Programmazione di un database

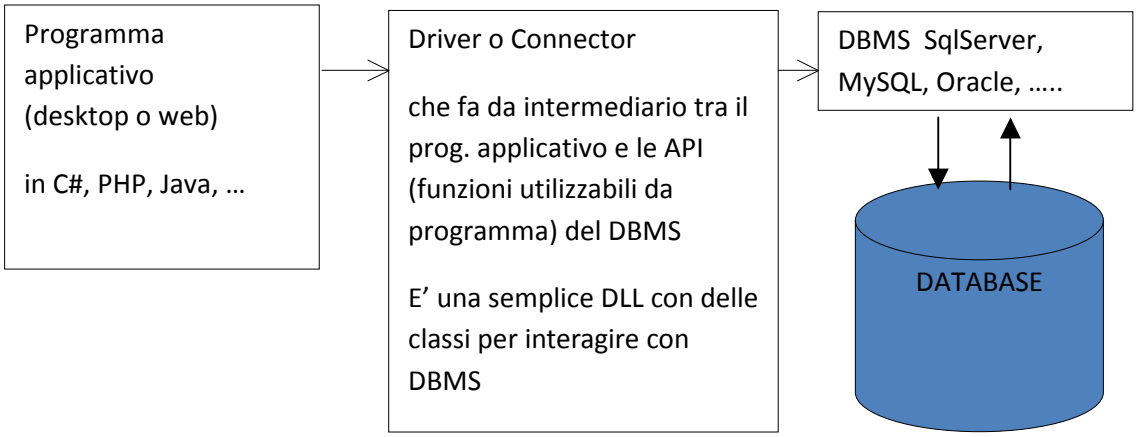
Autore BANDIERA ROBERTO - ***ver. 2 - febbraio 2014***

(nota sulla nuova versione: semplificato l'uso della classe PDO)

Questo lavoro rappresenta una guida alla comprensione e all'utilizzo delle moderne tecniche di realizzazione di programmi applicativi che accedono ai dati di un database. Verrà trattata la programmazione in linguaggio C# e in linguaggio PHP.

Per la programmazione dell'accesso ai dati di un database si può utilizzare uno dei seguenti due metodi:

1) Metodo tradizionale (per applicazioni "piccole" e applicazioni web)

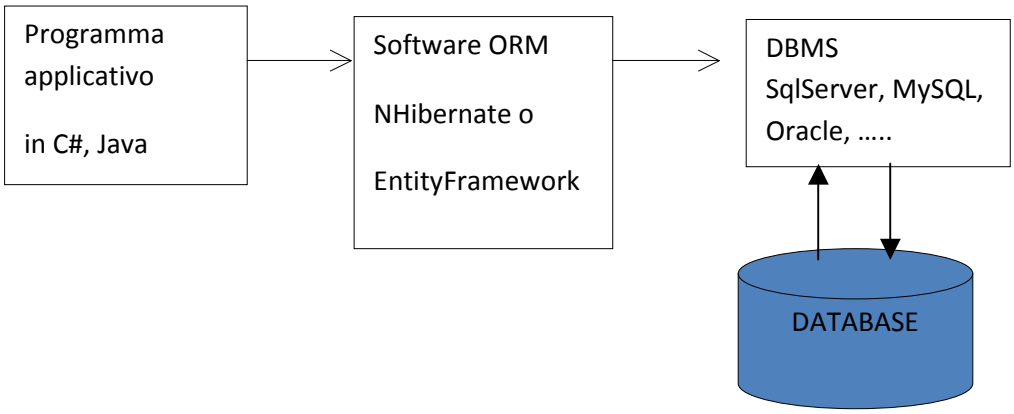


Il programma applicativo è indipendente dalla struttura logica e fisica dei dati e addirittura dal DBMS stesso nel senso che posso cambiare DBMS e il programma applicativo rimane funzionante, senza nessuna modifica ne' ricompilazione!!

Si interroga il DATABASE utilizzando delle API (classi apposite che lavorano con il DBMS) e si usa il linguaggio SQL Esempio: SELECT * FROM CLIENTI WHERE CITTA' = "TV"

2) Metodo "Nuovo" che rende trasparente il database: si utilizza un software ORM Object-Relational Mapper come NHibernate oppure EntityFramework di Microsoft

Questo metodo si usa tipicamente per applicazioni enterprise (che gestiscono database grandi!!!)



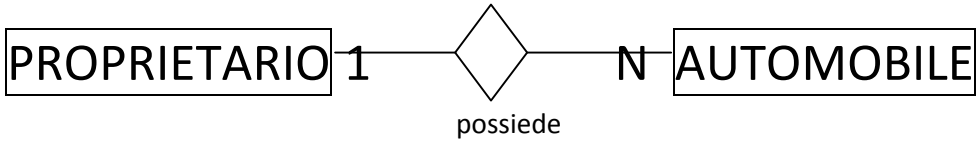
Il programma applicativo usa oggetti “persistenti” senza sapere che dietro c’è un database: si interagisce semplicemente con una collezione di oggetti in RAM.
Con EntityFramework, per cercare e aggiornare i dati si usa LINQ To Entities
Ad esempio: var elencoClienti = context.Clients.Where(x=>x.Città==”TV”)

METODO TRADIZIONALE

Abbiamo un piccolo DB di esempio di nome PRA con le seguenti due tabelle:

Proprietari (**codiceFiscale**, nome, cittàResidenza, AnnoPatente)
Automobili (**targa**, modello, cilindrata, codiceProprietario)

Si tratta di due tabelle che ipotizziamo essere collegate in modalità “uno a molti” (1-N) , ovvero si considera il caso semplificato dove ciascuna automobile ha un solo proprietario.



Per programmare l’accesso a questo DB mi servono delle classi:

- Classi DTO (Data Transfer Objects)
- Classi DAO (Data Access Objects)

Le classi DTO servono per rappresentare i dati estratti dal database: tipicamente si crea una classe per ogni tabella ma si potrebbero anche creare delle classi DTO specifiche per rappresentare dati ottenuti da sintesi e raggruppamenti di dati del database.

