Web Applications

Persistenza - JDBC – Pattern DAO

Giovanni Grasso e Kristian Reale

Gestione della persistenza

- Le nostre *applicazioni sono sviluppate con linguaggi OO* mentre *la persistenza dei dati è affidata ai DBMS relazionali*
  - OO: scalabilità e riusabilità
  - RDBMS: affidabilità, efficienza, efficacia
- Esistono differenze significative tra queste due tecnologie:
  - differenze tecnologiche
  - differenze culturali
- Si parla di *conflitto di impedenza*

Conflitto di impedenza (1/4)

- Definito anche come *“disaccoppiamento di impedenza”* (o *impedance mismatch*) *tra base di dati e linguaggio*
  - linguaggi: operazioni su *singole variabili o oggetti*
  - SQL: operazioni su *relazioni* (insiemi di ennuple)
- Differenze di accesso ai dati e correlazione:
  - linguaggio: dipende dal paradigma e dai tipi disponibili; ad esempio *scansione di liste o “navigazione” tra oggetti*
  - SQL: *join* (ottimizzabile)

Conflitto di impedenza (2/4)

- Differenze sui tipi di dato primitivi:
  - linguaggi: numeri, stringhe, booleani
  - SQL: CHAR, VARCHAR, DATE, ...
- Differenze sui costruttori di tipo:
  - linguaggio: dipende dal paradigma
  - SQL: *relazioni* e *ennuple*

- OID (trasparenti al programmatore) *vs.* chiavi primarie (visibili e manipolabili)
- Riferimenti *vs.* chiavi esterne
- Ereditarietà (senza controparte nel mondo relazionale)

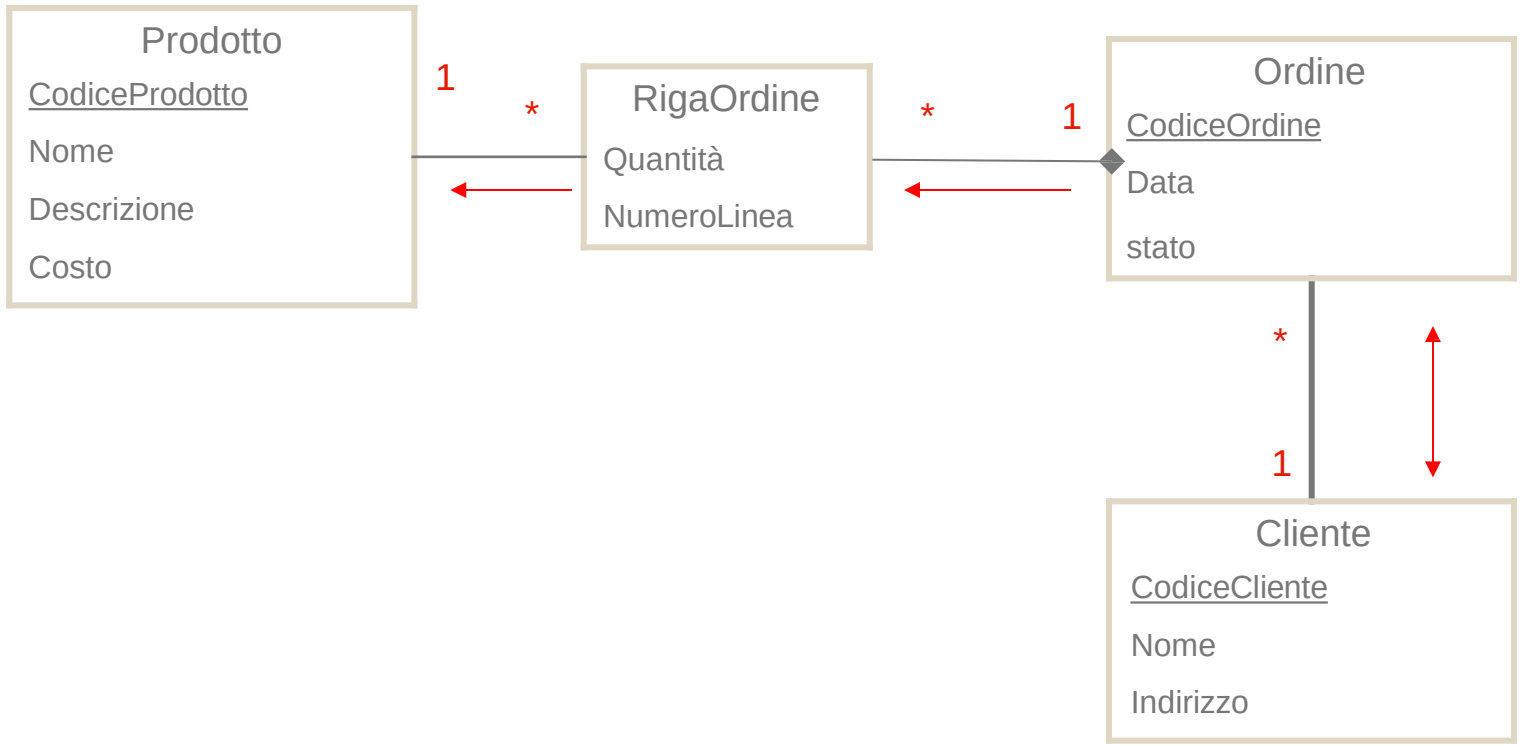
Conflitto di impedenza (3/4)

