ORM: Approccio DAO

Versione preliminare

Antonella Poggi

Dipartimento di informatica e Sistemistica

Sapienza Università di Roma

Progetto di Applicazioni Software

Anno accademico 2008-2009

Questi lucidi sono stati prodotti sulla base del materiale preparato per il corso di Progetto di Basi di Dati da D. Lembo e A. Calì.

Data Access Objects

- **Data Access Objects/Persistence classes** gestiscono l'accesso ai dati, incapsulando le modalità di comunicazione con il DB
- Le Data Access Objects si fanno carico di gestire il codice SQL, mentre tutto ciò è trasparente rispetto alle corrispondenti classi di dominio e di controllo
- A livello di logica dell'applicazione **siamo fortemente orientati agli oggetti**: ragioniamo solo in termini di Domain Objects, cioè dei concetti pertinenti al dominio dell'applicazione, e non possiamo mai utilizzare i metodi di accesso diretto alla base dati forniti dai DAO

ORM DAO – pattern

- Si crea una classe DAO per ogni classe che rappresenta entità del dominio di applicazione
- Questa classe DAO conterrà i metodi di interrogazione e manipolazione della corrispondente classe di dominio
- In particolare conterrà le funzionalità di base: CRUD
  - **C**reate
  - **R**ead
  - **U**pdate
  - **D**elete

ORM DAO: Definire le classi di dominio “persistenti”

- la classe di domino `C` espone i metodi `set` e `get` per operare sui campi (privati) che rappresentano le proprietà dell’entità di dominio
- se necessario, ridefinisce gli operatori di confronto e di generazione del codice di hashing (metodi `equals` e `hashCode`) (ad esempio in presenza di identificatori di classe significativi).

ORM DAO: Definire le classi di dominio "persistenti" (cont.)

Ciascuna classe di dominio espone inoltre, in genere, almeno i metodi

- `leggiDatidaDB`, che invoca il caricamento dal DB
- `inserisciDatisuDB`, che invoca l'inserimento sul DB
- `aggiornaDatisuDB`, che invoca l'aggiornamento sul DB
- `cancellaDatidaDB`, che invoca la cancellazione dal DB

Può esporre anche più metodi di uno stesso tipo (ad es., due versioni del metodo `leggiDatidaDB`) per consentire la realizzazione di politiche di lazy read.

ORM DAO – Data Control Services

- Nel caso di interrogazioni (o altre operazioni) che coinvolgano più di una classe di dominio, si definiscono nuove classi di accesso ai dati in cui queste operazioni siano raggruppate in maniera omogenea
  → Chiameremo queste classi **Data Control Services (DCS)**
- Le classi DCS sono sempre parte delle **persistence classes**
- Questo approccio generalizza l'approccio "classico" DAO che non prevede l'uso di Data Control Services

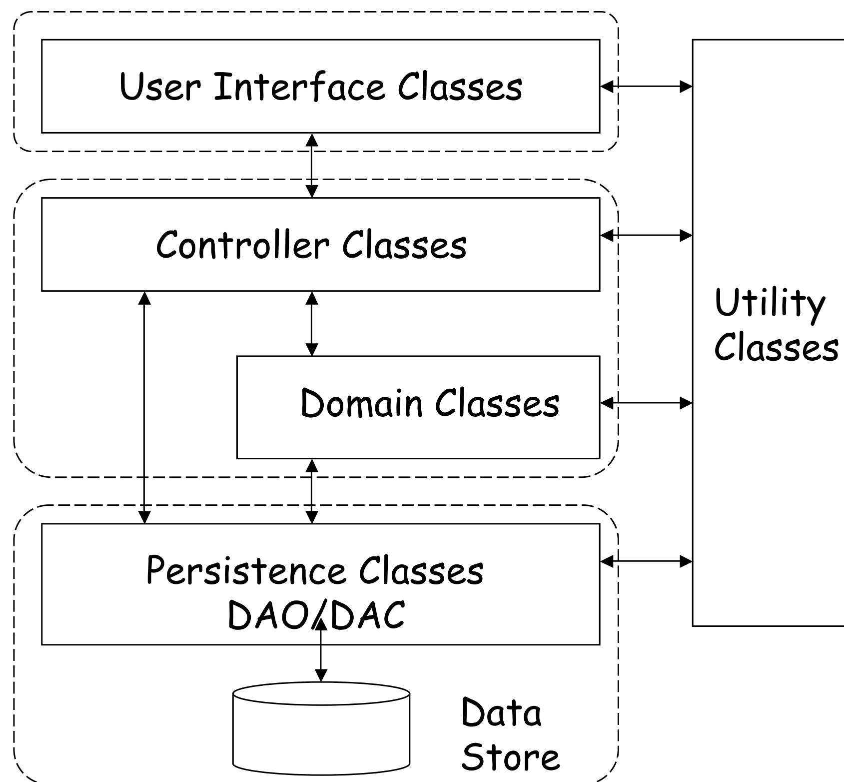
ORM DAO – DCS (cont.)

L'uso di DCS è una variante all'approccio DAO "puro" che prevede che i controller interagiscano esclusivamente con le Data Access Classes associate alle classi di dominio.

Tale variante permette di avere un codice più snello ed efficiente nel caso in cui molte operazioni richiedano il coinvolgimento di più classi di dominio.

I Data Access Objects sono accessibili **solo** tramite le Domain Classes, mentre i Data Control Services sono accessibili **solo** tramite Controller (o al più anche Domain Classes)

ORM: Architettura con DAO e DCS



ORM: DAO e DCS – esempio

- `Studente` ha responsabilità sullassociazione `Iscrizione`
- La classe DAO corrispondente a `Studente` conterrà i metodi necessari per linserimento, tra cui il metodo per linserimento della facoltà a cui è iscritto
  → una classe DAO può accedere di fatto a tutto il DB e non solo alla tabella che corrisponde alloggetto studente!

