Università degli studi di Bologna

Corso di laurea in Scienze e Tecnologie Informatiche

Progettazione e realizzazione di una soluzione ORM Multi Database con NHIbernate

Relatore: Chiar.mo Prof. Dario Maio

Laureando: Ghetti Emanuele

Indice

1. Introduzione

In un mondo dove l'utilizzo dei database va espandendosi sempre di più, le librerie di comunicazione rilasciate dalle aziende o da terze parti sono, in ambiente di programmazione, un oggetto di studio ampio e complesso.

Sin dall'inizio i metodi di comunicazione coi database da codice si limitavano all'apertura di connessioni specifiche ai database di produzione, l'esecuzione di query e la raccolta dei risultati. Nel tempo però, l'affermarsi sempre più solido del paradigma ad oggetti[1] e dei linguaggi rivolti a questo paradigma (C#, VB, Java etc…) hanno in parte rivoluzionato la comunicazione ai database.

Ciò che prima era semplicemente vista come una relazione, una tabella, inizia ad essere vista come una lista di oggetti, di cui ogni oggetto rappresenta ciò che precedentemente era una tupla[2]. Questo fa sì che le operazioni di aggiunta ed eliminazione di tuple vengano viste come aggiunta e cancellazione di oggetti, ed è su questo che si basano i nuovi sistemi di interfacciamento ai database definiti ORM[3].

Esistono vari vantaggi in questo tipo di approccio, come appresso elencato.

- Un'elevata portabilità rispetto alla tecnologia DBMS utilizzata: cambiando DBMS non devono essere riscritte le routine che implementano lo strato di persistenza; generalmente basta cambiare poche righe nella configurazione del prodotto per l'ORM utilizzato.

- Drastica riduzione della quantità di codice sorgente da redigere; l'ORM maschera dietro semplici comandi le complesse attività di creazione, prelievo, aggiornamento ed eliminazione dei dati (dette CRUD - *Create, Read, Update, Delete*). Tali attività occupano di solito una buona percentuale del tempo di stesura, testing e manutenzione complessivo. Inoltre, sono per loro natura molto ripetitive e, dunque, favoriscono la possibilità che vengano commessi errori durante la stesura del codice che le implementa.

Tutto questo si traduce naturalmente in una pulizia e correttezza maggiore all'interno del codice. Un piccolo esempio a riguardo può essere mostrato come segue *(Figura 1)*:

```
String sql = "SELECT ... FROM persons WHERE id = 10";
DbCommand cmd = new DbCommand(connection, sql);
Result res = cmd.Execute();
String name = res[0]["FIRST_NAME"];
```

Figura 1: ***Esempio di utilizzo di query***

[1] Paradigma che permette di definire oggetti software in grado di interagire gli uni con gli altri per mezzo di messaggi.
[2] In linguaggio riferito a DBMS relazionali, una "tupla" altro non è che una riga di una tabella di database.
[3] ORM è l'abbreviazione di Object Relational Mapping, cioè un mapping fra paradigma ad oggetti e DBMS Relazionali, i quali sono concettualmente molto diversi l'uno dall'altro.

Il codice in *Figura 1* mostra un esempio classico di comunicazione a un database, dove per prima cosa si specifica l'SQL da eseguire, si crea poi l'oggetto DbCommand per l'esecuzione della query, si raccoglie e si presenta il risultato.

Tutto ciò, utilizzando un sistema ORM basato su un'implementazione attraverso repository[4] può essere gestito come segue:

```
Person p = repository.GetPerson(10);
String name = p.FirstName;
```

Figura 2: *Esempio di utilizzo di query via ORM*

Ciò che prima poteva essere soggetto a numerosi errori, cioè scrittura errata della query SQL e accesso errato agli indici (ben 2 errori possibili in sole 4 linee di codice) diventa nel secondo caso un esempio molto più solido, e sebbene non esente da errori, molto più mantenibile.