- *In generale, i dati risiedono nel database (DB) mentre la logica applicativa viene implementata da oggetti*
- Dal modello concettuale si può derivare (una prima versione di)
  - classi che implementano la *logica applicativa* (diciamo che queste classi implementano il *modello di dominio*, o più semplicemente il *modello*)
  - schema relazionale *per memorizzare i dati*
    - questa attività può essere fatta anche successivamente
    - in alcuni casi (sistemi *legacy*) il database potrebbe esistere già e potrebbe non essere modificabile
      - quindi tutte le attività di progetto sono condizionate da questo vincolo

Metodologie per la gestione della persistenza

- In generale la nostra applicazione avrà metodi che
  - caricano tuple dalla base di dati e le usano per creare gli oggetti del modello
  - effettuano le operazioni della logica applicativa (es. evadi ordine)
  - salvano in maniera persistente gli oggetti del modello nella base di dati
- *L'interazione con il DB è una operazione critica*: esistono diverse metodologie (di *complessità crescente*) per realizzare questa interazione
  - forza bruta
  - pattern DAO
  - framework ORM (..al corso di Enterprise Applications)

Esempio (in UML)



Forza bruta

- È la tecnica più semplice per gestire la persistenza
  - forte accoppiamento con la sorgente dati
- Consiste nello scrivere dentro le classi del modello un insieme di metodi che implementano le operazioni CRUD
- *Operazioni CRUD*
  - **C**reate: inserimento di una tupla (che rappresenta un oggetto) nel database (INSERT)
  - **R**etrieve: ricerca di una tupla secondo un qualche criterio di ricerca (SELECT)
  - **U**pdate: aggiornamento di una tupla nel database (UPDATE)
  - **D**elete: eliminazione di una tupla nel database (DELETE)

Il pattern Data Access Object

- La soluzione *Forza bruta non è particolarmente conveniente*
  - L'accoppiamento tra la sorgente dati e le classi del modello è molto forte
  - Molto difficile iniziare a sviluppare senza preoccuparsi di *tanti* (*troppi* !) dettagli e senza un ambiente operativo complesso (java + dbms)
- Una soluzione "*naturale*" è quella di *affidare le responsabilità di interagire con il database ad opportune classi*
- Questo è l'approccio suggerito dal *pattern* Data Access Object (DAO)

DAO: Caratteristiche principali (1/2)

- I *valori scambiati tra DB e il resto dell'applicazione sono racchiusi in oggetti detti Data Transfer Object* (DTO):
  - campi privati per contenere i dati da leggere/scrivere su db
  - metodi getter e setter per accedere dall'esterno a tali campi
  - metodi di utilità (confronto, stampa, …)
- Le *operazioni che coinvolgono tali oggetti sono raggruppati in interfacce che definiscono i Data Access Object* (DAO) *disponibili*:
  - metodi CRUD
  - altri metodi

