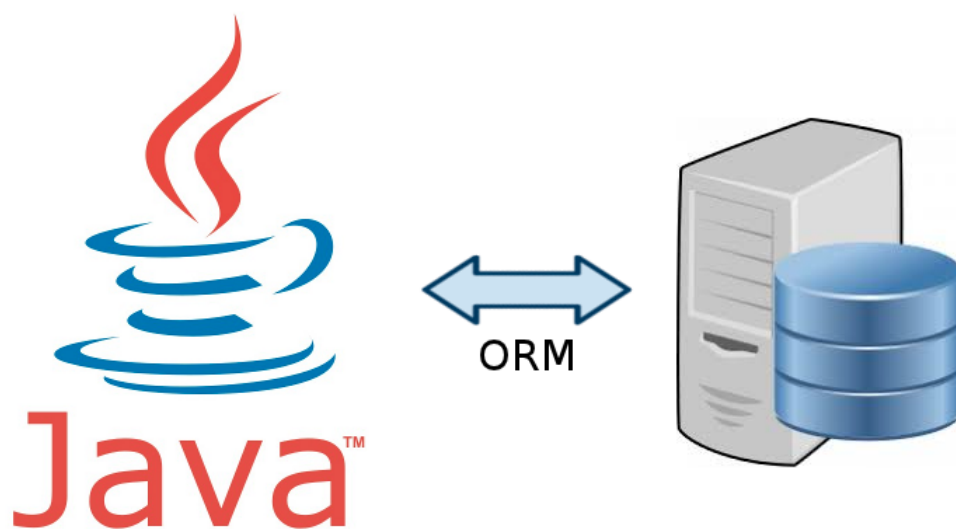

Accés a Dades

Tema 5: Eines de mapatge Objecte-Relacional (ORM)



1 - Introducció

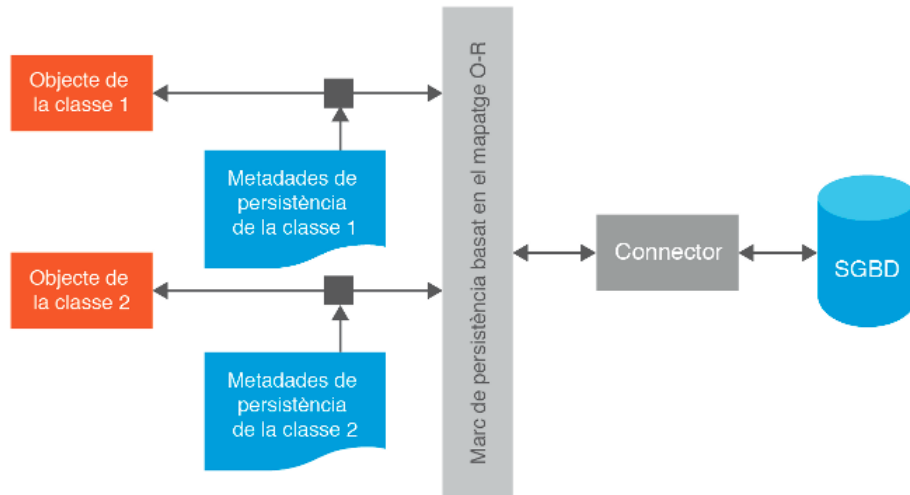
Ja hem vist que a través de JDBC podem connectar fàcilment a Bases de Dades Relacionals. Però tenim el problema del desfasament Objecte-Relacional. Com que en Java utilitzem objectes, ha d'haver una part important de programació en la conversió d'aquests objectes a taules, per a que es puguin guardar de forma permanent. Això ho hem intentat comprovar en l'**exercici 4.4**, destinat a fer la classe **GestionarRutesBD** de manera que siga transparent la persistència de les dades a qui utilitza aquesta classe. Sobre un exemple molt senzill, hem vist que la traducció entre els objectes i únicament dues taules, ja suposava una miqueta de esforç. Podem veure per tant que en una aplicació una miqueta més complicada, fer la traducció entre objectes i taules és una feina que es pot fer, però que costarà prou.

Les eines de **mapatge objecte-relacional (ORM: Object-Relational Mapping)** intenten minimitzar tant com siga possible el desfasament objecte-relacional, automatitzant el procés de traspàs d'un sistema a l'altre.

En certa manera podríem dir que implementen una Base de Dades Orientada a Objectes virtual perquè aporten característiques pròpies del paradigma OO, però el substrat on s'acaben guardant els objectes és un SGBD Relacional.

2 - Concepte de mapatge objecte-relacional

Les eines de mapatge objecte-relacional automatitzen els processos necessaris d'intercanvi de dades entre Sistemes Orientats a Objectes (OO) i Sistemes Relacionals.



Per tant ens ajuden a salvar el desfasament Objecte-Relacional de forma molt còmoda.

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

La descripció més senzilla que han de fer les metadades és la d'establir una correspondència directa entre classes i taules, i a nivell més bàsic entre els atributs de tipus primitius i els camps o columnes. També caldrà identificar l'atribut corresponent al camp que actuarà com a clau primària.

Lamentablement, no sempre es podrà establir aquest tipus de correspondències directes i caldrà que les metadades puguin expressar molta més complexitat. Aquests són alguns dels problemes que ens podem trobar:

- A vegades pot interessar guardar una propietat en més d'una columna o unes quantes propietats en una columna única.
- Pot haver propietats que no es guarden i camps guardats que no es veguen reflectits als objectes.
- Quan els atributs no siguin tipus de dades primitius caldrà saber també si haurem de guardar les dades en una taula diferent o en la mateixa taula i, en cas de que siga en taules diferents, quins atributs haurem de fer servir com a claus externes, qui tindrà la responsabilitat de realitzar l'emmagatzematge, etc.

3 - Eines de mapatge

Actualment hi ha moltes eines de mapatge en el mercat. Existeixen eines de mapatge per a la major part de llenguatges orientats a objectes PHP, Objective-C, C++, SmallTalk i evidentment Java.

Bàsicament són eines en les quals podem distingir tres aspectes conceptuals clarament diferenciats:

- Un sistema per expressar el mapatge entre les classes i l'esquema de la Base de Dades,
- Un llenguatge de consulta orientat a objectes, que en realitat accedirà a les taules, i per tant ens permetrà salvar el desfasament O-R
- Un nucli funcional que possibilita la sincronització dels objectes persistents de l'aplicació amb la Base de Dades.

Tècniques de mapatge

Entre les tècniques que aquestes eines fan servir per plasmar els mapes O-R, destaquem:

- Les que incrusten les definicions dins el codi de les classes
- Les que guarden les definicions en fitxers independents.

Les primeres solen ser tècniques molt vinculades al llenguatge de programació, així per exemple, en C++ se solen fer servir macros i en Java s'utilitzen anotacions.

Les tècniques de definició basades en fitxers independents del codi solen utilitzar el format XML, perquè és un llenguatge molt expressiu i fàcilment extensible.

La major part d'eines accepten les dues tècniques de mapatge anteriors, i fins i tot permeten la convivència dins d'una mateixa aplicació.

- Les tècniques que utilitzen el propi llenguatge de programació per incrustar el mapatge dins el codi presenten una corba d'aprenentatge més baixa per a desenvolupadors experimentats en el llenguatge amfitrió, però per contra no serà possible aprofitar les definicions si es decideix canviar de llenguatge de programació.
- En canvi, fent servir les tècniques basades en XML sí que es poden reutilitzar les definicions per a diferents llenguatges, sempre que l'eina utilitzada estiga disponible en el llenguatge de programació requerit o bé si el format de les definicions segueix alguna especificació estàndard.

A hores d'ara és difícil dir per on s'encaminaran els estàndards. Això sí, la major part d'eines utilitzen una sintaxi prou similar per a expressar les definicions del mapatge, encara que òbviament sempre hi ha diferències que no els fan del tot compatibles.

Llenguatges de consulta

El llenguatge de consulta més utilitzat per la major part d'eines és el llenguatge anomenat **OQL (Object Query Language)** o una variant del mateix. Es tracta d'un llenguatge especificat per l'ODMG. Presenta molta similitud amb SQL, ja que que els dos són llenguatges d'interrogació no procedimental, però l'OQL està totalment orientat a objectes. És a dir, en la consulta fem servir la sintaxi pròpia dels objectes i els resultats obtinguts retornen objectes o col·leccions d'objectes.

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

Tècniques de sincronització

La sincronització amb la base de dades és segurament un dels aspectes més crítics de les eines de mapatge. Solen ser processos molt complexos, on trobem implicades sofisticades tècniques de programació per a poder descobrir els

canvis que van patint els objectes (i així poder guardar-los), a crear i inicialitzar les noves instàncies que calga posar en joc dins l'aplicació d'acord amb les dades guardades o també a extreure la informació dels objectes per revertir-la a les taules del SGBD.

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

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

JPA (Java Persistence API) és també un estàndard de persistència. Es troba incorporat en el `JDK` des de la versió 5. Hi ha diverses biblioteques que el suporten, totes elles basades en el principi de la reflexió. Java és un llenguatge que en compilar les classes incorpora metadades amb informació pròpia de la classe, com ara l'estructura de dades interna, els mètodes que tinga disponibles, els paràmetres necessaris per invocar els mètodes, etc. La màquina virtual permet fer servir aquestes metadades per accedir a la informació real guardada en els objectes, per invocar algun dels seus mètodes, per construir noves instàncies, etc.

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

Malauradament, JDO no ha acabat de quallar i poques iniciatives comercials l'han seguit. Per contra JPA, és la tècnica que han utilitzat els dos gegants fortament implantats (Hibernate i EJB). Tot sembla apuntar que serà l'escollit per cobrir l'estandardització de la persistència en Java.

En aquestos apunts utilitzarem **Hibernate**, que es basa en JPA.

4 - Arquitectura Hibernate

El producte que utilitzarem serà **Hibernate**. Aquesta eina ens permetrà fer el mapatge entre objectes i les taules d'una Base de Dades. Aquestos objectes seran senzills, d'aquells que només tenen les propietats, i mètodes **get** i **set** per a accedir a elles.

Per a intercanviar la informació entre aquestos objectes i la Base de Dades, utilitzarem la classe **Session** (que seria l'equivalent de **Connection** de JDBC). Aquesta classe ens proporciona els mètodes:

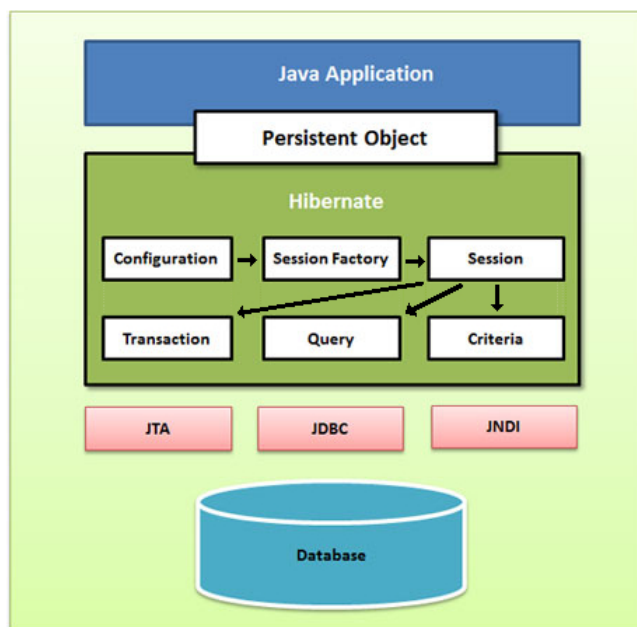
- **save(objecte)** , que guardarà l'objecte en la Base de Dades
- **createQuery(consulta)** , que crearà una consulta
- **beginTransaction()** , que començarà una transacció
- **close()**
- ...

Amb açò, interactuar amb la Base de Dades serà molt senzill, i no farà falta executar sentències SQL, tal i com ens passava en JDBC.

Session no consumeix molta memòria i la seua creació i destrucció és molt barata, cosa que ens pot animar a crear (i destruir) una sessió per a cada accés a la Base de Dades. Ho anirem veient.

El context que ens fa falta per a una aplicació serà:

- **Configuration**. S'utilitza per a configurar Hibernate. L'aplicació utilitza una instància de Configuration per a especificar la ubicació dels documents que descriuen el mapatge i també per a especificar propietats específiques d'Hibernate. A partir de Configuration es crearà el **SessionFactory**
- **SessionFactory**. Permet construir instàncies **Session**. Normalment utilitzarem només un SessionFactory en tota l'aplicació, i d'ell traurem totes les sessions que ens facen falta, a no ser que ens toque accedir a més d'una Base de Dades, on aleshores tindrem una SessionFactory per cada Base de Dades.
- **Session**. Com ja hem comentat representa una connexió a la Base de Dades.
- **Query**. Permet realitzar consultes a la Base de Dades. El llenguatge utilitzat serà **HQL** (Hibernate Query Language) o el SQL del SGBD, però preferiblement utilitzarem el primer. Es construeix a partir d'un **Session**
- **Transaction**. Permet assegurar que es fan totes les actualitzacions o cap d'elles. De pas ens assegura que en cas d'error en mig d'una transacció, no es faça cap de les operacions. També es construeix a partir d'un **Session**



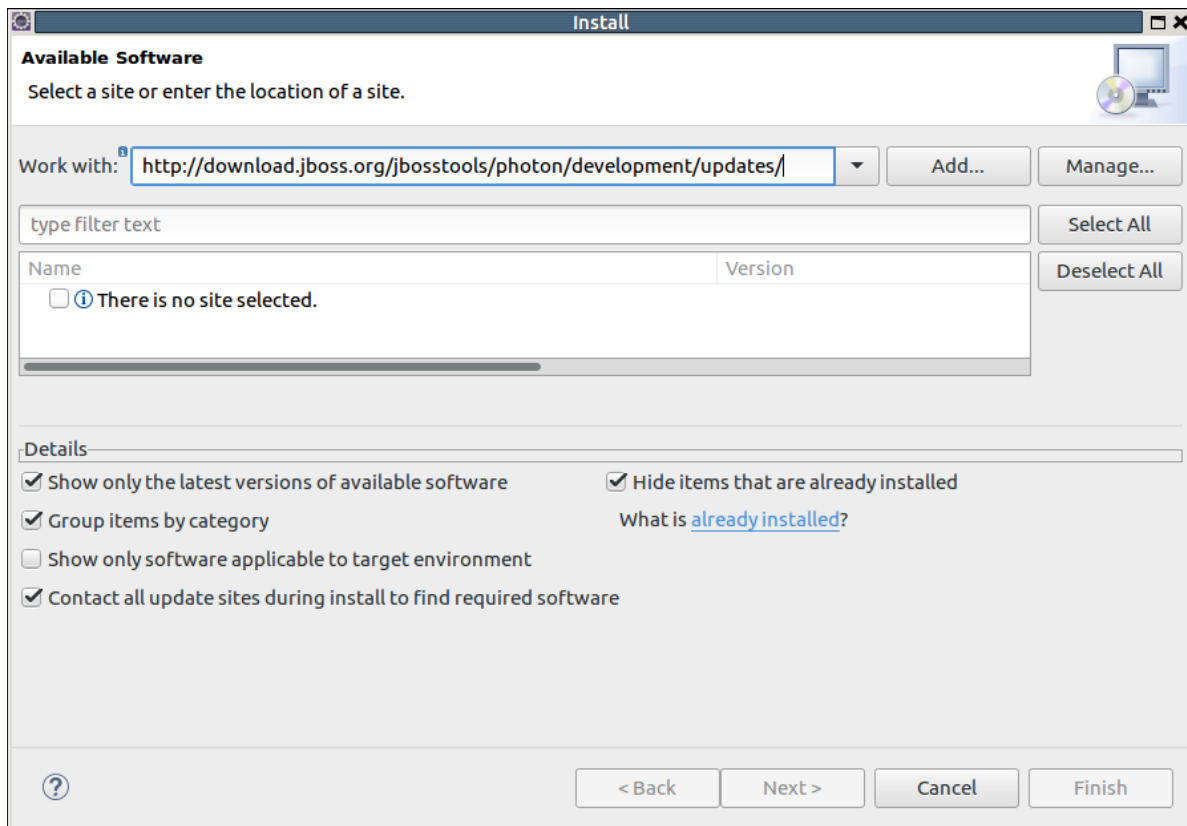
5 - Instal·lació i configuració d'Hibernate

Instal·lació

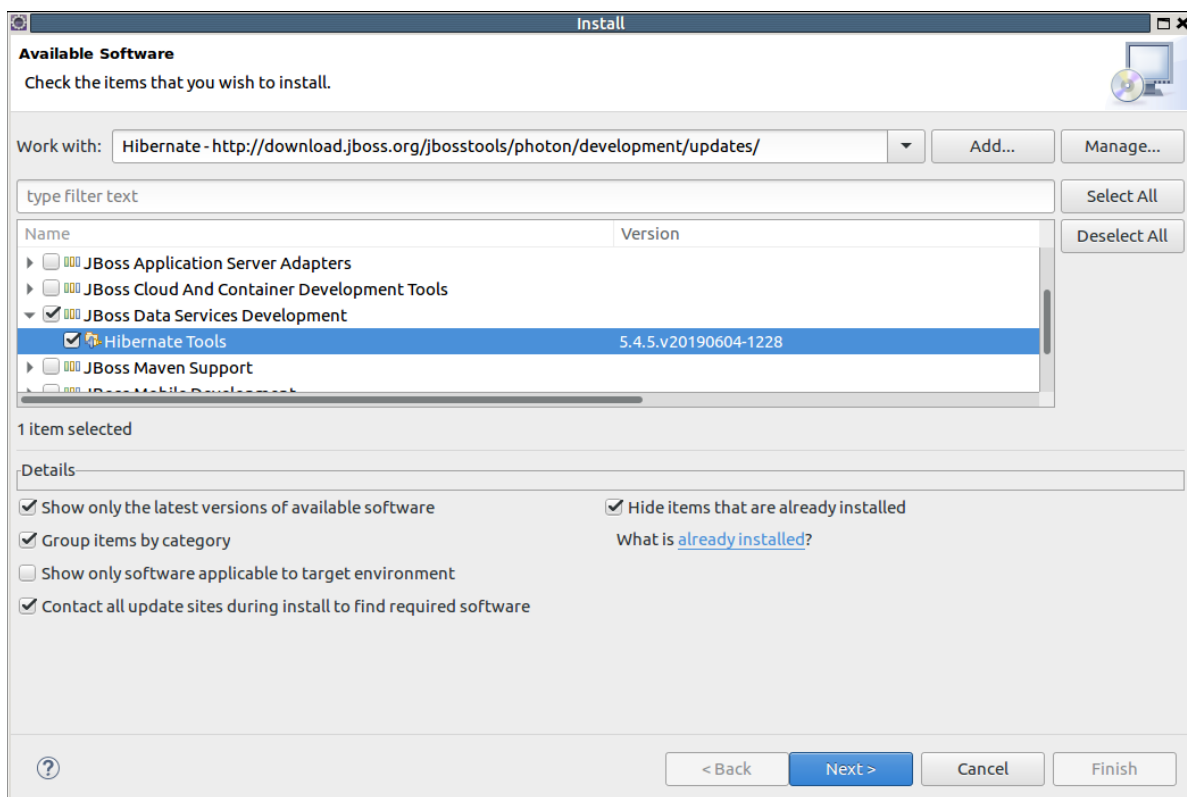
Anem ja a la instal·lació d'Hibernate en l'entorn d'Eclipse. En el moment de fer aquestos apunts, l'última versió disponible d'Eclipse és **2019-09**. Tanmateix, el següent vídeo és sobre la versió Neon d'Eclipse, prou anterior a l'actual, però no ofereix cap variació substancial. L'única diferència és que hi ha versions en les quals és suficient amb arrastrar el botó **Install** de la versió (imatges 2 a 4 del següent vídeo). En d'altres, com la **2019-06**, s'ha d'anar a **Help -> Install New Software**, i treballar amb l'adreça que ens dona en <http://tools.jboss.org/downloads>



Com déiem, copiarem l'adreça que ens dona l'anterior pàgina en **Help -> Install New Software** en el quadre de text on diu **Work with** i l'afegim.