Compresa l'importanza dei sistemi ORM, l'obiettivo che si pone questo progetto è la realizzazione di una soluzione multi database di comunicazione che permetta di appoggiarsi ad uno o più di questi sistemi di mapping relazionale di oggetti per la gestione di entità a livello di database.

La progettazione partirà dunque dalla formulazione delle necessità che questa libreria dovrà soddisfare, da cui seguirà uno studio delle soluzioni già disponibili per poi giungere alla progettazione e definizione del progetto, la cui presentazione sarà effettuata per mezzo di opportuni diagrammi di classe.

Al termine della progettazione sarà inoltre implementato un sistema di Unit Testing che permetterà ad eventuali utilizzatori futuri della libreria di avere sempre un quadro preciso del codice funzionante e del codice difettoso.

Per ogni progetto in grado di considerare una moltitudine di soluzioni già presenti, cioè, in questo caso, sia sistemi ORM, sia sistemi DBMS di database relazionali, un punto focale su cui concentrarsi è senza dubbio l'impostazione di un ambiente in grado di realizzare una statistica sulle prestazioni degli stessi, pertanto, in fase di Unit Testing, sarà anche presentato e impostato un metodo di utilizzo della libreria utilizzabile per lo scopo.

Finalizzato il progetto e la fase di test sarà infine lasciato spazio ad un'analisi sulle eventuali espansioni implementabili nella libreria, quali aggiunte, miglioramenti, ecc… e saranno tratte le conclusioni finali.

[4] Per "repository" si intende un contenitore di metadati, all'interno del quale, nel caso degli ORM, sono contenuti gli oggetti di database.

2. Obiettivi del progetto

In quanto libreria di classi, il primo obiettivo fondamentale è la portabilità. La libreria dovrà essere utilizzabile senza setup specifici sulla macchina locale. Semplicemente aggiungendo le librerie di classi opportune in riferimento al progetto su cui si sta lavorando dev'essere reso disponibile l'intero sistema in modo trasparente.

Obiettivo fondamentale è l'interoperabilità fra database. Impostati gli opportuni punti di ingresso alla libreria, il codice scritto dovrà operare nel modo più generico possibile, permettendo a codice scritto ad esempio per SqlServer di operare anche sotto Oracle con un cambiamento minimo di parametri d'esecuzione.

Come oggetto di studio, e per comodità dal punto di vista degli strumenti utilizzati, sono stati scelti come database: Oracle, MySql, Microsoft SQL Server, Ingres, DB2, SQLite e PostgreSQL, tutti i database di cui NHIbernate fornisce un Driver e un Dialect di base.

Per quanto riguarda il sistema ORM in uso, nonostante si sia deciso di affidarsi ad NHIbernate, si vuole progettare la libreria in modo che in futuro permetta l'implementazioni di altri sistemi ORM quali, ad esempio, Entity Framework.

3. Design Pattern, architettura ad oggetti e sviluppo software

Partendo dai requisiti esposti nel capitolo precedente, la progettazione della libreria scopo di questo progetto si presenta da subito non banale. È dunque opportuno sviluppare il concetto di Design Pattern, in modo da giustificare al meglio le scelte implementative effettuate.
In questo capitolo saranno esposti in un primo momento concetti generali di approccio alla scrittura di buon codice[5] fino a scendere nel dettaglio presentando i Design Pattern scelti per lo sviluppo, e cioè: Singleton Pattern, Data Mapper e Factory Method.

Per facilitare ulteriormente la stesura del codice sarà inoltre utilizzato un processo di sviluppo software noto con il nome di Test Driven Developement[6], esplicato al termine dell'esposizione dei design pattern.

3.a Design Pattern in generale

La progettazione ed implementazione di sistemi che seguono il paradigma Object Oriented non sempre è un compito banale, e in caso il software Object Oriented realizzato dovesse essere predisposto a riutilizzi in ambienti diversi, l'intera realizzazione può rivelarsi estremamente complessa, in quanto la soluzione al problema presentato dovrebbe essere solida sull'ambiente di primo utilizzo, ma anche abbastanza flessibile da permetterne una rielaborazione e reimplementazione su casi di studio futuri.