In questo caso si creano le classi Proprietario e Automobile.
Attenzione che i tipi di dati usati dal DB sono diversi da quelli del C#: ad esempio nel DB si ha CHAR(20) mentre nel C# ci sono solo stringhe a lunghezza variabile con il tipo stringa.
Le corrispondenze sono CHAR(n), VARCHAR(n) e TEXT con stringa, INT con int, NUMERIC(n,d) e DECIMAL(n,d) con Decimal, FLOAT e DOUBLE con double, DATE, TIME, DATETIME con DateTime

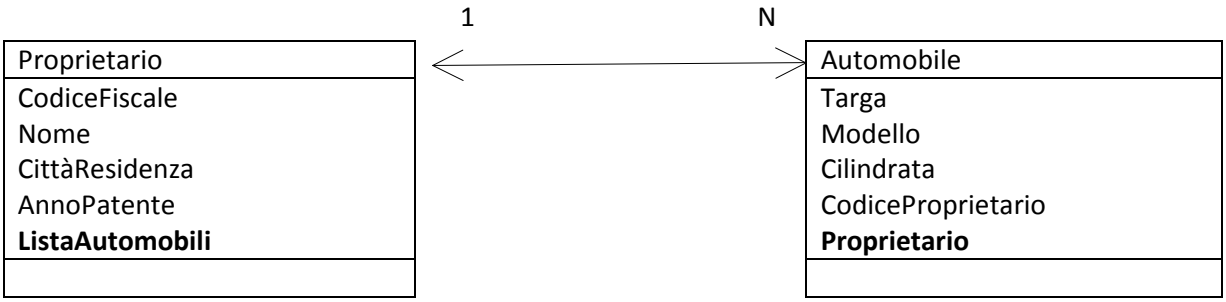
```
public class Proprietario
{       // una property per ogni attributo
        public  string CodiceFiscale {get; set;}
        public  string Nome {get; set;}
        public  string CittàResidenza {get; set;}
        public  int  AnnoPatente {get; set;}

  // navigation property  che consentono di “navigare” nella mia struttura dati in RAM
        public  List<Automobile>  ListaAutomobili {get; set;}
}
```

```
public class Automobile
{ // property
  public  string Targa {get; set; }
   public  string Modello {get; set;}
   public  int  Cilindrata {get; set;}
   public string CodiceProprietario {get;  set;}

    // navigation property  che consentono di "navigare" nella mia struttura dati in RAM
    public  Proprietario  proprietario {get; set; }
}
```

Diagramma delle classi UML



NOTA: non è detto che ci servano sempre i riferimenti (ListaAutomobili e Proprietario): potremmo anche non metterli o lasciarli con valore null e quindi non avere la possibilità di navigare tra gli oggetti. Si noti il fatto che mentre nel database la navigazione tra le tabelle e' sempre bidirezionale e si avvale dei valori della chiave esterna e della chiave primaria delle tabelle collegate, nella programmazione ad oggetti, invece **la navigazione è direzionale** ed è consentita dalla presenza esplicita in una classe di un riferimento, o puntatore, ad un oggetto di un'altra classe.

Le classi DAO servono per accedere ai dati del database e caricarli in RAM negli oggetti DTO oppure per salvare nelle tabelle del DB oggetti presenti in RAM nelle classi DTO.
Esse contengono i metodi CRUD (Create, Retrieve, Update, Delete).
Di solito si crea una classe DAO per ciascuna tabella del database (ma nessuno vi impedisce di creare un'unica classe DAO per l'intero DB anche se in questo caso vi trovereste parecchie decine di metodi CRUD).

GestioneProprietari
MessaggioErrore

// metodi CRUD
List<Proprietario> RecuperaTutti()
Proprietario RecuperaPerCodiceFiscale(unCodiceFiscale)
int Inserisci(unProprietario) // ritorna 0/1 se successo o no
…

GestioneAutomobili
MessaggioErrore
// metodi CRUD ….

I metodi CRUD sostanzialmente inviano al DBMS una query SQL e ottengono in risposta i dati richiesti.
Il .Net Framework fornisce una serie di classi astratte che si trovano nel Namespace **System.Data.Common** da cui gli specifici driver dei singoli DBMS costruiscono le classi specifiche di accesso al DBMS.

Classi astratte ➔ DBConnection, DBCommand, DBDataReader
Classi concrete per MySQL ➔ MySqlConnection, MySqlCommand, MySqlDataReader
Classi concrete per SQLServer ➔ SqlConnection, SqlCommand, SqlDataReader
Classi concrete per SQLServer Compact Edition ➔ SqlCEConnection, …..

Scriviamo il primo metodo CRUD per recuperare tutti i record della tabella Proprietari:

```
public List<Proprietario> RecuperaTutti()
{
    List<Proprietario> elenco = new List<Proprietario>();
    // bisogna gestire l'eventuale errore run time
    try{
        // aprire una connessione con il DBMS
        string stringaDiConnessione =
          "server=localhost; port=3306; database=PRA; user=root; password=root";
        MySqlConnection con = new MySqlConnection(stringaDiConnessione);
        con.Open();  // se ci sono errori di rete, di password  o di database si blocca!
        // scrivo la query da eseguire
        string query = "SELECT * FROM PROPRIETARI ORDER BY CodiceFiscale";
        MySqlCommand cmd = new MySqlCommand(query, con);
        // chiediamo al server di eseguire la query e otteniamo i dati richiesti
        MySqlDataReader reader = cmd.ExecuteReader();a
        // spazzoliamo i dati e li mettiamo in una lista
        while (reader.Read())
        {
            Proprietario p = new Proprietario();
            p.CodiceFiscale = (string) reader["CodiceFiscale"];
            p.Nome = (string) reader["Nome"];
            p.CittàResidenza = (string) reader["CittaResidenza"];
            p.AnnoPatente = (int) reader["AnnoPatente"];
            p.ListaAutomobili = null;
            elenco.Add(p);
        }
        reader.Close();
        con.Close();  // si chiude la connessione per liberare le risorse del server
    }
    catch(Exception ex)
    {…..}
    return elenco;
```

```
}
```