Una vegada ha analitzat les utilitats de l'adreça ens deixa triar quines volem instal·lar. La que ens interessa és **Hibernate Tools**, que està per exemple dins de **JBoss Data Services Development**



La resta de la instal·lació és idèntica a versions anteriors

En el següent vídeo està tot el procés d'instal·lació de la versió corresponent a Eclipse Neon

La instal·lació d'una versió anterior, en cas que treballem amb una versió diferent d'Eclipse, serà igual de senzilla. Només haurem d'anar a la pàgina de **All downloads** (<http://tools.jboss.org/downloads/overview.html>), on estan les versions d'Hibernate per a cada versió d'Eclipse. Millor si triem les versions estables. Ens hem de fixar en la columna de **JBoss Tools**. I si no està la que volem, podem anar al final de tot a: **Look for an older version? See the archives**

HomeDownloadsFeaturesBlogDocumentationCommunityGet InvolvedFollow Us

Home / Downloads / Recent Downloads

Recent Downloads

Eclipse Version	JBoss Developer Studio	JBoss Developer Studio Integration Stack	JBoss Tools	JBoss Tools Integration Stack
Eclipse Oxygen 4.7	11.0.0.GA 11.x.Nightly	11.0.0.GA	4.5.x.Nightly 4.5.1.AM3 4.5.1.AM2 4.5.0.Final 4.5.0.AM2 4.5.0.AM1	4.5.0.Final 4.5.0.AM1
Eclipse Neon 4.6.3	10.4.0.GA	10.3.0.GA	4.4.4.Final	4.4.3.Final
Eclipse Mars 4.5.2	9.1.0.GA	9.0.3.GA	4.3.1.Final	4.3.3.Final
Eclipse Luna 4.4.2	8.1.0.GA	8.0.7.GA	4.2.3.Final	4.2.7.Final

Looking for an older version ? See the archives.

Configuració

En aquest tema, en contra del criteri seguit en els temes anteriors, ens construirem un **projecte nou** per a cada connexió a cada Base de Dades. Aquest projecte el configurarem per a Hibernate creant el fitxer **Hibernate Configuration File (cfg.xml)**. Aquest fitxer contindrà tot el necessari per a realitzar la connexió a la Base de Dades que volem connectar.

El fitxer **cfg.xml** es crea amb l'opció **New -> Other -> Hibernate -> Hibernate Configuration File (cfg.xml)**.

Després del fitxer de configuració, també crearem la **Consola de configuració**.

La primera connexió la realitzarem a **PostgreSQL**, a la Base de Dades de prova que estem utilitzant: **geo_ad**. El nom que posarem al projecte és **Tema5_Hibernate_PostgreSQL_geo_ad**, un nom molt llarg, però que ens diu perfectament el que volem fer. El següent vídeo il·lustra tot el procés:

Les dades de connexió a PostgreSQL són aquestes (imatges 9 a 14 del vídeo anterior). Cuideu de triar la versió **5.2** d'Hibernate. Les posteriors en el moments de fer aquestos apunts (5.3 i 5.4) no funcionaven bé

Hibernate Configuration File (cfg.xml)
This wizard creates a new configuration file to use with Hibernate.

Container: /Tema5_Hibernate_PostgreSQL_geo_ad/src
File name: hibernate.cfg.xml
Hibernate version: 5.2
Session factory name: ConnexioPostgreSQL_geo_ad

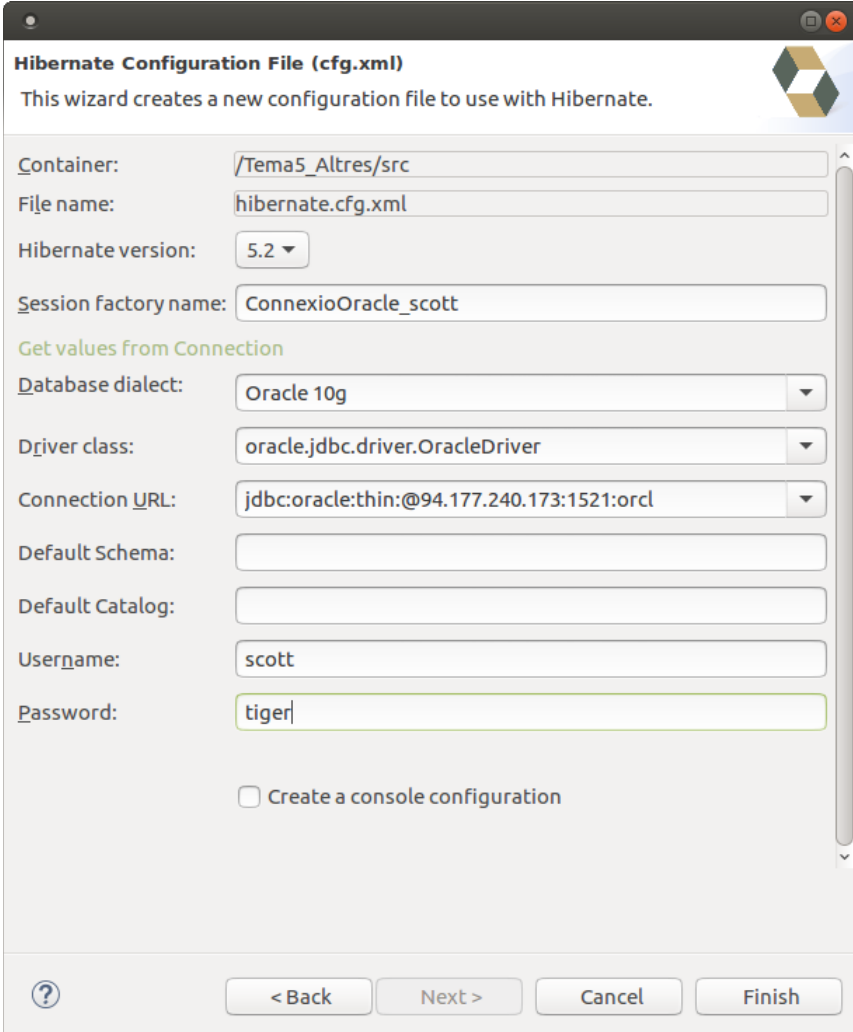
Get values from Connection

Database dialect: PostgreSQL
Driver class: org.postgresql.Driver
Connection URL: jdbc:postgresql://89.36.214.106:5432/geo_ad
Default Schema: public
Default Catalog:
Username: geo_ad
Password: geo_ad

☐ Create a console configuration

< Back Next > Cancel Finish

Si la connexió la férem a Oracle, les dades de connexió variarien lleugerament:



Hibernate Configuration File (cfg.xml)
This wizard creates a new configuration file to use with Hibernate.

Container: /Tema5_Altres/src
File name: hibernate.cfg.xml
Hibernate version: 5.2
Session factory name: ConnexioOracle_scott

Get values from Connection

Database dialect: Oracle 10g
Driver class: oracle.jdbc.driver.OracleDriver
Connection URL: jdbc:oracle:thin:@94.177.240.173:1521:orcl
Default Schema:
Default Catalog:
Username: scott
Password: tiger

☐ Create a console configuration

? < Back Next > Cancel Finish

I també ho farien per a MySQL:

Hibernate Configuration File (cfg.xml)
This wizard creates a new configuration file to use with Hibernate.

Container: /Tema5_Altres/src
File name: hibernate.cfg.xml
Hibernate version: 5.2
Session factory name: ConnexioMySQL_geo

Get values from Connection

Database dialect: MySQL
Driver class: com.mysql.jdbc.Driver
Connection URL: jdbc:mysql://89.36.214.106/geo
Default Schema:
Default Catalog:
Username: geo
Password: geo

☐ Create a console configuration

? < Back Next > Cancel Finish

No hi ha la possibilitat de connectar amb SQLite, ja que no és un servidor

Nota

Recordeu que heu de triar la **versió 5.2 d'Hibernate** en les pantalles anteriors de crear el fitxer de configuració **Hibernate Configuration File (cfg.xml)**. I també en la creació de la consola

5.1 - Generar les classes a partir de la Base de Dades

Una vegada configurat Hibernacle per al projecte, el següent pas ha de ser **generar** les classes de la Base de Dades. S'anomena procés d'enginyeria inversa, perquè a partir de les taules d'una Base de Dades generarà automàticament les classes de Java que es corresponen a les taules.

Per a això apretarem a la fletxeta del costat del botó nou que s'havia creat en instal·lar Hibernate, el de **Run As** ..., per a triar l'opció **Hibernate Code Generation Configurations**

El següent vídeo explica tot el procés:

5.2 - Perspectiva d'Hibernate

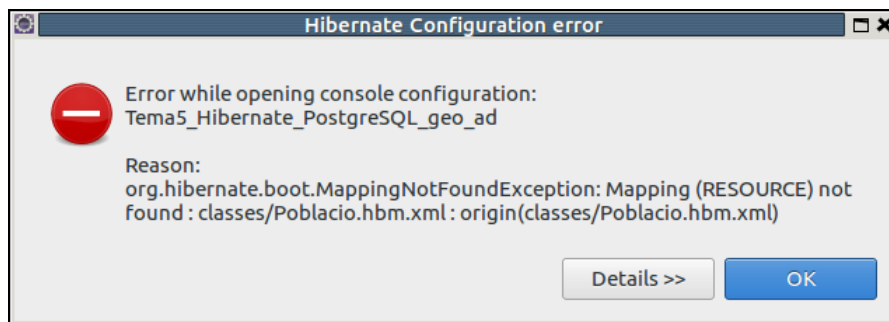
Hibernate ens proporciona una perspectiva pròpia que ens serà molt útil per a inspeccionar les classes generades. Podrem veure les propietats de cada classe, també els camps de cada taula, i també la correspondència entre propietats dels objectes i camps de les taules de forma gràfica. I també ens permetrà inspeccionar les dades.

Canviem a la perspectiva d'Hibernate del menú **Window -> Open Perspective -> Other... -> Hibernate**

A l'esquerra ens eixirà la consola generada a partir del fitxer de configuració (també la podríem generar ara). Quan la despleguem, podrem analitzar els objectes als quals ens dóna accés Hibernate.

NOTA IMPORTANT

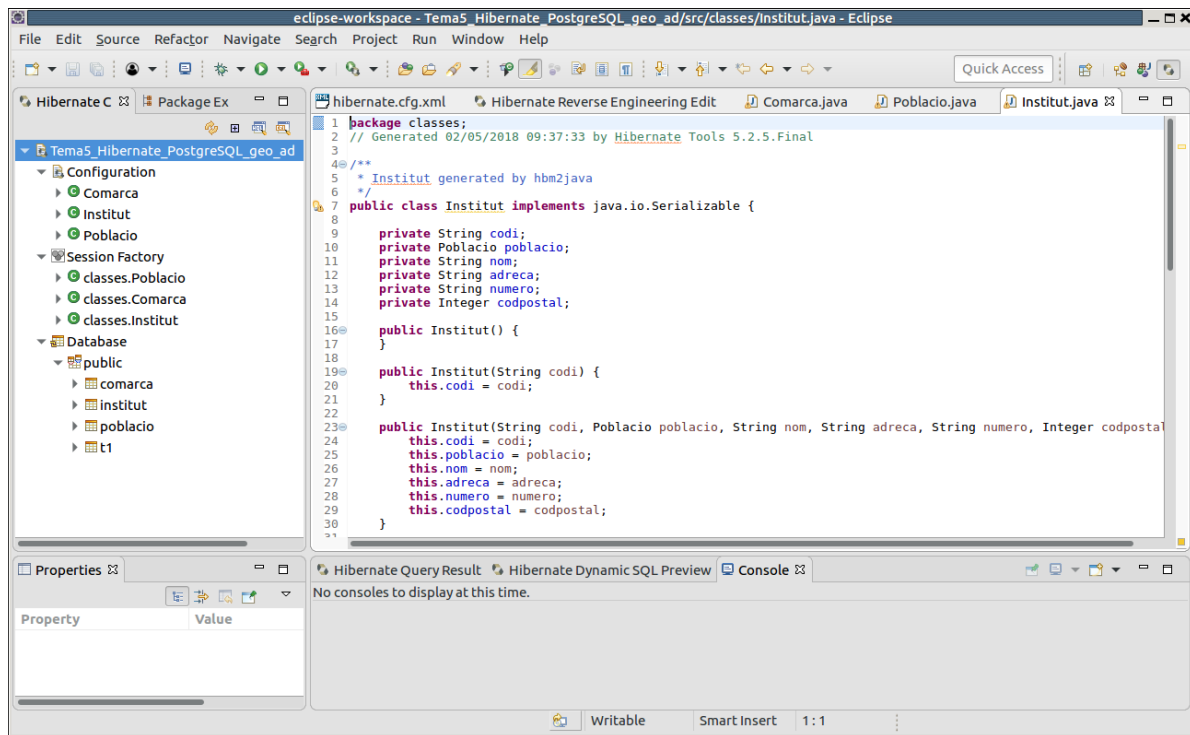
En la instal·lació d'Hibernate sobre Eclipse 2019-06 dóna un error en la perspectiva d'Hibernate immediatament després de generar les classes, marcant com que no troba els fitxers de mapatge



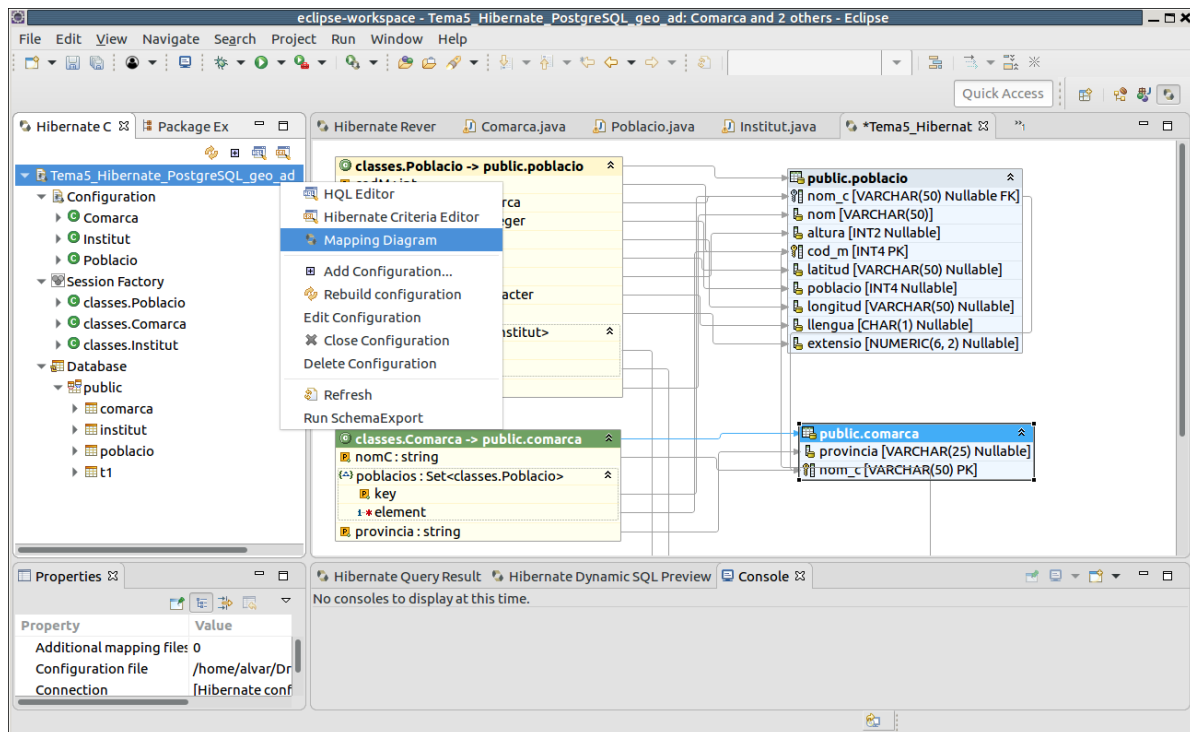
Tanmateix ho podem solucionar senzillament reconstruint el projecte: **Project -> Build All**

En aquesta perspectiva tindrem 4 grans zones:

- Esquerra: navegador d'objectes, on podrem inspeccionar les classes generades (en **Configuration** i **SessionFactory**), i les taules de la Base de Dades en **Database**
- Centre: on s'obren les classes, fitxers de configuració i també les consultes que farem (ho veurem en el següent punt)
- Baix centre: a banda de la consola, quan executem una consulta ací es veuran els resultats
- Baix esquerra: Propietats de l'objecte seleccionat