- Se invece chiedo tutti gli studenti iscritti ad una certa facoltà, ho bisogno di definire un DCS (perché `Facoltà` non ha responsabilità)

Approcci alternativi all'uso dei DCS

1. Eseguire operazioni complesse, ad es. query, a livello di applicazione (ad es., tramite join a livello applicazione)
   $\Rightarrow$ questa soluzione è in generale inefficiente

2. Definire delle ulteriori classi di dominio ("dummy" o "fantoccio") che incapsulano le funzionalità non "localizzabili", ciascuna associata ad una o più classi DAO per l'accesso al DB

   Le classi "fantoccio" hanno l'unico scopo di esporre ai controller metodi che non fanno altro che inovocare a loro volta i corrispondenti metodi DAO

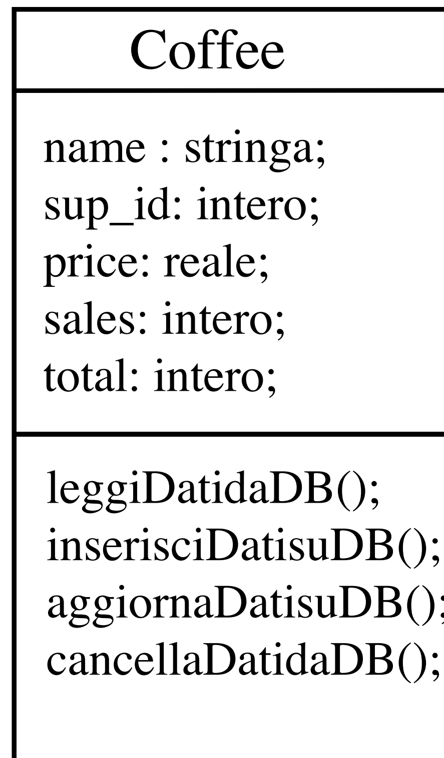
   $\Rightarrow$ favorisce il disaccoppiamento di moduli demandati a scopi differenti, ma in genere inefficiente e macchinoso

Data Control Services - Vantaggi

Per aumentare l'efficienza dell'applicazione è quindi in genere conveniente adottare un approccio meno vincolante, basato sull'uso di classi DCS (ed evitare così la realizzazione di classi di dominio dummy)

Questo approccio è particolarmente adatto nei casi in cui vi sono molte interrogazioni che coinvolgono più di una entità (raggruppate in gruppi omogenei, ciascuno implementato tramite una specifica classe DCS)

Esempio



Esempio

Costruiamo la classe di dominio `Coffee` come segue

```
public class Coffee {
     private final String name;
     private int sup_id;
     private double price;
     private int sales;
     private int total;

     public Coffee(String name) {
       this.name = name;
     }

    public String getName(){ return name; }
```

```
public int getSup_ID(){ return sup_id; }

public void setSup_ID(int sup_id){
  this.sup_id = sup_id;}

public double getPrice(){ return price; }

public void setPrice(double price){
  this.price = price;}

public int getSales(){ return sales; }

public void setSales(int sales){
  this.sales = sales;}

public int getTotal(){ return total; }
```

```
public void setTotal(int total){
  this.total = total;}

public void leggiDatidaDB() throws Exception
  {CoffeeDAO.load(this);}

public void inserisciDatisuDB() throws Exception
  {CoffeeDAO.insert(this);}

public void aggiornaDatisuDB() throws Exception
  {CoffeeDAO.update(this);}

public void cancellaDatidaDB() throws Exception
  {CoffeeDAO.delete(this);}
```

```
//Ridefinizione dell'operatore di uguaglianza
public boolean equals(Object o) {
    if (o == null)
        return false;
    if (!(o instanceof Coffee))
        return false;
    Coffee o1 = (Coffee) o;
    return this.name.equals(o1.name);
}

//Ridefinizione dell'operatore di hashing
public int hashCode() {
    return name.hashCode();}
```

```
    //Ridefinizione dell'operatore toString()
    public String toString() {
      return "cofName: "+this.name+" sup_id: "+this.sup_id+
       " price: "+this.price+" sales: "+this.sales+
       " total: "+total;
      }
}
```

Esempio

Supponiamo ora di avere la seguente tabella relazionale COFFEE memorizzata su una base di dati

```
CREATE TABLE COFFEE (
  COF_NAME VARCHAR(32) PRIMARY KEY,
  SUP_ID INTEGER UNIQUE,
  PRICE FLOAT,
  SALES INTEGER,
  TOTAL INTEGER
)
```

Esempio

La classe DAO corrispondente alla classe `Coffee` può essere implementata come segue

```
import java.util.*;
import java.sql.*;
/ * Data Access Object per l'entita' Coffee.
  * Incapsula le funzioni ed i tipi dato necessari
  * per manipolare le informazioni
  * della base dati pertinenti a detta entita'.
  * Si tratta di una utility class.
  */


public class CoffeeDAO {
  private CoffeeDAO() {}
```

```
// Comando SQL per l'inserimento di una nuova istanza
private static final String INSERT_SQL =
            "INSERT INTO COFFEES VALUES (?, ?, ?, ?, ?)";
public static void insert(Coffee coffee)
             throws SQLException, ClassNotFoundException{
  Connection con=null;
  PreparedStatement pstmt=null;
  con = ConnectionManager.getConnection();
  pstmt = con.prepareStatement(INSERT_SQL);
  pstmt.setString(1, coffee.getName());
  pstmt.setInt(2, coffee.getSup_ID());
  pstmt.setDouble(3, coffee.getPrice());
  pstmt.setInt(4, coffee.getSales());
  pstmt.setInt(5, coffee.getTotal());
  pstmt.executeUpdate();
  pstmt.close();
  con.close(); //si assume la modalita' autocommit
}
```

```
//Comando SQL per l'ottenimento di una nuova istanza
private static final String FIND_BY_NAME =
             "SELECT * FROM COFFEES WHERE COF_NAME = ?";
public static void load(Coffee coffee)
            throws SQLException, ClassNotFoundException{
  Connection con=null;
  PreparedStatement pstmt=null;
  ResultSet rs=null;
  con = ConnectionManager.getConnection();
  pstmt = con.prepareStatement(FIND_BY_NAME);
  pstmt.setString(1, coffee.getName());
  rs=pstmt.executeQuery();
  rs.next();
  coffee.setSup_ID(rs.getInt("SUP_ID") );
  coffee.setPrice(rs.getDouble("PRICE") );
  coffee.setSales(rs.getInt("SALES") );
  coffee.setTotal(rs.getInt("TOTAL") );
  rs.close();
  pstmt.close();
  con.close();
}
```

```
//Comando SQL per l'aggiornamento di una nuova istanza
private static final String UPDATE_BY_NAME =
  "UPDATE COFFEE SET SUP_ID=?,PRICE=?,SALES=?,TOTAL=? WHERE COF_NAME =
public static void update(Coffee coffee)
      throws SQLException, ClassNotFoundException{
  Connection con=null;
  PreparedStatement pstmt=null;
  ResultSet rs=null;
  con = ConnectionManager.getConnection();
  pstmt = con.prepareStatement(UPDATE_BY_NAME);
  pstmt.setString(5, coffee.getName());
  pstmt.setInt(1, coffee.getSup_ID());
  pstmt.setDouble(2, coffee.getPrice());
  pstmt.setInt(3, coffee.getSales());
  pstmt.setInt(4, coffee.getTotal());
  pstmt.executeUpdate();
  pstmt.close();
  con.close();
}
```

```
  //metodo lasciato per esercizio
  public static void delete(String cof_name){ }
}
```