Prima di vedere il codice completo della classe GestioneProprietari, si accenna a come si utilizza tale classe. Si pensi ad una Console Application con il metodo Main() che effettua un inserimento di un nuovo proprietario nel database, stampa l'elenco di tutti i proprietari e poi ricerca il proprietario che ha un determinato codice:

```
class Program
{
      static void Main(string[] args)
      {   // inserimento di un nuovo proprietario
          GestioneProprietari g = new GestioneProprietari();
          Proprietario nuovo = new Proprietario { CodiceFiscale = "ABC99", Nome = "osvaldo",
                                                  CittàResidenza = "Treviso", AnnoPatente = 2000) };
          int n = g.Inserisci(nuovo);
          Console.WriteLine("inseriti " + n + " record"); // esito dell'operazione
          Console.WriteLine(g.Errore);  // eventuale messaggio di errore
          // stampa dell'elenco dei proprietari
          List<Proprietario> elenco = g.RecuperaTutti();
          Console.WriteLine(g.Errore);
          foreach (Proprietario p in elenco)
          {
          Console.WriteLine(p.CodiceFiscale + " " + p.Nome + " " + p.CittàResidenza + " " +p.AnnoPatente);
          }
          Console.WriteLine();
          // ricerca di un proprietario in base al codice fiscale
          Proprietario x = g.RecuperaPerCodiceFiscale("ZZZ18");
          if (x != null)
          {
          Console.WriteLine(x.CodiceFiscale + " " + x.Nome + " " + x.CittàResidenza + " " + x.AnnoPatente);
          }
          Console.WriteLine(g.Errore);
          Console.ReadKey();
      }
   }
```

Per quanto riguarda la stringa di connessione al database, conviene che essa sia posta in un file di configurazione dell'applicazione in modo che essa possa essere agevolmente cambiata senza dover mettere le mani sul programma applicativo. Infatti, dopo aver sviluppato e testato il programma applicativo su una macchina di sviluppo corredata di server DBMS, il programma dovrà essere installato in altre macchine e dovrà interagire con un server DBMS collocato opportunamente su una apposito server aziendale.

Nella cartella dell'applicazione si crea il file XML di nome **App.config** con il seguente contenuto:

```
<?xml version="1.0" encoding="utf-8" ?>

<configuration>

 <appSettings>

 <add key="StringaDiConnessione" value="server=localhost;
port=3306; database=azienda; user=root; password=root" />

 </appSettings>

</configuration>
```

Dopo aver aggiunto al progetto il .Net Connector per MySQL che consiste nel file **MySQL.Data.DLL** si può scrivere il codice della classe GestioneProprietari:

```
using System.Collections.Generic;
using System.Configuration;
using MySql.Data;
using MySql.Data.MySqlClient;

class GestioneProprietari
{
    public string Errore { get; set; }
    public string StringaDiConnessione { get; set; }

    public GestioneClienti()
    {   Errore = "";
     // leggo la stringa di connessione dal file App.config
    StringaDiConnessione = ConfigurationSettings.AppSettings["StringaDiConnessione"];
    }
```

Osserva
"Only one SqlDataReader per associated SqlConnection may be open at a time, and any attempt to open another will fail until the first one is closed."
In other words: you will not be able to have multiple open data-readers on one connection.

Notare che quando è aperto un oggetto DataReader, l'oggetto Connection viene utilizzato esclusivamente da quell'oggetto DataReader.
Non sarà possibile eseguire alcun comando per l'oggetto Connection, né creare un altro DataReader fino a quando il DataReader originale non viene chiuso.

```
    public List<Proprietario> RecuperaTutti()
    {   // inizializzo la lista da ritornare al chiamante
       List<Proprietario> elenco = new List<Proprietario>();
       // si deve prevedere il verificarsi di errori run time
       try
       {   // il costrutto using consente di chiudere automaticamente l'oggetto creato
```

```
        using (MySqlConnection con = new MySqlConnection(StringaDiConnessione))
        {
           con.Open();
           string query = "SELECT * FROM proprietari";
           MySqlCommand cmd = new MySqlCommand(query, con);
           MySqlDataReader reader = cmd.ExecuteReader();
        // Il datareader è un oggetto che rappresenta la tabella dei dati ricevuti dal DBMS
        // dove si è posizionati su un record (il record corrente)
        // all'inizio si è posizionati prima del primo record
        // il metodo Read ritorna vero se c'è ancora un record da leggere e come effetto collaterale
        // avanza di un record il puntatore al record corrente
           while (reader.Read())
           {
              Proprietario c = new Proprietario();
              // è necessario un casting esplicito perché il reader fornisce Object
              c.CodiceFiscale = (string) reader["CodiceFiscale"];
              c.Nome = (string) reader["nome"];
              c.CittàResidenza = (string) reader["CittàResidenza"];
              c.AnnoPatente = (int) reader["AnnoPatente"];
              elenco.Add(c);
           }
           reader.Close();
        }
     }
     catch (Exception ex)
     { Errore = ex.Message; }
     return elenco;
  }


  public Proprietario RecuperaPerCodiceFiscale(string  unCodiceFiscale)
  {
     Proprietario c = null;
     try
     {
        using (MySqlConnection con = new MySqlConnection(StringaDiConnessione))
        {
           con.Open();
          // per evitare forme di attacco di tipo SQL INJECTION, si predispone una query
          //  parametrica (non una query ottenuta come concatenazione di stringhe!!!)
           string query = "SELECT * FROM Proprietari WHERE CodiceFiscale = @codice";
           MySqlCommand cmd = new MySqlCommand(query, con);
           cmd.Parameters.AddWithValue("@codice", unCodiceFiscale);
           MySqlDataReader reader = cmd.ExecuteReader();
            // nessun ciclo perchè al Massimo c'è un solo proprietario con quel codice
           if (reader.Read())
           {
              c = new Proprietario();
              c.CodiceFiscale = (string) reader["CodiceFiscale"];
              c.Nome = (string) reader["nome"];
              c.CittàResidenza = (string) reader["CittàResidenza"];
              c.AnnoPatente = (int) reader["AnnoPatente"];
```

```
                }
            reader.Close();
          }
        }
        catch (Exception ex)
        { Errore = ex.Message; }
        return c;
    }


     public int Aggiungi(Proprietario unProprietario)
     {
        int n = 0;  // predispongo il valore di ritorno
        try
        {
           using (MySqlConnection con = new MySqlConnection(StringaDiConnessione))
           {
              con.Open();
              string query = "INSERT INTO Proprietari(CodiceFiscale, Nome, CittàResidenza, AnnoPatente)
                              VALUES (@codice, @nome, @citta, @anno)";
              MySqlCommand cmd = new MySqlCommand(query, con);
              cmd.Parameters.AddWithValue("@codice", unProprietario.CodiceFiscale);
              cmd.Parameters.AddWithValue("@nome", unProprietario.Nome);
              cmd.Parameters.AddWithValue("@citta", unProprietario.CittàResidenza);
              cmd.Parameters.AddWithValue("@anno", unProprietario.AnnoPatente);
              n = cmd.ExecuteNonQuery();  // il DBMS restituisce il numero di record inseriti (0 o 1)
           }
        }
        catch (Exception ex)
        { Errore = ex.Message; }
        return n;
     }
}
```

