alguns dels aspectes són semblants, com ara l'accés a les dades emmagatzemades dins, que és semblant a com es faria amb un fitxer qualsevol. Aquest és el cas de la seva obertura, per tal de poder llegir les seves dades o escriure'n, i el seu posterior tancament quan ja no cal usar-lo més. El mètode estàtic `openFile` de la classe `Db4oEmbedded` permet que es pugui obrir aquest fitxer. Com succeeix en treballar amb fitxers, si aquest no existeix, se'n crea un de nou, amb la BDOO buida.

```
1 import com.db4o.*;
2 ...
3 ObjectContainer db = Db4oEmbedded.openFile("BDOOClients.db4o");
4 //Accions amb la BDOO
5 db.close();
```

Mitjançant l'objecte resultant de la crida, una instància d'`ObjectContainer` , es podran dur a terme totes les accions amb la BDOO. Fins a cert punt, és l'equivalent a una connexió a una BDR mitjançant JDBC. Un cop s'ha acabat de treballar, sempre cal tancar la BDOO usant el mètode `close`.

Ara bé, aquesta és una aproximació molt simple a l'ús d'una BD, ja que el fitxer ha d'estar emmagatzemat en local a la mateixa màquina que executa l'aplicació. Per poder tractar les dades des d'una altra màquina, caldrà copiar tant l'aplicació com també el fitxer de la BD. Normalment, el model de treball amb una BD es basa que hi ha un servidor, on s'executa la BD, i el desenvolupador genera el client, que s'hi connecta per xarxa. Un cop establerta la connexió, pot enviar peticions a la BD, i un cop dutes a terme totes les tasques, la tanca.

Les llibreries de *db4o* no proporcionen cap servidor en forma de programa que només cal instal·lar i configurar en un equip. Afortunadament, aquesta tasca és molt simple, ja que es pot fer en poques línies de codi. Dins aquest model d'accés a les dades, la BD continua essent un únic fitxer, en aquest cas emmagatzemat al servidor central, i les llibreries ja proporcionen tots els mecanismes necessaris per publicar el servei a la màquina i poder accedir-hi remotament de manera transparent, com si en realitat fos un fitxer en local en l'equip que executa l'aplicació client.

Per posar en marxa un servidor, cal usar el mètode estàtic `openServer` de la classe `Db4oClientServer` (al *package* `com.db4o.cs`). Un cop s'executa el mètode, el servei d'accés a la DBOO es posa en marxa a l'equip local on s'ha executat. Mentre el programa segueixi en execució, estarà disponible. Aquest mètode requereix tres paràmetres:

- Un nou objecte de configuració del servidor, que sempre es genera cridant `Db4oClientServer.newServerConfiguration()`.

- El nom del fitxer on es desa la BDOO. Si no existeix, se'n crearà un de nou buit.
- El port on s'executarà el servei.

Sovint, en engegar o apagar un servei db4o apareix un missatge de depuració per la consola d'errors del Java.

La creació d'un servidor retorna un objecte `ObjectServer`, a partir del qual es poden configurar certs aspectes del comportament del servei. El més important de tots és poder afegir usuaris i contrasenyes que limitin qui pot accedir a la BDOO remotament. Això es fa usant el mètode `grantAccess`.

Vegem un exemple de servidor *db4o*, molt senzill, però més que suficient per provar-ne el funcionament i els exemples d'aquest apartat si es desitja. En aquest cas, s'ha fet que el programa que executa el servei no finalitzi fins que l'usuari pitgi "Q" o "q". En fer-ho, el servei s'apaga i deixa d'estar accessible remotament.

```
1  import java.io.Scanner;
2  import com.db4o.*;
3  import com.db4o.cs.Db4oClientServer;
4
5  public class Server {
6    public static void main (String[] args) throws Exception {
7      ObjectServer sv = Db4oClientServer.openServer(Db4oClientServer.
           newServerConfiguration(), "BDRemota.db4o", 7000);
8      sv.grantAccess("usuari", "contrasenya");
9      Scanner in = new Scanner (System.in);
10     System.out.println("Pitja [Q] per tancar el servidor.");
11     while (in.hasNext()) {
12       if  ("Q".equalsIgnoreCase(in.next())) break;
13     }
14   }
15 }
```

Des del punt de vista de l'aplicació que vol accedir a la BD remota, cal usar el mètode `openClient` de la classe `Db4oClientServer`. Ara bé, calen alguns paràmetres per poder identificar on es vol accedir:

- L'identificador de la màquina remota.
- El port del servei, tal com s'ha configurat en fer `openServer`.
- Un nom d'usuari i una contrasenya vàlids.

Si algun d'aquests paràmetres no és correcte, es llançarà una excepció indicant que la connexió no s'ha pogut establir.

```
1  import com.db4o.*;
2  import com.db4o.cs.Db4oClientServer;
3  ...
4  ObjectContainer db = Db4oClientServer.openClient("lamevamaquina.domini.cat",
       7000, "usuari","contrasenya");
5  //Accions amb la BDOO
6  db.close();
```

Localhost

Per connectar-se a l'equip local, pel cas on tant el client com el servidor s'executen la mateixa màquina,es pot usar el nom de *host*, *localhost*. Això és útil per fer proves en un únic equip.