Esempio di classe di controllo

Il seguente controller espone un metodo per effettuare la modifica del prezzo di un tipo di caffè

```
public class CoffeeController {
    public void aggiornaPrezzo(String nomeCaffe, Double nuovoPrezzo)
     {
       Coffee caffe = new Coffee(nomeCaffe);
       caffe.SetPrice(nuovoPrezzo);
       try{
         caffe.aggiornaDatiSuDB();
         }
       catch(Exception ex){
          System.out.println("Eccezione ("+ex.getMessage()+")");
        }
    }
}
```

Esempio di Domain Control Service

Definiamo una classe DCS per implementare alcune funzionalità di ricerca sulla tabella `Coffee`

```
public class CoffeeDCS {
  private CoffeeDCS() {}

  //metodo ausiliario che crea un oggetto a partire
  //da una riga del result set
  private static Coffee objectFromCursor(ResultSet rs)
     throws SQLException {
     Coffee c = new Coffee(rs.getString(1));
     c.setSup_ID(rs.getInt(2));
     c.setPrice(rs.getFloat(3));
        c.setSales(rs.getInt(4));
        c.setTotal(rs.getInt(5));
        return c;
     }
```

```
//metodo ausiliario che costruisce una lista di oggetti
//di tipo Coffee a partire dalle righe di un result set
private static List processCollectionResultSet(ResultSet rs)
  throws SQLException {
  // La collezione e' una lista collegata per
  // ottimizzare le operazioni di inserimento
  LinkedList all = new LinkedList();
  while (rs.next()) {
    all.add(objectFromCursor(rs));
  }
  return Collections.unmodifiableList(all);
}
```

```
//Comando SQL per la funzione di ricerca
//di tutte le istanze
private static final String FIND_ALL_SQL ="SELECT * FROM COFFEES";

public static Collection findAll()
  throws SQLException, ClassNotFoundException{
  Collection all=null;
  Connection con=null;
  Statement stmt=null;
  ResultSet rs=null;
  con = ConnectionManager.getConnection();
  stmt = con.createStatement();
  rs = stmt.executeQuery(FIND_ALL_SQL);
  all = processCollectionResultSet(rs);
  stmt.close();
  rs.close();
  con.close();
  return all;
}
```

```
  // altri metodi (la cui definizione e' lasciata
  // come esercizio)

  public static Coffee findBySup_id(Double cof_sup_id) { }

  public static Collection findByPrice(Double cof_price) { }

  public static Collection findBySell(Int cof_sales){ }

  public static Collection findByTotal(Int cof_total){ }
}
```

La classe di controllo

Il controller espone anche un metodo per la stampa di tutti i caffè

```
public class CoffeeController {
 ...
 public static void stampaTutti() {
   try
     {Collection coll = CoffeeDCS.findAll() ;
      Iterator iter = coll.iterator();
      while(it.hasNext) {
        Coffee c = (Coffee) iter.next();
        System.out.println(c.toString());
       }
     }
   catch(Exception ex) {
     System.out.println("Eccezione ("+ex.getMessage()+")");}
   }
}
```

Incapsulare le eccezioni

- Lo strato che incapsula l'accesso al database (formato dalle classi DAO e DCS) rende **le classi di domino ed i controller** completamente **indipendenti dalla base dati**.
- Non solo in queste classi non si fa uso di comandi SQL, ma anche **l'API JDBC non viene utilizzata**.

→ Siamo costretti a catturare come eccezioni di classe `Exception` (quindi generiche) le eccezioni generatesi nelle classi DCS e DAO, che sono di classe `SQLException` (e di classe `ClassNotFoundException`).

Incapsulare le eccezioni (cont.)

- Nel momento in cui gestiamo l'eccezione **non possiamo quindi accedere ai metodi specifici di queste classi** ad es., `getErrorCode()`, `getSQLState()`).
- Inoltre, **l'interazione con la base di dati a seguito di una eccezione** (ad esempio per effettuare un `rollback`) può diventare macchinosa.

Incapsulare le eccezioni (cont.)

Una soluzione potrebbe essere catturare le eccezioni a livello di DAO e DCS

⇒ Questa soluzione implica però che a livello di Persistence Classes ci si deve occupare anche di informare l'utente del tipo di eccezione verificatasi, compito invece demandato alle User Interface Classes.

Per risolvere questo problema, **definiamo una classe di eccezioni specifica per l'applicazione**.