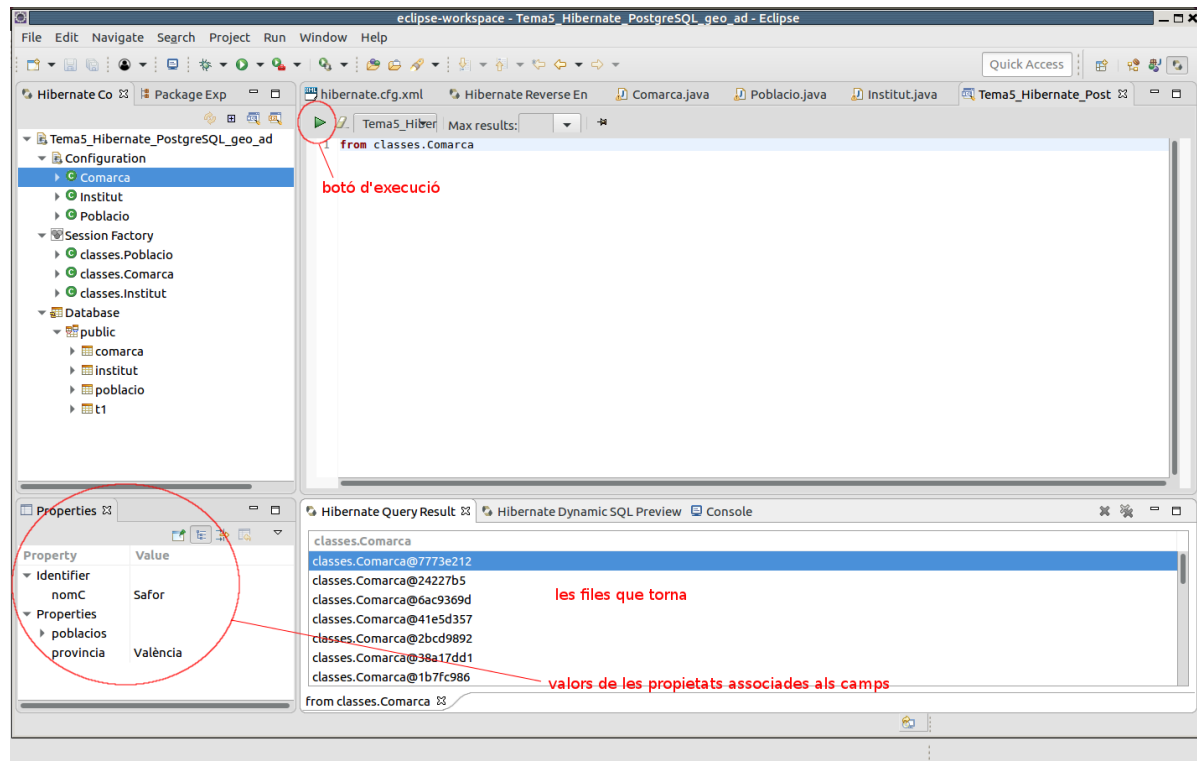
Podem veure gràficament tota la correspondència entre les taules i les classes des de la **perspectiva Hibernate**. Tenim prou amb apretar amb el botó de la dreta sobre la **Configuració** i triar **Mapping Diagram**:



5.3 - HQL: consultes senzilles

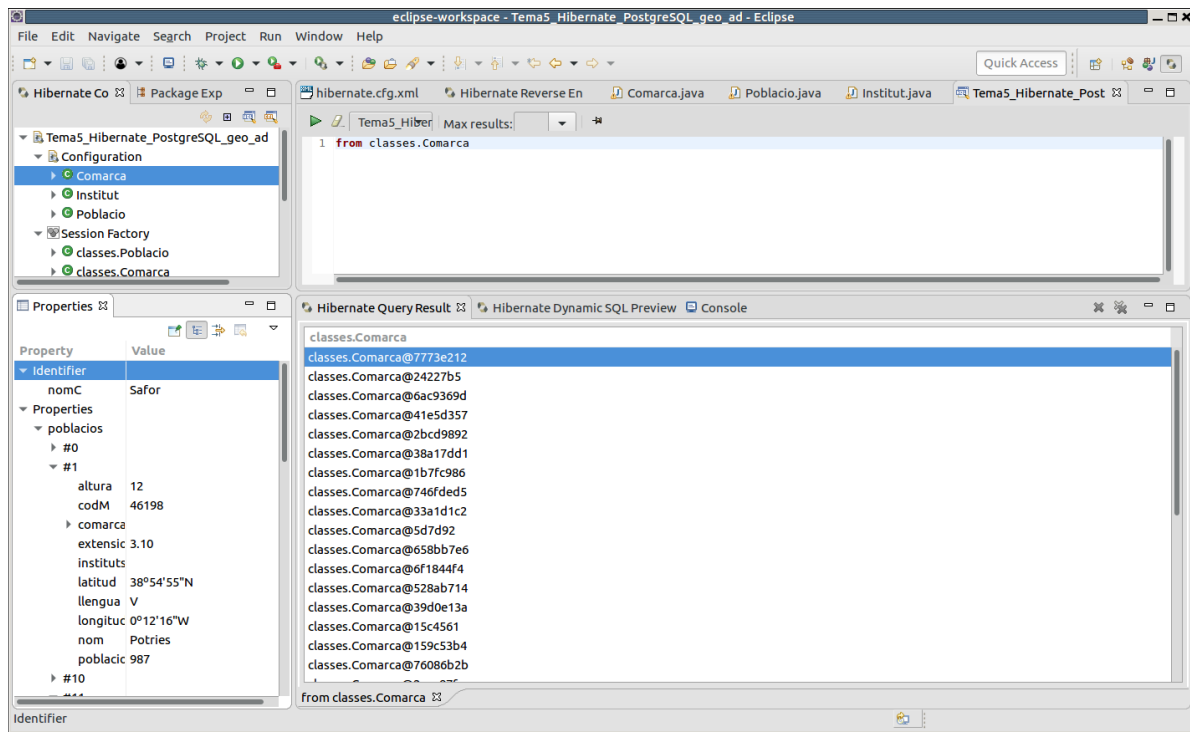
Com havíem comentat, des de la perspectiva Hibernate podrem veure les dades. No podrem veure directament el contingut de les taules, però sí que podrem fer sentències HQL per veure el contingut. Apretem amb el botó de la dreta sobre la configuració de la consola i triem **HQL Editor**.

Sobre la finestra podrem escriure les sentències (si apremem al botó de la dreta sobre una classe ens suggerirà la sentència HQL per a consultar-la). La primera podria ser aquesta **from Comarques**. Hem de respectar escrupolosament majúscules i minúscules, ja que en definitiva estem accedint a les classes Java.

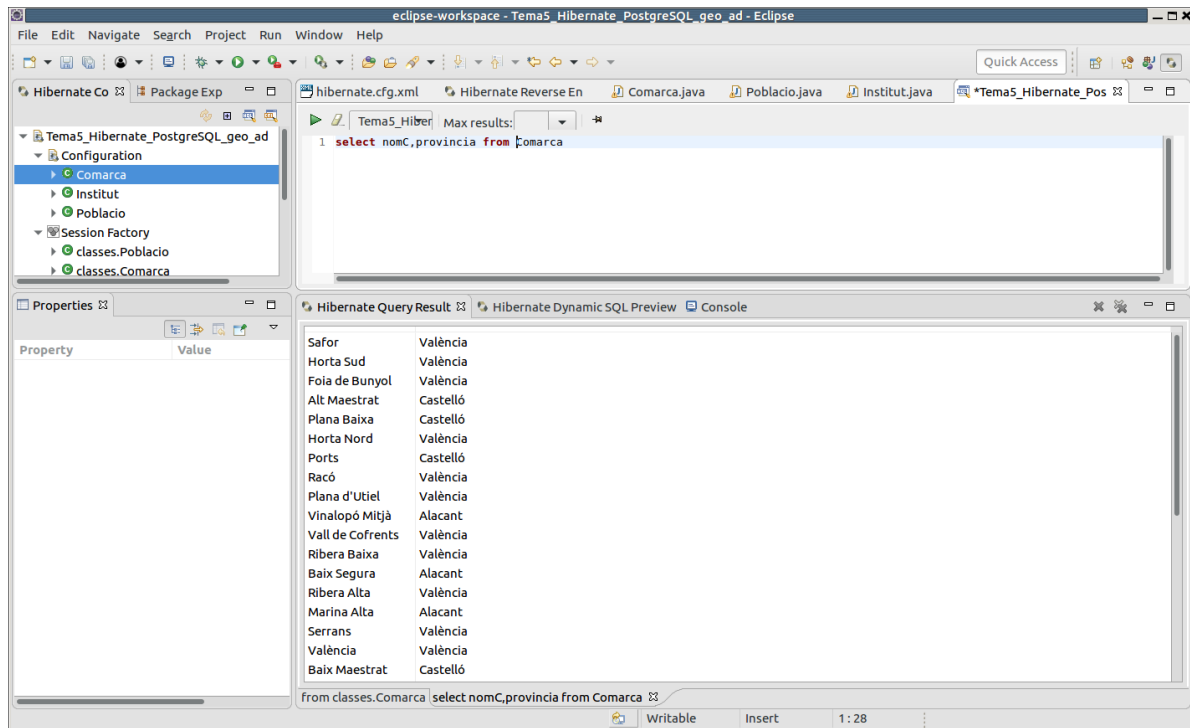


Com podem comprovar torna les files d'una forma un poc críptica, ja que en l'apartat de resultats només ens mostra els objectes (les referències als objectes). Però si seleccionem una determinada fila podrem veure a l'esquerra, en l'apartat de Propietats, el valor de cada camp, millor dit, de cada propietat (associada a un camp).

I comencem a veure el potencial de les propietats. Què fa quan troba una clau externa? A les classes generades havia posat un **Set** (conjunt) amb tots els objectes relacionats, que en aquest cas són totes les poblacions de la comarca. Si despleguem la propietat **poblacions** podrem veure totes les poblacions de la comarca seleccionada. Com veieu, molt útil, ja que a banda de que a partir de la població podem veure la seua comarca, també a partir de la comarca podem veure totes les seues poblacions.



Si la consulta fóra més completa, especificant els camps, ens eixiran directament en l'apartat de resultats. Observeu com se sembla a SQL : **select nomC,provincia from Comarques**



5.4 - Començant a programar

Per a poder començar a programar haurem d'incorporar les llibreries d'Hibernate al nostre projecte.

Com que en són prou, el que ens convé és col·locar-les totes en una llibreria nostra, creada per nosaltres. Així, per als propers projectes, només haurem d'incorporar aquesta llibreria creada per nosaltres.

També ens eixirà alguns avisos. Aquestos avisos, ens els llevarem de damunt per a que no molesten.

En el següent vídeo teniu el procés:

Depenent de la versió d'Hibernate, ens haurà tocat baixar-nos directament un jar, concretament el **ANTLR (*ANother Tool for Language Recognition*)** que serveix per a fer una primera anàlisi a les sentències HQL. Concretament a hores d'ara ens anirà bé la versió 3. Ací teniu l'enllaç al plugin: [antlr-3.5.2-complete.jar](#)

El programeta que hem utilitzat en el vídeo anterior està en un paquet nou, **Exemples**, i és:

```
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

import classes.Comarca;

public class PrimerAcces {

    public static void main(String[] args) {
        SessionFactory sf = new
Configuration().configure().buildSessionFactory();
        Session sessio = sf.openSession();
        Comarca com = (Comarca) sessio.load(Comarca.class, "Alt Maestrat");
    }
}
```

```
        System.out.print("Comarca " + com.getNomC() + ": ");  
        System.out.print(com.getProvincia());  
        System.out.println(" (" + com.getPoblacions().size() + " pobles)");  
        sessio.close();  
    }  
}
```

Nota

El mètode **buildSessionFactory** del mètode **configure()** de **Configuration** segurament el marcarà com a **deprecated**. De moment el deixarem d'aquesta manera per senzillesa. Ho arreglarem en la següent pregunta.

Com déiem, els avisos ens els hem llevat de damunt fent que es guarden en un fitxer. Per a això hem creat el fitxer **log4j.properties** dins de **src** amb el següent contingut:

```
# Root logger option  
log4j.rootLogger=INFO, file  
  
# Direct log messages to a log file  
log4j.appender.file=org.apache.log4j.RollingFileAppender  
log4j.appender.file.File=logging.log  
log4j.appender.file.MaxFileSize=1MB  
log4j.appender.file.MaxBackupIndex=1  
log4j.appender.file.layout=org.apache.log4j.PatternLayout  
log4j.appender.file.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %-5p  
%c{1}:%L - %m%n
```

I també notareu que ha anat una miqueta lent, ha tardat prou en traure el resultat. També ho arreglarem en la següent pregunta.

5.5 - SessionFactory únic: Singleton

En la pregunta 4 havíem comentat que si només accedíem a una Base de Dades, havíem de crear només un **SessionFactory**, i a través d'ell crear-nos totes les sessions que ens facen falta.

Per a assegurar-nos que només creem un objecte de SessionFactory anem a utilitzar la tècnica del **Singleton**. És una tècnica utilitzada en programació quan volem assegurar-nos que només instanciem un objecte d'una classe. Consistirà en crear una classe diferent que és la que crearà la instància de l'objecte. La primera vegada que es crida a aquesta classe crearà l'objecte, i quan ja estiga creat senzillament tornarà una referència a l'objecte creat.

Aprofitarem també per a arreglar el **deprecated** que teníem en el **buildSessionFactory**. El mètode que no està deprecated és **buildSessionFactory(ServiceRegistry)**

I també aprofitem per a que no costé tant la connexió. La raó de tardar tant és perquè per defecte vol carregar les metadades de la Base de Dades. Però en PostgreSQL en principi es poden veure totes les altres Bases de Dades que hi ha al servidor, encara que després no puguem connectar. Però igual vol carregar les metadades i tarda molt en intentar (i no poder) connectar. La manera de solucionar-ho serà dir-li que no carregue les metadades. Ho aconseguirem amb la sentència:

```
conf.setProperty("hibernate.temp.use_jdbc_metadata_defaults","false");
```

Ací tenim la classe que ens ajudarà a fer el singleton. L'anomenarem **SessioFactoryUtil**, i el podem col·locar també en el paquet **classes**:

```
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;
import org.hibernate.service.ServiceRegistry;
import org.hibernate.service.ServiceRegistryBuilder;

public class SessionFactoryUtil {

    private static SessionFactory sF;

    public static SessionFactory getSessionFactory() {
        if (sF == null) {
            Configuration conf = new Configuration().configure();

            conf.setProperty("hibernate.temp.use_jdbc_metadata_defaults","false");
            ServiceRegistryBuilder reg = new ServiceRegistryBuilder();
            reg.applySettings(conf.getProperties());
            ServiceRegistry serviceRegistry = reg.buildServiceRegistry();

            sF = conf.buildSessionFactory(serviceRegistry);
        }

        return sF;
    }
}
```

Des de fora sempre cridarem a **getSessionFactory** (i ens assurem perquè l'objecte és privat). La primera vegada, com és nul, el crearà. Però les següents vegades únicament tornarà la referència a l'objecte ja creat. A continuació mostrem el mateix exemple de l'apartat anterior, però ara utilitzant el Singleton:

```
import org.hibernate.Session;
import org.hibernate.SessionFactory;

import classes.Comarca;
import classes.SessionFactoryUtil;

public class SegonAcces {

    public static void main(String[] args) {

        SessionFactory sf = SessionFactoryUtil.getSessionFactory();
        Session sessio = sf.openSession();
        Comarca com = (Comarca) sessio.load(Comarca.class, "Alt Maestrat");
        System.out.print("Comarca " + com.getNomC() + ": ");
        System.out.print(com.getProvincia());
        System.out.println(" (" + com.getPoblacions().size() + " pobles)");
        sessio.close();
    }
}
```

}	}
---	---

6 - Classes generades

Comentem les classe que ha generat automàticament Hibernate. Posarà una propietat per cada camp de la taula, i alguna cosa més com veurem més endavant. També posarà un mètode **get** i un **set** per a cada propietat, ja que les propietats seran **private**.

També posa tres constructors: un sense paràmetres, un altre només amb la clau principal (i segurament els camps no nuls), i un altre complet.

A continuació mostrem 2 de les classes. En el cas de **Poblacio** no hem posat tots els mètodes **get** i **set**, ja que sempre són com cabria esperar. I en roig teniu el més destacable.

Comarca.java

```
package classes;

import java.util.HashSet;
import java.util.Set;

public class Comarca implements
java.io.Serializable {

    private String nomC;
    private String provincia;
    private Set<Poblacio> poblacios =
new HashSet<Poblacio>(0);

    public Comarca() {
    }

    public Comarca(String nomC) {
        this.nomC = nomC;
    }

    public Comarca(String nomC, String
provincia, Set<Poblacio> poblacios) {
        this.nomC = nomC;
        this.provincia = provincia;
        this.poblacios = poblacios;
    }

    public String getNomC() {
        return this.nomC;
    }

    public void setNomC(String nomC) {
        this.nomC = nomC;
    }

    public String getProvincia() {
        return this.provincia;
    }

    public void setProvincia(String
provincia) {
        this.provincia = provincia;
    }

    public Set<Poblacio>
getPoblacios() {
        return this.poblacios;
    }

    public void
setPoblacios(Set<Poblacio> poblacios)
{
        this.poblacios = poblacios;
    }
}
```

Poblacio.java

```
package classes;
// Generated 29/04/2018 13:28:33 by Hibernate
Tools 5.2.5.Final

import java.math.BigDecimal;
import java.util.HashSet;
import java.util.Set;

/**
 * Poblacio generated by hbm2java
 */
public class Poblacio implements
java.io.Serializable {

    private int codM;
    private Comarca comarca;
    private String nom;
    private Integer poblacio;
    private BigDecimal extensio;
    private Short altura;
    private String longitud;
    private String latitud;
    private Character llengua;
    private Set<Institut> instituts = new
HashSet<Institut>(0);

    public Poblacio() {
    }

    public Poblacio(int codM, String nom) {
        this.codM = codM;
        this.nom = nom;
    }

    public Poblacio(int codM, Comarca comarca,
String nom, Integer poblacio, BigDecimal
extensio, Short altura,
        String longitud, String latitud,
Character llengua, Set<Institut> instituts) {
        this.codM = codM;
        this.comarca = comarca;
        this.nom = nom;
        this.poblacio = poblacio;
        this.extensio = extensio;
        this.altura = altura;
        this.longitud = longitud;
        this.latitud = latitud;
        this.llengua = llengua;
        this.instituts = instituts;
    }

    ...

    public Comarca getComarca() {
        return this.comarca;
    }

    public void setComarca(Comarca comarca) {
        this.comarca = comarca;
    }

    ...
}
```

```
    public Set<Institut> getInstituts() {  
        return this.instituts;  
    }  
  
    public void setInstituts(Set<Institut>  
instituts) {  
        this.instituts = instituts;  
    }  
}
```

Observeu com els mètodes **get** i **set** sempre comencen per aquesta paraula seguida del nom de la propietat amb la inicial en majúscula. Per tant, Hibernate sabrà quin mètode gastar per a cada propietat

Sens dubte, el més destacable de la classe **Comarca** és que en les propietats, posa també un **conjunt d'objectes Poblacio**. Açò ens permetrà accedir comodíssimament a les poblacions d'una determinada comarca. Aquest conjunt sempre l'anomena com la classe a què fa referència, però en plural (en el nostre cas **poblacions**). Un conjunt (**Set**) té un comportament relativament senzill per a accedir als seus elements.