L'objecte retornat en aquest cas també és un `ObjectContainer`, per la qual cosa, un cop establerta la connexió amb la DB remota amb aquesta crida, totes les operacions que es poden dur a terme són exactament iguals que si es fessin accedint a un fitxer en local.

3.2.2 Emmagatzematge de nous objectes

Per emmagatzemar qualsevol objecte del vostre programa dins la BDOO, només cal cridar el mètode `store` que proporciona l'`ObjectContainer`, obtingut en obrir la BD (ja sigui en un fitxer local o remot). Aquest mètode només té un paràmetre, que és l'objecte a emmagatzemar. Mitjançant aquest mètode es pot desar qualsevol tipus d'objecte, sense que per aquest fet s'hagi de fer cap modificació al seu codi font.

Per exemple, suposeu que es volen gestionar els encàrrecs que duen a terme clients d'una empresa i, per a tal efecte, s'han generat les classes següents, que inicialment no es van desenvolupar amb el propòsit de ser integrades dins cap BDOO *db4o*. Noteu que la classe `Client` només permet canviar l'adreça electrònica un cop s'ha instanciat (usant el mètode `setAElectronica`).

```
1  import java.util.*;
2  public class Client {
3    private String nom;
4    private String aPostal;
5    private String aElectronica;
6    private String telefon;
7    private List<Encarrec> liComandes = new LinkedList<Encarrec>();
8    public Client(String n, String ap, String ae, String t) {
9      nom = n;
10     aPostal = ap;
11     aElectronica = ae;
12     telefon = t;
13   }
14   public String getNom() {
15     return nom;
16   }
17   public String getAPostal() {
18     return aPostal;
19   }
20   public String getAElectronica() {
21     return aElectronica;
22   }
23   public void setAElectronica(String ae) {
24     aElectronica = ae;
25   }
26   public String getTelefon() {
27     return telefon;
28   }
29   public int getNreComandes() {
30     return liComandes.size();
31   }
32   public void addComanda(Encarrec e) {
33     liComandes.add(e);
34   }
35   public List<Encarrec> getComandes() {
36     return liComandes;
37   }
38   @Override
39   public String toString() {
40     String res = nom + " : " + aPostal + " : (" + aElectronica + ", " + telefon
         + ")\n";
41     Iterator<Encarrec> it = liComandes.iterator();
42     while (it.hasNext()) {
43       Encarrec e = it.next();
44       res += e.toString() + "\n";
45      }
46     return res;
47   }
```

```java
import java.util.Date;

public class Encarrec {
  private String nomProducte;
  private int quantitat;
  private Date data;

  public Encarrec(String n, int q) {
    nomProducte = n;
    quantitat = q;
    data = new Date();
  }
  public String getNom() {
    return nomProducte;
  }
  public int getQuantitat() {
    return quantitat;
  }
  public Date getData() {
    return data;
  }

  @Override
  public String toString() {
    return getData()+ " — " + getNom() + " (" + getQuantitat() + ")";
  }
}
```

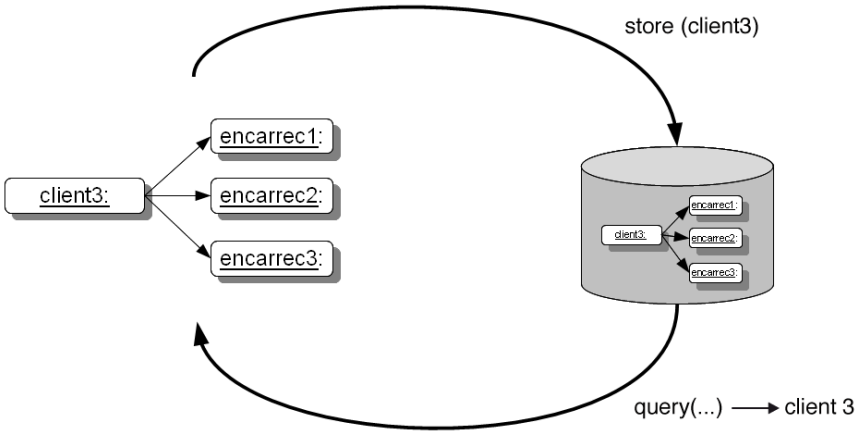
El codi d'un programa que emmagatzema quatre clients a la BDOO, entre els quals un d'ells ja té tres encàrrecs associats, seria el que hi ha a continuació, basat, senzillament, a fer crides successives a `store`, passant com a paràmetre cada objecte que es vol emmagatzemar. Per simplificar, se suposa que s'obre una BDOO ubicada en un fitxer local, però per al cas remot, seria exactament el mateix. Recordeu que sempre cal controlar les excepcions en accedir a les dades.

```java
import com.db4o.*;
public class EmmagatzemaClients  {
  public static void main(String[] args) throws Exception {
    ObjectContainer db = Db4oEmbedded.openFile("BDOOClients.db4o");
    try {
      Client[] clients = {
        new Client("Client1", "Adreça1", "e—mail1@domini.com", "+34931112233"),
        new Client("Client2", "Adreça2", "e—mail2@domini.com", "+34932223344"),
        new Client("Client3", "Adreça3", "e—mail3@domini.com", "+34931112233"),
        new Client("Client4", "Adreça3", "e—mail4@domini.com", "+34931112233")
      };
      clients[2].addComanda(new Encarrec("Impressora",1));
      clients[2].addComanda(new Encarrec("Toner Impressora",4));
      clients[2].addComanda(new Encarrec("Paquest A4", 20));
      for(int i = 0; i < clients.length; i++) {
        db.store(clients[i]);
      }
    } finally {
      db.close();
    }
  }
}
```