Questa classe fa parte dell'insieme delle Utility Classes usate dall'applicazione (classi di sistema e classi definite dal programmatore per l'applicazione in questione).

La classe **MyException**

```
public class MyException extends Exception {
   public MyException (String messaggio) {
      super(messaggio);
   }
}
```

Esempio

Nota: In tutti gli esempi che seguono, assumiamo che il livello di isolamento di default sia `REPEATABLE READ`

```
public class CoffeeDCS {
...
  public static void AggiornaTransaz(Collection coll)
                                  throws MyException {
    Connection con=null;
    PreparedStatement updateSales=null;
    PreparedStatement updateTotal=null;
    String updateString="UPDATE COFFEES "+
                "SET SALES=? WHERE COF_NAME=?";
    String updateStatement="UPDATE COFFEES "+
                "SET TOTAL=TOTAL+? WHERE COF_NAME=?";

    try{
      con=ConnectionManager.getConnection();
```

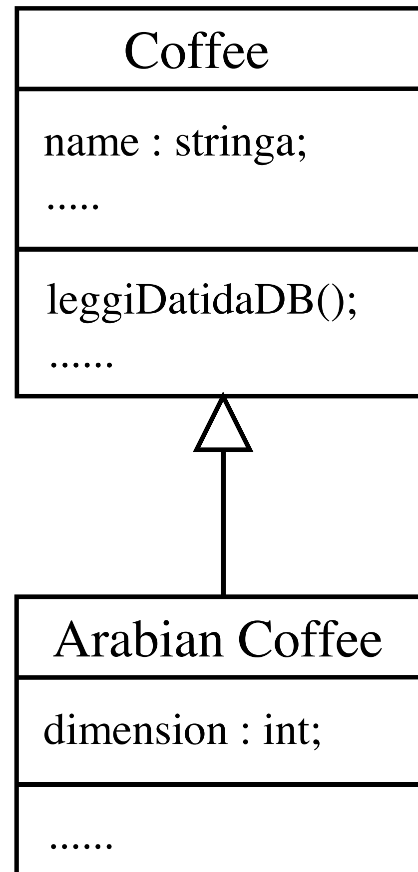
```
    updateSales=con.prepareStatement(updateString);
    updateTotal=con.prepareStatement(updateStatement);
    Iterator iter = coll.iterator();
    con.setAutoCommit(false);
    con.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED)
    while (iter.hasNext()){
      Coffee c = (Coffee) iter.next();
      updateSales.setInt(1,c.getSales());
      updateSales.setString(2,c.getName());
      updateSales.executeUpdate();
      updateTotal.setInt(1,c.getSales());
      updateTotal.setString(2,c.getName());
      updateTotal.executeUpdate();
      con.commit();
    }//end while
  }// end try


  catch (SQLException ex){
    if (con != null){
      try{ con.rollback(); }
```

```
          catch (SQLException excep) {
            throw new MyException("Error: "+excep.getMessage()); }
          }//end if
        }// end catch (ex)
      finally{
        if (con != null)
        try{
            updateSales.close();
            updateTotal.close();
            con.setAutoCommit(true);
            con.setTransactionIsolation(Connection.REPEATABLE_READ);
            con.close();
        }
        catch (SQLException ex2) {
            throw new MyException("Error: "+excep.getMessage());
        }
      }//end finally
    }// end AggiornaTransaz
  }//end class
```

La classe di controllo

```
public class CoffeeController {
 ...
  public void aggiornaVendite(int[] salesForWeek; String[] coffees){
    int len=coffees.length;
    //creaimo la collezione di caffe'
    LinkedList all = new LinkedList();
    for(int i=0; i<len; i++) {
      Coffee c = new Coffee(coffees[i]);
      c.setSales(salesForWeek[i]);
      all.add(c);
    }//end for
    try { AggiornaTransaz(all); }
    catch(MyException ex) {
      System.out.println("Eccezione ("+ex.getMessage()+")"); }
    }
  }
}
```

ISA – Esempio



ISA – Esempio (cont.)

```
public class Arabian extends Coffee {
  private int dimension;

  public void setDimension (int dim){ this.dimension = dim; }

  public int getDimension(){ return dimension; }

  public void leggiDatidaDB() throws MyException {ArabianDAO.load(this);}

  public void inserisciDatisuDB() throws MyException
                                  {ArabianDAO.insert(this);}

  public void aggiornaDatisuDB() throws MyException
                                  {ArabianDAO.update(this);}

  public void cancellaDatidaDB() throws MyException
                                  {ArabianDAO.delete(this);} }
```

ISA – Esempio (cont.)

Assumendo che nello schema relazionale sia presente la tabella

```
CREATE TABLE ARABIAN(
NAME VARCHAR(32) PRIMARY KEY REFERENCES COFFEE(COF_NAME),
DIMENSION INTEGER
)
```

creiamo la seguente classe DAO (solo il metodo `insert`, il resto si lascia come esercizio)

```
public class ArabianDAO {
  private ArabianDAO() {}

  // Comandi SQL per l'inserimento di una nuova istanza
  private static final String INSERT_COFFEE =
              "INSERT INTO COFFEES VALUES (?, ?, ?, ?, ?)";
  private static final String INSERT_ARABIAN =
              "INSERT INTO ARABIAN VALUES (?, ?)";
```

```
public static void insert(Arabian ar_coffee) throws MyException{
  Connection con=null;
  PreparedStatement pstmt1=null;
  PreparedStatement pstmt2=null;

  try{
    con = ConnectionManager.getConnection();
    con.setAutoCommit(false);
    con.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED)
    pstmt1 = con.prepareStatement(INSERT_COFFEE);
    pstmt1.setString(1, ar_coffee.getName());
    pstmt1.setInt(2, ar_coffee.getSup_ID());
    pstmt1.setDouble(3, ar_coffee.getPrice());
    pstmt1.setInt(4, ar_coffee.getSales());
    pstmt1.setInt(5, ar_coffee.getTotal());
    pstmt1.executeUpdate();
    pstmt2 = con.prepareStatement(INSERT_ARABIAN);
```

```
    pstmt2.setString(1, ar_coffee.getName());
    pstmt2.setInt(2, ar_coffee.getDimension());
    pstmt2.executeUpdate();
    con.commit();
  }



  catch (SQLException ex){
    if (con != null){
      try{ con.rollback(); }
      catch (SQLException excep) {
        throw new MyException("Error: "+excep.getMessage()); }
      }//end if
  }// end catch (ex)
```

```
    finally{
      if (con != null)
      try{
          pstmt1.close();
          pstmt2.close();
          con.setAutoCommit(true);
          con.setTransactionIsolation(Connection.REPEATABLE_READ);
          con.close();
      }
      catch (SQLException ex2) {
        throw new MyException("Error: "+excep.getMessage());
      }
    }//end finally
  }// end insert
}//end class
```