Esempio: ProdottoDAO

```
public interface ProdottoDAO {

        public List<Prodotto> doRetrieveAll()
                                throws PersistenceException;

        public Prodotto doRetriveByKey(String codice)
                                throws PersistenceException;

        public void saveOrUpdate(Prodotto prodotto)
                                throws PersistenceException;

        public void delete(Prodotto prodotto)
                                throws PersistenceException;

        // eventuali altri doRetriveBy...
 }
```

La classe Prodotto e' il DTO

Esempio: ProdottoDAOImpl

```
public class ProdottoDAOImpl implements ProdottoDAO {

        public List<Prodotto> doRetrieveAll()
                throws PersistenceException {
                List<Prodotto> prodotti;

                …
                try {
                        …//uses JDBC to connect to DB
                } catch (SQLException sqle) {
                        throw new PersistenceException(sqle);
                } finally {
                        …
                }
                return prodotti;
        }

        …
}
```

JDBC - Concetti Fondamentali

- Riferimenti
  - http://java.sun.com/javase/6/docs/technotes/guides/jdbc/

- Un Esempio
  - Driver e Driver Manager
  - Connection
  - PreparedStatement
  - ResultSet

Esempio

- Consideriamo la classe **`Studente`**:

```
import java.util.Date;
public class Studente {
   private String matricola;
   private String nome;
   private String cognome;
   private Date dataNascita;

   public Studente(){}
   public String getNome() {
return this.nome;
  }
   public void setNome(String nome) {
this.nome = nome;
  }
   // seguono tutti gli altri metodi getter e setter
}
```

Esempio

e il database *university:*

```
CREATE DATABASE university;

CREATE TABLE studente
(
  matricola character(8);
  nome character varying(64) NOT NULL,
  cognome character varying(64) NOT NULL,
  datanascita date NOT NULL,
  CONSTRAINT pk_studente PRIMARY KEY (matricola)
)
```

```
public List<Studente> findAll() {
    Connection connection = this.dataSource.getConnection();
    List<Studente> studenti = new
    LinkedList<>(); try {
        Studente studente;
        PreparedStatement statement;
        String query = "select * from studente";
        statement = connection.prepareStatement(query);
        ResultSet result = statement.executeQuery();
        while (result.next()) {
            studente = new Studente();
            studente.setMatricola(result.getString("matricola"));
            studente.setNome(result.getString("nome"));
            studente.setCognome(result.getString("cognome"));
            long secs = result.getDate("datanascita").getTime();
            studente.setDataNascita(new java.util.Date(secs));
            studenti.add(studente);
        }
    } catch (SQLException e) {
        throw new PersistenceException(e.getMessage());
    }    finally {
        try {
            connection.close();
        } catch (SQLException e) {
            throw new PersistenceException(e.getMessage());
        }
    }
    return studenti;
}
```

```
public DataSource dataSource(){
    DriverManagerDataSource ds = new DriverManagerDataSource();
    ds.setDriverClassName("com.mysql.jdbc.Driver");
    ds.setUrl("jdbc:mysql://localhost:3306/gene");
    ds.setUsername("root");
    ds.setPassword("root");
    return ds;
}
```

Operazione n.3
Istruzione SQL

- Aggiornamenti (insert, delete, update)
  - Si invoca il metodo **`executeUpdate()`** sull'oggetto **`PrepareStatement`**
- Esempio 1 (cont.)

```
PreparedStatement statement;
String insert_stm = "insert into studente(nome, cognome,
dataNascita, matricola) values (?,?,?,?)";
statement = connection.prepareStatement(insert_stm );
statement.setString(1, studente.getNome());
statement.setString(2, studente.getCognome());
long secs = studente.getDataNascita().getTime());
statement.setDate(3, new java.sql.Date(secs));
statement.setString(4, studente.getMatricola());
```

```
statement.executeUpdate();
```

