
# Università degli Studi di Bologna

## Facoltà di Ingegneria

*Tecnologie Web L-A*

*A.A. 2019 – 2020*

## Esercitazione 08

## DAO e Hibernate

# Agenda

- **Pattern DAO e libreria Hibernate**
  - **progetto d'esempio** relativo alla «gestione dei dati degli studenti»
  - veloce ripasso di alcuni concetti chiave già visti a lezione
  - **proposta esercizi da svolgere in autonomia** relativo alla «gestione dei corsi universitari», a partire dal progetto d'esempio
  - **esercizio guidato tratto dall'appello d'esame** del 25 luglio 2016: gestione persistenza basata su Hibernate

# Progetto di esempio

- Il file ***08_TecWeb.zip*** contiene lo scheletro di un semplice progetto di esempio basato sull'uso del pattern DAO e di Hibernate
  - creato con Eclipse, contiene già tutti i descrittori necessari a essere riconosciuto e configurato correttamente

- Importare il progetto come visto nelle precedenti esercitazioni, senza estrarre l'archivio su file system (lo farà Eclipse)
  - *File → Import → General → Existing Projects into Workspace → Next → Select archive file*

- Progetto (nella sua versione «completa») per la gestione del database di corsi universitari
  - elenco dei corsi attivi
  - elenco degli studenti
  - mapping tra corsi e studenti che li frequentano

# Metodologia “forza bruta”

- Nella scorsa esercitazione abbiamo visto come sia possibile gestire una tabella di studenti tramite API JDBC
- Metodi Java appositamente creati per
  - connettersi a database
  - generare/eliminare tabelle
  - modificare tali tabelle (insert, update, delete)
  - interrogare tali tabelle (ad esempio query per ottenere tutti gli studenti con un certo cognome)
- Utilizzare direttamente le API JDBC è certamente possibile, ma generalmente porta a codice
  - poco leggibile
  - di difficile manutenzione
  - complesso in fase di estensione (ad esempio, per supportare un nuovo DBMS)

# DAO & Hibernate

- Uso di Pattern (ad esempio, DAO) o librerie (ad esempio, Hibernate) rende più facile l'accesso a DBMS
  - uso di tecniche di programmazione o API che rendono più semplice e meno *error-prone* l'accesso a database
  - accesso a database trasparente (per quanto possibile) rispetto al particolare DBMS in uso
  - riconfigurazione facile e veloce se si vuole utilizzare un DBMS diverso da quello precedentemente in uso

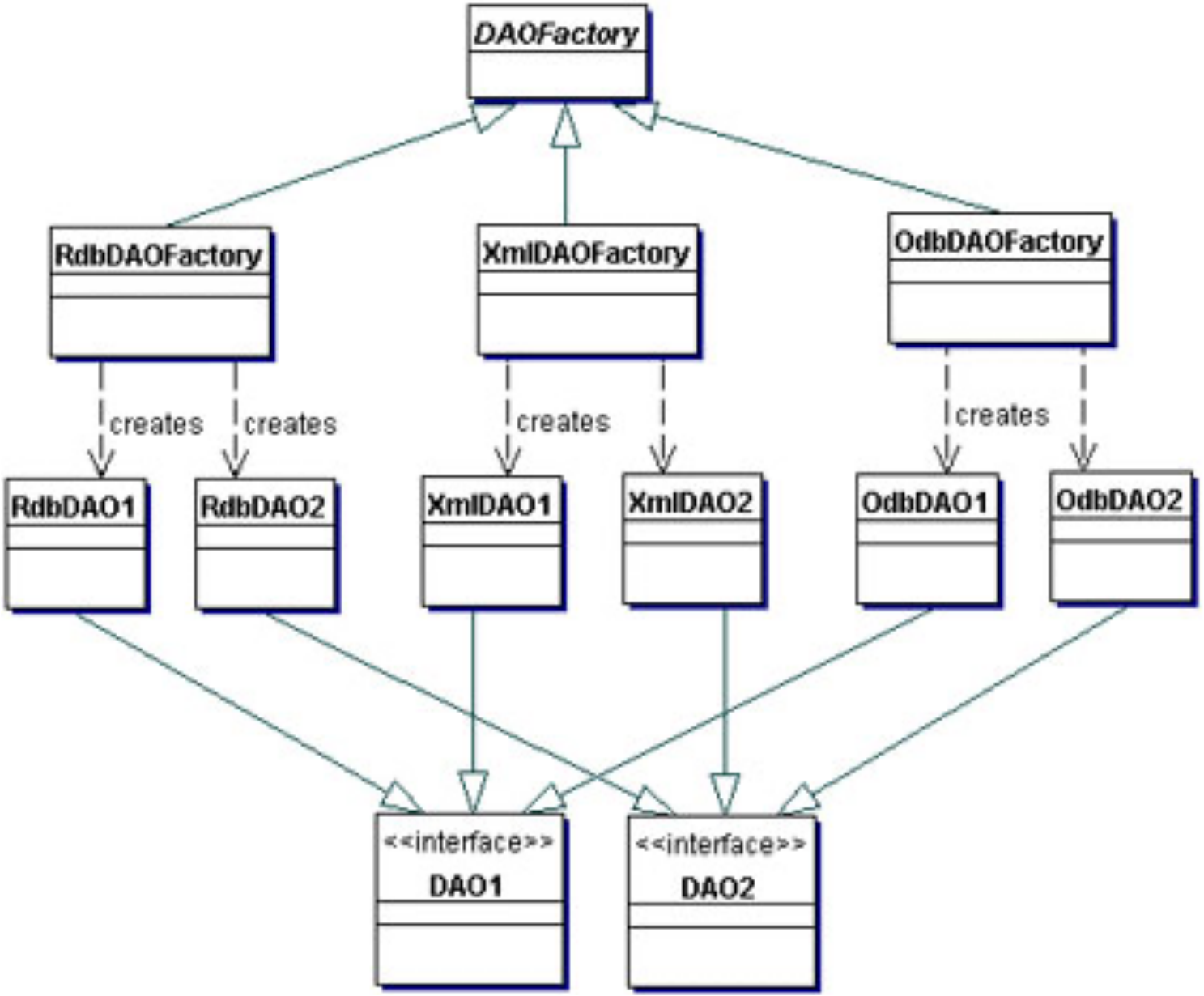
# Pattern DAO con classi Factory (1)

Veloce ripasso di alcuni concetti chiave già visti a lezione

- DTO: oggetto scambiato come Java Bean
- DAO: componente che offre i metodi per scambiare DTO tra applicazione Java e DBMS

- Una unica factory astratta
  - fornisce specifiche per le factory concrete
  - espone un metodo creazionale parametrico per ottenere factory concrete
- Una factory concreta per ogni tipo di DBMS supportato
  - permette di ottenere oggetti DAO appropriati al corrispondente tipo di DBMS
  - può gestire aspetti quali ottenimento della connessione, autenticazione, ...
- Un oggetto DTO per ogni tipo di entità che si vuole rappresentare
- Una interfaccia DAO per ogni oggetto DTO
- Una implementazione dell'interfaccia DAO di un DTO per ciascun DB supportato