ISA – Nota

Il seguente codice, che prevede di invocare il metodo `insert` della classe `CoffeeDAO` all'interno nel metodo `insert` della classe `ArabianDAO()` sarebbe stato errato:

```
...
try{con = ConnectionManager.getConnection();
    con.setAutoCommit(false);
    con.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
    CoffeeDAO.insert(ar_coffe);
    pstmt2 = con.prepareStatement(INSERT_ARABIAN);
    pstmt2.setString(1, ar_coffee.getName());
    pstmt2.setInt(2, ar_coffee.getDimension());
    pstmt2.executeUpdate();
    con.commit();
  }
...
```

ISA – Nota (cont.)

Infatti, nel momento in cui eseguiamo il metodo `CoffeeDAO.insert(ar_coffe);` **viene iniziata una nuova transazione** (se il metodo `ConnectionManager.getConnection()` lo consente – vedi slide su JDBC).

Questo fa perdere l'**atomicità** dell'operazione di inserimento di un oggetto di classe `Arabian`.

In generale, una transazione deve essere definita completamente all'interno di un metodo che risiede in una classe DAO o DCS, e non si può invocare al suo interno alcun altro metodo che accede al database. Questo infatti inizierebbe una nuova transazione.

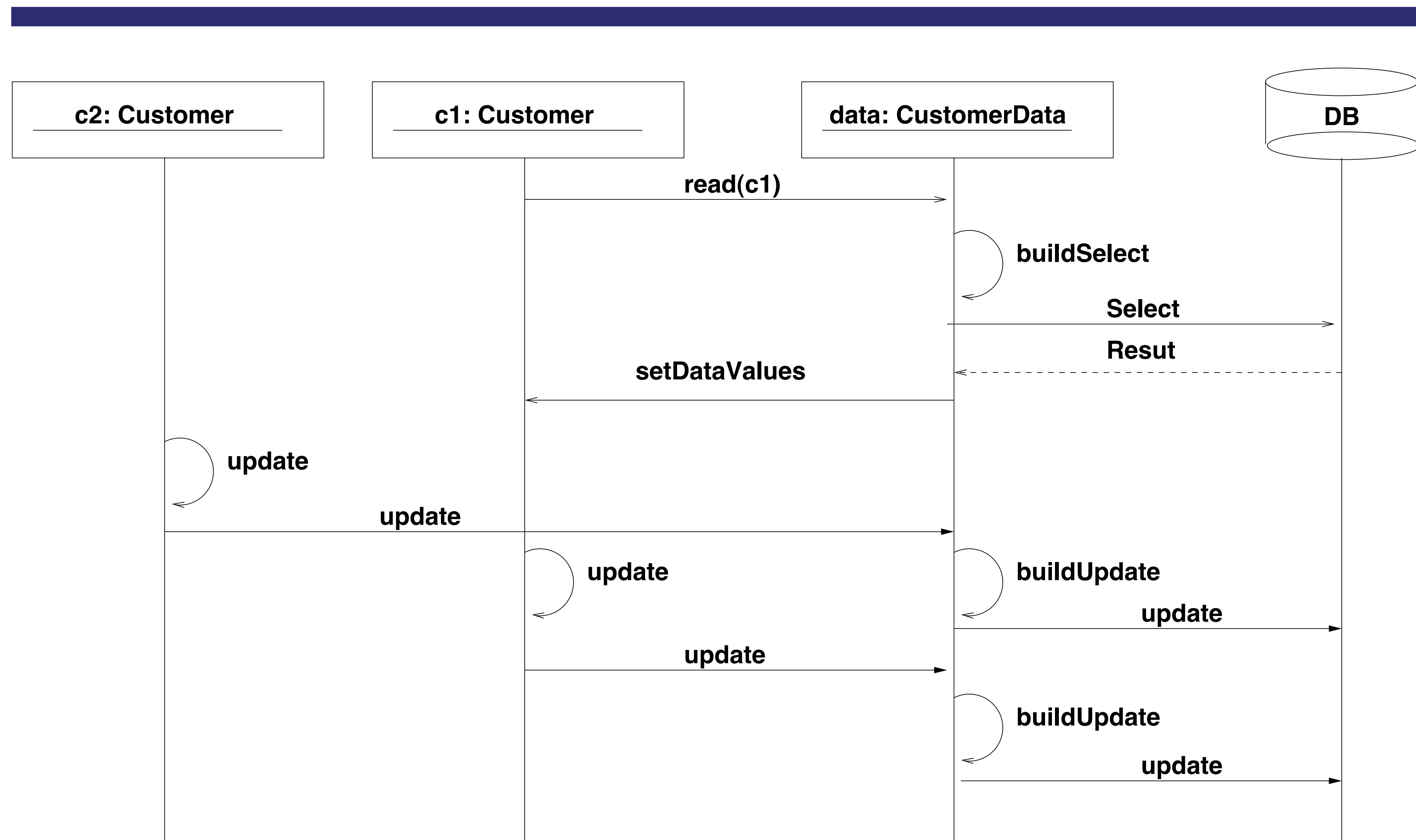
Transazioni e oggetti

Problema fondamentale

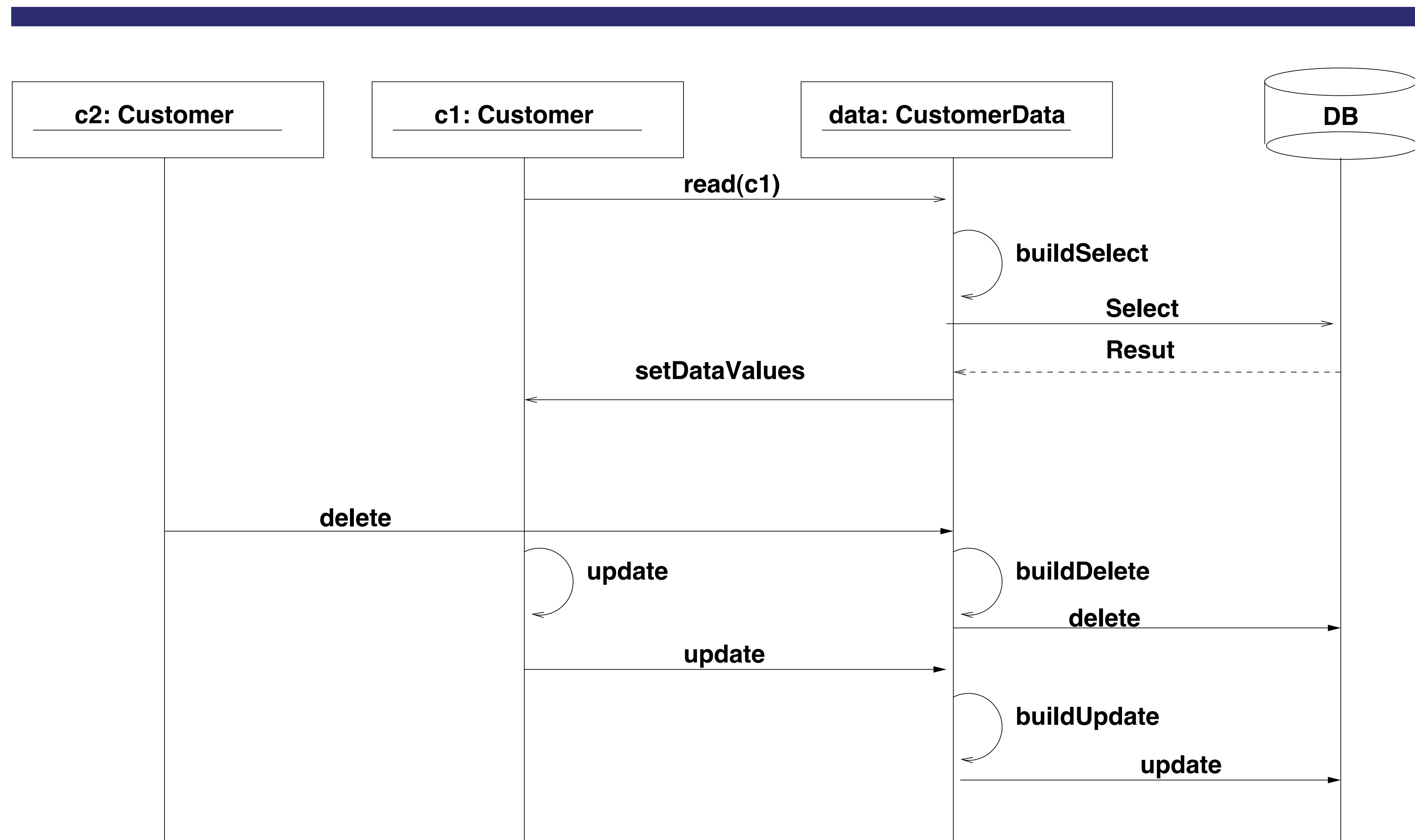
Gli oggetti della applicazione che rappresentano dati persistenti potrebbero essere inconsistenti rispetto ai dati nel DB, o tra loro

- Infatti, le operazioni sugli oggetti non sono immediatamente aggiornate nel DB