Aspetti Metodologici
Risolvere il conflitto di impedenza

- Gestione delle associazioni
  - Problemi
  - Lazy Load
- Gestione dei riferimenti
  - Chiavi naturali e chiavi surrogate
  - Generazione chiavi surrogate
- Operazioni in cascata
  - Insert, Update, Delete

Risolvere il conflitto di impedenza

- Nel modello OO, le relazioni tra oggetti sono realizzate con riferimenti, mentre nel modello relazionale sono realizzate con i valori
- Vediamo problemi e soluzioni relativamente a:
  - Retrieve: dal DB agli oggetti (ricostruzione degli oggetti a partire da dati persistenti)
- Create: dagli oggetti al DB
  - (memorizzazione persistente degli oggetti nel DB)

Esempio di riferimento

gruppo		
id	bigint	nullable=false PK
nome	varchar(255)	nullable=false

studente		
matricola	character(8)	nullable=false PK
nome	varchar(255)	nullable=false
cognme	varchar(255)	nullable=false
data_nascita	date	nullable=true
gruppo_id	bigint	FK gruppo.id

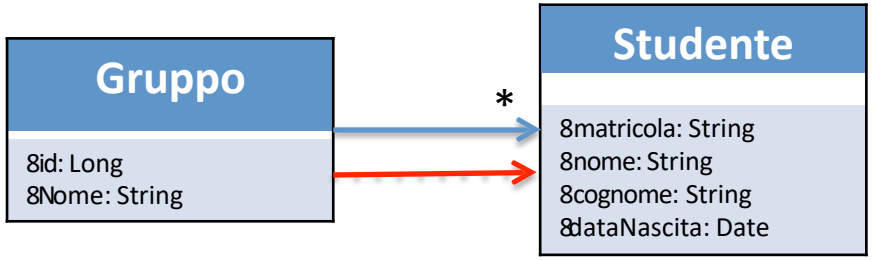
gruppo

ID	NOME
1	Le Mele
2	Microgeeks
…	…

studente

matricola	nome	cognome	data_nascita	gruppo_id
00000001	Roberto	Bianchi	20/05(1997	1
00000002	Gilia	Rossi	12/02(1997	1
…	…	…	…	

Esempio di riferimento



gruppo		
id	bigint	nullable=false PK
nome	varchar(255)	nullable=false

studente		
matricola	character(8)	nullable=false PK
nome	varchar(255)	nullable=false
cognme	varchar(255)	nullable=false
data_nascita	date	nullable=true
gruppo_id	bigint	FK gruppo.id

Esempio di riferimento

```
public class Gruppo {
    private Long id;
    private String nome;
    private Set<Studente> studenti;

    public Gruppo(){
        this.studenti = new HashSet<>();
    }

    // getter e setter

    public void addStudente(Studente studente){
        this.getStudenti().add(studente);
    }

    public void removeStudente(Studente studente){
        this.getStudenti().remove(studente);
    }
}
```

Esempio di riferimento

```
public class Studente {
    private String matricola;
    private String nome;
    private String cognome;
    private Date dataNascita;

    public Studente(){}

    // getter e setter
    // hashCode e equals

}
```

Ricostruzione degli oggetti dal db

- Consideriamo la ricostruzione di un oggetto **`Gruppo`**, ad esempio per definire il metodo **`Gruppo findByPrimaryKey(Long id)`**
  - nella classe **`Gruppo`**:
    - una collezione di riferimenti a oggetti **`Studente`**;
  - nel database:
    - tabella **`gruppo`**
    - tabella **`studente`** (foreign key con **`gruppo`**)
- Per ricostruire un oggetto **`Gruppo`** "completo" devo ricostruire la collezione di oggetti **`Studente`** (e anche questi oggetti devono essere "completi"?!)