En la classe **Poblacio**, tindrem les propietats que cabria esperar. I la referència a la comarca és un objecte de la classe **Comarca**. També tindrà un conjunt d'objectes **Institut**.

Per tant, i en resum, una cosa molt útil que fa Hibernate quan troba una clau externa, és posar en la classe corresponent a la taula on està la clau externa una propietat de la classe corresponent a la taula principal (cosa que en principi cabria esperar); però també en la classe corresponent a la taula principal, un conjunt d'objectes de la classe corresponent a l'altra taula, la qual cosa ens permetrà accedir fàcilment.

No en tots els SGBD el comportament serà tan bo. Sí que ho farà en **Oracle**, en **PostgreSQL**, i segurament en **MySQL**. En canvi no ho podrà fer en **SQLite**, en els qual només hi haurà propietats de tipus senzills per cadascun dels camps, i prou. Però com de moment tampoc podem connectar amb SQLite per mig d'Hibernate ...

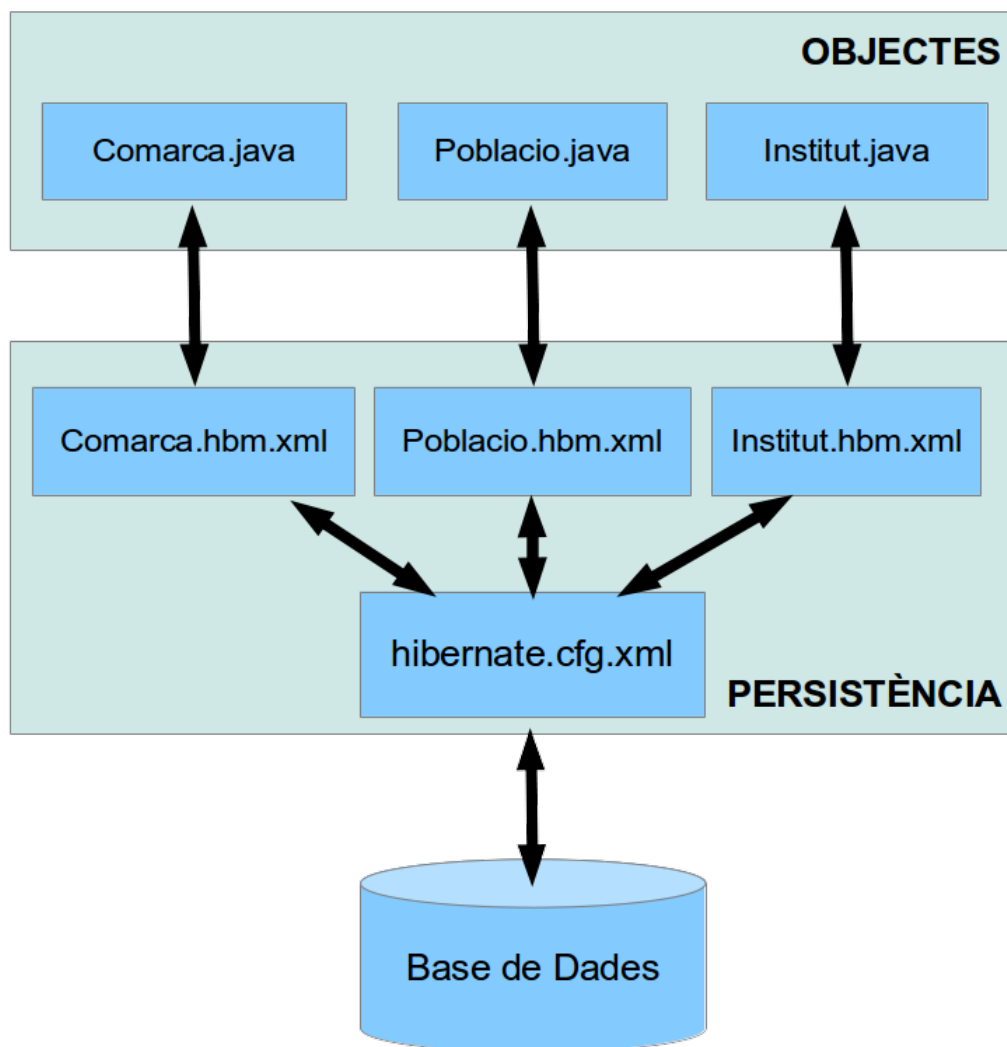
7 - Estructura dels fitxers de mapatge

En la pregunta 5.1 havíem vist les classes generades per Hibernate, i també havíem vist una imatge amb el diagrama de mapatge en la pregunta 5.2. Anem a veure-ho ara amb més detall, mirant totes les classes i fitxers generades en el procés de mapatge i com s'interrelacionen entre elles.

Els fitxers que serveixen de mapatge, Hibernate els genera de tipus **XML**, i són:

- **hibernate.cfg.xml**, ja comentat anteriorment i que estableix les condicions bàsiques de la connexió amb la Base de Dades (Driver, URL, usuari, contrasenya, ...).
- Un fitxer **.hbm.xml** per cada taula (i classe). Així en el nostre exemple tindrem **Comarques.hbm.xml**, **Poblacions.hbm.xml** i **Instituts.hbm.xml** i són els encarregats de dir la correspondència entre els camps de la taula (p.e. COMARQUES) i les propietats de l'objecte (p.e. Comarques.java).

La següent imatge explica aquesta manera d'enllaçar:



El fitxer hibernate.cfg.xml

El fitxer **hibernate.cfg.xml** estableix les condicions de la connexió:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
```

```

<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration
DTD 3.0//EN" "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property
name="hibernate.bytecode.use_reflection_optimizer">false</property>
    <property
name="hibernate.connection.driver_class">org.postgresql.Driver</property>
    <property name="hibernate.connection.password">geo_ad</property>
    <property
name="hibernate.connection.url">jdbc:postgresql://89.36.214.106:5432
/geo_ad</property>
    <property name="hibernate.connection.username">geo_ad</property>
    <property name="hibernate.default_schema">public</property>
    <property
name="hibernate.dialect">org.hibernate.dialect.PostgreSQLDialect</property>
    <property
name="hibernate.search.autoregister_listeners">true</property>
    <property name="hibernate.validator.apply_to_ddl">false</property>
    <mapping resource="classes/Poblacio.hbm.xml"/>
    <mapping resource="classes/Comarca.hbm.xml"/>
    <mapping resource="classes/Institut.hbm.xml"/>
  </session-factory>
</hibernate-configuration>

```

Podem observar com s'especifica el driver JDBC, la URL, l'usuari que es connecta, la contrasenya, l'esquema (que dependrà del SGBD; en altres farà falta el catàleg).

Els fitxers .hbm.xml

Els fitxers **.hbm.xml** estableixen la correspondència entre taula i classe, i dins d'elles entre camps de la taula i propietats de la classe. Mirem el contingut de **Comarques.hbm.xml**. Posem a un costat l'estructura de la taula, i a l'altre l'estructura de l'objecte:

Taula COMARCA	Comarca.hbm.xml	Comarca.java
<pre> TABLE COMARCA (nom_c VARCHAR(50) PRIMARY KEY, provincia VARCHAR(25)); </pre>	<pre> <?xml version="1.0" encoding="UTF-8" standalone="no"?> <hibernate-mapping auto-import="true" default- access="property" default-cascade="none" default- lazy="true"> <class dynamic-insert="false" dynamic-update="false" mutable="true" name="classes.Comarca" optimistic- lock="version" polymorphism="implicit" select-before- update="false" table="comarca"> <id name="nomC" type="string"> <column length="50" name="nom_c"/> <generator class="assigned"/> </id> <property generated="never" lazy="false" name="provincia" optimistic-lock="true" type="string" unique="false"> <column length="25" name="provincia"/> </property> <set embed-xml="true" fetch="select" inverse="true" lazy="true" mutable="true" name="poblacions" optimistic-lock="true" sort="unsorted" table="poblacio"> <key on-delete="noaction"> <column length="50" name="nom_c"/> </key> <one-to-many class="classes.Poblacio" embed- xml="true" not-found="exception"/> </set> </class> </hibernate-mapping> </pre>	<pre> package classes; import java.util.HashSet; import java.util.Set; public class Comarca implements java.io.Serializable { private static final long serialVersionUID = 1L; private String nomC; private String provincia; private Set<Poblacio> poblacions = new HashSet<Poblacio>(0); public Comarques() { } ... } </pre>

Comentem un poc les coses:

- **<hibernate-mapping>** és l'element arrel; tot el mapatge està contingut dins d'ell.
- **<class>** Equival a tota la classe per una banda i tota la taula per una altra. Dins d'aquest element estaran totes les correspondències de camps i propietats.
 - L'atribut **name** conté el nom de la classe.
 - L'atribut **table** conté el nom de la taula corresponent.
 - Pot haver algun altre atribut, indicant el catàleg (depèn del SGBD)
- **<id>** especifica el camp que és clau principal. És diferent als camps normals.
 - L'atribut **name** indica el nom de la propietat
 - L'atribut **type** el tipus de dades.
 - L'element **column** és un element buit amb la propietat **name**, que contindrà el camp de la taula
 - L'element **generator** és un element buit amb un atribut **class** indicant la manera de generar aquesta clau principal. Els valors possibles són:
 - **assigned** que indica que l'usuari ha d'assignar un valor a aquesta propietat
 - **increment** que indica que la Base de Dades proporciona el valor autogenerat
- **<property>** especifica un camp normal, que no és clau principal. Només s'ha d'especificar:
 - L'atribut **name** indica el nom de la propietat
 - L'atribut **type** el tipus de dades.
 - L'element **column** és un element buit amb la propietat **name**, que contindrà el camp de la taula

Els tipus de dades no són ni de la Base de Dades ni tampoc Java. Són tipus anomenats **tipus de mapatge Hibernate**, i s'encarrega Hibernate de fer la correspondència entre tipus Java i de la Base de Dades de forma automàtica.

I ens deixem per al final la manera d'enllaçar la clau externa. Recordem que la clau externa està definida en **POBLACIO** i que apunta a **COMARCA**. En el mapatge de **Comarca** tindrem una línia nova que és la que indica que la propietat **poblacions** ha de contenir les poblacions de la comarca. És per tant un conjunt (**set**) de moltes poblacions. Les coses més importants són les següents:

- **<set>** que indica que serà un conjunt
 - L'atribut **name** serà el nom de la propietat
 - L'atribut **table** el nom de la taula on ha de buscar les dades, és a dir la que té la clau externa (en aquest cas **POBLACIO**)
 - L'atribut **order-by** ens permet que les dades vinguin ordenades
 - L'element **key** especifica la clau externa, amb l'element **column** i atribut **name**, en aquest cas **nom_c**.
 - L'element **one-to-many** indica que un departament pot tenir molts empleats (per això era un conjunt), i especifica de quina classe són amb l'atribut **class**

Fem especial menció a l'atribut **order-by**, que ens permet que les dades vinguin ordenades. Ho aplicarem en la pregunta 8.3.

En el cas de les poblacions, taula **POBLACIO** i classe **Poblacio**, les coses seran molt similars. La diferència més significativa és la manera de representar la clau externa. Ara serà una relació **many-to-one**, indicant que la població pot estar en una única comarca, és a dir farà referència a un únic objecte **Comarca**.

```
<many-to-one class="classes.Comarca" embed-xml="true" fetch="select"
insert="true" name="comarca" not-found="exception" optimistic-lock="true"
unique="false" update="true">
  <column length="50" name="nom_c"/>
</many-to-one>
```

- **<many-to-one>** en la propietat que fa referència a l'altra classe
 - L'atribut **name** indica el nom de la propietat.
 - L'atribut **class** indica la classe a què fa referència, en aquest cas **Comarca**.
 - L'element **column** és un element buit amb la propietat **name**, que contindrà el camp de la taula

En resum, dins de l'element **class** que equival a la classe (i la taula), tindrem els elements:

- **id** per a la clau principal
- **property** per als camps normals
- **many-to-one** per als que són clau externa
- **set** per als que són apuntats per una clau externa, indicant el conjunt.

8 - Sessions i objectes hibernate

En aquesta secció comentarem el comportament de les sessions i dels objectes Hibernate (les classes creades per Hibernate).

Recordem que el procés per a arribar a una sessió era **Configuration -> SessionFactory -> Session**. Les sentències serien aquestes:

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

La primera sentència carrega el fitxer de configuració **hibernate.cfg.xml**, i inicialitza l'entorn d'Hibernate. A partir d'aquest moment podem utilitzar l'objecte Configuration per crear el **SessionFactory**. Ja hem comentat en el punt 5.5 que normalment només utilitzarem un objecte SessionFactory, i per tant podríem obligar a que només hi haja una instància d'aquesta classe amb la tècnica del **Singleton** (que utilitzarem a partir d'ara amb la classe **SessionFactoryUtil.java**). També hem comentat que **buildSessionFactory** està desferrat, però el posem ací per claredat i senzillesa. En el Singleton **SessionFactoryUtil.java** ja ho havíem arreglat.

8.1 - Transaccions

El primer que hauríem de fer notar és que si el SGBD admet transaccions, les possibles modificacions que fem sobre una Base de Dades a través d'Hibernate no es guardaran si no posem en marxa una transacció. Per tant, si volem fer actualitzacions (insercions, esborrats i/o modificacions), el primer que haurem de fer en la sessió serà començar una transacció. Al final podrem confirmar amb **commit** o rebutjar totes les actualitzacions amb **rollback**. En el següent codi inserim una nova comarca. Utilitzem el mètode **save** de la sessió, però ja el veurem més avant. Ara només volem il·lustrar una transacció.

```
SessionFactory sf = SessionFactoryUtil.getSessionFactory();
Session sessio = sf.openSession();
Transaction t = sessio.beginTransaction();

Comarques com = new Comarques();
com.setNomC((String) "Columbretes");
com.setProvincia("Castelló");
sessio.save(com);

t.commit();
sessio.close();
```

En cas de no haver posat les dues sentències en roig, no hauria hagut cap modificació. El mètode **beginTransaction()** ha fet començar la transacció, i el mètode **commit()** l'ha confirmada definitivament. Si en aquest moment vulguérem començar una altra transacció seria suficient amb **t.begin()**.

Per a no haver d'estar sempre pendents de transaccions, es podria posar en mode **autocommit**, és a dir, que automàticament faça un commit després de cada actualització (després de cada **save()**, **delete()** o **update()**). Seria canviar en el fitxer **hibernate.cfg.xml**, posant la següent propietat entre les del **<session-factory>**:

```
<property name="hibernate.connection.autocommit">true</property>
```

Tanmateix no és tan fiable, i us recomane que utilitzeu transaccions, és a dir, tenir el autocommit a false com estava per defecte i gestionar les transaccions manualment.

8.2 - Estats d'un objecte Hibernate

Els objectes de les classes que Hibernate ens ha ajudat a crear poden estar en més d'un estat, si mirem la seua sincronització amb la Base de Dades:

- **Transitori:** quan l'objecte s'ha creat, però no s'ha associat a cap sessió. És a dir, ja s'ha creat amb **new** i potser estiga buit o no (ja se li ha posat algun valor a alguna propietat amb els mètodes set), però encara no s'ha associat a cap sessió. Es podrà associar per exemple amb **save()** per a guardar l'objecte com hem vist en l'exemple de l'apartat anterior, o per exemple amb **load()**, com havíem vist en exemples anteriors per a agafar des de la Base de Dades.
- **Persistent:** quan l'objecte ja s'ha associat amb una sessió i per tant té una representació en la Base de Dades i un valor identificador. Es pot haver guardat o carregat. Hibernate detectarà qualsevol canvi fet sobre un objecte persistent i sincronitzarà l'estat quan es complete la unitat de treball. I per a això ja s'encarregarà ell de fer els INSERT, DELETE o UPDATE en la Base de Dades. Nosaltres no ens haurem de preocupar.
- **Separat:** és un objecte persistent, però que la seua sessió ha finalitzat. Es pot utilitzar, i fins i tot canviar l'estat, però no tindrà efecte aquestos canvis la Base de Dades, a no ser que es torne a associar una altra vegada amb una sessió, és a dir, si es torna a fer persistent. De fet, és una tècnica de treball, utilitzar objectes separats quan l'usuari puga estar molt de temps sense interactuar, i quan torne a interactuar, tornar a fer-lo persistent.

Anem a veure un exemple on es vegen els tres tipus d'estat. Tampoc cal que el feu vosaltres, està únicament a nivell il·lustratiu:

```
SessionFactory sf = SessionFactoryUtil.getSessionFactory();
Session sessio = sf.openSession();
Transaction t = sessio.beginTransaction();

Comarques com = new Comarca(); // A partir d'ací l'objecte és
transitori

com.setNomC((String) "Columbres");
com.setProvincia("Castelló");
sessio.save(com);

sessio.save(com); // A partir d'ací és persistent

t.commit();

sessio.close(); // A partir d'ací és separat, igual es
pot utilitzar, // però no estarà sincronitzat