# Pattern DAO con classi Factory (2)



# DAO Project (1)

Il progetto riprende lo **StudentRepository** della scorsa settimana

- L'oggetto **`it.unibo.tw.dao.StudentDTO`** **funge da trasporto** tra l'applicazione e la tabella **students** (è un Java Bean al 100%; suffisso DTO aggiunto solo per chiarezza)
- Sono previste **tre versioni degli oggetti DAO**, una per DB2 (completa), una per HSQLDB (appena accennata) e una per MySQL (appena accennata)
  - ai fini di questa semplice applicazione, i tre DBMS accedono ai database nello stesso modo
- **La scelta dell'implementazione da utilizzare** dipende da un particolare parametro specificato in fase di instanziazione del DAOFactory

# DAO Project (2)

- Vogliamo accedere alla tabella di nome "xxx" (ad esempio "students" che risiede in un DBMS di tipo "dbmsName" (ad esempio "DB2")

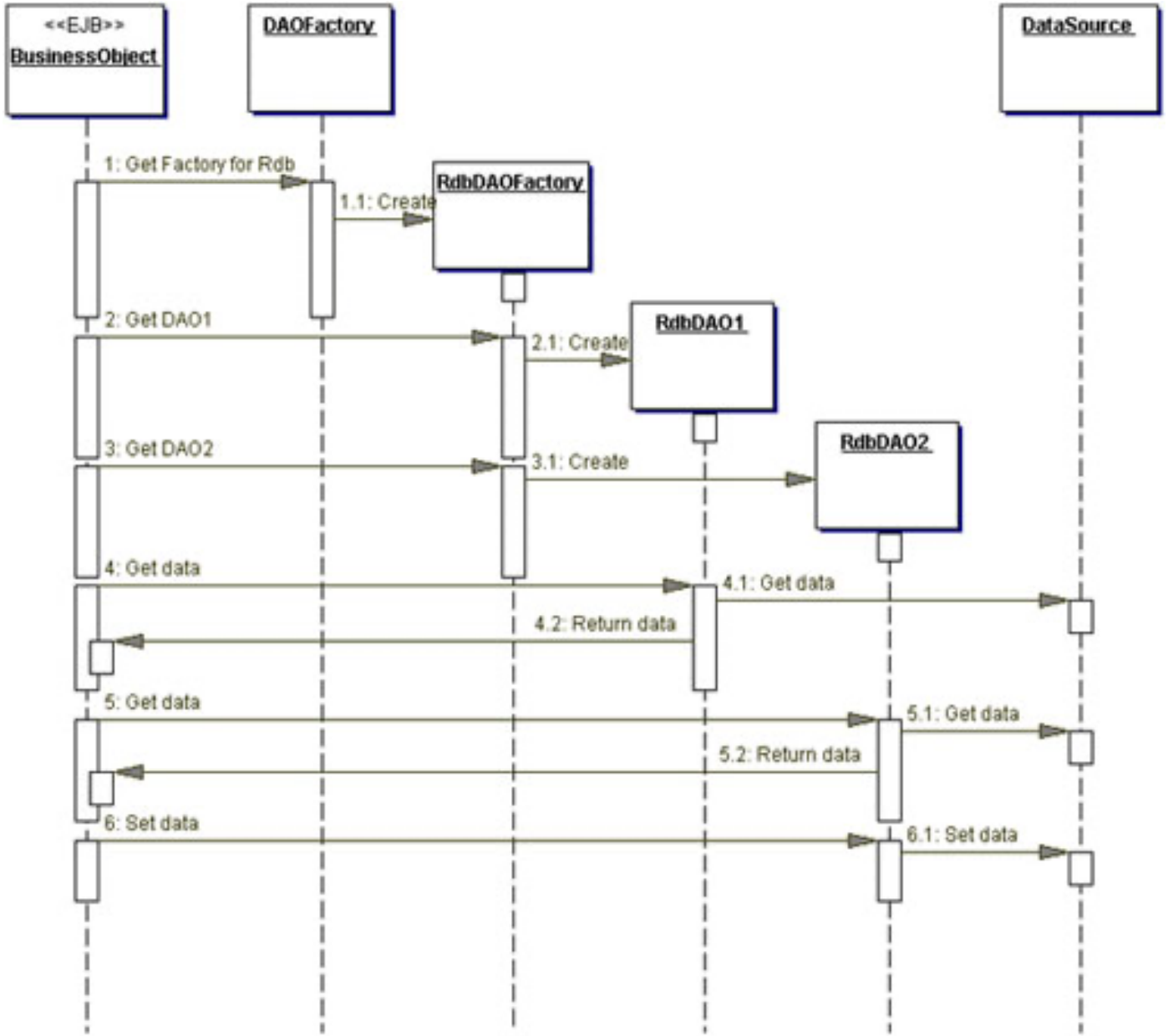
- Un unico **DAOFactory** per applicazione
  - entry-point ai componenti che supportano la persistenza
- Un oggetto Java Bean **xxxDTO** per tabella
  - rappresentazione object-oriented di **una riga** della tabella "xxx"
- Una interfaccia **xxxDAO** per tabella, coi metodi d'accesso alla tabella
  - interfaccia unica per tutti i DBMS, ma **differente tabella per tabella**
- Un dbmsNameDAOFactory per DBMS che si vuole supportare
  - instanziato da DAOFactory, specifica driver JDBC, URI, username, password...
- Per ciascun DBMS, un dbmsNamexxxDAO che implementa l'interfaccia xxxDAO corrispondente
  - implementazione concreta dei metodi dichiarati in xxxDAO per accedere alla tabella "xxx" del DBMS "dbmsName"
  - l'implementazione dei metodi può variare a seconda del dialetto SQL utilizzato dallo specifico DBMS

# DAO Project (3)

- Il package **it.unibo.tw.dao** contiene classi e interfacce comuni a tutti i DBMS
- Il package **it.unibo.tw.dao.dbmsName** contiene le classi specifiche necessarie per accedere al DBMS dbmsName

- *All'interno della classe DAOTest, osservate come avviene l'accesso alla tabella **students***
  1) tramite metodo statico DAOFactory si ottiene un riferimento ad una classe **specifica per un particolare DBMS** che estende la classe astratta DAOFactory
  2) tramite tale instanza di DAOFactory si ottiene un riferimento ad una classe che **implementa l'interfaccia StudentDAO**
  3) infine tramite un'instanza della classe che implementa l'interfaccia StudentDAO, si accede alla tabella tramite i metodi offerti dall'oggetto ottenuto

# Pattern DAO con classi Factory: diagramma di sequenza



# Implementazione degli oggetti DAO

Un consiglio su come procedere nello scrivere il codice degli oggetti DAO basati su JDBC

**Ogni metodo**

1. dichiari una variabile dove collocare il proprio risultato
2. controlli la bontà dei parametri attuali ricevuti
3. apra la connessione al database
4. formuli gli statement SQL che lo riguardano e imposti il risultato
5. preveda di gestire le eventuali eccezioni
6. rilasci SEMPRE E IN OGNI CASO la connessione in uso
7. restituisca il risultato (eventualmente di fallimento)