LAZY o EAGER?

Se si vuole recuperare dal database l'elenco dei proprietari, ciascuno con l'elenco delle proprie automobili, si possono seguire due diverse strade:

1. metodo "pigro" (**lazy**) che inizialmente effettua una query per recuperare l'elenco dei proprietari (con un metodo CRUD nella classe GestioneProprietari) e poi, quando l'utente dell'applicazione clicca sul nome del proprietario, si effettua una seconda query per recuperare l'elenco delle sue automobili (utilizzando un apposito metodo CRUD situato nella classe GestioneAutomobili)

2. metodo "voglioso" (**eager**) che recupera da subito, con un'unica query, l'elenco di tutti i proprietari con le loro automobili (basta un unico metodo CRUD nella classe GestioneProprietari)

Il metodo pigro ha il vantaggio di caricare in memoria solo i dati via via richiesti, ma facendo molteplici query costringe il DBMS a lavorare molto in modo poco efficiente.

Il metodo voglioso ha il vantaggio di effettuare un'unica query molto efficiente che porta in memoria molti dati, alcuni dei quali potrebbero non essere utilizzati.

In questo esempio vediamo un metodo CRUD voglioso, che utilizza il JOIN tra le tabelle Proprietari e Automobili per recuperare tutte i proprietari con le rispettive automobili possedute:

```
public List<Proprietario> RecuperaTuttiConAutomobili()
{
   List<Proprietario> elenco = new List<Proprietario>();
   try
   {
      using (MySqlConnection con = new MySqlConnection(StringaDiConnessione))
      {
         con.Open();
         string query = "SELECT * FROM Proprietari INNER JOIN Automobili
                              ON Proprietari.CodiceFiscale = Automobili.CodiceProprietario
                               ORDER BY Proprietari.CodiceFiscale, Autombili.Targa";
         MySqlCommand cmd = new MySqlCommand(query, con);
         MySqlDataReader reader = cmd.ExecuteReader();
         // uso la tecnica classica della "rottura di codice"
         // per capire quando si inizia con un altro proprietario
        Proprietario p = new Proprietario();
        p.CodiceFiscale = null;
         while (reader.Read())
         {   // ad ogni iterazione sul datareader incontro una nuova automobile con il suo
             // proprietario, che potrebbe essere lo stesso dell'auto precedente oppure no
            string CodiceFiscaleCorrente = (string) reader["Proprietari.CodiceFiscale"];
            if (! CodiceFiscaleCorrente.Equals(p.CodiceFiscale))
            {  // rottura di codice ➔ si tratta di un  proprietario diverso dal precedente
               If (p.CodiceFiscale != null)   {  elenco.Add(p);  }
              p  = new Proprietario();
              p.CodiceFiscale = CodiceFiscaleCorrente;
              p.Nome = (string) reader["Proprietari.Nome"];
              p.CittàResidenza = (string) reader["Proprietari.CittàResidenza"];
              p.AnnoPatente = (int) reader["Proprietari.AnnoPatente"];
              p.ListaAutomobili = new List<Automobile>();
            }
            Automobile a = new Automobile();
            a.Targa = (string) reader["Automobili.Targa"];
            a.Modello = (string) reader["Automobili.Modello"];
            a.Cilindrata = (string) reader["Automobili.Cilindrata"];
            a.CodiceProprietario = (string) reader["Automobili.CodiceProprietario"];
            a.Proprietario = p;
            p.ListaAutomobili.Add( a );
         }
       // non dimenticarsi di caricare nell'elenco anche l'ultimo proprietario
       if (p.CodiceFiscale != null)   {  elenco.Add(p);  }
       reader.Close();
      }
    }
   catch (Exception ex)
```

```
        { Errore = ex.Message; }
      return elenco;
}
```

Esempio di utilizzo di questo metodo:

```
GestioneProprietari g  = new GestioneProprietari();
List<Proprietario> elenco Proprietari = g.RecuperaTuttiConAutomobili();
foreach (Proprietario p in elencoProprietari)
{   Console.WriteLine(p.CodiceFiscale + " " + p.Nome +  " " + p.CittàResidenza + " " + p.AnnoPatente);
    Console.WriteLine("possiede le seguenti automobili: ");
    foreach(Automobile a in p.ListaAutomobili)
      {
          Console.WriteLine( a.Targa + " " + a.Modello + " " + a.Cilindrata);
      }
   Console.WriteLine();
}
```

La modalità pigra avrebbe richiesto le seguenti istruzioni, **molto meno efficienti** (si può stimare, per 1000 proprietari, 1000 volte il tempo richiesto dalla precedente, dovendo fare 1000 connessioni al database anziché una sola):

```
GestioneProprietari g  = new GestioneProprietari();
GestioneAutomobili h = new GestioneAutomobili();
List<Proprietario> elenco Proprietari = g.RecuperaTutti();
foreach (Proprietario p in elencoProprietari)
{   Console.WriteLine(p.CodiceFiscale + " " + p.Nome +  " " + p.CittàResidenza + " " + p.AnnoPatente);
    Console.WriteLine("possiede le seguenti automobili: ");
    List<Automobile> lista = h.RecuperaPerCodiceProprietario(unCodiceProprietario);
     foreach(Automobile a in lista)
    {
          Console.WriteLine( a.Targa + " " + a.Modello + " " + a.Cilindrata);
    }
   Console.WriteLine();
}
```