[5] Per buon codice si intende codice il più possibile comprensibile, mantenibile e riutilizzabile nel tempo.
[6] Letteralmente "Sviluppo Guidato dai Test".

Queste condizioni danno vita a una generale difficoltà per un programmatore nello scrivere codice che risponda alle esigenze funzionali alla prima stesura, e spesso l'approccio Try And Fail[7] porta alla finale scrittura di buon codice Object Oriented solo dopo una quantità di tempo che rischia di essere persino proibitivo in ambiente aziendale.

Pertanto, nel tempo, l'approccio dei programmatori più esperti si è attestato sul non risolvere i problemi partendo da zero, ma riusando soluzioni, o parti di esse sulle quali hanno lavorato in passato, utilizzando pezzi di codice già consolidati e testati.

Proprio il riutilizzo di questo tipo di elementi è l'idea alla base del Design Pattern: il semplice riutilizzo del codice, infatti, porta ad indiscutibili vantaggi sia a livello di tempo che difficoltà implementativa, e la possibilità di utilizzare pezzi di design già consolidati facilita di molto la stesura di un nuovo progetto.

Come è noto, infatti, la fase di design rappresenta una delle fasi più complesse del ciclo di sviluppo, i cui errori possono essere ridondati al punto da obbligare il team di sviluppo a riprogettare parte, o addirittura l'interezza dell'applicazione in caso di Application Fault[8], mentre d'altro canto, la stesura in se del codice, escludendo algoritmi matematici di elevata complessità, è raro che incontri complicazioni insormontabili.

Queste considerazioni, unite all'osservazione della tendenza a riutilizzare più volte soluzioni uguali e simili fra di loro in contesti diversi, ha portato all'idea di collezionare e documentare queste soluzioni a problemi ricorrenti, in modo da migliorare e rendere più solido lo sviluppo di software.

Per facilitare il riutilizzo di alcuni elementi di progettazione che sono già stati sviluppati si ricorre pertanto ai design pattern, che hanno lo scopo di dare un nome, spiegare e valutare pezzi importanti e ricorrenti nella progettazione del software Object Oriented.

I design pattern non solo aiutano a costruire software riusabile ed evitare scelte che compromettano l'utilizzo dello stesso, ma permettono di migliorare la documentazione e manutenzione dei sistemi già esistenti, aiutando dunque a progettare nel modo migliore, in minor tempo.

La prima e più famosa collezione di design pattern è contenuta nel libro di Gamma[1], più noto come libro della "Gang of 4", dove sono contenuti e documentati 23 tipologie diverse di design pattern.

3.b Design Pattern nello specifico: Singleton Pattern

Nel campo dell'ingegneria del software il Singleton Pattern è un Design Pattern che restringe l'inizializzazione di una classe ad un solo oggetto. L'utilità di questo approccio consiste nel generare un singolo oggetto comune a tutta la soluzione in grado di coordinare le azioni del sistema che si va a sviluppare.

[7] Con "Try And Fail" si intende un tipo di approccio che fin da subito punti più sulla funzionalità del codice, che non sulla sua buona stesura. Questo porterà a numerosi passaggi di refactoring per migliorare via via lo stato del codice.
[8] Per "Application Fault" si intende un caso non gestito dall'applicazione sviluppata, che porta dunque ad una rielaborazione a scopo integrativo della stessa.

Nel tempo sono state mosse numerose critiche[2] nei confronti di questo design pattern, in quanto spesso impone restrizioni specifiche in situazioni dove non è in realtà richiesta una sola istanza di un dato oggetto, oltre all'introduzione di uno stato globale[9] all'interno dell'applicazione.