En aquest codi hi ha un aspecte molt important en què val la pena fixar-se en detall. Si l'examineu, veureu que, tot i que els objectes que cal emmagatzemar són tant els clients com els encàrrecs, enlloc es fa cap `store` per als encàrrecs. Només es fa per als clients. Això es deu al fet que, en les BDOO, en fer persistent un objecte, aquesta persistència es propaga a tots els objectes enllaçats

també, de manera transitiva, fins que tot el mapa d'objectes, el *graf* d'enllaços que parteix de l'objecte emmagatzemat es troba al complet a la BDOO. Això es fa automàticament sense necessitat que ho faci el programador. Aquest comportament, esquematitzat a la figura 3.1, també succeeix quan es recuperen les dades amb alguna cerca, com veureu properament.

FIGURA 3.1. Tractament dels mapes d'objectes sobre una BDOO



Una pregunta que pot sorgir és: què passa si, *a posteriori*, s'emmagatzema un objecte que, a causa d'aquest comportament, ja existeix a la BDOO? Per exemple, si es fa un `store` d'un dels tres encàrrecs, quan aquest de fet ja està a la BDOO, ja que s'ha desat automàticament en emmagatzemar el client 3. La resposta és que no passa res. La BDOO ja detecta que es tracta del mateix objecte i, per tant, no es generen dues còpies.

Això és possible ja que, recordeu que una de les bases de la OO és "Tot és un objecte, amb una identitat pròpia". O sigui, tot objecte s'identifica amb una única referència. Aquesta pot estar replicada en diferents variables, però totes apunten a un únic objecte a memòria. Això també es compleix dins la BDOO, i, per tant, aquesta és capaç d'identificar diferents operacions amb un mateix objecte.

Ara bé, aquest comportament té unes altres implicacions que cal tenir també ben presents. Suposeu que aquest mateix programa l'executeu 3 vegades consecutives. Quants objectes client hi haurà emmagatzemats a la BDOO després de la darrera execució? La resposta és que n'hi haurà 12, ja que els objectes de cada execució són independents entre si. Tot i que el contingut dels objectes en successives execucions és exactament el mateix, l'objecte en si és diferent, tenen diferents referències, i, per tant, es considera un nou element a la BDOO. En conseqüència, cal anar amb molt de compte en emmagatzemar nous objectes entre execucions diferents del programa, ja que això sempre implicarà la creació d'un nou element a la BDOO.

3.2.3 Cerca d'objectes

Els mecanismes de lectura d'una BD solen ser els més importants, ja que normalment són els que s'usen més sovint. La llibreria *db4o* ofereix dos sistemes diferents per dur a terme aquest procés, diferenciats únicament per com es discrimina quins objectes cal retornar de la BDOO. En qualsevol dels dos casos, el que es retorna és un conjunt d'objectes, exactament tal com els heu definit a les vostres classes, empaquetat dins un contenidor específic de *db4o* anomenat `ObjectSet<T>`. Aquesta és una classe genèrica, de manera que, en declarar-ne una variable, cal establir sempre el tipus d'objectes que contindrà (com passa amb altres contenidors del Java:`List`, `Set`...). Per exemple, si es volen consultar clients, caldrà usar la definició `ObjectSet<Client>`.

Una de les característiques més interessants de les cerques dins una BDOO és la recuperació d'objectes enllaçats entre si, de manera que si, en recuperar un objecte, aquest contenia a la vegada referències a altres objectes, aquests objectes també són recuperats.

I així fins a un cert nivell de profunditat, que per defecte val 5, però que es pot modificar utilitzant el codi:

```
1 EmbeddedConfiguration conf = Db4oEmbedded.newConfiguration();
2 conf.common().activationDepth(novaProfunditat);
```

L'exemple més senzill d'aquest comportament són els atributs de tipus `String`, que també són pròpiament objectes, i són restaurats junt amb l'objecte original. Però aquest comportament també es compleix per a objectes de qualsevol altra classe, ja sigui del Java o creada per vosaltres.

Cerques per exemple

Les cerques per exemple (*Query-By-Example*) són les més senzilles. Es basen a crear una instància del tipus d'objecte a cercar, i només assignar valors als atributs sobre els quals es vol cercar una coincidència exacta. La resta, es posen a `null` (en el cas de valors numèrics, a 0). Llavors, s'invoca el mètode `queryByExample`, usant com a paràmetre aquesta instància.