APPROFONDIMENTO: Pur utilizzando un metodo LAZY, un notevole miglioramento dei tempi di accesso al database si otterrebbe utilizzando **un'unica connessione al database** sempre attiva (prelevata da un "connection pool") su cui eseguire 1000 volte la STESSA QUERY PRECOMPILATA ("prepared query")

Notare che con il metodo voglioso si possono caricare in RAM anche ampie porzioni di un database, costituendo una memorizzazione CACHE del database. Si deve stare però molto attenti che i dati presenti in RAM costituiscono la "fotografia" dei dati del database valida nell'istante in cui essa è stata fatta. In una situazione di accesso concorrente tramite rete al server DBMS da parte di molteplici utenti, tale CACHE del database è da ritenersi presto superata e non più affidabile. Pertanto si dovrà interrogare nuovamente il DBMS per avere dati aggiornati più attendibili.

La gestione di questa forma di CACHE potrebbe però essere vista come uno strumento per migliorare l'efficienza dell'accesso al database. I software ORM, tra le altre cose, si occupano tipicamente di gestire in modo affidabile una CACHE dei dati del database, con l'accortezza di marcarla come "non valida" non appena qualche utente effettua un qualche aggiornamento ai dati del database (questo lavoro in effetti risulta piuttosto gravoso per l'ORM, e non tutti trovano conveniente farlo!).

Un altro esempio di metodo CRUD voglioso, che utilizza il JOIN tra le tabelle Proprietari e Automobili per recuperare una automobile assieme ai dati del suo proprietario.

```
public Automobile RecuperaAutomobileConProprietario(string Targa)
{
   Automobile auto = null;
   try
   {
      using (MySqlConnection con = new MySqlConnection(StringaDiConnessione))
      {
         con.Open();
         string query = "SELECT * FROM Proprietari INNER JOIN Automobili
                           ON Proprietari.CodiceFiscale = Automobili.CodiceProprietario
                        WHERE Automobili.Targa";
         MySqlCommand cmd = new MySqlCommand(query, con);
         MySqlDataReader reader = cmd.ExecuteReader();
         auto  = new Automobile();
         if (reader.Read())
         {
              p  = new Proprietario();
              p.CodiceFiscale =  (string) reader[“Proprietari.CodiceFiscale”];
              p.Nome = (string) reader["Proprietari.Nome"];
              p.CittàResidenza = (string) reader["Proprietari.CittàResidenza"];
              p.AnnoPatente = (int) reader["Proprietari.AnnoPatente"];
              auto.Targa = (string) reader[“Automobili.Targa”];
              auto.Modello = (string) reader[“Automobili.Modello”];
              auto.Cilindrata = (string) reader[“Automobili.Cilindrata”];
              auto.CodiceProprietario = (string) reader[“Automobili.CodiceProprietario”];
              auto.Proprietario = p;
         }
       reader.Close();
```

```
        }
      }
      catch (Exception ex)
        { Errore = ex.Message; }
      return elenco;
}
```

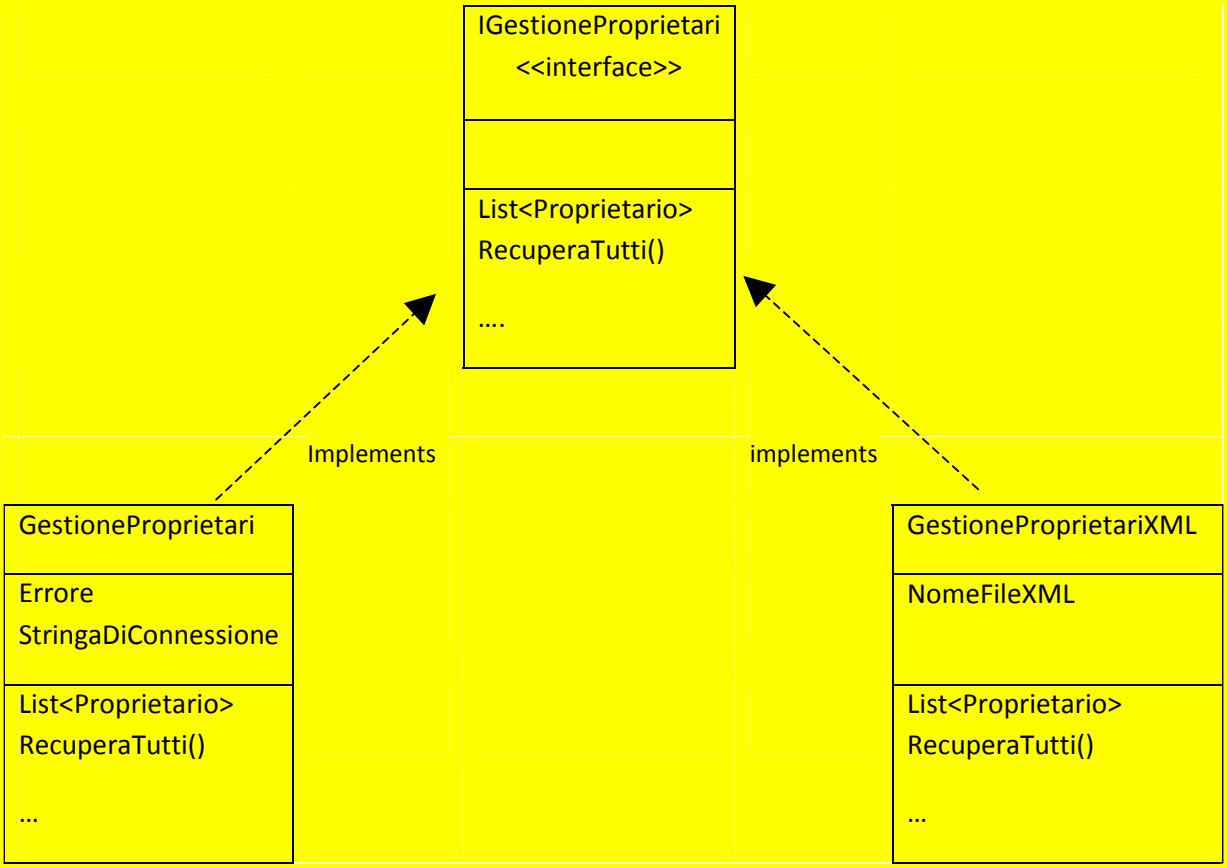
un esempio di utilizzo del suddetto metodo:

```
GestioneProprietari g  = new GestioneProprietari();
Automobile a = g.RecuperaAutombileConProprietario("TV123456");
Proprietario p = a.Proprietario;
Console.WriteLine("dati dell'automobile:");
Console.WriteLine(a.Targa + " " + a.Modello + " " + a.Cilindrata);
Console.WriteLine("dati del proprietario: ");
Console.WriteLine(p.CodiceFiscale + " " + p.Nome +  " " + p.CittàResidenza + " " + p.AnnoPatente);
```