L'implementazione di un Singleton può essere sviluppata in diversi modi. Il più banale è l'utilizzo di una classe statica con scope globale, in questo modo al primo accesso ad essa verrà eseguito il costruttore e saranno inizializzate le strutture che verranno interrogate dall'esterno per ottenere i parametri di connessione. Ulteriori chiamate all'oggetto utilizzeranno l'istanza creata la prima volta, senza rigenerarla nuovamente.

Nel caso dell'applicazione in oggetto, si è scelto di utilizzare questo tipo di design pattern per generare un provider di connessione comune all'applicazione. L'implementazione avverrà per mezzo di classe statica.

3.c Design Pattern nello specifico: Data Mapper

Nel paragrafo 1 si è introdotto e spiegato il concetto di ORM e del ruolo che investe nella progettazione e realizzazione di software di medie-grandi dimensioni.

In questo paragrafo viene invece descritta l'implementazione di un ORM mediante un pattern definito Data Mapper. Il Data Mapper è un livello di software implementato allo scopo di separare gli oggetti risiedenti in memoria dal database utilizzato, ha il compito di trasferire i dati da un estremo all'altro e deve inoltre garantire che i dati siano rispettivamente isolati.

Grazie a questo pattern gli oggetti in memoria (definiti oggetti di dominio) non hanno bisogno di conoscere l'implementazione del database sottostante per funzionare. In caso di modifica al DBMS sarà il Data Mapper ad assumersi il compito di aggiornare la comunicazione fra i due mondi.

Questo fa sì che gli oggetti in memoria non si interfaccino mai direttamente al database via query SQL e che la conoscenza dello schema del database non sia di loro competenza.

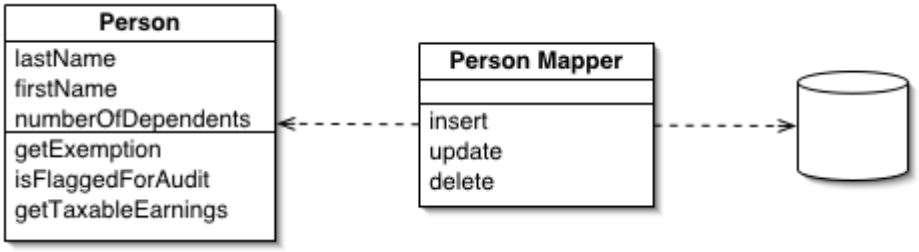


Figura 3: *Esempio di Data Mapper*

Nell'immagine in *Figura 3[3]* viene rappresentato un classico esempio di implementazione di questo design pattern.

La classe Persona, sulla destra, non conosce nulla dell'implementazione del database utilizzato dall'applicazione, ma si limita ad esporre le proprietà di un oggetto che dovrà essere rappresentato a livello di database. All'interno del Data Mapper, d'altro canto, troviamo il codice funzionale di interfacciamento al database: insert, update, e delete.

[9] Per stato globale si intende una variabile globale al'interno dell'applicazione, potenzialmente accessibile da ogni punto a meno che non venga mascherata.

3.d Design Pattern nello specifico: Factory Method

Nell'ambito di un linguaggio di programmazione ad oggetti basato sul processo di specifica di classi, il Factory Method è un pattern di sviluppo che si occupa del compito di istanziare oggetti senza che sia necessario specificarne la classe specifica.

Ciò può essere realizzando alternativamente alla chiamata al costruttore creando gli oggetti attraverso uno strumento chiamato Factory, che può essere implementato a livello di interfaccia o realizzato come override[10] nelle classi che ereditano da una specifica classe base .

Il concetto su cui si basa questo particolare Design Pattern è quello di definire un'interfaccia di creazione oggetti, ma lasciare alle classi che implementano tale interfaccia il compito di decidere quale oggetto specifico istanziare.

Il vantaggio di tutto ciò sta nel trasferire la complessità di creazione di un oggetto, che in alcuni casi può essere elevata, dall'oggetto stesso a un punto di controllo centralizzato, la cui unica occupazione sarà quella di creare l'istanza giusta di una specifica classe a runtime.