Per exemple, per cercar tots els clients que tenen com a adreça la cadena de text "Adreça3" es faria d'acord al codi que hi ha a continuació. En executar-se el codi, atesos els quatre clients d'exemple existents a la BDOO, dins l'`ObjectSet` hi haurà 2 objectes `Client`: el que té com a `Client3` i el que té `Client4`.

```
1 ObjectContainer db = Db4oEmbedded.openFile("BDOOClients.db4o");
2   ...
3   Client ex = new Client(null, "Adreça3", null, null);
4   ObjectSet<Client> result = db.queryByExample(ex);
5   ...
```

L'`ObjectSet` pot ser recorregut seqüencialment mitjançant els mètodes `hasNext()`, que indica si encara hi ha elements per recórrer, i `next()`, que llegeix un element i avança una posició. En aquest sentit, es comporta exactament igual que un `Iterator` de les llibreries estàndard del Java. Per exemple, per tal de mostrar per pantalla tots els elements obtinguts per la consulta, es podria fer el següent:

```
1 ObjectSet<Client> result = ...
2 while (result.hasNext()) {
3   Client cli = result.next();
4   System.out.println(cli);
5 }
```

Atès que `ObjectSet` és una classe genèrica, el mètode `next()` retorna directament una instància del tipus indicat, i no cal fer cap "cast". Ara bé, cal anar amb compte a definir sempre el tipus dels elements de l'`ObjectSet`, en declarar-ne la variable, de la mateixa classe que la instància usada en invocar `queryByExample`. En cas contrari, es produirà un error per manca de concordança de tipus.

Si bé aquesta mena de cerques són molt senzilles de fer, no permeten res més que la comparació directa d'atributs. Tanmateix, tampoc no permeten fer cerques sobre valors que siguin `null` o 0, ja que són les condicions per ignorar-los com a criteri de cerca. També tenen la restricció que no poden usar-se a partir d'objectes que no permeten inicialitzar atributs a valors `null` o 0.

Cerques natives

Normalment, les cerques que es voldran fer dins una BD van més enllà de les simples concordances directes entre valors d'atributs, i es desitjarà poder avaluar tota mena de condicions, a gust del desenvolupador, tal com permet SQL (o més). Hi ha dos mecanismes per dur a terme cerques d'acord a criteris complexos dins *db4o*, però la més potent, i, a la vegada, la més senzilla, són les cerques natives (*Native Queries*). Es basen en l'execució d'un codi Java per avaluar si un objecte dins la BDOO compleix la condició de cerca o no. En basar-se només en codi Java, la seva versatilitat és la mateixa que en qualsevol programa possible, o sigui, molt gran.

Per executar una cerca nativa s'usa el mètode `query`, que necessita com a paràmetre una implementació de la classe `Predicate<T>` (pertanyent al *package* `com.db4o.query`).

`Predicate<T>` és una classe genèrica abstracta, per la qual cal indicar quina mena d'objectes és capaç de processar en emprar-la. L'únic mètode abstracte que cal implementar és `match`, que s'ha de fer que s'avaluï a `true` si es considera que l'objecte passat com a paràmetre compleix el criteri de cerca, o `false` en cas contrari. Aquest mètode s'executarà passant com a paràmetre tots els objectes de la BDOO que es corresponguin al tipus escollit, un per un. Ara bé, el codi d'aquest mètode l'heu d'implementar vosaltres, ja que és el que ha de prendre la decisió de si un objecte compleix o no el criteri de cerca, i, per tant, el podeu fer al vostre gust. Per a cada cerca diferent que es vol fer al programa, caldrà crear una implementació diferent d'aquesta classe.

Tot seguit, es mostra un exemple d'implementació de la classe `Predicate<T>`, de manera que avalua si, donat un client, aquest té com a adreça "Adreça3", ignorant majúscules i minúscules (cosa que no es pot fer amb una cerca per exemple, ja que compara textos estrictament). Normalment, per implementar aquesta classe s'usa una classe anònima, de manera que tan bon punt es declara una variable d'aquest tipus, ja s'indica el seu codi immediatament, en lloc de fer-ho en un fitxer a part.

```
ObjectContainer db = Db4oEmbedded.openFile("BDOOClients.db4o");

//Declaració de la implementació de Predicate<T> com a classe anònima
Predicate p = new Predicate<Client>() {
  @Override
  public boolean match(Client c) {
    //Codi pel criteri de cerca
    return "Adreça3".equalsIgnoreCase(c.getAPostal());
  }
};
//Fi de la declaració

ObjectSet<Client> result = db.query(p);
...
```