System.out.println(com.getNomC() + " (" + com.getProvincia() + ")");
```

8.3 - Càrrega d'objectes: mètodes load() i get()

La càrrega d'objectes s'aconsegueix amb els mètodes **load()** i **get()**. Són similars excepte en una qüestió: el primer dóna un error si no es troba la fila, mentre que el segon contindrà null. Per tant, si no estem segurs de trobar la fila, hauríem d'utilitzar **get**, a no ser que vulguem provocar l'excepció, clar.

Tant si utilitzem un com l'altre, haurem d'especificar 2 paràmetres: la **classe** que volem buscar, i el **valor** de la clau principal de qui volem trobar. Aquest valor s'ha de passar amb el mateix tipus que la propietat corresponent a la clau principal, i en moltes ocasions haurem de canviar de tipus (**cast**). En el següent exemple, ja utilitzat anteriorment, trobem la comarca **Alt Maestrat** (valor de **nom_c**, que correspon a la clau principal). En aquesta ocasió no cal canviar de tipus, ja que és string:

```
SessionFactory sf = SessionFactoryUtil.getSessionFactory();
Session sessio = sf.openSession();

Comarca com = (Comarca) sessio.load(Comarca.class, "Alt Maestrat");

System.out.print("Comarca " + com.getNomC() + ": " +
com.getProvincia());
sessio.close();
```

Com comentàvem, si no existira la comarca Alt Maestrat, donaria un error. Ací tenim el mateix exemple, però utilitzant el mètode **get()** i comprovant després si **com** és null, i així esquivem l'error:

```
SessionFactory sf = SessionFactoryUtil.getSessionFactory();
Session sessio = sf.openSession();

Comarca com = (Comarca) sessio.get(Comarca.class, "Maestrat mitjà");

if (com==null)
    System.out.print("No existeix la comarca");
else
    System.out.print("Comarca " + com.getNomC() + ": " +
com.getProvincia());
sessio.close();
```

Anem a mirar la potencialitat que ens proporciona la manera que té Hibernate de construir les classes, amb el conjunt (o col·lecció) de poblacions que té un objecte de la classe Comarca.

Els **Set** es poden recórrer de més d'una manera. Una molt estesa és utilitzar un **iterator**, utilitzant els mètodes **hasNext()** que ens diu si hi ha més elements en la col·lecció, i **next()** que torna el següent element de la col·lecció. En el següent programa anem a visualitzar informació de la comarca de l'**Alcalatén**, i també els seus pobles. En roig està marcada la utilització del **Iterator**:

```
import java.util.Iterator;
import org.hibernate.Session;
import org.hibernate.SessionFactory;

import classes.Comarca;
import classes.Poblacio;
import classes.SessionFactoryUtil;

public class TercerAcces {

    public static void main(String[] args) {

        SessionFactory sf = SessionFactoryUtil.getSessionFactory();
        Session sessio = sf.openSession();
        Comarca com = (Comarca) sessio.load(Comarca.class, "Alcalatén");
        System.out.print("Comarca " + com.getNomC() + ": ");
        System.out.print(com.getProvincia());
        System.out.println(" (" + com.getPoblacions().size() + " pobles)");
        System.out.println();
        System.out.println("Llista de pobles");
```



```

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

        Iterator<Poblacio> it = com.getPoblacios().iterator();

        while (it.hasNext()){
            Poblacio p = it.next();
            System.out.println(p.getNom() + " (" + p.getPoblacio() + "
habitants)");
        }

        sessio.close();
    }
}

```

Però més senzilla encara és la utilització del bucle **foreach**, que fa el recorregut ell sol, tenint disponible cada element. La sintaxi en Java és **for (Classe e : conjunt)** i **e** anirà agafant tots els valors de la col·lecció.

Així, en el programa anterior podem substituir la declaració del iterator i el bucle (és a dir, les marcades amb roig) pel següent:

```

        for (Poblacio p : com.getPoblacios())
            System.out.println(p.getNom() + " (" + p.getPoblacio() + "
habitants)");

```

I quedaria de la següent manera:

```

import java.util.Iterator;

import org.hibernate.Session;
import org.hibernate.SessionFactory;

import classes.Comarca;
import classes.Poblacio;
import classes.SessionFactoryUtil;

public class QuartAcces {

    public static void main(String[] args) {

        SessionFactory sf = SessionFactoryUtil.getSessionFactory();
        Session sessio = sf.openSession();
        Comarca com = (Comarca) sessio.load(Comarca.class, "Alcalatén");
        System.out.print("Comarca " + com.getNomC() + ": ");
        System.out.print(com.getProvincia());
        System.out.println(" (" + com.getPoblacios().size() + " pobles)");
        System.out.println();
        System.out.println("Llista de pobles");
        System.out.println("-----");

        for (Poblacio p : com.getPoblacios())
            System.out.println(p.getNom() + " (" + p.getPoblacio() + "
habitants)");

        sessio.close();
    }
}

```

En ambdós, tant en **TercerAcces** com en **QuartAcces** casos el resultat haurà estat:

```

Comarca Alcalatén: Castelló (9 pobles)

Llista de pobles
-----
Atzeneta del Maestrat (1321 habitants)
Vistabella del Maestrat (384 habitants)
Figueroles (549 habitants)
Benafigos (156 habitants)
Costur (562 habitants)
Useres, les (992 habitants)
Xodos (126 habitants)
Llucena (1417 habitants)
Alcora, l' (10672 habitants)

```

Com veieu apareixen tots els pobles de la comarca que hem llegit, però com és un conjunt (Set) no podem assegurar l'ordre. De fet, podria ser que a cadascú de nosaltres li apareguen els pobles en ordre distint.

Estaria bé que aparegueren ordenats. Sempre ho podríem fer per mig d'una consulta i ordenar pel nom de la població

com veurem més avant. Però si ens interessa que els pobles apareguen sempre ordenats, podem fer una altra cosa: modificant el fitxer de mapatge, li podem dir que ens apareguen sempre ordenats per un camp. Ho havíem comentat en la pregunta 7, i ara anem a aplicar-lo.

Per mig d'una senzilla indicació en el fitxer de mapatge **Comarca.hbm.xml**, podem fer que les dades ens vinguin ordenades, és a dir, que les poblacions d'una comarca ens vinguin per ordre alfabètic. Senzillament serà posar **order-by="nom"** en la definició del **set poblacios**.

És a dir hauríem de substituir la línia

```
<set embed-xml="true" fetch="select" inverse="true" lazy="true"
mutable="true" name="poblacios" optimistic-lock="true" sort="unsorted"
table="poblacio">
```

per

```
<set embed-xml="true" fetch="select" inverse="true" lazy="true"
mutable="true" name="poblacios" optimistic-lock="true" sort="unsorted"
table="poblacio" order-by="nom">
```

I així li estem dient que les poblacions vinguin ordenades per nom, que és el camp de la taula **POBLACIO** pel qual ens interessa ordenar.

Només fent aquesta modificació, el resultat tant del programa **TercerAcces** com el de **QuartAccess** serà:

```
Comarca Alcalaén: Castelló (9 pobles)

Llista de pobles
-----
Alcora, l' (10672 habitants)
Atzeneta del Maestrat (1321 habitants)
Benafigos (156 habitants)
Costur (562 habitants)
Figueroles (549 habitants)
Llucena (1417 habitants)
Useres, les (992 habitants)
Vistabella del Maestrat (384 habitants)
Xodos (126 habitants)
```

8.4 - Inserció, modificació i esborrat d'objectes

En totes aquestes operacions d'actualització, recordem que farà falta posar-les en una **transacció** per a que tinguin efecte.

Inserció

Es fa amb el mètode **save()** de la sessió, passant-li l'objecte que volem guardar. Posem el mateix exemple d'apartats anteriors:

```
import org.hibernate.Session;
import org.hibernate.Transaction;

import classes.Comarca;
import classes.SessionFactoryUtil;

public class Insercio {

    public static void main(String[] args) {

        Session sessio = SessionFactoryUtil.getSessionFactory().openSession();
        Transaction t = sessio.beginTransaction();
        Comarca com = new Comarca();

        com.setNomC((String) "Columbres");
        com.setProvincia("Castelló");

        sessio.save(com);

        t.commit();
        sessio.close();
    }
}
```

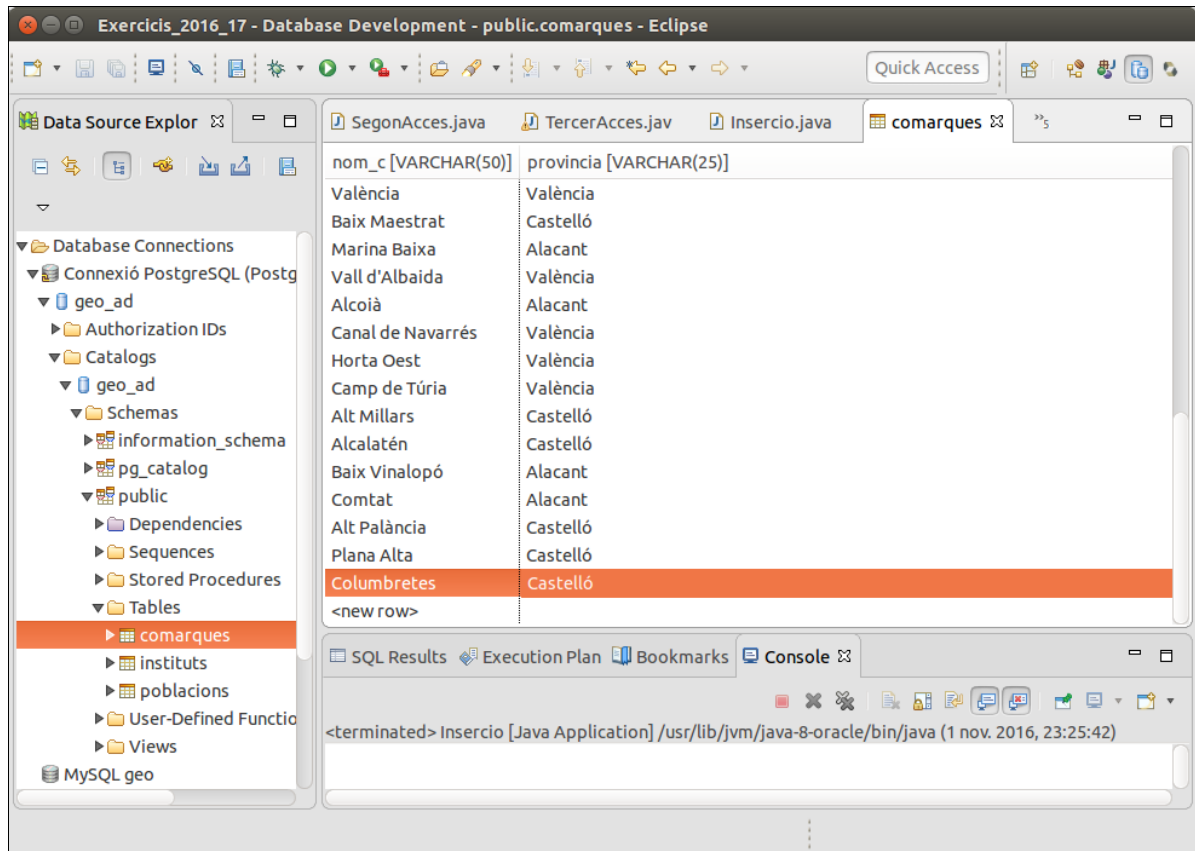
Podem comprovar en la perspectiva d'Hibernate com l'última comarca és la que acabem d'introduir:

The screenshot shows the Eclipse IDE interface. The 'Hibernate Query Result' window displays the results of the query 'from classes.Comarques'. The results are a list of 11 'classes.Comarques' objects, each with a unique ID. The last object, 'classes.Comarques@3d3c0f97', is highlighted in orange, indicating it is the most recently added record.

Property	Value
Identifier	nomC
Properties	poblacions
	provincia
	Castelló

I també podem comprovar en la perspectiva Database com aquesta comarca efectivament s'ha introduït en la taula

COMARQUES

Esborrat

Amb el mètode **delete**, passant-li com a paràmetre l'objecte. Prèviament s'ha d'haver carregat l'objecte amb els mètodes **load()** o **get()**.

```
import org.hibernate.Session;
import org.hibernate.Transaction;

import classes.Comarca;
import classes.SessionFactoryUtil;

public class Esborrat {

    public static void main(String[] args) {

        Session sessio = SessionFactoryUtil.getSessionFactory().openSession();
        Transaction t = sessio.beginTransaction();
        Comarca com = (Comarca) sessio.load(Comarca.class, "Columbretes");

        sessio.delete(com);

        t.commit();
        sessio.close();
    }
}
```

Modificació

Amb el mètode **update()**, passant-li com a paràmetre l'objecte. Prèviament haurà d'haver estat carregat amb els mètodes **load()** o **get()**, igual que en el cas d'esborrar.

En el següent exemple estem canviant la província del Camp de Morvedre. Per a que no tinga efecte aquest canvi, i no modificar les dades que està utilitzant la resta de companys, observeu com al final estem fent un **rollback**:

```
import org.hibernate.Session;
import org.hibernate.Transaction;

import classes.Comarca;
import classes.SessionFactoryUtil;

public class Modificacio {

    public static void main(String[] args) {

        Session sessio = SessionFactoryUtil.getSessionFactory().openSession();
        Transaction t = sessio.beginTransaction();
        Comarca com = (Comarca) sessio.load(Comarca.class, "Camp de
Morvedre");

        com.setProvincia("Castelló");
        sessio.update(com);

        t.rollback();
        sessio.close();
    }
}
```

El mètode **update()** és específic per a fer actualitzacions d'un objecte que ja existia a la Base de Dades. Però també podríem utilitzar **save()**. És a dir, **save()** sempre funciona. Si no existeix l'objecte el crea (crea les files oportunes a la Base de Dades), i si ja existia, l'actualitza. En canvi **update()** només funciona amb actualitzacions.

9 - Consultes

Hem vist com podem accedir molt còmodament a les classes que equivalen a una taula. Però en moltes ocasions ens farà falta accedir no exactament a una taula sinó a una combinació de taules, o en definitiva voldrem informació més elaborada. Per a poder "interrogar" a la Base de Dades amb consultes més complexes, Hibernate suporta un llenguatge de consulta Orientat a Objectes anomenat **HQL (Hibernate Query Language)**, molt paregut a SQL ja que és una extensió Orientada a Objectes d'aquest.

En aquest sentit s'ha intentat fer un estàndard de llenguatge de consulta anomenat **OQL (Object Query Language)**, desenvolupat per un grup amb ànim de crear un estàndard **ODMG (Object Data Management Group)**. Aquest estàndard no l'ha implementat al 100% cap producte comercial, i en realitat tenim subconjunts d'aquest estàndard.

Utilitzarem la classe **Query**, i invocarem al mètode **createQuery()** de **Session**, que justament torna una Query. Aquest seria un exemple:

```
Query q = sessio.createQuery("from Comarca");
```

Per a recuperar les dades tenim dues possibilitats, utilitzant dos mètodes de la query:

- Mètode **list()**: torna tots els resultats de la consulta en una col·lecció (**List**). Aquest mètode fa una crida única al SGBD i es duren totes les dades. Haurà d'haver, per tant, memòria suficient per a que càpiguen tots els resultats. Si és una quantitat gran de resultats, tardarà molt en executar-se.
- Mètode **iterate()**: torna un iterador per a poder recórrer els resultats de la consulta. Hibernate executa la sentència, però només torna els identificadors de les files del resultat, i cada vegada que fem **next()** del iterador, s'executa realment la consulta tornant la següent fila. Per tant fa falta molta menys memòria. Per contra es fan més accessos a la Base de Dades, encara que cadascun tardarà molt poc, però en total tardarà més. Es pot fins i tot fixar la quantitat de files a tornar amb el mètode **setFetchSize()**.

En el següents dos exemples, que són equivalents, es trau una llista de totes les comarques. El primer utilitza el mètode **list()** i el segon el mètode **iterate()**. I en aquesta ocasió utilitzem un iterador per a recórrer la llista (**List**) en compte de **foreach**, per veure més d'una manera, i per similitud amb el segon exemple,

```
import java.util.Iterator;
import java.util.List;

import org.hibernate.Query;
import org.hibernate.Session;

import classes.Comarca;
import classes.SessionFactoryUtil;

public class AccésAmbList {

    public static void main(String[] args) {

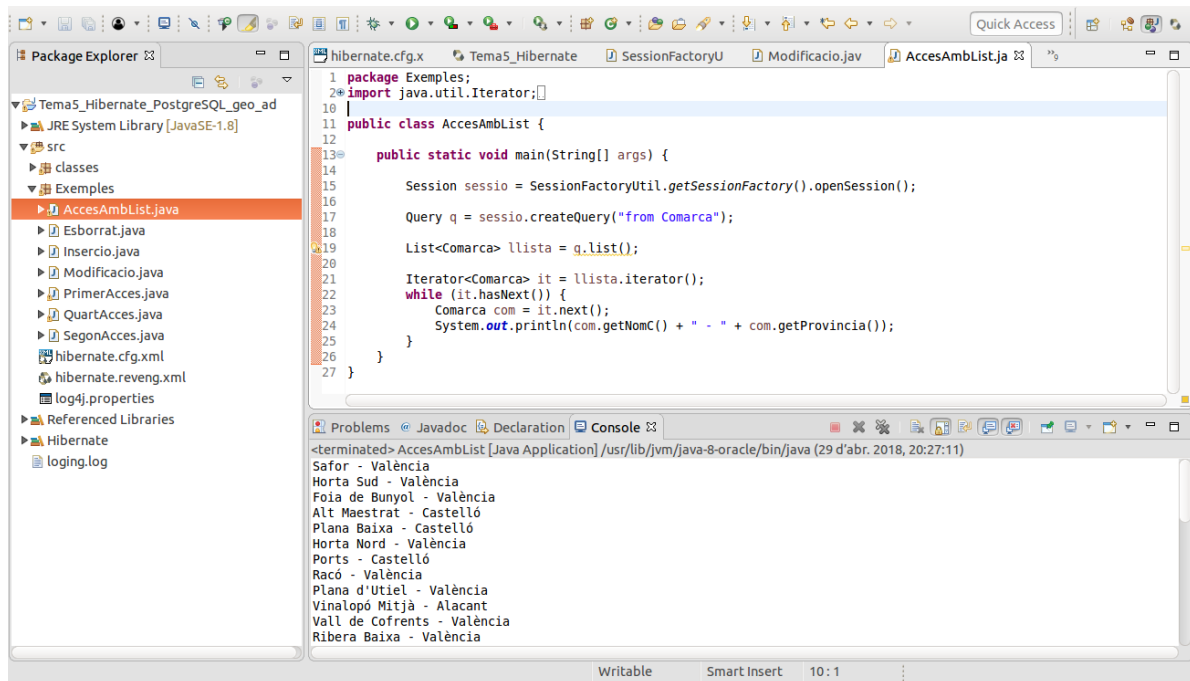
        Session sessio = SessionFactoryUtil.getSessionFactory().openSession();