Ricostruzione degli oggetti dal db

- Come ricostruiamo l'oggetto **`Gruppo`** "completo"?
- Se voglio ricostruire la collezione di riferimenti agli studenti
  - eseguo un join tra le tabelle **`gruppo`** e **`studente`**
  - itero sul **`ResultSet`** e costruisco i vari oggetti **`Gruppo`** e **`Studente`** (aggiungendone i riferimenti alla collezione)

Esempio con Join

```
public Gruppo findByPrimaryKeyJoin(Long id) {
    Connection connection = this.dataSource.getConnection();
    Gruppo gruppo = null;
    try {
        PreparedStatement statement;
        String query = "SELECT g.id AS g_id, g.nome AS g_nome, "
                     + "s.matricola AS matricola, s.nome AS nome, "
                     + "s.cognome AS cognome, "
                     + "s.datanascita AS datanascita "
                     + "FROM gruppo g LEFT OUTER JOIN "
                     + "studente s ON g.id=s.gruppo_id "
                     + "WHERE g.id = ?";
            statement = connection.prepareStatement(query);
            statement.setLong(1, id);
            logger.debug(statement);
            ResultSet result = statement.executeQuery();
```

Esempio con Join (cont.)

```
    boolean primaRiga = true;
    while (result.next()) {
        if (primaRiga) {
            gruppo = new Gruppo();
            gruppo.setId(result.getLong("g_id"));
            gruppo.setNome(result.getString("g_nome"));
            primaRiga = false;
        }
        if(result.getString("matricola")!=null){
            Studente studente = new Studente();
            studente.setMatricola(result.getString("matricola"));
            studente.setNome(result.getString("nome"));
            studente.setCognome(result.getString("cognome"));
            long secs = result.getDate("datanascita").getTime();
            studente.setDataNascita(new java.util.Date(secs));
            gruppo.addStudente(studente);
        }
    }
} catch (SQLException e) { ///…
```

Ricostruzione degli oggetti dal db

- Ma che cosa è un "oggetto completo"?
  - con un modello di dominio complesso questa strategia può portarci a caricare una rete molto vasta di oggetti
    (esempio: Professore←Corso←**Gruppo**→Studente→…)
- Fino a quando devo costruire oggetti "inseguendo i riferimenti"?
  - e quindi calcolando costosi join?

Problemi soluzione Join

- La soluzione con il join ha un (potenzialmente grosso) problema di prestazioni
- Non è scontato che le operazioni che devo fare sui dati in memoria richiedano i dati degli oggetti collegati (ad esempio mi servono solo i nomi dei gruppi)
- Per ovviare a questo problema, facciamo in modo che gli oggetti della collezione siano caricati solo se necessario, cioè solo se viene invocato un metodo che richiede l'accesso alla collezioni
  - nel nostro esempio il metodo getStudenti()

Lazy Load (caricamento pigro)

- Consideriamo la classe **`Gruppo`**
- Se non è ragionevole pensare che per caricare i dati di un gruppo si debba ricostruire l'intera rete di oggetti collegati dai riferimenti nelle collezioni, possiamo pensare che la collezione di studenti associati al gruppo possa essere caricata solo quando richiesta:
  - quando invochiamo il metodo **`getStudenti()`** su un oggetto **`Gruppo`**
- Questa strategia si chiama *Lazy Load* (caricamento pigro)

Lazy Load (caricamento pigro)

- In pratica quello che vogliamo fare è caricare i dati solo quando questi sono effettivamente richiesti
- Vediamo come dovrebbe agire il metodo **`getStudenti()`** della classe **`Gruppo`** seguendo questa strategia
  - quando invocato il metodo **`getStudenti()`** esegue la query per costruire gli oggetti **`Studente`** associati al gruppo, aggiunge i riferimenti a questi oggetti nella collezione dell'oggetto **`Gruppo`**, e restituisce la collezione
- E' importante osservare che se il metodo non viene mai invocato, il costo di caricare i dati relativi agli ordini non viene mai pagato
- Non possiamo però implementare questa strategia nella classe **`Gruppo`**: metteremmo codice di gestione della persistenza in una classe del dominio
- Per implementare bene la strategia Lazy Load senza ridurre la coesione nella classe del modello è utile fare riferimento al pattern Proxy

Lazy Load e classi *proxy*

- I DAO non restituiscolo un oggetto “completo” ma un *proxy*
  - Il *proxy* ha il compito di gestire il caricamento degli oggetti collegati
  - Il *proxy* deve contenere tutte le informazioni necessarie per effettuare il caricamento “lazy”
  - Gli oggetti collegati vengono caricati solo su richiesta
  - Il *proxy* nasconde l'accesso al database