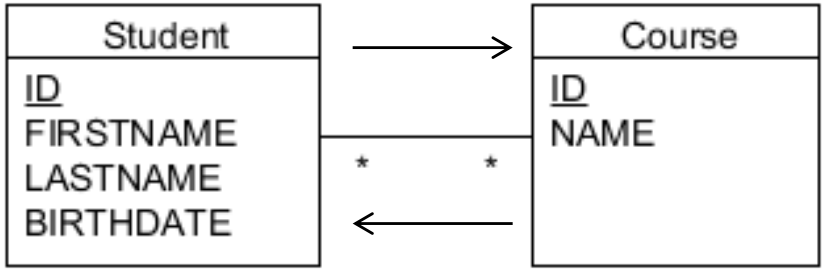
E per quanto riguarda gli **statement SQL** veri e propri

1. crei (se senza parametri) o prepari (se con parametri) lo statement da proporre al database
2. pulisca e imposti i parametri (se ve ne sono, ovviamente)
3. esegua l'azione sul database ed estragga il risultato (se atteso)
4. cicli sul risultato (se presente) per accedere a ogni sua tupla e impostare il proprio risultato con i valori in essa contenuti
5. rilasci la struttura dati del risultato stesso
6. rilasci la struttura dati dello statement

- Potete ritrovare questo schema di operazioni nei commenti a corredo del codice

# Ora a voi: DAO (1/2)

- Partendo dal progetto di esempio, e considerando il diagramma UML di seguito riportato relativo alla “gestione dei corsi universitari”, si richiede di




- estendere le funzionalità dell'applicazione esistente in modo tale da fornire una soluzione alla gestione della persistenza basata su Pattern DAO
  - in grado di “mappare” efficientemente il modello di dominio rappresentato dai Java Bean **Student**, **Course** e **S-CMapping** (ovvero il mapping tra studenti e corsi) con le corrispondenti tabelle **Student**, **Course** e **S-CMapping** contenute nel DB **TW_STUD**

# Ora a voi: DAO (2/2)

- Nel dettaglio, dopo aver creato gli schemi delle tabelle all'interno del proprio schema nel database **TW_STUD**, implementato i Java Bean e realizzato le classi relative al Pattern DAO per l'accesso CRUD alle tabelle, si richiede la realizzazione di un metodo che permetta
  - dato l'ID di uno studente, di fornire l'elenco dei corsi frequentati
  - dato l'ID di un corso, di fornire l'elenco degli studenti che lo frequentano

- Note:
  - per ciascuna nuova tabella xxx è necessario creare un nuovo xxxDTO ed un nuovo xxxDAO nel package it.unibo.tw.dao
  - implementare xxxDAO almeno per il DBMS DB2, creando quindi una classe DB2CourseDAO ed una classe DB2MappingDAO nel package it.unibo.tw.dao.db2

# Hibernate: hibernate.cfg.xml

- Nel package **it.unibo.tw.hibernate** accesso a DBMS con hibernate
- La classe **Student** è un Java Bean, identico a **StudentDTO**

- Il file **hibernate.cfg.xml** specifica tutto il necessario per identificare ed accedere ad un database
  - driver JDBC, URI, username, password, dialetto SQL...
  - posizionato di default nella directory **src**

```
<property name="connection.driver_class">
   com.ibm.db2.jcc.DB2Driver
</property>
<property
   name="connection.url">jdbc:db2://diva.deis.unibo.it:50000/tw_stud
</property>
<property name="connection.username">username</property>
<property name="connection.password">password</property>
```

# Hibernate: Student.hbm.xml

- Inoltre il file **hibernate.cfg.xml** identifica altri file XML di configurazione che, a loro volta, dichiarano il mapping tra classi Java Bean e specifiche tabelle presenti nel database

```
<mapping resource="it/unibo/tw/hibernate/Student.hbm.xml"/>
```

- Il file **Student.hbm.xml** specifica il mapping tra oggetto Java Bean e tabella del database
  - posizionato di default nella directory corrispondente al package del Java Bean
  - nome della tabella
  - primary key della tabella
  - mapping tra proprietà del Java Bean e colonna della tabella

```
<class name="it.unibo.tw.hibernate.Student" table="students">
   <id name="id" column="id" />
   <property name="firstName" column="firstName"/>
   <property name="lastName" column="lastName"/>
   <property name="birthDate" column="birthDate"/>
</class>
```

# Hibernate: classe HibernateTest

- La classe **HibernateTest** crea e rende persistente due Java Bean Student e poi effettua query sul database
  1) si ottiene un sessione
  2) si inizia una transazione
  3) si richiede la persistenza di uno o più Java Bean
  4) si termina la transazione tramite commit
  5) eventualmente si inizia una nuova transazione (punto 2)
  6) infine si termina la sessione (vedi **finally**)
  - in caso di errore si richiede il rollback della transazione (vedi **catch**)

- Dopo il commit, viene richiesto **l'elenco degli studenti che hanno cognome "Gialli"**
  - Hibernate supporta query di tipo HQL ed SQL
    - con **HQL** il nome della tabella viene identificato tramite mapping via Java Bean e XML
    - con **SQL** è necessario specificare il nome della tabella (come in JDBC)
  - in alternativa è possibile effettuare la query specificando criteri di restrizione, in modo object-oriented al 100%
    - Criteria su classe Student (e quindi corrispondente tabella students), specificando la Restriction che il lastName deve essere Gialli

# Ora a voi: Hibernate

Prendendo spunto dai file XML di configurazione del progetto di esempio, estendete le funzionalità dell'applicazione analogamente a quanto già fatto per il progetto DAO

- Creare **due nuovi file di mapping**, uno per i corsi ed una per il mapping tra i corsi esistenti e gli studenti che li frequentano
- Modificare il **file di configurazione hibernate.cfg.xml** per includere anche i due nuovi file di configurazione appena creati
- **Estendere la classe HibernateTest** per ottenere
  - l'**elenco dei corsi** frequentati da uno studente (identificato tramite l'ID dello studente)
  - **l'elenco degli studenti** che frequentano un corso (identificato tramite l'ID del corso)

- API ufficiali di Hibernate (versione 4.3) al seguente link
  - http://docs.jboss.org/hibernate/core/4.3/javadocs/

# Variante esercizio: DAO/Hibernate

Risolvere l'esercizio precedente **utilizzando solo due Java Bean**

- **Student** e
- **Course**

ovvero **mantenendo esplicitamente i mapping M-N specificati nel diagramma UML di partenza**

# Appendice

## (esercizio guidato tratto dall'appello d'esame del 9 giugno 2016: gestione della persistenza basata su Patter DAO)

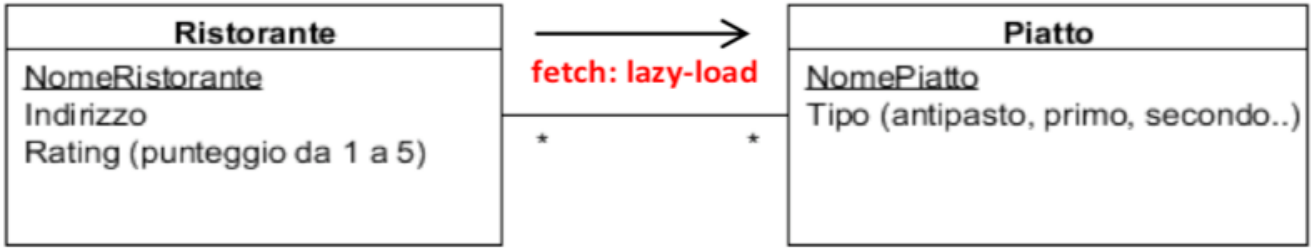
# Agenda

- Presentazione del Testo d'Esame
- Primo Approccio all'Esercizio
- Passi Principali:
  - Dall'UML al Pattern DAO
  - La Relazione M-N e il Verso di Percorrenza
  - Il Fetch Lazy-Load
  - Le Query