Un esempio estremamente triviale e di semplice assimilazione in quanto presenta riscontro diretto nella vita di tutti i giorni, contenuto nel libro Head First Design Patterns[5] si può trovare esaminando una generica classe "Pizzeria".

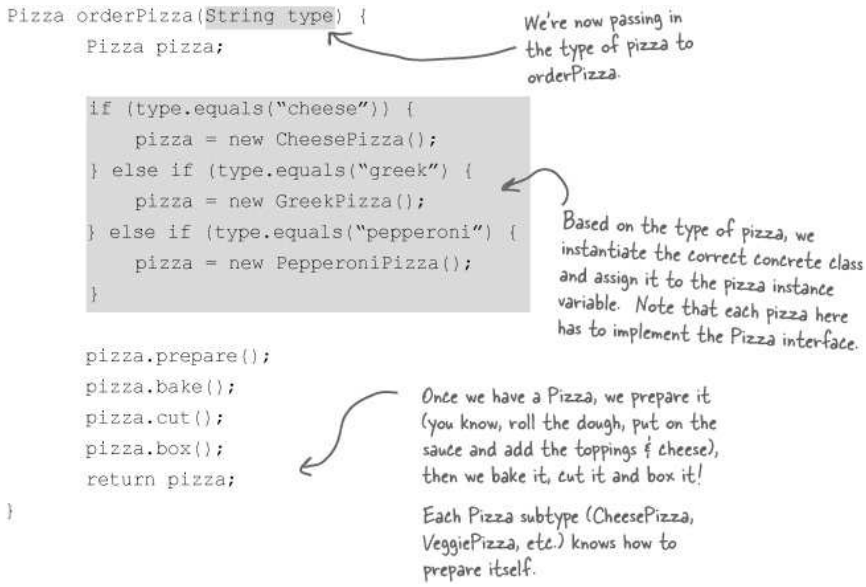


Figura 4: Creazione di una pizza in base all'ordine ricevuto

Poniamoci nella situazione di dover gestire una pizzeria con un certo listino. Gli ordini in cui è contenuto il nome della pizza ordinata arrivano e in base ad esso dev'essere prodotta la giusta tipologia di pizza. L'implementazione di una soluzione base si può notare in *Figura 4*: in base al comando ricevuto si istanzia il tipo specifico della pizza desiderata, che è poi elaborato e restituito incapsulato in un interfaccia denominata Pizza.

[10] Per "override" si intende il comportamento di sovrascrittura di un metodo definito abstract di una classe padre da parte di una classe figlia, in modo da specializzarne l'implementazione.