L'avantatge d'usar classes anònimes és que permeten incloure, dins el seu propi codi, atributs declarats dins la mateixa classe que les conté. Això atorga gran flexibilitat si es volen fer cerques en base a variables, i no a valors constants. Això permet crear cerques parametritzades mitjançant objectes `Predicate`. Per exemple, suposem que es volen cercar els clients que han superat cert valor en el nombre de comandes, però aquest valor depèn d'una variable dins el codi del vostre programa, ja que es demana pel teclat i per tant pot ser diferent en diferents execucions. El codi següent fa tot just això.

```
import com.db4o.*;
import com.db4o.query.Predicate;
import java.util.Scanner;

public class CercaParametritzada  {
  private int valor = 0;

  public void cercaClients() throws Exception {
    ObjectContainer db = Db4oEmbedded.openFile("BDOOClients.db4o");
    try {
      Predicate p = new Predicate<Client>() {
       @Override
       public boolean match(Client c) {
         return valor <= c.getNreComandes();
       }
      };
      ObjectSet<Client> result = db.query(p);
      while (result.hasNext()) {
        Client cli = result.next();
        System.out.println(cli);
      }
    } finally {
      db.close();
    }
  }

  public void setValor(int v) {
    valor = v;
  }

  public static void main(String[] args) throws Exception {
      CercaParametritzada cp = new CercaParametritzada();
```

```
      Scanner in = new Scanner(System.in);
      System.out.print("Quin és el valor mínim a cercar? ");
      cp.setValor(in.nextInt());
      cp.cercaClients();
  }
}
```

En fer cerques, recordeu que a la BDOO no només hi ha aquells objectes dels quals s'ha fet una crida `store` explícita, sinó que també es troben disponibles els objectes emmagatzemats implícitament a causa d'enllaços amb altres objectes. Per tant, a la BDOO, també es poden fer cerques sobre encàrrecs. El següent codi permet fer una cerca parametritzada d'encàrrecs dins el sistema, en base a un valor mínim en la seva quantitat.

```
import com.db4o.*;
import com.db4o.query.Predicate;
import java.util.Scanner;

public class CercaParametritzada  {

  private int valor = 0;

  public void cercaEncarrecs() {
    ObjectContainer db = Db4oEmbedded.openFile("BDOOClients.db4o");
    try {
      Predicate p = new Predicate<Encarrec>() {
       @Override
       public boolean match(Encarrec c) {
         return valor <= c.getQuantitat();
       }
      };
      ObjectSet<Encarrec> result = db.query(p);
      while (result.hasNext()) {
        Encarrec e = result.next();
        System.out.println(e);
      }
    } finally {
      db.close();
    }
  }

  public void setValor(int v) {
    valor = v;
  }

public static void main(String[] args) throws Exception {
      CercaParametritzada cp = new CercaParametritzada();
      Scanner in = new Scanner(System.in);
      System.out.print("Quin és el valor mínim a cercar? ");
      cp.setValor(in.nextInt());
      cp.cercaEncarrecs();
    }
}
```

En aquests exemples, el codi per establir si cada objecte compleix o no el criteri de la cerca és relativament simple, d'una sola línia, però el codi del mètode `match` pot ser tan complex com es desitgi i basat en qualsevol informació disponible dins el programa. Ara bé, malauradament, això vol dir que no es pot disposar de funcions executables directament a la BD (com passava amb MAX, AVG... a SQL).

3.2.4 Actualitzacions d'objectes

Normalment, l'actualització d'objectes té sentit quan, primer de tot, s'afegeix un element a la BDOO, i, en posteriors execucions del programa, el contingut dels objectes emmagatzemats veuen modificats els seus valors al llarg del seu cicle de vida (un client canvia la seva adreça electrònica, o va afegint encàrrecs). És el que dóna sentit a la persistència dels objectes en una BD, en definitiva.

L'actualització d'objectes amb *db4o* es porta a terme usant la crida `store`, tal com s'ha usat per emmagatzemar un nou objecte, només que, en aquest cas, en lloc de ser un objecte nou, és un que ja existia prèviament a la BD. Per fer això, primer cal carregar a memòria l'objecte des de la BD, usant una cerca, i un cop es disposa de la seva referència, ja s'hi pot accedir per fer canvis usant els mètodes que proporciona la seva classe, tal com es faria normalment. Aquests canvis no es propagaran a la BD fins tornar a executar `store`.

Podeu comprovar que s'ha modificat executant els exemples anteriors de cerca amb un valor que llisti tots els clients (com el 0).

Per exemple, el següent codi permet canviar l'adreça electrònica d'un client, donat el seu nom (suposarem que el nom ha de ser únic perquè funcioni). Per millorar la llegibilitat, s'han omès algunes comprovacions d'errors (si no s'escriu res amb el teclat, per exemple).

```
1  import com.db4o.*;
2  import java.util.Scanner;
3
4  public class ModificaAElectronica {
5
6    public static void main(String[] args) throws Exception {
7      ObjectContainer db = Db4oEmbedded.openFile("BDOOClients.db4o");
8      Scanner in = new Scanner(System.in);
9      System.out.print("Quin és nom del client? ");
10     String nom = in.nextLine();
11
12     //Cercar clients a la BDOO i obtenir-los a memòria com a objectes del
           programa
13     Client qbe = new Client(nom, null, null, null);
14     ObjectSet<Client> clients = db.queryByExample(qbe);
15
16     if (clients.size() != 1) {
17       System.out.println("No es pot modificar aquest nom.");
18     } else {
19       System.out.print("Quina és la nova adreça? ");
20       String ad = in.nextLine();
21       Client c = clients.next();
22       c.setAElectronica(ad);
23       db.store(c);
24     }
25     db.close();
26   }
27
28 }
```

L'única excepció a aquest comportament és si l'objecte enllaçat és una cadena de text (com s'ha vist, precisament, a l'exemple anterior).