# Il Testo d'Esame

**ESERCIZIO 3 (11 punti)**

Partendo dalla realtà illustrata nel **diagramma UML** di seguito riportato, si fornisca una soluzione alla gestione della persistenza basata su **Pattern DAO** in grado di "mappare" efficientemente e con uso di ID surrogati il modello di dominio rappresentato dai **JavaBean Ristorante e Piatto del diagramma UML** con le corrispondenti **tabelle relazionali derivate dalla progettazione logica del diagramma** stesso.




Nel dettaglio, dopo aver creato da applicazione Java gli **schemi delle tabelle** all'interno del proprio schema nel database **TW_STUD** di **DB2** (esplicitando tutti i **vincoli** opportuni), **implementato** i **JavaBean** e **realizzato le classi** relative al **Pattern DAO** per **l'accesso CRUD** alle tabelle**,** si richiede l'implementazione di **opportuni metodi per il supporto delle seguenti operazioni**:

- per ogni ristorante sito in Bologna, si richiede la lista dei piatti di tipo "primo" offerti nel rispettivo menù; si richiede quanti ristoranti, nella fascia di rating compresa tra 4 e 5, offrono come tipo di secondo piatto "seppie con i piselli".

Si crei poi un **main di prova** che: (i) inserisca due o piu' tuple nelle tabelle di interesse; (ii) faccia uso corretto dei metodi realizzati al punto precedente al fine di produrre una stampa del risultato sul file **ristorante.txt**.

# Primo Approccio all'Esercizio

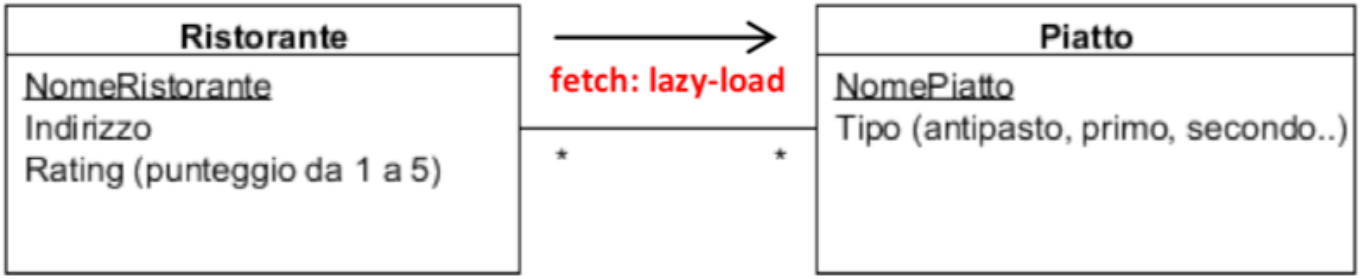
Leggendo il testo in prima battuta saltano subito all'occhio alcuni punti salienti:

- L'utilizzo del Pattern DAO
- La relazione M-N e il suo verso di percorrenza
- La tipologia di fetching: **Lazy-Load**

**Note:** abbiamo già visto cosa vuol dire utilizzare gli ID surrogati, la scrittura su file di testo dei risultati la diamo per assodata.

# Dall'UML al Patter DAO

Sicuramente la prima cosa da fare per affrontare l'esercizio è progettare lo schema relazionale dal diagramma UML utilizzando il Pattern DAO.




Dalla teoria sappiamo che il Pattern DAO ha due tipologie di entità Java (Classi ed Interfacce) principali:

- **Data Transfer Object (DTO):** sono gli oggetti che incapsulano i dati veri e propri
- **Data Access Object:** sono interfacce ed oggetti che mappano le operazioni relative alla persistenza sugli oggetti DTO

# Dall'UML al Patter DAO (2)

Come detto gli oggetti DTO incapsulano i dati veri e propri, nel nostro esercizio quindi avremo:

- **RistoranteDTO**
- **PiattoDTO**

A livello implementativo le classi Java conterranno tutti gli attributi indicati nell'UML, in più però, RistoranteDTO (solamente) conterrà una Collezione di PiattoDTO. In più c'è un altro DTO nascosco.

Dopo vedremo il perchè ;-)

Per le entità DAO bisognerà fare un'interfaccia ed una classe Java di implementazione (almeno ;-)) per ogni classe del diagramma UML, quindi nel nostro caso:

- Le interfacce: **RistoranteDAO** e **PiattoDAO**
- Le implementazioni: **Db2RistoranteDAO** e **Db2PiattoDAO**

# Dall'UML al Patter DAO: un po' di codice

```java
public class RistoranteDTO implements Serializable {

    /**
     *
     */
    private static final long serialVersionUID = 1L;

    private int id;
    private String nomeRistorante;
    private String indirizzo;
    private int rating;
    private List<PiattoDTO> piatti;
```

```java
public class PiattoDTO {

    private String nomePiatto;
    private String tipo;
    private int id;
```

```java
public interface RistoranteDAO {

    // --- CRUD ------------

        public void create(RistoranteDTO ristorante);

        public RistoranteDTO read(String nome);

        public boolean update(RistoranteDTO ristorante);

        public boolean delete(String nome);


        // ---------------------------------
        public List<RistoranteDTO> getResturantByCity(String citta);

        public List<RistoranteDTO> getRatedResturant(int stars);
        // ---------------------------------

        public boolean createTable();

        public boolean dropTable();

}
```

```java
public interface PiattoDAO {
    // --- CRUD ------------

    public void create(PiattoDTO piatto);

    public PiattoDTO read(String nome);

    public boolean update(PiattoDTO piatto);

    public boolean delete(String nome);


    // ---------------------------------



    public boolean createTable();

    public boolean dropTable();
}
```

# La relazione M-N e il Verso di Percorrenza

Abbiamo già visto che una relazione UML molti a molti si mappa con una classe extra nel mondo relazionale Java, detta classe di mapping.

Nel Pattern DAO una relazione molti a molti si mappa alla stessa maniera, aggiungendo un'interfaccia DAO e la sua implementazione

Nel nostro caso:

- RistorantePiattoMappingDAO
- Db2RistorantePiattoMappingDAO

Il verso di percorrenza (la freccia sopra la relazione) indica la navigabilità della relazione, ovvero il verso di "lettura" dei dati. Nel nostro caso da Ristorante verso Piatto:

Ecco perchè solo RistoranteDTO ha una Collezione di PiattoDTO!!