- Occorre gestire con accortezza gli oggetti dell'applicazione che rappresentano dati persistenti

Esempio



Esempio



Osservazioni

Gli oggetti c1 e c2 aggiornano (o cancellano) i proprî dati nel DB, attraverso un oggetto che incapsula l'accesso ai dati.

Può accadere che i due aggiornamenti confliggano?

Sì, se c1 e c2 rappresentano il medesimo cliente (Customer), oppure se rappresentano clienti legati per mezzo di qualche vincolo

Osservazioni

- Abbiamo a che fare con transazioni **su oggetti** oltre che su dati
- Dobbiamo garantire che una transazione eseguita sugli oggetti venga poi trasferita in modo "opportuno" nel DB
- Possiamo avvalerci del supporto transazionale del DBMS

Osservazioni

- Si noti che una transazione per il DBMS può essere definita solo a livello del DAO
- L'applicazione deve quindi "suddividere" il lavoro in transazioni per il DBMS che vengono innescate con opportune chiamate ai metodi delle classi del DAO
- In alcuni casi questo non è semplice, soprattutto per quelle operazioni che richiedono più accessi ai dati dipendenti da alcune scelte fatte a livello di applicazione (magari interagendo con l'utente)

Concetto fondamentale

Lo stato degli oggetti coinvolti in una transazione costituisce il **contesto** della transazione

Esempio L'oggetto di tipo Customer nell'esempio precedente contiene i dati sui quali è effettuato l'aggiornamento. Tali dati costituiscono il contesto della transazione.