Ara bé, en el cas d'objectes que contenen enllaços a altres objectes (que, a la vegada, poden tenir enllaços a altres objectes, i així fins a molts nivells de profunditat), el comportament de *db4o* no és aquest. Per poder garantir un rendiment òptim, és obligat, en el cas d'accés a fitxers en local, canviar una mica la declaració de l'obertura de la BD, indicant que ha d'estar configurada per acceptar

actualitzacions en cascada. Això es fa usant un constructor diferent i una inicialització prèvia d'un objecte `EmbeddedConfiguration`. En aquesta inicialització, cal llistar totes les classes on es vol que la BDOO controli actualitzacions en cascada. O sigui, que es vol que, si es fa un `store` sobre un objecte d'aquest tipus, també es comprovi si cal actualitzar tots els seu *graf* d'objectes enllaçats.

El codi d'inicialització és el que hi ha a continuació. On diu "NomClasse1", "NomClass2"... caldria posar el nom de la classe a controlar les actualitzacions en cascada. Hi haurà tants registres de classes com classes es volen controlar. Per al cas de qualsevol classe que no s'enumeri explícitament a la inicialització, amb una línea corresponent, en fer un `store`, només se n'actualitzaran els atributs que siguin tipus primitius o cadenes de text, però no altres objectes.

```
EmbeddedConfiguration config = Db4oEmbedded.newConfiguration();

  //Configurar totes les classes on cal propagar canvis
  config.common().objectClass(NomClasse1.class).cascadeOnUpdate(true);
  config.common().objectClass(NomClasse2.class).cascadeOnUpdate(true);
  //etc.

  ObjectContainer db = Db4oEmbedded.openFile(config, "BDOOClients.db4o");
```

Per exemple, el codi següent permet afegir un encàrrec a un client, donat el seu nom (suposant que aquest és únic). Com que els encàrrecs estan enllaçats dins una llista, cal forçar les actualitzacions en cascada per a aquesta classe.

```
import com.db4o.*;
import com.db4o.config.EmbeddedConfiguration;
import java.util.Scanner;

public class AfegirEncarrec {
  public static void main(String[] args) throws Exception {
    EmbeddedConfiguration config = Db4oEmbedded.newConfiguration();

    config.common().objectClass(Client.class).cascadeOnUpdate(true);

    ObjectContainer db = Db4oEmbedded.openFile(config, "BDOOClients.db4o");

    Scanner in = new Scanner(System.in);
    System.out.print("Quin és nom del client? ");
    String nom = in.nextLine();

    //Cercar clients a la BDOO i obtenir-los a memòria com a objectes del
        programa
    //S'usa una cerca per exemple
    Client qbe = new Client(nom, null, null, null);
    ObjectSet<Client> clients = db.queryByExample(qbe);

    if (clients.size() != 1) {
      System.out.println("No es pot modificar aquest nom.");
    } else {
      System.out.print("Quin és nom del producte? ");
      String prod = in.nextLine();
      System.out.print("Quants en vols encarregar? ");
      String txtQuan = in.nextLine();
      int quant = Integer.parseInt(txtQuan);

      Encarrec ne = new Encarrec(prod, quant);
      Client c = clients.next();
      c.addComanda(ne);
      db.store(c);
    }
    db.close();
```

```
  }
}
```

3.2.5 Esborrat d'objectes

Per esborrar un objecte s'aplica la idea general de les actualitzacions, però en lloc del mètode `store`, cal usar el mètode `delete`. Primer cal recuperar l'objecte de la BD, carregar-lo a memòria, i després esborrar-lo usant la seva referència. Per exemple, per esborrar un client, d'entrada, es podria fer així:

```
import com.db4o.*;
import java.util.Scanner;

public class EsborraClient {
  public static void main(String[] args) throws Exception {
    ObjectContainer db = Db4oEmbedded.openFile("BDOOClients.db4o");
    Scanner in = new Scanner(System.in);
    System.out.print("Quin és nom del client? ");
    String nom = in.nextLine();
    //Cercar clients a la BDOO i obtenir-los a memòria com objectes del
        programa
    Client qbe = new Client(nom, null, null, null);
    ObjectSet<Client> clients = db.queryByExample(qbe);
    if (clients.size() != 1) {
      System.out.println("No es pot modificar aquest nom.");
    } else {
      Client c = clients.next();
      db.delete(c);
    }
    db.close();
  }
}
```

Si mostreu els clients que hi ha a la BDOO, veureu que ja no existeix el client que heu escrit en executar-lo. Ara bé, aquest programa no és del tot correcte. Si esborreu el client 3, executeu el programa que fa cerca de comandes i cerqueu totes les comandes amb una quantitat superior a 0, observareu que les comandes del client 3 encara estan a la BD. La crida a `delete` esborra, estrictament, l'objecte associat a aquest client, però absolutament res més.