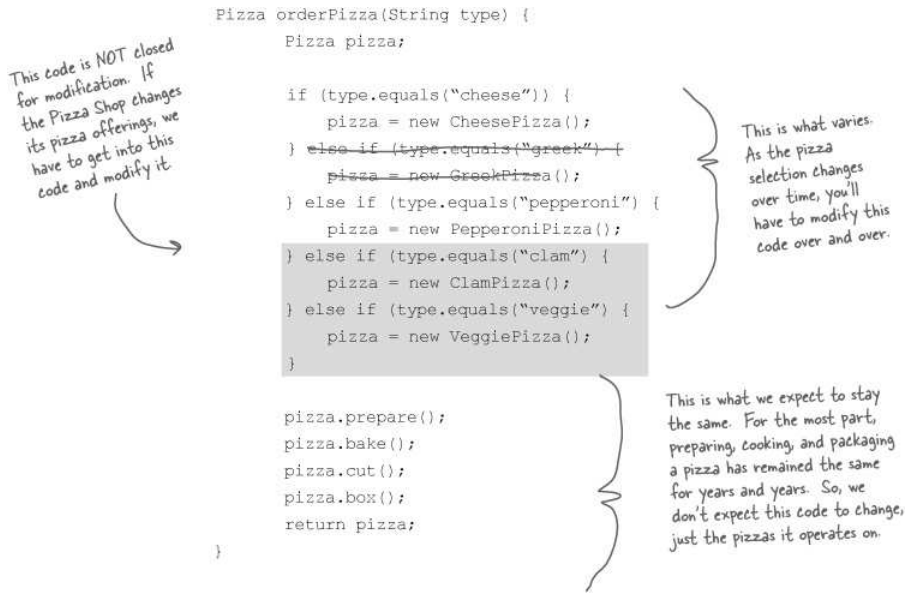


Figura 5: Risposta a cambiamento di produzione

Questo approccio chiaramente funziona, ma in caso la pizzeria decidesse di modificare il listino, si dovrebbe accedere e modificare il sistema di creazione ed elaborazione dell'entità generica Pizza, che quindi non risulta più blindato alle modifiche od esente da errori con massima sicurezza, come si può notare dalla *Figura 5.*

Ed è proprio sotto queste condizioni che inizia a diventare necessaria ed efficace la creazione di una classe che si occupi solo ed esclusivamente di generare pizza per i clienti che la richiedono. Sarà dunque possibile spostare il codice di creazione in un oggetto centralizzato (*Figura 6*) che sarà modificato in caso si decida di aggiungere o rimuovere pizze dal listino.

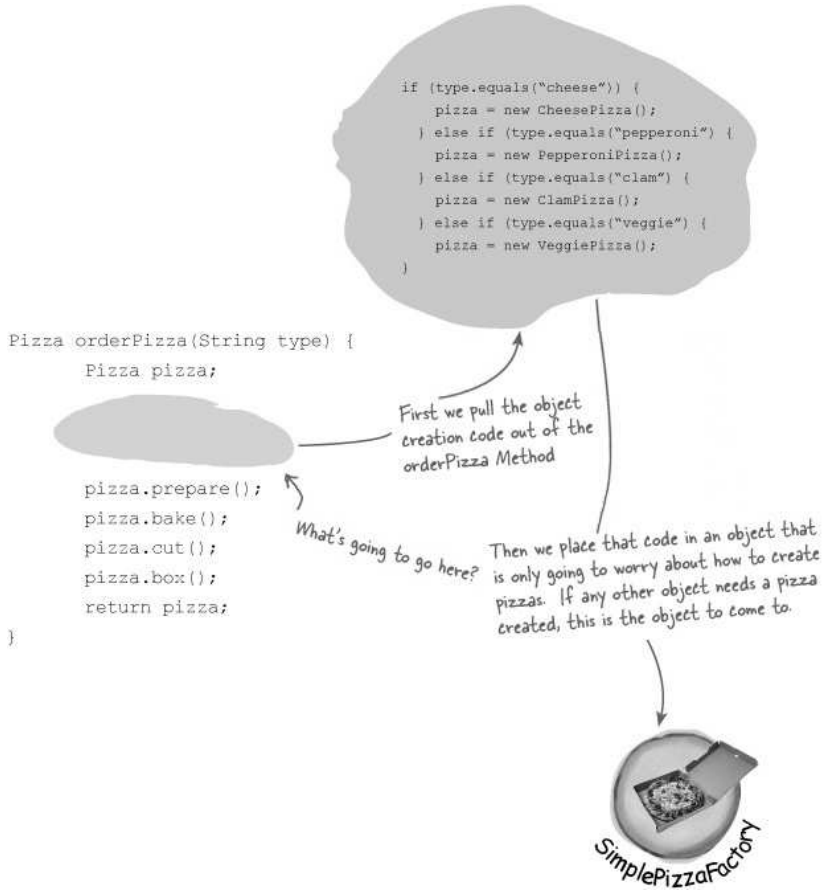


Figura 6: Estrazione del codice soggetto a modifiche

3.e Sviluppo del software: Test Driven Development

Il test driven development (TDD) è un processo di sviluppo software che si basa sull'iterazione di un ciclo di sviluppo estremamente limitato: per prima cosa il programmatore scrive un test automatizzato rappresentante una miglioria o nuova implementazione all'interno dell'applicazione, questo test in un primo momento fallirà a causa della mancanza dell'effettiva implementazione.