APPROFONDIMENTO: GESTIRE DIVERSE FONTI DATI

Per consentire una gestione più flessibile della fonte dei dati, conviene definire delle interfacce (classi astratte senza attributi e con solo metodi astratti) come IGestioneProprietari e IGestioneAutomobili con i metodi CRUD astratti.

Si possono poi definire le classi concrete GestioneProprietari e GestioneAutomobili che implementano tali metodi utilizzando un DBMS come fonte dati, e anche altre classi concrete come GestioneProprietariXML e GestioneAutomobiliXML che implementano tali metodi utilizzando come fonti dati dei file XML, e così via.



In questo modo risulta immediato passare da una fonte dati ad un'altra modificando una sola riga del programma:

```
IGestioneProprietari g = new GestioneProprietari();
// oppure
IGestioneProprietari g = new GestioneProprietariXML();
// e poi utilizzo il metodo RecuperaTutti() per ottenere l'elenco dei proprietari
List<Proprietario> elenco = g.RecuperaTutti();
```

Transazioni

Una **transazione** per un database è una "sequenza di operazioni di lettura e modifica dei dati" che deve essere trattata in modo unitario, nel senso che la sua esecuzione non deve essere interrotta e deve concludersi con un successo (COMMIT) oppure con il suo integrale annullamento (ROLLBACK).
Nel caso di accesso concorrente ai dati, una transazione deve essere "isolata" dalle altre con una politica (isolation level) "pessimistica" oppure "ottimistica".
La politica "pessimistica" garantisce che finchè una transazione sta operando sui dati, nessun altra transazione possa agire sugli stessi dati. La politica "ottimistica" consente a due transazioni di agire sugli stessi dati contemporaneamente, ma se i dati sono stati modificati dalla prima transazione mentre la seconda li stava ancora leggendo, quest'ultima viene avvisata (mediante una eccezione run time) che i dati che ha letto non sono più validi e che quindi una loro eventuale modifica non può concludersi.
La politica "pessimistica" costringe le diverse transazioni ad aspettare la terminazione del lavoro di chi in quel momento sta agendo sui dati, mentre quella "ottimistica" da' l'illusione di poter procedere in modo più spedito, salvo fallire poi!!!
Solitamente i DBMS applicano la politica ottimistica :-)

Tenere presente le seguenti regole:

- Ogni singola istruzione SQL è considerata una transazione.
- Se una istruzione SQL attiva dei TRIGGER, l'insieme dell'istruzione SQL più i TRIGGER attivati costituiscono automaticamente una transazione.
- Se si vuole realizzare una transazione costituita da 2 o più istruzioni SQL si deve inserire nel codice applicativo una apposita istruzione BeginTransaction che verrà poi terminata da Commit oppure, in caso di fallimento, da Rollback.

Un esempio classico di transazione è dato dal giroconto bancario costituito dal prelievo di denaro da un conto con conseguente versamento dello stesso in un altro conto: non è ammesso per nessun motivo che tale transazione venga interrotta a metà lavoro, lasciando il database in un pericoloso stato di inconsistenza.

In un metodo CRUD si crea un oggetto di tipo MySqlTransaction associato alla connessione, a cui vengono associate anche le varie query da eseguire. In caso di fallimento della transazione viene automaticamente avviato il Rollback.

```
try{
using (MySqlConnection con = new MySqlConnection(StringaDiConnessione))
    {
        con.Open();
        MySqlTransaction transaction = con.BeginTransaction();

        string query1 = "UPDATE ……";
        string query2 = "UPDATE ……";

         MySqlCommand cmd1 = new MySqlCommand(query1, con, transaction);
         MySqlCommand cmd2 = new MySqlCommand(query2, con, transaction);

        cmd1.ExecuteNonQuery();
        cmd2.ExecuteNonQuery();

        transaction.Commit();
```

```
      }
  }
  catch (Exception ex)
  { Errore =  ex.Message; }
```

che equivale al seguente codice, dove viene esplicitamente richiesto il Rollback in caso di fallimento della transazione:

```
try{
using (MySqlConnection con = new MySqlConnection(StringaDiConnessione))
    {
        con.Open();
        MySqlTransaction transaction = con.BeginTransaction();

     string query1 = "UPDATE ……";
     string query2 = "UPDATE ……";

      MySqlCommand cmd1 = new MySqlCommand(query1, con, transaction);
      MySqlCommand cmd2 = new MySqlCommand(query2, con, transaction);

     try{
        cmd1.ExecuteNonQuery();
        cmd2.ExecuteNonQuery();

        transaction.Commit();
     }
     catch(Exception ex2)
     {  transaction.Rollback(); }
    }
  }
  catch (Exception ex)
  { Errore =  ex.Message; }
```

Non tutti i DBMS supportano le transazioni. Ad esempio MySQL supporta le transazioni solo se si lavora con tabelle **InnoDB**, mentre non le supporta per le tabelle **MyISAM.**