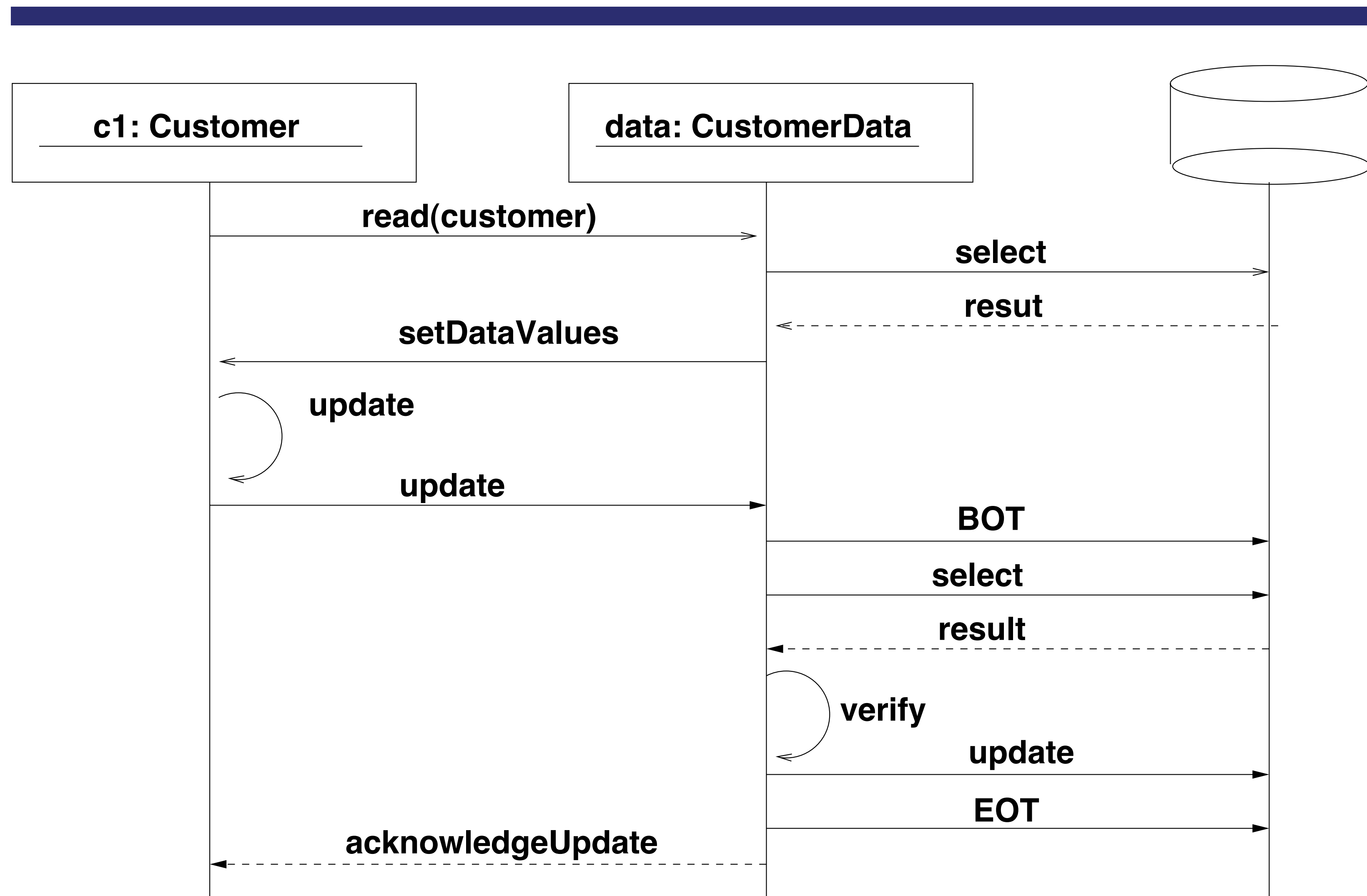
Long-lived transactions

- Tipicamente passa un lungo intervallo di tempo tra il momento in cui inizializziamo gli oggetti leggendo dal DB e il momento in cui eseguiamo degli aggiornamenti.
- Dal punto di vista degli oggetti, stiamo eseguendo una transazione con vita estremamente lunga (**long-lived transaction**)
- Questo è un problema, perché se facciamo corrispondere a tale transazione una transazione sul DB (quando possibile), ciò è estremamente inefficiente, a meno che non si usi un livello di isolamento basso.
- Lo stesso accade se implementiamo (con fatica) un meccanismo di **lock** sugli oggetti dell'applicazione.