# La relazione M-N e il Verso di Percorrenza: nella pratica

```java
public interface RistorantePiattoMappingDAO {

    // --- CRUD ------------

        public void create(int idr, int idp);

        //public RistorantePiattoMappingDTO read(int idRistorante, int idPiatto);

        //public boolean update(CourseDTO student);

        public boolean delete(int idRistorante, int idPiatto);


        // ---------------------------------

        public List<PiattoDTO> getPiattiFromResturant(int id);


        // ---------------------------------


        public boolean createTable();

        public boolean dropTable();
}
```

```java
public class Db2RistorantePiattoMappingDAO implements
        RistorantePiattoMappingDAO {

    // === Costanti letterali per non sbagliarsi a scrivere !!! ===========
        static final String TABLE = "ristoranti_piatti";
        // ------------------------------------------------------------------
        static final String ID_R = "idRistorante";
        static final String ID_P = "idPiatto";

        // == STATEMENT SQL ==================================================

        // INSERT INTO table ( idCourse, idStudent ) VALUES ( ?,? );
        static final String insert = ..

        // SELECT * FROM table WHERE idcolumns = ?;
        static String read_by_ids = ..

        // SELECT * FROM table WHERE idcolumns = ?;
        static String read_by_ristoranteID = ..

        // SELECT * FROM table WHERE idcolumns = ?;
        static String read_by_piattoID = ..

        // SELECT * FROM table WHERE idcolumns = ?;
        static String dish_query = ..

        // SELECT * FROM table WHERE stringcolumn = ?;
        static String read_all = ..

        // DELETE FROM table WHERE idcolumn = ?;
        static String delete = ..

        // UPDATE table SET xxxcolumn = ?, ... WHERE idcolumn = ?;
        /*static String update = ..

        // SELECT * FROM table;
        static String query = ..

        // CREATE entrytable ( code INT NOT NULL PRIMARY KEY, ... );
        static String create =
            "CREATE " +
                "TABLE " + TABLE +" ( " +
                    ID_R + " INT NOT NULL, " +
                    ID_P + " INT NOT NULL, " +
                    "PRIMARY KEY (" + ID_R +","+ ID_P + " ), " +
                    "FOREIGN KEY ("+ID_R+") REFERENCES ristoranti(id), "+
                    "FOREIGN KEY ("+ID_P+") REFERENCES piatti(id) "+
                ") "
            ;

        static String drop = ..
```

# Il Fetch Lazy-Load

Come sappiamo dalla teoria la tipologia di fetching Lazy-Load prevede Il caricamento dei dati di tipo “pigro”, ovvero questi vengono caricati solo quando sono strettamente necessari, ovvero immediatamente prima che siano processati.

Come si implementa il fetching Lazy-Load in Java, e come si integra nel Pattern DAO?

Serve una classe PROXY, ovvero occorre:

Estendere RistoranteDTO creando una classe proxy Db2RistoranteDTOProxy

Questa classe farà l’override della getPiatti() della superclasse andando a caricare dal db tutti i piatti che serve lo specifico oggetto RistoranteDTO.

La query per il caricamento dei piatti del ristorante sarà contenuta nell’interfaccia RIstorantePiattoMappingDAO

# Il Fetch Lazy-Load: codice (1)

La classe proxy Db2RistoranteDTOProxy

```
public class Db2RistoranteDTOProxy extends RistoranteDTO {

    public Db2RistoranteDTOProxy() {
        super();
        // TODO Auto-generated constructor stub
    }


    @Override
    public List<PiattoDTO> getPiatti()
    {
        if(isAlreadyLoaded())
            return super.getPiatti();
        else
        {
            RistorantePiattoMappingDAO rpm = new Db2RistorantePiattoMappingDAO();
            isAlreadyLoaded(true);
            return rpm.getPiattiFromResturant(this.getId());
        }
    }

}
```

Ad ogni chiamata del metodo getPiatti(), la classe controlla che i dati non siano già stati caricati precedentemente, evitando, in tal caso, di andare un'altra volta sul DB

# Il Fetch Lazy-Load: codice (2)

## Il lazy-load lato DB

```
public interface RistorantePiattoMappingDAO {

    // --- CRUD ------------

        public void create(int idr, int idp);

        //public RistorantePiattoMappingDTO read(int idRistorante, int idPiatto);

        //public boolean update(CourseDTO student);

        public boolean delete(int idRistorante, int idPiatto);


        // --------------------------------

        public List<PiattoDTO> getPiattiFromResturant(int id);

        // --------------------------------


        public boolean createTable();

        public boolean dropTable();
}
```

```
static String dish_query =
    "SELECT * " +
        "FROM " + TABLE + " RP, piatti P " +
        "WHERE RP.idPiatto = P.id AND " + ID_R + " = ? ";
```

```
@Override
public List<PiattoDTO> getPiattiFromResturant(int id) {
    List<PiattoDTO> result = null;
    if ( id< 0 )  {
        System.out.println("read(): cannot read an entry with a negative id");
        return result;
    }
    Connection conn = Db2DAOFactory.createConnection();
    try {
        PreparedStatement prep_stmt = conn.prepareStatement(dish_query);
        prep_stmt.clearParameters();
        prep_stmt.setInt(1, id);
        ResultSet rs = prep_stmt.executeQuery();

        result = new ArrayList<PiattoDTO>();
        while ( rs.next() ) {
            PiattoDTO entry = new PiattoDTO();
            entry.setId((rs.getInt("id")));
            entry.setNomePiatto(rs.getString("nome"));
            entry.setTipo(rs.getString("tipo"));
            result.add(entry);
        }
        rs.close();
        prep_stmt.close();
    }
    catch (Exception e) {

        e.printStackTrace();
    }
    finally {
        Db2DAOFactory.closeConnection(conn);
    }
    return result;
}
```

# Le Query

Una volta progettata bene tutta l'architettura relazionale in Java, le query non risultano di particolare difficoltà:

- Restituire tutti i primi piatti dei ristoranti siti in Bologna
- Contare i ristoranti con voti tra 4 e 5 (compresi) che servono "Seppie e Piselli"

# La Prima Query: Primi Piatti a Bologna (1)

Metodo nella classe di Test: riceve tutti i ristoranti di Bologna e ne prende i primi piatti

```
public static String ListPrimiPiattiDeiRistorantiDiBologna(RistoranteDAO r, RistorantePiattoMappingDAO rpm)
{
    List<RistoranteDTO> ristorantiBolognesi = r.getResturantByCity("Bologna");
    String result="";
    for(RistoranteDTO risto : ristorantiBolognesi)
    {
        boolean trovato=false;
        List<PiattoDTO> primiPiatti = risto.getPiatti();
        for(PiattoDTO p : primiPiatti)
        {
            if(p.getTipo().compareTo("primo")==0)
            {

                result = result+p+"\n";
                trovato=true;
            }
        }
        if(trovato)
        {

            result = result+"Questo/i piatto/i è/sono preparato/i da: "+risto.getNomeRistorante()+"\n";
        }
    }
    return result;
}
```