Com passava amb les actualitzacions d'objectes que enllacen altres objectes, aquest cas s'ha de tractar d'una manera una mica especial. Malauradament, no hi ha cap paràmetre de configuració que resolgui el problema. Per tal d'eliminar objectes, el programador ha de fer un codi que, manualment i un per un, vagi recorrent el mapa d'objectes i eliminant els que correspongui. Per als clients, el programa correcte hauria de ser el següent:

```
import com.db4o.*;
import java.util.*;

public class EsborraClient {

  public static void main(String[] args) throws Exception {
    ObjectContainer db = Db4oEmbedded.openFile("BDOOClients.db4o");
    Scanner in = new Scanner(System.in);
    System.out.print("Quin és nom del client? ");
```

```
10      String nom = in.nextLine();
11
12      //Cercar clients a la BDOO i obtenir-los a memòria com objectes del
            programa
13      Client qbe = new Client(nom, null, null, null);
14      ObjectSet<Client> clients = db.queryByExample(qbe);
15
16      if (clients.size() != 1) {
17        System.out.println("No es pot modificar aquest nom.");
18      } else {
19        Client c = clients.next();
20        List<Encarrec> li = c.getComandes();
21        Iterator<Encarrec> it = li.iterator();
22        //anem esborrant tots els encàrrecs, un per un
23        while (it.hasNext()) {
24          Encarrec e = it.next();
25          db.delete(e);
26        }
27        //Ja es pot esborrar el client
28        db.delete(c);
29      }
30      db.close();
31    }
32  }
```

En aquest cas concret, la tasca no és gaire complicada, ja que només hi ha un nivell d'objectes enllaçats, i donat un client, les seves comandes només les gestiona ell i cap altre objecte. Ara bé, cal ser conscients que gestionar l'esborrat correcte d'objectes enllaçats dins la BDOO es pot arribar a complicar força en casos complexos, ja que sempre cal garantir la consistència de tot el mapa d'objectes emmagatzemat. Si un objecte és referenciat per més d'un altre objecte, cal anar amb molt de compte de no esborrar-lo, perquè es podrien deixar referències a `null` sense voler, cosa que comportaria errors d'execució en el futur. Per exemple, si els encàrrecs es poguessin compartir entre més d'un client, en esborrar un client no es podrien esborrar alegrement tots els seus encàrrecs. Caldrà comprovar que cada encàrrec només està assignat a un únic client i, si cal, l'esborrarem, però, en cas contrari, no.

3.3 JDO (Java Data Objects)

Normalment, el motiu principal per treballar amb una BDDO és poder generar codi que s'integri de manera natural en un programa orientat a objectes. O sigui, poder desar i cercar directament objectes dins la base de dades, de manera que si estan enllaçats a altres objectes mitjançant referències, també es recuperin automàticament. A més a més, els mètodes d'accés a la BD ja retornen directament els conjunts d'objectes que es vol cercar.

En contraposició, l'ús d'una BDR implica haver de fer traduccions d'objectes a taules, i les cerques retornen files de valors de taules, que cal recuperar, i a partir d'aquests cal instanciar les classes desitjades. Per desar un objecte, cal fer exactament el mateix, però a la inversa: extreure'n els atributs i convertir-los a valors dins una taula. En cas de voler desar objectes enllaçats entre si, la tasca es pot arribar a complicar força. A part, cal fer conversions de tipus Java a SQL i

viceversa. En general, la seva integració dins el codi Java no és gens natural, ja que s'està usant un sistema basat en taules per desar elements que, a memòria, no són pas a taules.

Per tant, des del punt de vista de simplicitat del codi, l'ús d'una BDOO dins un programa orientat a objectes ofereix enormes avantatges. Malauradament, el seu grau de maduresa i estandardització no arriba ni de bon tros al que actualment tenen les BDR, per la qual cosa no és tan senzill decidir quin tipus usar. Ara bé, el Java disposa d'una especificació pròpia per definir la persistència d'objectes usant una estratègia semblant a l'accés a una BDOO, però sense lligar-se a un tipus concret de mecanisme d'emmagatzematge.

Java Data Objects (JDO) API és una interfície estàndard basada en l'abstracció del model de persistència de Java. Les aplicacions escrites amb l'API de JDO són independents del sistema d'emmagatzematge subjacent. Diferents implementacions poden donar suport a diferents tipus de bases de dades, incloent bases de dades relacionals i d'objecte, XML, arxius...

Una BDOO que es basa en JDO, per a l'accés a les seves dades, és JDOInstruments.

Aquesta especificació només serveix per al llenguatge Java i no és portable a cap altre llenguatge, en contrast amb l'ODL i OQL, que intenten ser llenguatges genèrics independents del llenguatge (com l'SQL). De fet, l'objectiu d'aquesta especificació és haver d'obviar la necessitat d'aprendre cap llenguatge extra que no sigui el mateix Java i res més. En aquesta secció es presenta només una breu introducció dels aspectes generals del seu funcionament.