Lazy Load

```
public class GruppoDaoJDBC {
// …
public Gruppo findByPrimaryKey(Long id) {
        Connection connection = this.dataSource.getConnection();
        Gruppo gruppo = null;
        try {
            PreparedStatement statement;
            String query = "select * from gruppo where id = ?";
            statement  =  connection.prepareStatement(query);
            statement.setLong(1, id);
            ResultSet result = statement.executeQuery();
            if (result.next()) {
                gruppo = new GruppoProxy();
                gruppo.setId(result.getLong("id"));
                gruppo.setNome(result.getString("nome"));
            }
        } catch (SQLException e) {…          }
        return gruppo;
    }
}
```

Lazy Load: implementazione Proxy

```
public class GruppoProxy extends Gruppo
     { private DataSource dataSource;

     public GruppoProxy() {
          this.dataSource = new DataSource();
     }

     public Set<Studente> getStudenti() {
          Set<Studente> studenti = new
          HashSet<>();
          Connection connection = this.dataSource.getConnection();
          try {
               PreparedStatement statement;
               String query = "select * from studente where gruppo_id = ?";
               statement =
               connection.prepareStatement(query);
               statement.setLong(1, this.getId());
               ResultSet result =
               statement.executeQuery(); while
               (result.next()) {
                    Studente studente = new Studente();
                    studente.setMatricola(result.getString("matricola"))
                    ;  studente.setNome(result.getString("nome"));
                    studente.setCognome(result.getString("cognome"));
                    long secs = result.getDate("datanascita").getTime();
                    studente.setDataNascita(new java.util.Date(secs));
                    studenti.add(studente);
               }
          } catch (SQLException e) { …
          }
          this.setStudenti(studenti);
          return super.getStudenti();
     }
```

Rendere persistenti gli oggetti

- L'operazione CREATE (save/insert/persist) ha lo scopo di rendere persistenti gli oggetti della applicazione
- Problemi:
  - persistenza dei riferimenti
  - propagazione degli aggiornamenti

Persistenza dei riferimenti

- Come rendiamo persistenti i riferimenti?
  - se il dominio offre identificatori naturali per tutte le entità, il problema è limitato (ma potremmo fare considerazioni su prestazioni e portabilità)
  - se non tutte le classi hanno identificatori naturali ("semplici") è necessario introdurre chiavi surrogate

Persistenza dei riferimenti

- Chiavi surrogate
  - la chiave può essere vista come una rappresentazione persistente dell'OID
  - in parte cerchiamo di avvicinare i due mondi (relazionale ed oggetti) senza stravolgerli, armonizzando le differenze
- Questa soluzione offre vantaggi in termini di
  - prestazioni (in un db relazionale, usando come chiavi degli interi le operazioni di accesso sono più efficienti)
  - portabilità ed evoluzione (gli identificatori naturali possono cambiare nel tempo*)

*Esempio: è allo studio una revisione del codice fiscale.

Persistenza dei riferimenti

- Chi ci dà il valore di una chiave surrogata?
  - nb: la chiave viene creata al momento dell'inserimento di un oggetto nel db

- Diverse soluzioni:
  - uso campi auto-incrementanti
  - uso una sequenza SQL che interrogo per farmi dare un nuovo id **prima** di ogni inserimento.

Persistenza dei riferimenti

- **campi auto-incrementanti**
  - dobbiamo aggiornare il valore della variabile id nell'oggetto, quindi dopo l'inserimento dobbiamo interrogare il db per riottenerle: funziona, ma è macchinoso (vedi librerie Jdbc per ottenere il valore del campo autoincrementante)
- **sequenza SQL che interrogo per farmi dare un nuovo id prima di ogni inserimento.**
  - efficiente (il dbms offre anche funzionalità di caching)
  - semplice da implementare
  - qualche limite di portabilità (nonostante le sequenze siano standard SQL, i vari DBMS differiscono un po' nell'uso di questo costrutto) facilmente superabile scrivendo le istruzioni SQL in un file di configurazione