# La Prima Query: Primi Piatti a Bologna (2)

Lato DB,

classe

Db2RistoranteDAO

```java
@Override
public List<RistoranteDTO> getResturantByCity(String citta) {
    // TODO Auto-generated method stub
    List<RistoranteDTO> result = null;
    // --- 2. Controlli preliminari sui dati in ingresso ---
    if ( citta.isEmpty() || citta == null )  {
        return result;
    }
    // --- 3. Apertura della connessione ---
    Connection conn = Db2DAOFactory.createConnection();
    // --- 4. Tentativo di accesso al db e impostazione del risultato ---
    try {
        // --- a. Crea (se senza parametri) o prepara (se con parametri) lo statement
        PreparedStatement prep_stmt = conn.prepareStatement(query);
        // --- b. Pulisci e imposta i parametri (se ve ne sono)
        // --- c. Esegui l'azione sul database ed estrai il risultato (se atteso)
        ResultSet rs = prep_stmt.executeQuery();
        result = new ArrayList<RistoranteDTO>();
        // --- d. Cicla sul risultato (se presente) per accedere ai valori di ogni sua tupla
        String address;
        while ( rs.next() ) {
            address = rs.getString("indirizzo").toLowerCase();
            if(address.contains("bologna "))
            {
                RistoranteDTO entry = new Db2RistoranteDTOProxy();
                entry.setId(rs.getInt(ID));
                entry.setIndirizzo(address);
                entry.setRating(rs.getInt(RATING));
                entry.setNomeRistorante(rs.getString(NOMERISTORANTE));
                result.add(entry);
            }
        }
        // --- e. Rilascia la struttura dati del risultato
        rs.close();
        // --- f. Rilascia la struttura dati dello statement
        prep_stmt.close();
    }
    // --- 5. Gestione di eventuali eccezioni ---
    catch (Exception e) {
        e.printStackTrace();
    }
    // --- 6. Rilascio, SEMPRE E COMUNQUE, la connessione prima di restituire il controllo al chiamante
    finally {
        Db2DAOFactory.closeConnection(conn);
    }
    // --- 7. Restituzione del risultato (eventualmente di fallimento)
    return result;
}
```

# La Seconda Query: Seppie e Piselli gourmet (1)

Metodo nella classe di Test: riceve tutti i ristoranti votati almeno con 4 stelle, e controlla se servono “Seppie e Piselli”

```
public static String CountRatedResturantsWithSeppieEPiselli(RistoranteDAO r, RistorantePiattoMappingDAO rpm)
{
    List<RistoranteDTO> ristorantiStellati = r.getRatedResturant(4);
    int counter=0;

    for(RistoranteDTO risto : ristorantiStellati)
    {

        List<PiattoDTO> primiPiatti = risto.getPiatti();
        for(PiattoDTO p : primiPiatti)
        {
            if(p.getNomePiatto().compareTo("Seppie e Piselli")==0)
            {
                counter++;
                break;
            }
        }

    }
    return "Sono stati trovati "+counter+" ristoranti con almeno 4 stelle che preparano Seppie e Piselli";

}
```

# La Seconda Query: Seppie e Piselli gourmet (2)

Lato DB,

classe

Db2RistoranteDAO

```
static String find_resturant_over_rate =
        read_all +
            "WHERE " + RATING + " > ? ";
```

```
@Override
public List<RistoranteDTO> getRatedResturant(int stars) {
            List<RistoranteDTO> result = null;
            // --- 2. Controlli preliminari sui dati in ingresso ---
            if ( stars<1 || stars >5 )  {

                return result;
            }
            // --- 3. Apertura della connessione ---
            Connection conn = Db2DAOFactory.createConnection();
            // --- 4. Tentativo di accesso al db e impostazione del risultato ---
            try {
                // --- a. Crea (se senza parametri) o prepara (se con parametri) lo statement
                PreparedStatement prep_stmt = conn.prepareStatement(find_resturant_over_rate);
                // --- b. Pulisci e imposta i parametri (se ve ne sono)
                prep_stmt.clearParameters();
                prep_stmt.setInt(1, stars-1);
                // --- c. Esegui l'azione sul database ed estrai il risultato (se atteso)
                ResultSet rs = prep_stmt.executeQuery();
                result = new ArrayList<RistoranteDTO>();
                // --- d. Cicla sul risultato (se presente) per accedere ai valori di ogni sua tupla
                while ( rs.next() ) {
                        RistoranteDTO entry = new Db2RistoranteDTOProxy();
                        entry.setId(rs.getInt(ID));
                        entry.setIndirizzo(rs.getString(INDIRIZZO));
                        entry.setRating(rs.getInt(RATING));
                        entry.setNomeRistorante(rs.getString(NOMERISTORANTE));
                        result.add(entry);
                }
                // --- e. Rilascia la struttura dati del risultato
                rs.close();
                // --- f. Rilascia la struttura dati dello statement
                prep_stmt.close();
            }
            // --- 5. Gestione di eventuali eccezioni ---
            catch (Exception e) {
                e.printStackTrace();
            }
            // --- 6. Rilascio, SEMPRE E COMUNQUE, la connessione prima di restituire il controllo al chiamante
            finally {
                Db2DAOFactory.closeConnection(conn);
            }
            // --- 7. Restituzione del risultato (eventualmente di fallimento)
            return result;
}
```

# Appendice

## (esercizio guidato tratto dall'appello d'esame del 25 luglio 2016: gestione della persistenza basata su Hibernate)

# Agenda

- Presentazione del Testo d'Esame
- Primo Approccio all'Esercizio
- Passi Principali:
  - Dall'UML al Mapping XML di Hibernate
  - Le Relazioni in Hibernate
  - Hibernate Config
  - Il Codice Java e le Query

# Il Testo d'Esame

**ESERCIZIO 3 (11 punti)**
Partendo dalla realtà illustrata nel **diagramma UML** di seguito riportato, si fornisca una soluzione alla gestione della persistenza basata su **Hibernate** in grado di "mappare" efficientemente e con uso di ID surrogati il modello di dominio rappresentato dai **JavaBean del diagramma UML** con le corrispondenti **tabelle relazionali derivate dalla progettazione logica del diagramma** stesso.


Nel dettaglio, dopo aver creato da applicazione Java gli schemi delle tabelle all'interno del proprio schema nel database **TW_STUD** di DB2 (esplicitando tutti i vincoli derivati dal diagramma UML), implementato i **JavaBean**, definiti i **file XML di mapping** e il **file XML di properties**, si richiede la realizzazione di una classe di prova facente uso delle **API Hibernate** in grado di:

- inserire due o più tuple nelle tabelle di interesse;
- determinare i) elenco dei nomi degli accertamenti di tipo "Analisi di Laboratorio" erogati presso l'ospedale Policlinico S.Orsola-Malpighi di Bologna; ii) per ogni ospedale, nome, città, indirizzo e numero totale di accertamenti erogabili presso la struttura;
- stampare i risultati ottenuti al punto precedente sul file **Ospedale.txt**;

il tutto, mediante opportuna gestione delle **transazioni**.

**N.B.** La soluzione deve **sfruttare i mapping M-N e 1-N specificati nel diagramma UML**. Ogni ulteriore scelta fatta dallo studente deve essere opportunamente giustificata mediante commenti nel codice.