Les classes de JDO pertanyen al paquet `javax.jdo`. Aquest no pertany a la distribució estàndard del Java.

JDO defineix tres tipus de classes:

- **Habilitades per a persistència**. Representen les classes els objectes de les quals poden passar a un estat persistent. Al llarg de l'execució de l'aplicació poden passar de la memòria a ser emmagatzemades en la BD, i viceversa.
- **Conscients de persistència**. Són les classes que manipulen el tipus anterior. Concretament, la classe `JDOHelper` proporciona diferents mètodes per descobrir si un objecte concret es troba en un estat persistent o no.
- **Normals**. Els seus objectes no poden passar a un estat persistent, només existiran en la memòria de l'aplicació.

Objectes transitoris

S'anomena objectes transitoris els que només són en la memòria, però no en la BD, i objectes buits (*hollow*) els que només estan en la BD, sense representació en la memòria.

Els objectes de classes habilitades per a la persistència poden passar per diferents estats dins el seu cicle de vida, depenent de diversos factors: si només es troben en la memòria o també estan representats en la BD, si la seva representació actual en la memòria difereix de l'escrita en la BD, etc. La transició entre estats és controlada per la classe `PersistenceManager`, que serveix d'*interface* primària entre l'aplicació i la BD. Aquesta classe és, fins a cert punt, l'equivalent a la connexió JDBC en el cas d'accés a BDR.

Totes les accions cap a la BD vindran determinades a partir de crides a mètodes definits en aquesta classe. Alguns dels seus mètodes més significatius són:

- `void makePersistent(Object o)`. Fa un objecte persistent a la BD de manera que quan es manipulin les seves dades, quedaran desades.
- `void makeTransient(Object o)`. Fa que un objecte deixi de ser persistent en la BD i passi a ser transitori.
- `void deletePersistent(Object o)`. Esborra un objecte persistent de la BD.

3.3.1 Instanciació d'un objecte PersistenceManager

Per instanciar un objecte `PersistenceManager` no es pot usar directament una sentència `new`, cal usar la classe `PersistenceManagerFactory`:

```
1 Properties props = new Properties();
2 props.put(...);
3 PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory(props);
4 PersistenceManager pm = pmf.getPersistenceManager();
```

Un cop disposem de l'objecte `PersistenceManager`, ja es pot començar a operar amb la BDOO.

3.3.2 Interacció amb la BD

Per interactuar amb la BD, el JDO usa transaccions, per modificar l'estat d'un objecte, i consultes, per accedir a les dades persistents. Aquesta filosofia no és gaire diferent de la de JDBC. Les transaccions estan representades per la classe `Transaction`, mentre que les consultes per la classe `Query`. Els objectes d'ambdós tipus s'obtenen a partir de crides al `PersistenceManager`:

- **Transaction currentTransaction()**. Obté una transacció per modificar l'estat d'objectes dins l'aplicació.
- **Query newQuery(java.lang.Class cls)**. Crea una consulta, per ser executada posteriorment per accedir als objectes persistents dins la BDOO. El paràmetre cls indica la classe esperada dels objectes que es vol consultar. Aquest mètode es troba sobrecarregat per poder proporcionar diferents opcions.

L'aspecte més important en el procés d'emmagatzemament i recuperació de dades en la BD és que tot codi Java generat segueix exactament la mateixa filosofia orientada a objectes com si sempre s'operés sobre instàncies a memòria (només s'opera amb referències). El mecanisme de persistència és transparent.

El mètode `rollback()` permet fer enrere una transacció en cas d'error durant el procés.

Tot seguit es mostra com un objecte passa de ser transitori a persistent. El resultat final és que aquest queda emmagatzemat en la BDOO.

```
Client cli = new client("Client1", "Adreça1",...);
Transaction tr;
try {
  tr = pm.currentTransaction();
  tr.begin();
  pm.makePersistent(cli);
  tr.commit();
  } catch (Exception e) {
    if(tx.isActive()) {
      tx.rollback();
   }
  }
```

El mètode `closeAll()` finalitza la consulta, alliberant els objectes resultants.

Per executar una consulta cal cridar el mètode `execute()` definit en la classe `Query`. Aquest retorna una col·lecció d'objectes resultants. Per exemple, per recuperar objectes `Client` emmagatzemats es faria:

```
try {
  Query query = pm.getQuery(Client.class);
  query.setFilter("(nom == param)");
  query.declareParameters("String param");
  Collection c = (Collection)query.execute("Client1");
  Iterator it = c.iterator();
  while(it.hasNext()) {
    Client cli = (Client)it.next();
    System.out.println("S'ha trobat el client:" + cli);
  }                              closeAll()
  query.closeAll();
} catch (Exception e) {
  ...
}
```

Els mètodes `setFilter` i `declareParameters` permeten establir condicions de cerca. Amb el primer s'estableix la condició d'acord amb certs paràmetres d'entrada (param) que s'usaran en fer la consulta i es compararan amb els camps dels objectes (nom). Amb el segon establim els tipus dels paràmetres d'entrada (String param). En fer execute, s'indiquen els valors dels paràmetres (Client1).