Esempio (con chiave surrogata)

- Consideriamo la classe **`Gruppo`**:

```
public class Gruppo {
   private Long id;
   private String nome;
   private Set<Studente> studenti;

   public Gruppo(){
     this.studenti = new HashSet<>();
    }

   public Long getId() {
     return this.id;
   }
   public void setId(Long id) {
     this.id = id;
   }
   // seguono tutti gli altri metodi getter e setter
}
```

Persistenza dei riferimenti: esempio

```
public class GruppoDaoJDBC {
  …
  private void save(Gruppo gruppo) {
    …
    try {
      PreparedStatement statement = null;
      long id = IdBroker.getId(connection);
      gruppo.setId(id);
      String insert = "insert into gruppo (id, nome) values (?,?)";
      statement = connection.prepareStatement(insert);
      statement.setLong(1, gruppo.getId());
      …
      statement.executeUpdate();
    ….
    }  catch (SQLException e) {
  …
}
```

Questo oggetto non ha un id

Genero un nuovo id persistente

IdBroker: esempio

```
public class IdBroker {
    // Standard SQL (queste stringhe andrebbero scritte in un file di configurazione
    // private static final String query = "SELECT NEXT VALUE FOR SEQUENZA_ID AS id";
    private static final String query = "SELECT nextval('sequenza_id') AS id";   // postgresql

    public static Long getId(Connection connection) { Long id = null;
        try {
            PreparedStatement statement = connection.prepareStatement(query);
            logger.debug(statement);
            ResultSet result = statement.executeQuery(); result.next();
            id = result.getLong("id");
        } catch (SQLException e) {
            throw new PersistenceException(e.getMessage());
        }
        return id;
    }
}
```

Nel Database

- Statement SQL Per la creazione della sequenza
  (in Postgresql e Oracle):

```
CREATE SEQUENCE sequenza_id
START WITH 1 INCREMENT BY 1
MINVALUE 0
MAXVALUE 9999999
NO CYCLE;
```

Propagazione degli aggiornamenti

- Quando effettuiamo una operazione di insert (ma analogamente update o delete) propaghiamo *"in cascata"* gli aggiornamenti a tutta la rete di oggetti?
- Ad esempio consideriamo la classe Gruppo:
  - quando rendiamo persistenti i dati di un gruppo, propaghiamo l'operazione a tutti gli oggetti Studente della collezione?
- Anche in questo caso dipende dai casi d'uso

Propagazione degli aggiornamenti

- Attenzione alle prestazioni!
- Supponiamo di propagare l'operazione save() di un gruppo a tutti gli studenti della collezione:
  - dobbiamo fare save di tutti gli oggetti della collezione
  - ma alcuni potrebbero già essere presenti nel db
  - quindi prima di ogni save() dobbiamo controllare se l'oggetto è già stato salvato nel database
- Vediamo il codice

```
public void save(Gruppo gruppo) {
    Connection connection = this.dataSource.getConnection(); try {
        Long id = IdBroker.getId(connection);
        gruppo.setId(id);
        String insert = "insert into gruppo(id, nome) values (?,?)";
        PreparedStatement statement =   connection.prepareStatement(insert);
        statement.setLong(1, gruppo.getId());
        statement.setString(2, gruppo.getNome());

        statement.executeUpdate();
        // salviamo anche tutti gli studenti del gruppo in CASCATA
        this.updateStudenti(gruppo, connection);

    } catch (SQLException e) {
```

```
private void updateStudenti(Gruppo gruppo, Connection connection)
                                                    throws SQLException {
    StudenteDao studenteDao = new StudenteDaoJDBC();
    for (Studente studente : gruppo.getStudenti()) {
        if (studenteDao.findByPrimaryKey(studente.getMatricola()) == null){
            studenteDao.save(studente);
        }
        String update = "update studente SET gruppo_id = ? WHERE matricola = ?";
        PreparedStatement statement = connection.prepareStatement(update);
        statement.setLong (1, gruppo.getId());
        statement.setString (2, studente.getMatricola());
        statement.executeUpdate();
    }
}
```