# Primo Approccio all'Esercizio

Dal testo emergono immediatamente i punti salienti che l'esercizio richiede di affrontare:

- Mapping delle **classi** UML in **Hibernate**
  - tramite **file XML**
- Mapping delle **relazioni** in **Hibernate** e verso di percorrenza
- Inserimento di dati e risoluzione di query attraverso **Hibernate**

# Dall'UML ad HIBERNATE: Mapping delle Classi

Come sappiamo, il mapping delle classi in Hibernate richiede la definizione di opportuni file XML

- Questi file indicano al framework come "mappare" le entità del dominio nelle corrispondenti tabelle DB
- Per ogni Java Bean è **necessario** un file XML

# Dall'UML ad HIBERNATE: Mapping delle Classi

Nel dettaglio, per la parte Hibernate:

- Bisogna creare un file XML chiamato:

  **<NomeClasseJava>.hbm.xml**

- Questo file istruirà il framework su come mappare il Java Bean nella tabella corrispondente

- Specificare nome della tabella, nome e tipo degli attributi, chiave primaria, chiavi esterne, vincoli di unicità, ecc.

# Dall'UML ad HIBERNATE: Dal Bean all'XML (1)

Prendiamo come esempio la classe Accertamento…

La parte del Java Bean ormai la consideriamo standard

```
public class Accertamento implements Serializable{

    /**
     *
     */
    private static final long serialVersionUID = 1L;
    private int accId;
    private int codiceAcc;
    private String nome;
    private String descrizione;

    private TipoAccertamento tipoAccertamento;
```

Oltre agli attributi che ci aspettiamo, saltano all'occhio due cose:

- L'implementazione dell'interfaccia Serializable (Utile ad Hibernate)
- L'attributo di tipoAccertamento, che dettagliamo in seguito

# Dall'UML ad HIBERNATE: Dal Bean all'XML (2)

Ora il file **Accertamento.hbm.xml** l(hbm – HiBernate Mapping)

```xml
<?xml version="1.0"?>

<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
  <class name="hibernate.Accertamento" table="accertamenti">
    <id name="accId" column="accId" />
    <property name="codiceAcc" column="codiceAcc"/>
    <property name="nome" column="nome"/>
    <property name="descrizione" column="descrizione"/>

     <many-to-one name="tipoAccertamento" class="hibernate.TipoAccertamento" fetch="select">
            <column name="tipoAccId" not-null="true" />
        </many-to-one>

  </class>
</hibernate-mapping>
```

# Dall'UML ad HIBERNATE: Dal Bean all'XML (3)

Analizziamo il file:

- Hibernate-mapping: Root del file, tutti i file di mapping di Hibernate hanno questo tag come root.
- Class: Specifica il mapping tra la classe Java e la tabella del DB
  - Attributo Name: Nome della classe Java (compresa di package)
  - Attributo Table: Nome della tabella su cui verrà mappata la classe Java
- Id: Indica quale attributo della classe Java farà da chiave primaria della tabella
  - Attributo Name: Nome dell'attributo della classe Java
  - Attributo Column: Nome della colonna della tabella sulla quale verrà mappata la proprietà della classe Java
- Property: Indica il mapping degli attributi comuni
  - Gli attributi sono gli stessi di Id.

Ed i tipi di dato? I vincoli di unicità ? E di not-null? …non è tutto qui…

# Dall'UML ad HIBERNATE: Dal Bean all'XML (4)

- Dall'esempio mostrato emerge che Hibernate sia in grado di derivare i tipi di dato degli attributi delle tabelle da quelli dichiarati nella relativa classe Java.
- I vincoli di unicità e di not null sono degli attribute booleani

Esempio completo:

```
<property name="stockName" type="string">
    <column name="STOCK_NAME" length="20" not-null="true" unique="true" />
</property>
<property name="priceChange" type="java.lang.Float">
    <column name="PRICE_CHANGE" precision="6" />
</property>
<property name="volume" type="java.lang.Long">
    <column name="VOLUME" />
</property>
<property name="date" type="date">
    <column name="DATE" length="10" not-null="true" unique="true" />
</property>
```

Abbiamo tralasciato solo una cosa: **Il mapping delle relazioni**!!

# HIBERNATE: Il mapping delle Relazioni

Nel diagramma UML presente nel testo **non ci sono frecce per i versi di percorrenza**, sappiamo già che vuol dire:

**Relazioni Bidirezionali**

- Ecco perchè la classe Java “Accertamento” ha un attributo di tipo “TipoAccertamento”

In Hibernate, dobbiamo mappare la bidirezionalità della relazione tra “Accertamento” e “TipoAccertamento”

La relazione sarà:

- Many-to-One da “Accertamento” verso “TipoAccertamento”
- One-to-Many da “TipoAccertamento” verso “Accertamento”

**Queste relazioni dovranno essere dichiarate nei rispettivi file di mapping Hibernate delle classi**

- ora vediamo come si fa…

# HIBERNATE: Il mapping delle Relazioni (2)

Nel file Accertamento.hbm.xml:

```
<many-to-one name="tipoAccertamento" class="hibernate.TipoAccertamento" fetch="select">
        <column name="tipoAccId" not-null="true" />
    </many-to-one>
```

- Many-to-one: Tipologia di relazione
  - Attributo Name: il nome della proprietà all'interno della classe Java mappata dal file xml in oggetto
  - Attributo Class: Nome della classe Java riferita (esterna)
- Elemento Column: Colonna di riferimento della taballa esterna
  - Attributo Name: Nome della Colonna della primary key della tabella esterna. **N.B. Ovviamente I nomi devono essere uguali!!**

Vedremo un esempio completo...

# HIBERNATE: Il mapping delle Relazioni (3)

Nella classe Java “TipoAccertamento”, essendo una relazione One-to-Many, ci sarà una collezione di oggetti “Accertamento”

Vediamo come una relazione One-To-Many si mappa in Hibernate, nel file TipoAccertamento.hbm.xml:

```
<set name="accertamenti" table="accertamenti"
            inverse="true" lazy="true" fetch="select">
        <key>
            <column name="tipoAccId" not-null="false" />
        </key>
        <one-to-many class="hibernate.Accertamento" />
    </set>
```

- Set: Indica che si va a riferire una molteplicità di oggetti
  - Attributo Name: Nome della proprietà della Collezione nella classe Java
  - Attributo table: Nome della tabella esterna
- Elemento Key: Indica la chiave esterna
  - Attributo Name di Column: Nome della Colonna di chiave della tabella stessa
- One-to-many: Tipologia della Relazione
  - Attributo Class: Classe Java esterna

# HIBERNATE: Il mapping delle Relazioni (4)

Esempio completo:

Accertamento.java

Accertamento.hbm.xml

TipoAccertamento.java

TipoAccertamento.hbm.xml

```
public class Accertamento implements Serializable{

    /**
     *
     */
    private static final long serialVersionUID = 1L;
    private int accId;
    private int codiceAcc;
    private String nome;
    private String descrizione;

    private TipoAccertamento tipoAccertamento;
```

```
private static final long serialVersionUID = 1L;
private int tipoAccId;
private int codiceTipoAcc;
private String  descrizione;

private Set<Ospedale> ospedali = new HashSet<Ospedale>(0);
private Set<Accertamento> accertamenti = new HashSet<Accertamento>(0);
```

```
<many-to-one name="tipoAccertamento" class="hibernate.TipoAccertamento" fetch="select">
    <column name="tipoAccId" not-null="true" />
</many-to-one>
```

```
<set name="accertamenti" table="accertamenti"
            inverse="true" lazy="true" fetch="select">
        <key>
            <column name="tipoAccId" not-null="false" />
        </key>
        <one-to-many class="hibernate.Accertamento" />
    </set>
```