        Query q = sessio.createQuery("from Comarca");

        List<Comarca> llista = q.list();

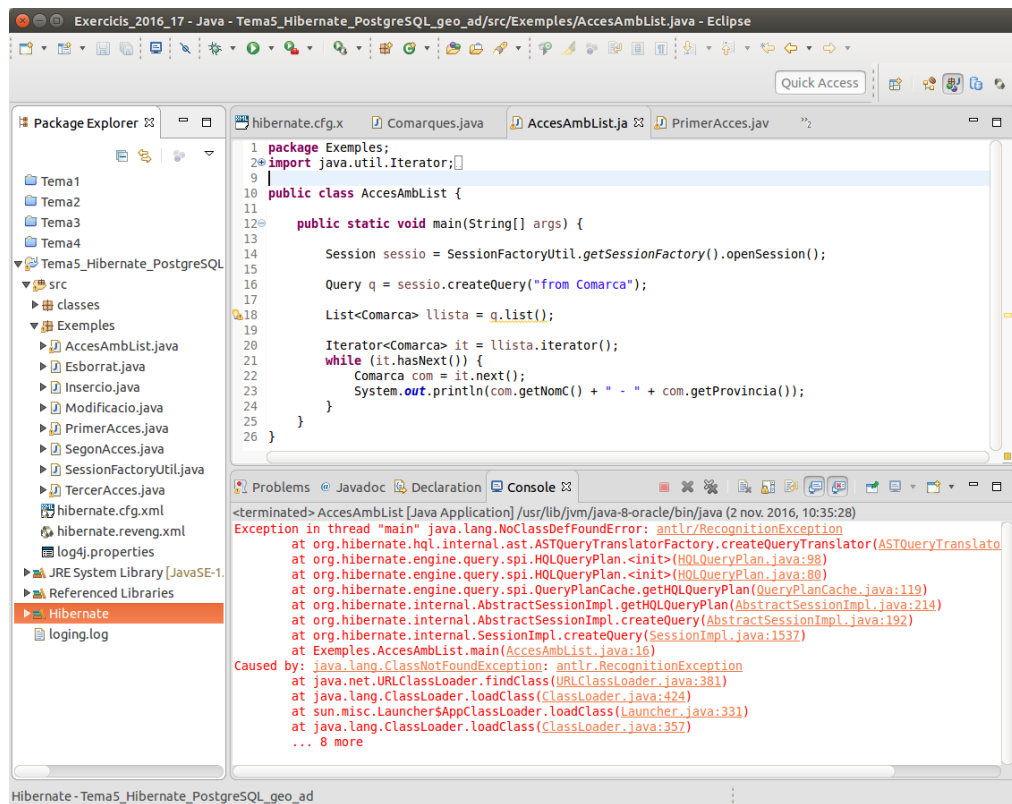
        Iterator<Comarca> it = llista.iterator();
        while (it.hasNext()) {
            Comarca com = it.next();
            System.out.println(com.getNomC() + " - " + com.getProvincia());
        }
    }
}
```

Aquest serà el resultat



Nota

En cas de no haver incorporat el plugin [antlr-3.5.2-complete.jar](#), necessari per a fer una anàlisi prèvia de la sentència HQL, ens donaria el següent error:



I ho solucionariem baixant-nos el plugin anterior i incorporant-lo a la nostra llibreria d'Hibernate

Ara amb el mètode **iterate()**:

```
import java.util.Iterator;
```

```

import org.hibernate.Query;
import org.hibernate.Session;

import classes.Comarca;
import classes.SessionFactoryUtil;

public class AccésAmbIterate {

    public static void main(String[] args) {

        Session sessio = SessionFactoryUtil.getSessionFactory().openSession();

        Query q = sessio.createQuery("from Comarca");

        Iterator<Comarca> it = q.iterate();

        while (it.hasNext()) {
            Comarca com = it.next();
            System.out.println(com.getNomC() + " - " + com.getProvincia());
        }

    }

}

```

Encara que la manera més curta i potser més clara és utilitzant un bucle **foreach** amb el mètode **list()**. Només haurem d'anar amb compte de fer un **cast** per a que sàpiga quin tipus de llista és:

```

import java.util.List;

import org.hibernate.Query;
import org.hibernate.Session;

import classes.Comarca;
import classes.SessionFactoryUtil;

public class AccésAmbListForEach {

    public static void main(String[] args) {

        Session sessio = SessionFactoryUtil.getSessionFactory().openSession();

        Query q = sessio.createQuery("from Comarca");

        for (Comarca com : (List<Comarca>) q.list())
            System.out.println(com.getNomC() + " - " + com.getProvincia());

    }

}

```

Si sabem que la consulta tornarà únicament una fila, podem assignar aquesta fila a un objecte de la classe de la taula afectada, posant el mètode **uniqueResult()** en la creació de la query, així ens estavem passos.

```

import org.hibernate.Session;

import classes.Comarques;

public class AccésAmbUniqueResult {

    public static void main(String[] args) {

        Session sessio = SessionFactoryUtil.getSessionFactory().openSession();

        Comarques d = (Comarques) sessio.createQuery("from Comarques where nomC='Alcalatén').uniqueResult();

        System.out.println(d.getNomC() + " - " + d.getProvincia());

    }

}

```

Però en realitat en aquest exemple poca cosa hem guanyat, perquè per a agafar l'objecte corresponent a una única fila d'una taula, ja ho fèiem amb **session.load()**. Més endavant veurem consultes més complicades on trobarem la utilitat.

9.1 - Paràmetres en les consultes

createQuery() admet també la utilització de paràmetres, igual que feia les sentències **PreparedStatement** del tema anterior. La manera de posar-les és molt similar al vist en aquella ocasió. Ara, però, ho ampliarem un poc.

La manera de posar valor als paràmetres serà utilitzant els mètodes **setTipus()** que tindran 2 paràmetres, el primer per a assenyalar el paràmetre, i el segon per a indicar el valor. De moment tot igual que en el **PreparedStatement**.

Però ara veurem 2 maneres de posar paràmetres en la consulta:

- Amb **?**, utilitzant el mètode **setTipus(int, valor)** i indicant el número d'ordre d'aparició del paràmetre, que **comença en aquesta ocasió per 0**.
- Amb **:nom**, utilitzant el mètode **setTipus(string, valor)** i indicant el nom del paràmetre

Ho veurem molt més clar en un exemple, millor dit, en dues versions del mateix exemple. Intentarem traure les poblacions d'una determinada comarca, que tenen una altura determinada o major. I aquestos dos valors els posarem coma paràmetres, i així podrem practicar les dues formes. Els valors que posarem per a la comarca serà **Alcalatén**, i l'altura **500**

En aquesta primera versió assenyalarem els paràmetres amb **?** a l'estil de **JDBC**.

```
import java.util.List;

import org.hibernate.Query;
import org.hibernate.Session;

import classes.Poblacio;
import classes.SessionFactoryUtil;

public class ConsultesAmbParametres1 {

    public static void main(String[] args) {

        Session sessio = SessionFactoryUtil.getSessionFactory().openSession();

        Query q = sessio.createQuery("from Poblacio where altura>=? and
comarca.nomC=?");
        q.setInteger(0, 500);
        q.setString(1, "Alcalatén");

        for (Poblacio p : (List<Poblacio>) q.list()){
            System.out.println(p.getNom() + " --> " + p.getAltura());
        }

    }

}
```

En la segona posem els paràmetres de l'altra manera.

```
import java.util.List;

import org.hibernate.Query;
import org.hibernate.Session;

import classes.Poblacio;
import classes.SessionFactoryUtil;

public class ConsultesAmbParametres2 {

    public static void main(String[] args) {

        Session sessio = SessionFactoryUtil.getSessionFactory().openSession();

        Query q = sessio.createQuery("from Poblacio where altura>=:alt and
comarca.nomC=:com");
        q.setInteger("alt", 500);
        q.setString("com", "Alcalatén");

        for (Poblacio p : (List<Poblacio>) q.list()){
            System.out.println(p.getNom() + " --> " + p.getAltura());
        }

    }

}
```

}

9.2 - Consultes diferents a una classe ja definida

De moment en les consultes sempre hem tornat una fila o més però sempre **d'una única taula**, o millor dit, **una classe** equivalent a una taula.

Però hi haurà moltes ocasions en què voldrem fer una consulta diferent, on intervinga més d'una taula (o classe), que torne funcions de grup (SUM, AVG, ...), en definitiva que no siga una sentència equivalent a **SELECT * FROM ...**

Per tant ara no tornarà un objecte o una col·lecció d'objectes dels que ja tenim definits. Podem tenir més d'un cas, i a continuació els intentarem tractar:

Consultes que tornen més d'un objecte dels definits

Són consultes en les quals hi haurà una reunió de taules (millor dit, de les classes equivalents), i es tornen tots els camps (totes les propietats).

Podríem definir-nos una classe que englobara totes les propietats de tots els objectes, però hi ha una solució més ràpida. Els resultats s'obtenen en un **array d'objectes**, on el primer objecte serà l'equivalent de la primera taula, el segon l'equivalent de la segona, etc. Aquest exemple és només il·lustratiu. Està clar que es pot fer molt més curt i senzill agafant únicament les Poblacions i a partir d'elles traure la comarca.

```
import java.util.Iterator;

import org.hibernate.Query;
import org.hibernate.Session;

import classes.Comarca;
import classes.Poblacio;
import classes.SessionFactoryUtil;

public class ConsultaComplexa1 {

    public static void main(String[] args) {

        Session sessio = SessionFactoryUtil.getSessionFactory().openSession();
        Query q = sessio.createQuery("from Poblacio p, Comarca c where
p.comarca.nomC=c.nomC order by p.nom");

        Iterator it = q.iterate();
        while (it.hasNext()) {
            Object[] tot = (Object[]) it.next();
            Poblacio p = (Poblacio) tot[0];
            Comarca c = (Comarca) tot[1];
            System.out.println(p.getNom() + " (" + c.getNomC() + ". " +
c.getProvincia() + ")");
        }
    }
}
```

Com veiem, un poc "rotllo"...

Consultes que tornen un únic valor

Aquest és el cas més senzill, ja que el valor tornat el podem agafar del tipus bàsic del valor tornat. Normalment és després d'utilitzar una funció de grup. Recordem que ha de tornar un únic valor (una única fila i una única columna, per dir-lo així). Haurèm d'utilitzar el mètode **uniqueResult()** ja comentat amb anterioritat. L'exemple serà per a calcular l'altura mitjana de totes les poblacions.

```
import org.hibernate.Query;
import org.hibernate.Session;

import classes.SessionFactoryUtil;
```

```

public class ConsultaComplexa2 {
    public static void main(String[] args) {
        Session sessio = SessionFactoryUtil.getSessionFactory().openSession();
        Query q = sessio.createQuery("select avg(altura) from Poblacio");
        Double mitjana = (Double) q.uniqueResult();
        System.out.println("Altura mitjana: " + mitjana);
    }
}

```

Consultes amb qualsevol resultat

Una de les solucions seria crear una classe amb totes les propietats que torna la consulta, posant els mètodes get i set de cadascuna de les propietats. No veurem aquesta solució per tediosa.

Farem com en el primer apartat d'aquesta pregunta, gestionant l'**array d'objectes** que s'obtenen, però en aquesta ocasió cada objecte de l'array serà un tipus simple.

```

import java.util.List;
import org.hibernate.Query;
import org.hibernate.Session;

import classes.SessionFactoryUtil;
public class ConsultaComplexa3 {
    public static void main(String[] args) {

        Session sessio = SessionFactoryUtil.getSessionFactory().openSession();

        Query q = sessio.createQuery("select
c.nomC,count(p.codM),avg(p.altura) "
                                + "from Comarca c , Poblacio p "
                                + "where c.nomC=p.comarca.nomC "
                                + "group by c.nomC "
                                + "order by c.nomC");

        List<Object[]> llista = q.list();
        for (Object[] tot : llista)
            System.out.println("Comarca: " + tot[0] + ". Núm. pobles: " +
tot[1] + ". Altura mitjana: " + tot[2]);
    }
}

```

Hem utilitzat una altra vegada el **foreach**, i en aquesta ocasió només hem d'anar amb compte amb el tipus **Object[]**.

10 - Consultes d'actualització

Aquesta pregunta no té tanta importància com les altres, ja que actualitzacions ja sabem fer amb els mètodes **save()**, **delete()** i **update()** de la sessió.

Pràcticament l'única utilitat serà fer actualitzacions massives, que afecten a més d'una fila (millor dit, més d'un objecte). HQL també permet fer-les. Mirem la seua sintaxi:

DELETE

```
DELETE [FROM] NomEntitat  
[WHERE condició]
```

Esborrarà tots els objecte que acompleixen la condició. Observeu que:

- FROM és opcional
- La clàusula WHERE és opcional. Si no es posa s'esborraran tots els objectes.
- En el FROM només es pot posar una entitat. No val a posar més d'una ni reunions ni res. Sí que es podran posar subconsultes en la clàusula WHERE, i en les consultes sí que es pot fer el que es vulga (reunions, ...).

Per a executar la consulta s'utilitzarà el mètode **Query.executeUpdate()** (en compte de **Query.list()** per exemple). Aquest mètode torna un enter que serà el número de files (objectes) esborrades.

I recordeu que s'ha de confirmar la transacció per a que tinguin efecte els canvis.

Ací tenim un exemple, al qual li hem aplicat **ROLLBACK** al final, per a no fer efectiu l'esborrat. Però sí que diu quantes files "s'esborrarien".

El següent exemple "esborraria" els Instituts de la província de Castelló, que són els que el seu codi comença per 12, i es diu quants serien els afectats, però després es rebutgen els canvis en fer **rollback()**.

```
import org.hibernate.Session;  
import org.hibernate.Transaction;  
  
import classes.SessionFactoryUtil;  
  
public class EsborratMassiu {  
    public static void main(String[] args) {  
        Session sessio =  
SessionFactoryUtil.getSessionFactory().openSession();  
  
        Transaction tr = sessio.beginTransaction();  
  
        int files = sessio.createQuery("delete Institut where codi like  
'12%'").executeUpdate();  
  
        System.out.println("S'han esborrat " + files + " files.");  
  
        tr.rollback();  
    }  
}
```

UPDATE

```
UPDATE [FROM] NomEntitat  
SET propietat = valor [, propietat2 = valor2, ...]  
[ WHERE condicio ]
```

Les observacions fetes en l'apartat de DELETE també valen ací. I observeu que podem modificar més d'un camp, separant-los per comes. Ací tenim un exemple, en el qual fem també **rollback()** per a deixar les coses com estaven:

```
import org.hibernate.Session;
import org.hibernate.Transaction;

import classes.SessionFactoryUtil;

public class ModificacioMassiva {

    public static void main(String[] args) {

        Session sessio = SessionFactoryUtil.getSessionFactory().openSession();

        Transaction tr = sessio.beginTransaction();

        int files = sessio.createQuery("update Poblacio set poblacio =
poblacio *1.05 where poblacio < 200").executeUpdate();

        System.out.println("S'han modificat " + files + " files.");

        tr.rollback();

    }

}
```

INSERT

```
INSERT INTO NomEntitat (llista_propietats)
    Sentència select
```

Ara tindrem els següents condicionants:

- No és vàlida la sentència INSERT INTO ... VALUES ... que servia per a introduir una fila donant-li els valors. Si volem fer això, millor utilitzar **save()**. Per tant només podrem introduir valors procedents d'una altra taula.
- La sentència SELECT pot ser qualsevol sentència HQL. Només hem de cuidar que els tipus tornats per la consulta siguin els que espera INSERT (és a dir, del mateix tipus que les propietats de la classe associada a la taula).
- Per a la propietat **id**, que és la que correspon a la clau principal, podem tenir 2 opcions:
 - es pot especificar en la llista de propietats, i aleshores li haurem de proporcionar un valor.
 - es pot ometre, sempre i quan la propietat estiga definida com **increment** (no **assigned**); això vol dir que es correspon a un camp autoincrementat. Aleshores serà el SGBD qui li assignarà el valor

Aquest seria un exemple. En ell intentem introduir una nova comarca a partir d'una hipotètica taula de comarques noves (*NovesComarques*). No la podrem provar perquè no existeix aquesta taula, millor dit, aquesta classe. L'hauríem de crear, fer-li el mapatge, i posteriorment executar açò. Per tant ho veiem únicament a mode il·lustratiu:

```
Transaction tr = sessio.beginTransaction();

int files = sessio.createQuery("insert into Comarca (nomC, provincia) "
    + " select n.nomC, n.provincia from
    NovesComarques n").executeUpdate();

System.out.println("S'han introduït " + files + " files.");

tr.rollback();
```

11 - Resum del llenguatge HQL

Veurem d'una manera molt ràpida el llenguatge HQL. Sobretot insistirem en les diferències amb SQL, al qual se sembla molt.

- La clàusula **SELECT** és opcional. Si no la posem equival a agafar totes les propietats (és a dir, equivalent a **SELECT *** de SQL). És una operació molt habitual, ja que d'aquesta manera la llista d'objectes tornats és de la classe marcada en el from.
- Es poden utilitzar àlies de les classes, posant **AS** després de la classe i abans de l'àlies. També podem prescindir del AS. Si utilitzem àlies, per a fer referència a la classe haurem d'utilitzar sempre l'àlies. Per exemple **FROM Comarques AS c**, o senzillament **FROM Comarques c**.
- La clàusula **WHERE** és opcional. Si no la posem equival a agafar de totes les instàncies de la classe associada (és a dir, de totes les files de la taula).
- S'admeten les clàusules **GROUP BY**, **HAVING** i **ORDER BY**, que funcionen igual que en SQL.
- S'admeten **subconsultes**, si és que el SGBD les admet, clar.

Ací teniu uns quants exemples:

```
from Comarca
```

Torna totes les instàncies de Comarques

```
from Comarca order by nomC
```

Torna totes les instàncies de Comarques ordenades per nom de comarca

```
from Comarca where provincia='Castelló'
```

Torna totes les comarques de la província de Castelló

```
from Comarca where provincia in ('Castelló','Alacant')
```

Torna totes les comarques de les províncies de Castelló i Alacant

```
from Poblacio where altura between 700 and 750
```

Torna totes les poblacions amb una altura entre 700 i 750 m,

```
from Institut where codpostal is null
```

Torna els instituts que no tenen codi postal (no hi ha ningun)

```
from Comarca where nomC like 'A%'
```

Torna les comarques el nom de les quals comença per A

```
select provincia, count(*) from Comarca group by provincia
```

Torna les províncies i el número de comarques en cadascuna

```
select poblacio.nom , count(*) from Institut group by poblacio.nom
```

Torna el nom de la població i el número d'instituts que té (només d'aquelles que tenen algun institut)

```
select comarca.nomC , avg(altura) from Poblacio group by comarca.nomC having  
avg(altura) > 700
```

Torna el nom de la comarca i l'altura mitjana d'aquelles comarques que tenen una altura mitjana més gran que 700

```
from Poblacio where altura > (select avg(altura) from Poblacio)
```

Torna les poblacions que tenen una altura major que la mitjana

```
from Poblacio p1 where p1.altura > (select avg(altura) from Poblacio p2 where  
p2.comarca=p1.comarca)
```

Torna les poblacions que tenen una altura major que la mitjana de la seua comarca

HQL també admet reunions en les classes. Per a fer una reunió normal, podem posar la condició en el where i ja està, però si volem fer una reunió externa (un **left join** per exemple) no ens valdrà aquesta opció, i haurèm de fer la reunió.

La sintaxi de la reunió és diferent en HQL.

En **SQL** posàvem la condició en l'apartat **ON**, després d'haver posat les taules, i posàvem una condició d'igualtat entre els camps d'una i altra taula

```
SELECT ... FROM POBLACIO LEFT JOIN INSTITUT ON POBLACIO.cod_m=INSTITUT.cod_m
```

En canvi en **HQL** després de les taules no posem **ON**, sinó que posarem l'enllaç entre les dues taules. Aquest enllaç coincideix amb el nom de la col·lecció d'elements de l'altre taula. En el cas de l'exemple que estem veient és **institutses**. Per tant una reunió entre **Poblacions** i **Instituts** es definiria així

```
from Poblacio p left join p.instituts
```

Vejam un exemple un poc més elaborat, on traurem el nom de la població amb el número d'instituts que té, fins i tot d'aquells que no tenen institut

```
select p.nom, count(i.codi) from Poblacio p left join p.instituts i group by  
p.nom order by p.nom
```


12. Generació de taules a partir de les classes (voluntari)

Durant aquest tema, i també en els exercicis, hem treballat sobre projectes en els quals a partir de les taules d'una Base de Dades ja creada en qualsevol Sistema Gestor de Bases de Dades (excepte SQLite) hem generat les classes. Funciona molt bé aquesta generació de classes amb Hibernate.

També és possible el procés invers, és a dir, **generar les taules** en una Base de Dades **a partir de les classes** creades en un projecte. Veurem un exemple molt senzill de com és possible. Tanmateix hem de dir ja que aquesta manera de funcionar no és tan efectiva com la inversa, i si l'estructura de classes és complexa, podem tenir problemes per a generar les taules. Per això l'exemple que posarem és molt senzill.

Aquesta pregunta és totalment voluntària. Si no teniu ganes o temps, no fa falta que feu l'exemple mostrat a continuació. El que sí que és obligatori és la generació de classes a partir de les taules, vist anteriorment.

L'exemple que intentarem implementar és una part de l'exemple de la biblioteca, vist ja en el tema 4. Només treballarem sobre 2 classes **Editorial** i **Llibre**, i fins i tot no posarem totes les propietats, ja que si ho posàrem tot, segurament ens donaria problemes a l'hora de la generació de taules.

Aquestes són les classes, millor dit les propietats de les classes, com veieu amb una estructura molt senzilla:

classe Editorial	classe Llibre
<pre>public class Editorial { String codi; String nom; String web; }</pre>	<pre>public class Llibre { String isbn; String titol; int pagines; Editorial editorial; }</pre>

Com veieu tenim una propietat en la classe **Llibre** que és de tipus **Editorial**, per a marcar l'editorial del llibre. El més lògic seria posar un `ArrayList` de tipus **Llibre** en la classe **Editorial** per a posar tots els llibres de l'editorial. Però açò segurament ens donaria problemes en el moment de la generació de taules. Per tant, farem aquest exemple tan senzill a mode il·lustratiu, i prou.

En aquestes classes, com es mostra en el vídeo posterior, s'han d'incorporar els mètodes **get** i **set** per a totes les propietats, així com **constructors**, un sense paràmetres, i un altre amb paràmetres per a inicialitzar totes les propietats.

Posteriorment s'han de preparar les classes per a poder generar les taules. Consistirà en un procediment automàtic que s'encarregarà d'afegir les **anotacions**, que són sentències que Java ignorarà i serveixen per a afegir la informació addicional necessària. Les anotacions comencen sempre per **@**.

Després de les anotacions, les classes quedaran així (hem marcat les anotacions en roig):

classe Editorial	classe Llibre
<pre>import javax.persistence.Entity; import javax.persistence.Id; @Entity public class Editorial { @Id String codi; String nom; String web; public Editorial() { super(); } }</pre>	<pre>import javax.persistence.Entity; import javax.persistence.Id; import javax.persistence.ManyToOne; @Entity public class Llibre { @Id String isbn; String titol; int pagines; @ManyToOne Editorial editorial; }</pre>

<pre> } public Editorial(String codi, String nom, String web) { super(); this.codi = codi; this.nom = nom; this.web = web; } public String getCodi() { return codi; } public void setCodi(String codi) { this.codi = codi; } public String getNom() { return nom; } public void setNom(String nom) { this.nom = nom; } public String getWeb() { return web; } public void setWeb(String web) { this.web = web; } } </pre>	<pre> public Llibre() { super(); } public Llibre(String isbn, String titol, int pagines, editorial) { super(); this.isbn = isbn; this.titol = titol; this.pagines = pagines; this.editorial = editorial; } public String getIsbn() { return isbn; } public void setIsbn(String isbn) { this.isbn = isbn; } public String getTitol() { return titol; } public void setTitol(String titol) { this.titol = titol; } public int getPagines() { return pagines; } public void setPagines(int pagines) { this.pagines = pagines; } public Editorial getEditorial() { return editorial; } public void setEditorial(Editorial editorial) { this.editorial = editorial; } } </pre>
---	---

Amb aquesta informació es podran generar les sentències SQL que crearan les taules en la Base de Dades. Aquestes sentències serien:

taula EDITORIAL	taula LLIBRE
<pre> create table Editorial (codi varchar(255) not null, nom varchar(255), web varchar(255), primary key (codi)); </pre>	<pre> create table Llibre (isbn varchar(255) not null, pagines int4 not null, titol varchar(255), editorial_codi varchar(255), primary key (isbn)); alter table Llibre add constraint FKap3159swqtuj1i2q43xm0wk2 foreign key (editorial_codi) references Editorial; </pre>

Observeu com ha creat la clau externa en LLIBRE que apunta a EDITORIAL

Aquest vídeo arreplega tot el procés:

Exercicis



Exercici 5.1

Crea un projecte nou anomenat **Tema5_PostgreSQL_Rutes**. Inclou el driver JDBC de PostgreSQL i la llibreria nostra d'Hibernate.

- Fes el mapatge de les taules **RUTA** i **PUNT**, situades en la Base de Dades **rutes** a la qual pot accedir l'usuari **rutes** (contrasenya **rutes**) i situada en el servidor de l'Institut (**89.36.214.106**). Guarda-les en un paquet anomenat **dades**.
- Hauràs observat que ha creat les classes **Ruta.java**, **Punt.java** i també **PuntId.java**. Aquesta última l'ha creada perquè la clau principal de la taula **PUNT** està formada per **num_r + num_p**. No ens afectarà a nosaltres, que podem utilitzar únicament **Ruta.java** i **Punt.java**.
- Intenta visualitzar dades des de la perspectiva Hibernate.
- Copia't el fitxer **log4j.properties** per a que no es visualitzin els avisos.
- Crea't la classe **SessionFactoryUtil**, que serà un **singleton** per a poder crear una única SessionFactory. L'has d'utilitzar en totes les altres classes, sempre que et faci falta un SessionFactory. Ha d'estar en el mateix paquet **dades**.
- En un paquet anomenat **Exercicis**, crea't la classe **VeureRutes**, que serà un programa senzill que visualitzi les rutes, i el número de punts, ordenades per nom de ruta. Ha de ser per mig d'una **consulta HQL** (el més senzilla possible), i evidentment el programa ha de ser independent del número de rutes existent en aquest moment a la Base de Dades. El resultat seria aquest:

```
La Magdalena - 7 punts
Pelegrins de Les Useres - 6 punts
Pujada a Penyagolosa - 5 punts
```



Exercici 5.2

Sobre el projecte de **Tema5_Hibernate_PostgreSQL_geo_ad**, on tenim el mapatge de les taules **COMARQUES**, **POBLACIONS** i **INSTITUTS**, creeu-vos un paquet anomenat **Exercicis**, on col·loqueu les classes d'aquest exercici.

Per a visualitzar les coses d'una forma un poc més agradable anem a utilitzar les llibreries gràfiques **Java AWT** i **Java Swing**. Amb aquestes llibreries incorporarem:

- **JPanel** i **JScrollPane**, per a contenir altres objectes (aquest últim amb botons de scroll si són necessaris).
- **JLabel**, per a etiquetes
- **JText**, per a quadres de text (on poder posar informació)
- **JTextArea**, és un quadre de text, però més gran. Nosaltres l'utilitzarem per a visualitzar informació extensa, i per tant el farem no editable.
- **JButton**, per a botons. Estarà esperant a que l'apremem (ActionListener)

Tot açò anirà en una classe que anomenarem **Pantalla_Veure_Pobles_Comarca** que estendrà **JFrame**, i que implementarà **ActionListener**.

Haurem de tenir una classe principal, **Veure_Pobles_Comarca.java**, que únicament crearà un objecte **Pantalla_Veure_Pobles_Comarca** i l'iniciará (mètode **iniciar()**), on col·loquem les primeres coses i prepararem el botó per a que escolte si s'apreta.

Per tant, tindrem una classe **Veure_Pobles_Comarca.java** tan senzilla com açò:

```
package Exercicis;
```

```

public class Veure_Pobles_Comarca {

    public static void main(String[] args) {
        final Pantalla_Veure_Pobles_Comarca finestra = new
Pantalla_Veure_Pobles_Comarca();
        finestra.iniciar();
    }

}

```

I ara anem a per la classe **Pantalla_Veure_Pobles_Comarca.java**. Vosaltres haureu de completar el mètode **VisualitzaCom(String comarca)**, per a que si no existeix la comarca, es diga que no existeix dins del JTextArea, i si existeix, que es vegi **els seus pobles** (si vols pots millorar-lo per a que es vegi en ordre alfabètic).

```

package Exercicis;

import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.Arrays;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;

public class Pantalla_Veure_Pobles_Comarca extends JFrame implements
ActionListener {

    private static final long serialVersionUID = 1L;
    JLabel etiqueta = new JLabel("Comarca:");
    JLabel et_ini = new JLabel("Introdueix la comarca:");
    JTextField com = new JTextField(15);
    JButton consultar = new JButton("Consultar");
    JTextArea area = new JTextArea();

    // en iniciar posem un contenidor per als elements anteriors
    public void iniciar() {
        this.setBounds(100, 100, 450, 300);
        this.setLayout(new BorderLayout());
        // contenidor per als elements
        JPanel panell1 = new JPanel(new FlowLayout());
        panell1.add(et_ini);
        panell1.add(com);
        panell1.add(consultar);
        getContentPane().add(panell1, BorderLayout.NORTH);

        JPanel panell2 = new JPanel(new BorderLayout());
        panell2.add(etiqueta, BorderLayout.NORTH);
        area.setForeground(Color.blue);
        JScrollPane scroll = new JScrollPane(area);
        panell2.add(scroll, BorderLayout.CENTER);
        getContentPane().add(panell2, BorderLayout.CENTER);

        setVisible(true);
        consultar.addActionListener(this);
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == consultar) {
            etiqueta.setText("Comarca: " + com.getText());
            visualitzaCom(com.getText());
        }
    }

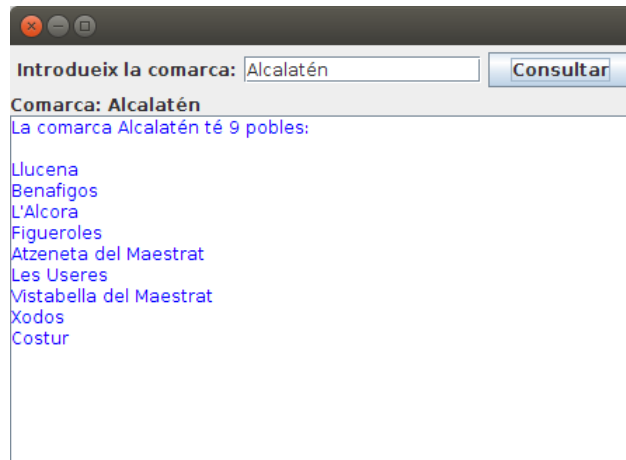
    private void visualitzaCom(String comarca) {

        // Instruccions per a llegir la comarca introduïda (s'ha de
deixar en un objecte Comarques).
        // S'ha de cuidar que si no existeix la comarca, en el JTextArea
es pose que no existeix.
        // La manera d'anar introduint informació en el JTextArea és
area.append("Linia que es vol introduir ")

```

```
}
}
```

Aquest seria un exemple d'utilització:



Exercici 5.3

Anem a fer unes quantes millores a l'exemple anterior. Copia't la classe **Pantalla_Veure_Pobles_Comarca.java** a aquesta altra: **Pantalla_Veure_Pobles_Comarca_Combo_List.java**

Les millores seran les següents:

- (70%) Substitueix el quadre de text per un **JComboBox**, i elimina el botó. Recorda que el component **JComboBox** funciona de la següent manera:
 - La manera de saber si ha estat el ComboBox qui ha provocat l'esdeveniment en el **actionPerformed** és idèntica que en el cas del botó: **if (e.getSource()==nom_ComboBox)**
 - Per a obtenir l'element seleccionat utilitzarem el mètode **.getSelectedItem()**
 - Per tant, per a omplir el ComboBox haurem de fer una consulta prèvia amb els noms de les comarques (en principi amb els noms és suficient). El resultat de la consulta pot ser un List de strings, ja que només volem el nom de la comarca. Aquesta llista la podem convertir en un array amb el mètode **.toArray()**, que és el que li fa falta al constructor del ComboBox. Però també podeu anar afegint elements al ComboBox amb el mètode **.addItem()**
- (15%) Substitueix el **JTextArea** on mostrem els pobles de la comarca per un **JList**. Aquest component serà una llista, i podrem seleccionar cada element de la llista. El seu funcionament és així:
 - Per a anar omplint el **JList**, no és tant senzill com el **JComboBox**. Ens fa falta un **DefaultListModel**, i construir el **JList** a partir d'ell. Posteriorment afegirem elements al **DefaultListModel**, i això suposarà que es veuran en el **JList**.
 - Per a agafar els pobles de la comarca, pot donar-se el cas que hi haja cometes simples en el nom (p.e. Plana d'Utiel). Açò faria que la consulta HQL falle (ja que la cometa simple és el delimitador de les constants de text). Una manera de solucionar-lo és utilitzar un paràmetre en un **PreparedStatement**, ja que d'aquesta manera no hi haurà problema amb les cometes. Una altra manera de solucionar-lo és posar dos cometes simples. Podem utilitzar el mètode **replaceAll()** de la classe String, substituint tota cometa simple per dues cometes simples. Si per exemple tenim el nom de la comarca en el String **comarca**, fariem:

```
comarca.replaceAll("'", "' ' ") :
```

- (15%) Quan se seleccione un element del **JList**, fes que es mostre baix de tot el número d'Instituts del

poble seleccionat

- Per a poder controlar si s'ha seleccionat un element del **JList** la classe principal haurà d'implementar també **ListSelectionListener**, a banda del **ActionListener**. Prepararem que volem escoltar aquest esdeveniment amb **area.addListSelectionListener(this)**; (si el **JList** s'anomena **area**, clar). Posteriorment sobreescrivem el mètode **valueChanged(ListSelectionEvent e)**, que és on posarem les accions per a mostrar el nombre d'Instituts del poble.
- El nom del poble també pot dur cometes simples, per tant farem la mateixa consideració d'abans. Ho solucionem amb un paràmetre de **PreparedStatement**, o posar dues cometes simple substituint el nom del poble per:

```
poble.replaceAll("'", "'') :
```

L'esquelet quedarà ara d'aquesta manera:

```
package Exercicis;

import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.Arrays;
import java.util.List;

import javax.swing.DefaultListModel;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JList;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextField;
import javax.swing.event.ListSelectionEvent;
import javax.swing.event.ListSelectionListener;

import org.hibernate.Session;

import dades.*;

public class Pantalla_Veure_Pobles_Comarca_Combo_List extends JFrame
implements ActionListener, ListSelectionListener{

    private static final long serialVersionUID = 1L;
    JLabel etiqueta = new JLabel("Comarca:");
    JLabel et_ini = new JLabel("Introdueix la comarca:");
    JComboBox com = new JComboBox();
    DefaultListModel listModel = new DefaultListModel();
    JList area = new JList(listModel);
    JTextField peu = new JTextField();

    // en iniciar posem un contenidor per als elements anteriors
    public void iniciar() {
        this.setBounds(100, 100, 450, 300);
        this.setLayout(new BorderLayout());
        // contenidor per als elements
        JPanel panell1 = new JPanel(new FlowLayout());
        panell1.add(et_ini);
        panell1.add(com);
        getContentPane().add(panell1, BorderLayout.NORTH);