PHP e la classe PDO

Dalla versione 5 del linguaggio PHP è stata introdotta la classe PDO (PHP Database Object) che consente di gestire in modo uniforme l'accesso ai diversi DBMS, nello stesso modo in cui già da tempo si poteva fare in Java e in C#.
Prima della classe PDO, i diversi DBMS prevedevano istruzioni PHP completamente diverse per essere utilizzati, facendo perdere di flessibilità al codice prodotto: ad esempio nel caso di passaggio da SQLite a MySQL si doveva riscrivere gran parte del codice dell'applicazione. Ora con la classe PDO questo non avviene più!!!
Lavorando ad oggetti, in PHP si può lavorare in modo pressochè identico a come si lavorava in Java o C#: si creano le classi DTO e le classi DAO con i metodi CRUD.
Ad esempio la classe Proprietario in PHP, memorizzata nel file Proprietario.php:

```
<?php
class Proprietario
{
    public $CodiceFiscale;
    public $Nome;
    public $CittàResidenza;
    public $AnnoPatente;
}
?>
```

e la classe GestioneProprietari, memorizzata nel file GestioneProprietari.php, che utilizza come file di configurazione config.php:

```
// config.php
<?php
// definiamo le costanti per l'accesso al database di MySQL
define("SERVER", "localhost");
define("DBNAME", "azienda");
define("USER", "root");
define("PASSWORD", "");
?>
```

```
<?php
include_once("Proprietario.php");
include_once("config.php");

class GestioneProprietari
{
    // attributo per la stringa di errore
        public $errore;

   // costruttore
        public function __construct()
        { $errore = ""; }

   // metodi CRUD per l'accesso al database
```

```
// metodo che ritorna un array
 public function recupera_tutti()
 {
    $array = array();
try{
        $con = "mysql:host=".SERVER.";dbname=".DBNAME;
        // usa la classe PDO per connettersi al database
        $db = new PDO($con, USER, PASSWORD);
        // solleva una eccezione in caso di errore (per default non fa niente!)
        $db->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
        // scrivo la query
        $sql = "SELECT * FROM Proprietari ORDER BY CodiceFiscale";
        // predispongo l'oggetto che invierà la query al DBMS
        $pdostatement = $db->prepare($sql);
        // eseguo la query
        $pdostatement->execute();
        // l'oggetto PDO è molto potente!!!!
        // e riempie l'array di proprietari con una sola istruzione
        $array = $pdostatement->fetchAll(PDO::FETCH_CLASS, "Proprietario");
        // chiudo la connessione con il database
        $db = null;
        }
        catch(Exception $e)
        {
            // catturo gli errori di connessione e di accesso al database
            $this->errore = $e->getMessage();
        }
    return $array;
}
```

In alternativa potrei lavorare in stile C# spazzolando il result set, senza bisogno di casting vari!

```
....
$pdostatement->execute();
$res = $pdostatement->fetchAll();
foreach($res as $row)
{
   $p = new Proprietario();
  $p->CodiceFiscale = $row["CodiceFiscale"];
   $p->Nome = $row["nome"];
   $p->CittàResidenza = $row["CittàResidenza"];
  $p->AnnoPatente = $row["AnnoPatente"]);
   $array[] = $p;
}
```

```
// metodo che ritorna un oggetto oppure false se non c'è il record cercato
 public function recupera_proprietario_per_codice($codice)
 {
   $proprietario = false;
try{
```

```
            $con = "mysql:host=".SERVER.";dbname=".DBNAME;
            $db = new PDO($con, USER, PASSWORD);
            // solleva una eccezione in caso di errore (per default non fa niente!)
            $db->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
           // query parametrica per evitare attacchi stile SQL INJECTION
            $sql = "SELECT * FROM Proprietari WHERE CodiceFiscale = ?";
            $pdostatement = $db->prepare($sql);
            $pdostatement->bindParam(1, $codice);
            $pdostatement->execute();
            // se non c'è nessun proprietario con il codice richiesto ritorna FALSE
              $cliente = $pdostatement->fetchObject("Proprietario");
            // chiudo la connessione con il database
            $db = null;
           }
           catch(Exception $e)
           {
              $this->errore = $e->getMessage();
           }
            return $proprietario;
           }


          // metodo che ritorna un numero intero
           public function inserisci($proprietario)
           {
             $n = 0; // numero di record effettivamente inseriti
          try{
            $con = "mysql:host=".SERVER.";dbname=".DBNAME;
            $db = new PDO($con, USER, PASSWORD);
            // solleva una eccezione in caso di errore (per default non fa niente!)
            $db->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
          // query parametrica per evitare rischi di SQL INJECTION
            $sql = "INSERT INTO Proprietari(CodiceFiscale, Nome, CittàResidenza, AnnoPatente)
                  VALUES(?, ?, ?, ?)";
            $pdostatement = $db->prepare($sql);
            $pdostatement->bindParam(1, $proprietario->CodiceFiscale);
            $pdostatement->bindParam(2, $proprietario->Nome);
            $pdostatement->bindParam(3, $cproprietario->CittàResidenza);
            $pdostatement->bindParam(4, $proprietario->AnnoPatente);
            $pdostatement->execute();
            $n = $pdostatement->rowCount();  // sarà 1
            // chiudo la connessione con il database
            $db = null;
           }
           catch(PDOException $e)
           {
              $this->errore = $e->getMessage();
           }
              return $n;
           }
}
?>
```

Nel caso di transazioni si scrive qualcosa come

```
        try{
            $con = "mysql:host=".SERVER.";dbname=".DBNAME;
            $db = new PDO($con, USER, PASSWORD);
            // inizia la transazione
             $db->beginTransaction();

            $sql1 = "UPDATE ….";
            $sql2 = "UPDATE ….";

            $pdostatement1 = $db->prepare($sql1);
            $pdostatement2 = $db->prepare($sql2);

        try{
            $pdostatement1->execute();
            $pdostatement2->execute();
           $db->commit();
           }
        catch (Exception $e2)
        {    $db->rollBack();     }
       // chiudo la connessione con il database
      $db = null;
}
catch(Exception $e)
{
    // catturo gli errori di connessione con il database
   $this->errore = $e->getMessage();
}
```

In definitiva, con PHP 5 la programmazione dei database è pressochè identica a quella che si effettua con gli altri linguaggi di programmazione come Java e C#.