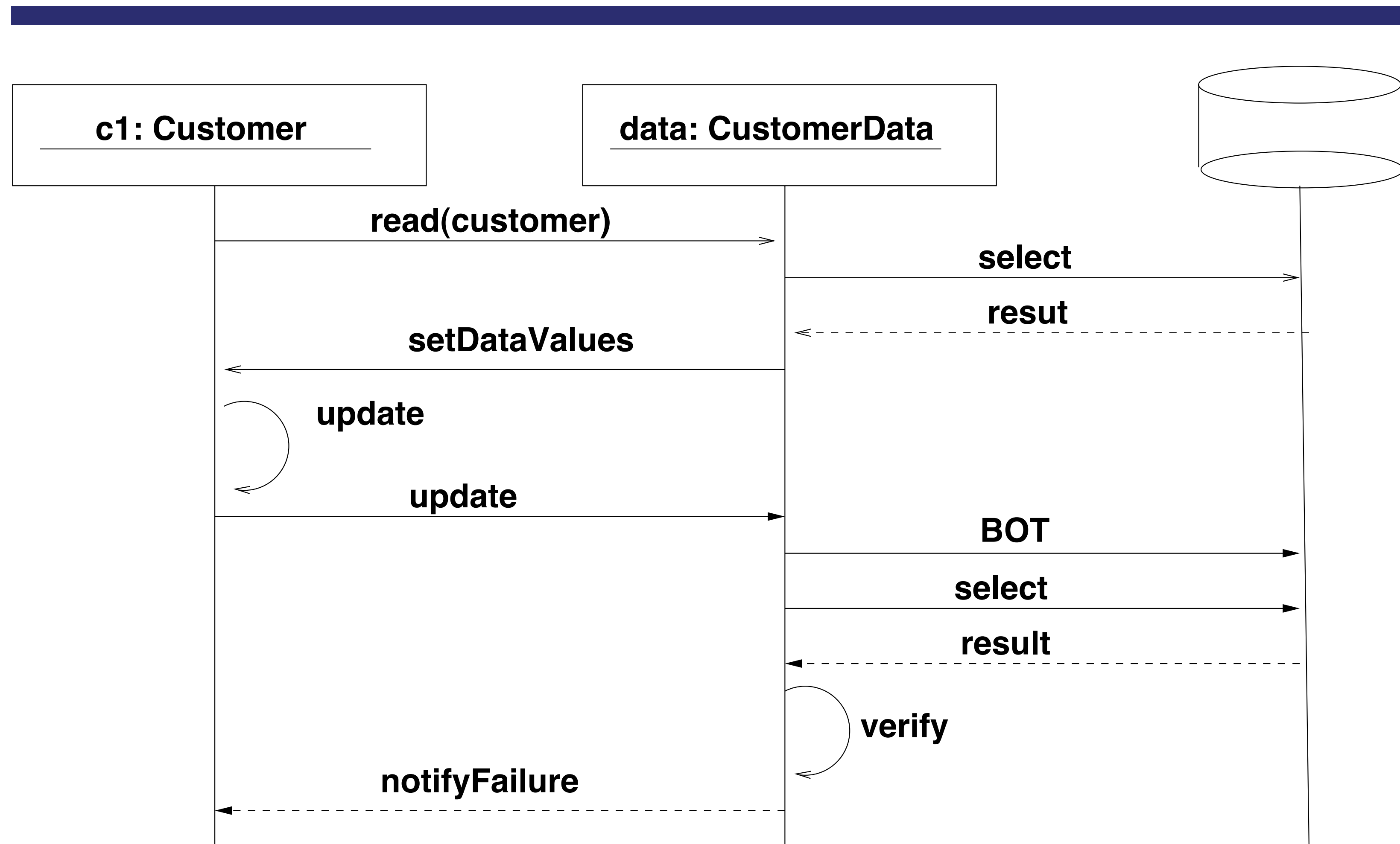
Una possibile soluzione

- Prima di aggiornare i dati nel DB, si verifica che i dati nel DB siano consistenti con quelli letti quando l'oggetto è stato inizializzato. Se questo non si verifica, si annulla l'operazione.
- Per procedere in questo modo abbiamo bisogno di memorizzare opportunamente lo stato degli oggetti.
- La verifica al punto precedente e l'aggiornamento successivo (se la verifica ha esito positivo) vengono effettuati all'interno di una transazione del DB.
- Il controllo di concorrenza è affidato al DBMS, e il livello di isolamento è posto al **livello minimo necessario**.

Esempio



Esempio



Esempio

- Ritorniamo all'esempio del caffè, in cui **non** ci siamo finora occupati di controllare eventuali accessi concorrenti alla base dati
- In particolare, potrebbe accadere che nel momento in cui aggiorniamo il valore del caffè sulla base dati, sovrascriviamo un aggiornamento concorrente (questo è in genere indesiderabile)

Esempio (cont.)

Modifichiamo la classe di dominio `Coffee`, aggiungendo le seguenti variabili di istanza

```
private int oldSup_id;
private float oldPrice;
private int oldSales;
private int oldTotal;
```

Esempio (cont.)

Inoltre, modifichiamo tutti i metodi `set` della classe `Coffee` in modo da mantenere il vecchio stato a seguito di modifiche (assumiamo per semplicità che da applicazione venga si possa aggiornare solo il prezzo di un caffè). Ad esempio,

```
public void setPrice(float price){
  this.oldPrice = this.price;
  this.price = price;
}
```

Esempio (cont.)

Riscriviamo il metodo `update` della classe `CoffeeDAO` in modo che effettui il confronto con il vecchio stato memorizzato

```
//Comando SQL per l'ottenimento di una nuova istanza
private static final String FIND_BY_NAME =
           "SELECT * FROM COFFEES WHERE COF_NAME = ?";

//Comando SQL per l'aggiornamento di una istanza
private static final String UPDATE =
        "UPDATE COFFEE set PRICE= ? WHERE NAME = ?"


public static void update(Coffee caffe) throws MyException{
  try{
      con = ConnectionManager.getConnection();
      con.setAutoCommit(false);
```

```
con.setTransactionIsolation(
      Connection.TRANSACTION_REPEATABLE_READ);
pstmt = con.prepareStatement(FIND_BY_NAME);
pstmt.setString(1, caffe.getName());
rs=pstmt.executeQuery();
if (rs.next())
   if (rs.getFloat(3)==caffe.getOldPrice()){
      pstmt = con.prepareStatement(UPDATE);
      pstmt.setFloat(1, caffe.getPrice());
      pstmt.setString(2, caffe.getName());
      pstmt.executeUpdate();
      con.commit();
     }
   else{
      // lancia una eccezione poiche' la verifica
      // sullo stato e' fallita
     con.setAutoCommit(true);
     con.setTransactionIsolation(Connection.REPEATABLE_READ);
     throw new MyException("errore di aggiornamento");
     }
}
```

```
    catch (SQLException ex){
       if (con != null){
         try{ con.rollback(); }
         catch (SQLException excep)
            { throw new MyException("Error: "+excep.getMessage()); }
       }//end if
     }// end catch (ex)
     finally{
       if (con != null)
       try{
           pstmt.close();
           con.setAutoCommit(true);
           con.setTransactionIsolation(Connection.READ_COMMITED);
           con.close();
           }
       catch (SQLException ex2)
           { throw new MyException("Error: "+excep.getMessage());}
     }//end finally
   }// end insert
 }//end class
```

Transazioni su più oggetti

- Il contesto è rappresentato da un **insieme** di oggetti
- La classe di accesso ai dati dovrà avere un metodo che ha in ingresso una **collezione** di oggetti, e aggiorna gli oggetti di tale collezione tutti in una medesima transazione
- Ovviamente, ciò va fatto **solo** se è richiesto che l'aggiornamento sia effettivamente una operazione atomica