        JPanel panell2 = new JPanel(new BorderLayout());
        panell2.add(etiqueta, BorderLayout.NORTH);
        area.setForeground(Color.blue);
        JScrollPane scroll = new JScrollPane(area);
        panell2.add(scroll, BorderLayout.CENTER);
        getContentPane().add(panell2, BorderLayout.CENTER);
        getContentPane().add(peu, BorderLayout.SOUTH);

        agafarComarques();

        setVisible(true);

        com.addActionListener(this);
        area.addListSelectionListener(this);
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == com) {
```

```

        etiqueta.setText("Llista de pobles de la comarca: " +
com.getSelectedItemAt());
        visualitzaCom(com.getSelectedItemAt().toString());
    }

    @Override
    public void valueChanged(ListSelectionEvent e){
        JList l = (JList) e.getSource();
        if (l.getSelectedIndex() >= 0){
            visualitzaInstituts(l.getSelectedValue().toString());
        }
    }

    private void visualitzaCom(String comarca) {

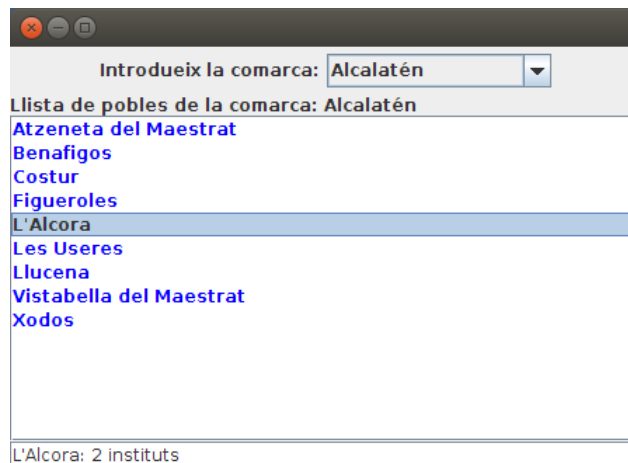
        // Instruccions per a llegir la comarca introduïda (s'ha de
deixar en un objecte Comarques).
        // S'ha de cuidar que si no existeix la comarca, en el JList es
pose que no existeix.
        // La manera d'anar introduint informació en el JList és a
través del DefaultListModel:
        // listModel.addElement("Linia que es vol introduir ")
        // Una manera de solucionar el problema de la cometa simple és
utilitzar comarca.replaceAll("'", "'").
        // Una altra és utilitzar paràmetres amb PreparedStatement
    }

    private void agafarComarques(){
        // Instruccions per a posar en el ComboBox el nom de totes les
comarques, millor si és per ordre alfabètic
    }

    private void visualitzaInstituts(String poble){
        // Instruccions per a mostrar el número d'Instituts del poble
seleccionat
        // Una manera de solucionar el problema de la cometa simple és
utilitzar poble.replaceAll("'", "'").
        // Una altra és utilitzar paràmetres amb PreparedStatement
    }
}

```

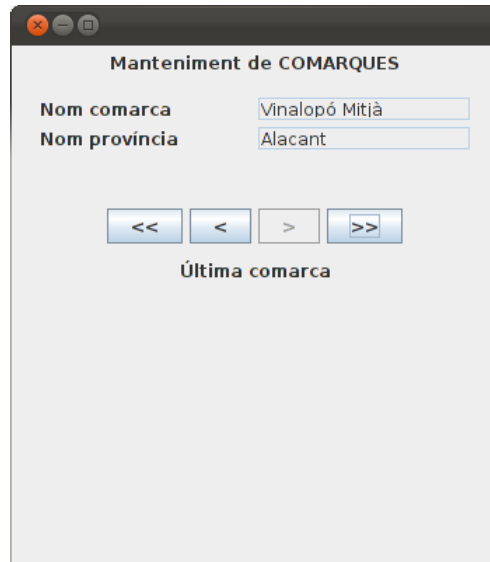
Aquest seria un exemple d'execució:



Exercici 5.4

Anem a intentar un programa per al manteniment de les comarques (no intentarem mantenir els pobles de la comarca, únicament el nom i la província de la comarca). L'anomenarem **Manteniment_Comarques** al principal, i **Manteniment_Comarques_Pantalla** al que ho contindrà tot i estendrà de JPanel. Ambdós han d'estar en el paquet **Exercicis** del projecte **Tema5_Hibernate_PostgreSQL_geo_ad**.

En una primera fase únicament posarem 2 quadres de text per al nom i la província (amb 2 etiquetes) i 4 botons per a desplaçar-nos: al primer, a l'anterior, al següent i a l'últim.



Ves fent progressivament les següents coses:

- Organitza les diferents seccions en diferents JPanelels.
- Per a que queden aliniats els JLabel i JTextField, segurament el més senzill és col·locar-los en un JPanel d'estil **GridLayout**, amb 2 columnes: **JPanel panell2 = new JPanel(new GridLayout(2,2,5,5));**
- Col·loca els JLabel i JText necessaris en el JPanel anterior.
- Posa en una llista de Comarques tots els objectes de les comarques (per mig d'una consulta HQL). Inicialment situa't en la primera.
- Posa i implementa els 4 botons de desplaçament.
- Controla que no es puga eixir fora del rang deshabilitant els botons corresponents quan s'estiga en la primera o l'última comarca. No t'oblides d'habilitar-los quan toque. Opcionalment pots posar una etiqueta que diga si s'està en la primera o última comarca.

Ací tens el programa principal, així com "l'esquelet" del JFrame. Observa que la sessió està creada en la zona de declaracions de les propietats. Podríem haver-la creat en el mètode **afafarComarques()**, que és quan per mig d'una sentència **HQL** agafarem totes les comarques i les posarem en un **ListArray<Comarques>**. D'aquesta manera podríem tancar la sessió en el mateix mètode **agafarComarques()** i no utilitzar més recursos dels necessaris. Hem optat per tenir la sessió oberta durant tota la durada del programa per fer-lo igual que en el següent exercici, en el qual sí que ens convé tenir la sessió oberta mentre dure el programa. Observeu també que a banda de **llistaComarques**, l'arraylist on estan tots els objectes comarques, també tenim la propietat **indActual**, on guardem l'índex actual, és a dir, l'índex de la comarca que s'ha d'estar visualitzant en aquest moment.

```
package Exercicis;

public class Manteniment_Comarques {
    public static void main(String[] args) {
        Manteniment_Comarques Pantalla finestra = new
Manteniment_Comarques_Pantalla();
        finestra.iniciar();
    }
}
```

```
package Exercicis;

import java.awt.BorderLayout;
import java.awt.FlowLayout;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.ArrayList;

import javax.swing.JButton;
import javax.swing.JFrame;
```

```

import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;

import org.hibernate.Query;
import org.hibernate.Session;

import Exemples.SessionFactoryUtil;
import dades.Comarca;

public class Manteniment_Comarques_Pantalla extends JFrame implements
ActionListener{
    JLabel etIni= new JLabel("Manteniment de COMARQUES");
    JLabel etNom= new JLabel("Nom comarca");
    JLabel etProv= new JLabel("Nom província");

    JTextField nomComarca = new JTextField();
    JTextField nomProvíncia= new JTextField();

    JButton primer = new JButton("<<");
    JButton anterior = new JButton("<");
    JButton seguent = new JButton(">");
    JButton ultim = new JButton(">>");

    JPanel pDalt = new JPanel(new FlowLayout());
    JPanel pCentre = new JPanel(new GridLayout(8,0));
    JPanel pDades = new JPanel(new GridLayout(2,2));
    JPanel pBotonsMov = new JPanel(new FlowLayout());

    Session s = SessionFactoryUtil.getSessionFactory().openSession();

    ArrayList<Comarca> llistaComarques = new ArrayList<Comarca>();
    int indActual=0;

    public void iniciar(){
        this.setBounds(100, 100, 350, 400);
        this.setLayout(new BorderLayout());

        this.getContentPane().add(pCentre, BorderLayout.CENTER);
        this.getContentPane().add(new JPanel(new FlowLayout()),
BorderLayout.WEST);
        this.getContentPane().add(new JPanel(new FlowLayout()),
BorderLayout.EAST);

        pDalt.add(etIni);
        pCentre.add(pDalt);

        pDades.add(etNom);
        pDades.add(nomComarca);
        pDades.add(etProv);
        pDades.add(nomProvíncia);
        pCentre.add(pDades);

        nomComarca.setEditable(false);
        nomProvíncia.setEditable(false);

        pCentre.add(new JPanel(new FlowLayout()));

        pBotonsMov.add(primer);
        pBotonsMov.add(anterior);
        pBotonsMov.add(seguent);
        pBotonsMov.add(ultim);
        pCentre.add(pBotonsMov);

        pCentre.add(new JPanel(new FlowLayout()));

        llistaComarques = agafarComarques();
        visComarca(indActual);

        this.setVisible(true);
        this.setDefaultCloseOperation(EXIT_ON_CLOSE);

        primer.addActionListener(this);
        anterior.addActionListener(this);
        seguent.addActionListener(this);
        ultim.addActionListener(this);
    }

    private void visComarca(int index) {
        // Mètode per a visualitzar la comarca marcada per l'índex que
ve com a paràmetre
        // Es pot aprofitar per a activar/desactivar els botons
anterior i seguent, si s'està en la primera o última comarca

```

```

    }

    private ArrayList<Comarca> agafarComarques() {
        ArrayList<Comarca> llista = null;
        // ací aniran les sentències per a omplir (i retornar) la
        llista de comarques
        return llista;
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        //ací aniran les accions de moviment primer, anterior, següent
        i últim
        if (e.getSource()==primer){
        }

        if (e.getSource()==anterior){
        }

        if (e.getSource()==següent){
        }

        if (e.getSource()==ultim){
        }
    }
}

```



Exercici 5.5 (voluntari)

Modifica l'anterior construint **Manteniment_Comarques_Avançat_Pantalla** per a posar també botons per a Inserir noves comarques, esborrar o modificar. En un primer moment aquestos botons prepararan l'acció, però no la realitzaran.

- Col·loca primer els 3 botons.
- Quan s'aprete (qualsevol dels 3) es visualitzaran uns altres dos (ja creats, ara es visualitzaran) que seran els d'Acceptar i Cancel·lar. A més d'això:
 - el d'inserir ha de netejar els quadres de text i si havíeu posat els quadres de text com a que no es puguin editar, doncs ara que sí.
 - el de modificar ha de permetre l'edició de la província (si no ho estava)
- En apretar Cancel·lar, s'ha de tornar a l'estat anterior, i fer invisibles els botons d'Acceptar i Cancel·lar.
- En apretar Acceptar, s'ha de distingir quina acció s'està fent, inserir, esborrar o modificar, i realitzar tal acció en la Base de Dades. Posteriorment s'hauran de fer invisibles els botons d'Acceptar i Cancel·lar
- Seria molt útil que en quan s'està fent una actualització (Inserir, Modificar o Esborrar), es desactiven tant els botons de moviment com d'actualització. En apretar Acceptar o Cancel·lar a més de fer les accions

ciutades abans, s'haurien de tornar a activar els botons de moviment i d'actualització.

Ací està "l'esquelet" del JFrame. Podeu utilitzar el mateix programa principal.

```
package Exercicis;

import java.awt.BorderLayout;
import java.awt.FlowLayout;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.ArrayList;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JTextField;

import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.Transaction;

import dades.SessionFactoryUtil;
import dades.Comarca;

public class Manteniment_Comarques_Avançat_Pantalla extends JFrame
implements ActionListener{
    JLabel etIni= new JLabel("Manteniment de COMARQUES");
    JLabel etNom= new JLabel("Nom comarca");
    JLabel etProv= new JLabel("Nom província");

    JTextField nomComarca = new JTextField();
    JTextField nomProvíncia= new JTextField();

    JButton primer = new JButton("<<");
    JButton anterior = new JButton("<");
    JButton següent = new JButton(">");
    JButton ultim = new JButton(">>");

    JButton inserir = new JButton("Inserir");
    JButton modificar = new JButton("Modificar");
    JButton esborrar = new JButton("Esborrar");

    JButton acceptar = new JButton("Acceptar");
    JButton cancel·lar = new JButton("Cancel·lar");

    JPanel pDalt = new JPanel(new FlowLayout());
    JPanel pCentre = new JPanel(new GridLayout(8,0));
    JPanel pDades = new JPanel(new GridLayout(2,2));
    JPanel pBotonsMov = new JPanel(new FlowLayout());
    JPanel pBotonsAct = new JPanel(new FlowLayout());
    JPanel pBotonsAccCanc = new JPanel(new FlowLayout());

    Session s = SessionFactoryUtil.getSessionFactory().openSession();
```

```

        ArrayList<Comarca> llistaComarques = new ArrayList<Comarca>();
        int indActual=0;
        String accio=null;

        public void iniciar(){
            this.setBounds(100, 100, 350, 400);
            this.setLayout(new BorderLayout());

            this.getContentPane().add(pCentre, BorderLayout.CENTER);
            this.getContentPane().add(new JPanel(new FlowLayout()),
BorderLayout.WEST);
            this.getContentPane().add(new JPanel(new FlowLayout()),
BorderLayout.EAST);

            pDalt.add(etIni);
            pCentre.add(pDalt);

            pDades.add(etNom);
            pDades.add(nomComarca);
            pDades.add(etProv);
            pDades.add(nomProvincia);
            pCentre.add(pDades);

            nomComarca.setEditable(false);
            nomProvincia.setEditable(false);

            pCentre.add(new JPanel(new FlowLayout()));

            pBotonsMov.add(primer);
            pBotonsMov.add(anterior);
            pBotonsMov.add(següent);
            pBotonsMov.add(ultim);
            pCentre.add(pBotonsMov);

            pBotonsAct.add(inserir);
            pBotonsAct.add(modificar);
            pBotonsAct.add(esborrar);
            pCentre.add(pBotonsAct);

            pBotonsAccCanc.add(acceptar);
            pBotonsAccCanc.add(cancelar);
            pCentre.add(pBotonsAccCanc);
            pBotonsAccCanc.setVisible(false);

            pCentre.add(new JPanel(new FlowLayout()));

            llistaComarques = agafarComarques();
            visComarca(indActual);

            this.setVisible(true);
            this.setDefaultCloseOperation(EXIT_ON_CLOSE);

            primer.addActionListener(this);
            anterior.addActionListener(this);
            següent.addActionListener(this);
            ultim.addActionListener(this);

            inserir.addActionListener(this);
            modificar.addActionListener(this);
            esborrar.addActionListener(this);

            acceptar.addActionListener(this);
            cancelar.addActionListener(this);
        }

        private void visComarca(int index) {
            // Mètode per a visualitzar la comarca marcada per l'índex que
ve com a paràmetre
            // Es pot aprofitar per a activar/desactivar els botons
anterior i següent, si s'està en la primera o última comarca
        }

        private ArrayList<Comarca> agafarComarques() {
            ArrayList<Comarca> llista = null;
            // ací aniran les sentències per a omplir (i retornar) la
llista de comarques
            return llista;
        }

        @Override
        public void actionPerformed(ActionEvent e) {
            //ací aniran les accions de moviment primer, següent
i últim

```

```

        if (e.getSource()==primer){
        }

        if (e.getSource()==anterior){
        }

        if (e.getSource()==seguent){
        }

        if (e.getSource()==ultim){
        }

        if (e.getSource()==inserir){
        }

        if (e.getSource()==modificar){
        }

        if (e.getSource()==esborrar){
        }

        if (e.getSource()==acceptar){
        }

        if (e.getSource()==cancelar){
        }
    }

    private int BuscaCom(String text) {
        // Busca la comarca passada com a paràmetre en llistaComarques,
        // tornant el seu índex. Per a situar-se després d'una inserció.
        int ind =0;
        // Ací haurien d'anar les sentències
        return ind;
    }

    private void activarBotons(boolean b) {
        //Mètode per activar o desactivar (segons el paràmetre) els
        //botons de moviment i també els d'actualització
    }
}

```



Exercici 5.6 (voluntari)

Crea un projecte nou anomenat **Tema5_PostgreSQL_Autoescola**. Inclou el driver JDBC de PostgreSQL i la llibreria nostra d'Hibernate.

- Fes el mapatge de totes les taules, situades en la Base de Dades **autoescola** a la qual pot accedir l'usuari **autoescola** (contrasenya **autoescola**) i situada en el servidor de l'Institut (**89.36.214.106**). Guarda-les en un paquet anomenat **dades**. El mapatge serà de 5 taules: **ALUMNE**, **EXAMEN**, **PRACTIQUES**, **PROFESSOR** i **VEHICLE**.
- Crea't la classe **SessionFactoryUtil**, que serà un **singleton** per a poder crear una única SessionFactory. L'has d'utilitzar en totes les altres classes, sempre que et faci falta un SessionFactory. Ha d'estar en el mateix paquet **dades**.
- En un paquet anomenat **Exercicis**, crea't la classe **ConsultaAutoescola**, que serà un programa senzill que visualitzarà els professors de l'autoescola i els seus alumnes, ordenats pel nom del professor. Completarem la informació amb els quilòmetres realitzats pels alumnes en la totalitat de les seues pràctiques i el número d'exàmens que ha realitzat. El resultat seria aquest:

Duch Ramell, Albert		
Contreras Mifsud, Beatriu	52 km. de pràctiques.	1 exàmens.
Villalonga Mas, Carme	31 km. de pràctiques.	0 exàmens.
March Albiol, Lluïsa		
Abad Tomàs, Anna	50 km. de pràctiques.	1 exàmens.
Montoliu Pujol, Sara	68 km. de pràctiques.	2 exàmens.
Romeu Gas, Victòria	0 km. de pràctiques.	0 exàmens.
Segarra Martí, Pere	76 km. de pràctiques.	2 exàmens.
Vinaixa Garbell, Robert	0 km. de pràctiques.	0 exàmens.
Pou Bernat, Andreu		
Renau Gisbert, Àngel	102 km. de pràctiques.	3 exàmens.
Saura Gumbau, Rosa	84 km. de pràctiques.	1 exàmens.
Sorribes Vinaixa, Ricard	15 km. de pràctiques.	1 exàmens.

Llicenciat sota la [Llicència Creative Commons Reconeixement NoComercial CompartirIgual 2.5](#)