METODO NUOVO (ORM)

L'uso di un software ORM consente di semplificare notevolmente il lavoro dello sviluppatore di software perchè non si dovranno più scrivere i metodi CRUD e addirittura le classi DTO vengono create in modo pressochè automatico da tale software.
Si tratta di "framework" ovvero di ambienti di lavoro molto potenti ma anche piuttosto vincolanti che costringono lo sviluppatore a seguire le loro convenzioni e il loro metodo di lavoro.
I benefici sono tuttavia notevoli e pertanto si vedrà in futuro un sempre maggiore ricorso a questi strumenti software.
Il framework che ha fatto scuola è NHibernate e solo recentemente Microsoft ha corredato Visual Studio dell'Entity Framework che con il rapido succedersi delle versioni ha via via ha colmato il gap iniziale con il suddetto software. Attualmente ci sono NHibernate 3 e Entity Framework 6.
Entity Framework è costituito dal file System.Data.Entity.dll (fino alla versione 5) oppure EntityFramework.dll per la versione 6.

Con Entity Framework viene creato automaticamente un Entity Data Model (EDM) associato al database di interesse (Entity Framework 6 lavora solo con SqlServer). Questo EDM è costituito da una serie di classi DTO che rappresentano le singole tabelle del database corredate anche delle Navigation Property (come già discusso nella prima parte di questo lavoro). Si tratta di classi POCO (Plain Old CLR Object) ovvero di semplici classi "normali" del tutto ignare dell'esistenza del database e del problema della persistenza degli oggetti (persistance ignorance).
Viene inoltre creata una classe speciale che rappresenta il contesto (di tipo DBContext) a cui sono associate delle classi che rappresentano le diverse collezioni di oggetti estratti dalle tabelle del database (si tratta di classi di tipo DBSet)
Ad esempio, nel contesto del database PRA ci saranno due DBSet di nome Proprietari e Automobili che conterranno, rispettivamente, una collezione di oggetti di tipo Proprietario e di tipo Automobile.
Per interrogare il database si utilizzano le funzioni di LINQ to Entities.
Ad esempio, per consultare l'elenco di tutti i proprietari basta scrivere una sola istruzione:

```
using (context = new PRAContext())
{
   List<Proprietario> elenco = context.Proprietari.ToList();
}
```

senza necessità di scrivere alcun metodo CRUD !!!!

Esiste anche una query syntax (che tenta di imitare la sintassi di SQL) che consente di scrivere:

```
List<Proprietario> elenco = from p in context.Proprietari
                            select p;
```

Per avere i dati in ordine di Nome: `var elenco = context.Proprietari.OrderBy(x =>x.Nome).ToList();`

Per avere i dati filtrati in base all'anno di patente: `var elenco = context.Proprietari.Where(x=>x.AnnoPatente == 2000).OrderBy(x=>x.Nome).ToList();`

che nella query syntax si scrive

```
var elenco = from p in context.Proprietari
             where p.AnnoPatente == 2000
             orderby p.Nome
             select p;
```

L'accesso al database viene effettuato in modo "pigro", ovvero solo al momento della necessità:

```
Proprietario  p = context.Proprietari.Find("ABC77");  // ricerca basata sulla chiave primaria
List<Automobili> lista = p.Automobili.ToList();
```

Tuttavia, per maggiore efficienza, può convenire un accesso "voglioso" che viene espresso con la funzione Include() come nel seguente esempio:

```
Proprietario  p = context.Proprietari.Include(x=>x.Automobili).Find("ABC77");
List<Automobili> lista = p.Automobili.ToList();
```

Per aggiornare i dati si utilizzano le funzioni Add() e Remove().
La modifica avviene sui dati in memoria RAM e viene salvata nel database con la funzione SaveChanges() che agisce con un "lock ottimistico".

Esempio di inserimento di un nuovo proprietario, con una nuova auto

```
Proprietario p = new Proprietario();
p.CodiceFiscale = "ZZZ33"; p.Nome = "ugo"; p.CittàResidenza="Verona"; p.AnnoPatente= 2013;
Automobile a = new Automobile();
a.Targa = "1234"; a.Modello = "Fiat Panda"; a.Cilindrata = 900;
p.Automobili.Add(a);
context.Proprietari.Add(p);
try{ context.SaveChanges(); }
catch(Exception ex){ ……. }
```

Esempio di modifica della città di residenza di ugo:

```
Proprietario p = context.Proprietari.Find("ZZZ33");
p.CittàResidenza = "Treviso";
try{
context.SaveChanges();
}
catch(Exception ex) {……}
```

Esempio di cancellazione dell'auto 1234:

```
Automobile a = context.Automobili.Find("1234");
context.Automobili.Remove(a);
try{
context.SaveChanges();
}
catch(Exception ex) {……}
```

un modo più efficiente di cancellare l'auto 1234 crea un oggetto fittizio per evitare di effettuare una ricerca nel database:

```
Automobile a = new Automobile(); a.Targa = "1234";
context.Automobili.Attach(a);  // aggiunge l'oggetto fittizio al contesto
context.Automobili.Remove(a);
try{
context.SaveChanges();
}
catch(Exception ex) {……}
```

APPROFONDIMENTO
Finora si è ragionato secondo la modalità di lavoro Model First, ovvero dapprima si crea il database e poi le classi per accedervi. Dalla versione 4 di Entity Framework è possibile anche agire al contrario, ovvero secondo la modalità Code First che consiste nel definire innanzitutto le classi che rappresentano gli oggetti dell'applicazione e successivamente creare il corrispondente database per gestire la persistenza di tali dati.

Riferimenti

G. Mecca – Materiale didattico del corso Programmazione del DBMS – Univ. Basilicata - http://www.db.unibas.it/users/mecca/corsi/2004-2005/sviluppoWeb/index.html

D. Bochicchio e altri – C# 4 Guida completa per lo sviluppatore – Ed. Hoepli, 2010

E. Lecky-Thompson e altri – PHP 6 Professional - Wiley Publishing, Inc., 2009