…Abbiamo tralasciato le relazioni Many-to-Many…

# HIBERNATE: Relazione Many-To-Many

"TipoAccertamento" ha anche una relazione Many-to-Many con "Ospedale"

Come abbiamo visto, nella classe Java "TipoAccertamento" c'è anche una collezione di oggetti "Ospedale"

Vediamo come questa collezione e questo tipo di relazione si mappa in Hibernate, sempre il file TipoAccertamento.hbm.xml:

```
<set name="ospedali" table="tipoAccertamento_ospedale" inverse="false" lazy="true"
    fetch="select" cascade="all">
        <key column="tipoAccId" />
        <many-to-many column="ospId" class="hibernate.Ospedale"/>
</set>
```

Il significato degli elementi e degli attributi sono gli stessi della relazione che abbiamo già visto

**Attenzione:** L'attributo **Table** di **Set** riferisce la **"tabella di mapping"** (ovvero la tabella DB derivata dal passo di progettazione logica applicato alla associazione N-M del diagramma UML di partenza)

Ora vediamo la stessa relazione dalla parte di "Ospedale"…

# HIBERNATE: Relazione Many-To-Many (2)

La classe Java “Ospedale”:

```
private int ospId;
private int codice;
private String nome;
private String citta;
private String indirizzo;

private Set<TipoAccertamento> tipiAccertamento = new HashSet<TipoAccertamento>(0);
```

Mentre in Ospedale.hbm.xml:

```
<set name="tipiAccertamento" table="tipoAccertamento_ospedale" inverse="true">
        <key column="ospId" />
        <many-to-many column="tipoAccId" class="hibernate.TipoAccertamento"/>
</set>
```

Ovviamente è il duale di “TipoAccertamento”

Vediamo ora l’esempio complete…

# HIBERNATE: Relazione Many-To-Many (4)

Esempio completo:

TipoAccerttamento.java

TipoAccertamento.hbm.xml

Ospedale.java

Ospedale.hbm.xml

```java
private static final long serialVersionUID = 1L;
private int tipoAccId;
private int codiceTipoAcc;
private String  descrizione;

private Set<Ospedale> ospedali = new HashSet<Ospedale>(0);
private Set<Accertamento> accertamenti = new HashSet<Accertamento>(0);
```

```java
private int ospId;
private int codice;
private String nome;
private String citta;
private String indirizzo;

private Set<TipoAccertamento> tipiAccertamento = new HashSet<TipoAccertamento>(0);
```

```xml
<set name="ospedali" table="tipoAccertamento_ospedale" inverse="false" lazy="true"
    fetch="select" cascade="all">
        <key column="tipoAccId" />
        <many-to-many column="ospId" class="hibernate.Ospedale"/>
</set>
```

```xml
<set name="tipiAccertamento" table="tipoAccertamento_ospedale" inverse="true">
        <key column="ospId" />
        <many-to-many column="tipoAccId" class="hibernate.TipoAccertamento"/>
</set>
```

# Configurare Hibernate: Hibernate Config

…I file xml da scrivere non sono finiti…. L'ultimo file rimanente serve per configurare il framework di Hibernate

```xml
<?xml version="1.0" encoding="utf-8"?>
<!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.connection.username">User</property>
    <property name="hibernate.connection.password">password</property>
  <!--  <property name="connection.pool_size">"1"</property> -->
    <property name="hibernate.dialect">org.hibernate.dialect.DB2Dialect</property>
<!--    <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/tw_stud</property>
    <property name="hibernate.connection.username">root</property>
    <property name="hibernate.connection.password"></property>
    <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>-->
    <property name="show_sql">true</property>
    <property name="format_sql">true</property>
    <mapping resource="hibernate/Accertamento.hbm.xml"/>
    <mapping resource="hibernate/Ospedale.hbm.xml"/>
    <mapping resource="hibernate/TipoAccertamento.hbm.xml"/>
</session-factory>
</hibernate-configuration>
```

Questo file contiene tutti i parametri di configurazione di Hibernate, tra i quali il "dialetto" di SQL che si vuole usare, credenziali d'accesso al DB fisico, e dove si trovano i file di mapping

# Il Codice Java e le Query

Una volta impostato e configurato bene il Hibernate framework ed il mapping, le query richieste da questo esercizio non sono di particolare difficoltà:

- L'elenco dei nomi degli accertamenti di tipo "analisi di laboratorio" effettuate al Sant'Orsola di Bologna

- Per ogni Ospedale, restituire: il nome, la città, l'indirizzo, e il numero totale di "Accertamenti" erogabili

Nelle due query che vedremo nelle prossime due slide, è stato utilizzato un approccio più Java-oriented, mantenendo le query a livello DB molto semplice…

ma una soluzione DB-oriented puoò risultare più efficiente ☺

# Prima Query: Le Analisi del Sant'Orsola

Codice soluzione della prima query

```
Query firstQuery = session.createQuery("from "+Ospedale.class.getSimpleName()+
        " where citta = ? and nome = ?");
firstQuery.setString(0, "Bologna");
firstQuery.setString(1, "S.Orsola");
List<Ospedale> ospedaliRes = firstQuery.list();
String firstQueryResult ="";
String secondQueryResult ="";
for(Ospedale os : ospedaliRes)
{
    Set<TipoAccertamento> tipiAcc = os.getTipiAccertamento();
    for(TipoAccertamento tipo : tipiAcc)
    {
        if(tipo.getDescrizione().compareTo("analisi di laboratorio")==0)
        {
            Set<Accertamento> accertamenti = tipo.getAccertamenti();
            for(Accertamento acc :accertamenti)
            {
                firstQueryResult=firstQueryResult+acc.getNome()+"\n";
            }
        }
    }
}
```

# Seconda Query: Gli Ospedali ed Accertamenti

Codice soluzione della seconda query

```
Query secondQuery = session.createQuery("from "+Ospedale.class.getSimpleName());
ospedaliRes= secondQuery.list();
for(Ospedale os : ospedaliRes)
{
    secondQueryResult = secondQueryResult+os.getNome()+ " "
            +os.getIndirizzo() +" "+os.getCitta()+"\n Numero Accertamenti erogabili: ";
    Set<TipoAccertamento> tipiAcc = os.getTipiAccertamento();
    int numAccCounter=0;
    for(TipoAccertamento tipo : tipiAcc)
    {
        numAccCounter += tipo.getAccertamenti().size();
    }
    secondQueryResult=secondQueryResult+numAccCounter +"\n";
}
```

# Ulteriori Utili Riferimenti

Approfondimento sugli attributi delle relazioni:

https://www.mkyong.com/tutorials/hibernate-tutorials/

Approfondimenti ed esempi completi di mapping e relazioni:

https://www.mkyong.com/hibernate/hibernate-one-to-many-relationship-example/

https://www.mkyong.com/hibernate/hibernate-many-to-many-relationship-example/