A questo punto il programmatore implementerà la minima quantità di codice in grado di portare a conclusione con successo il test appena scritto, per poi effettuare un refactoring del codice portandolo ad accettabili standard di qualità.

Il ciclo di sviluppo, presentato da Kent Beck nel 2003[4] dovrà essere ripetuto fino all'aggiunta di tutti i moduli in grado di far funzionare i test presenti nell'applicazione.

In questo progetto si è fatto estensivo uso di questa tecnica di software developing, specie nelle implementazioni dei sistemi di comunicazione per ogni singolo database. L'argomento verrà trattato in maniera più completa in seguito, nel paragrafo *6.h*

Di seguito, in *Figura 7* viene presentato in maniera visuale il workflow relativo a questa tipologia di sviluppo software.

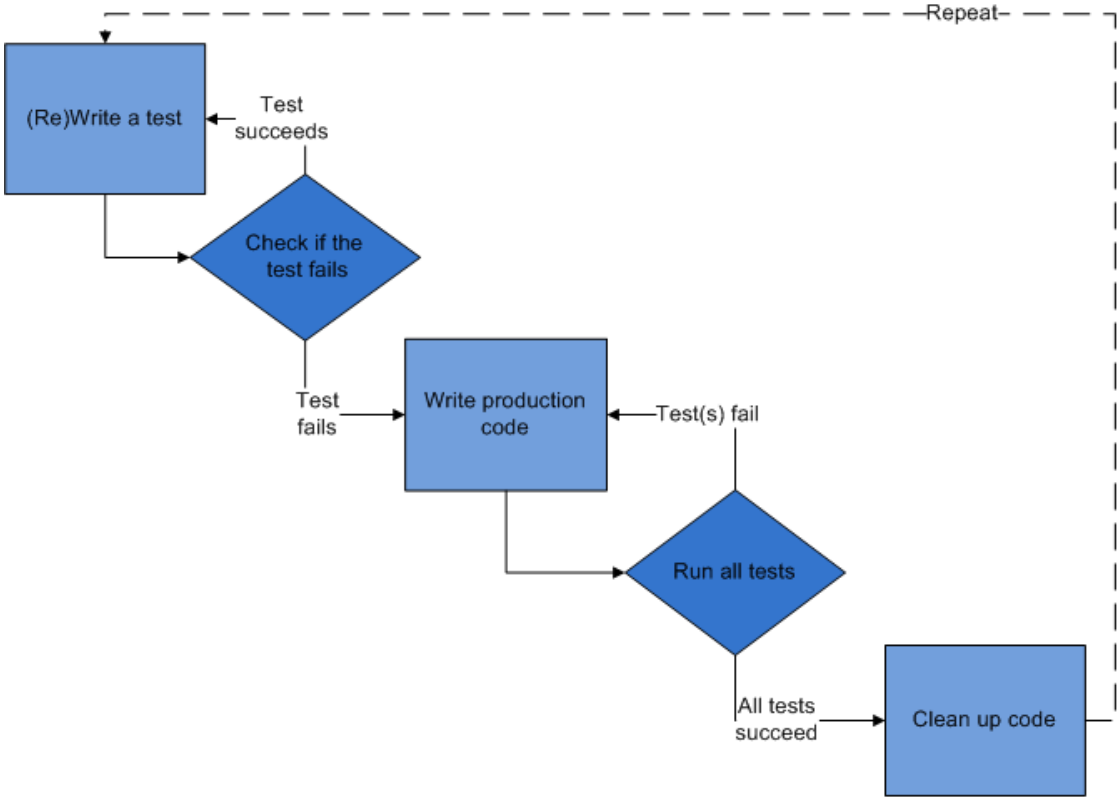


Figura 7: *Workflow di sviluppo software basato su Test Driven Development*

4. Stato dell'arte dei sistemi multi database

Attualmente le soluzioni ORM che permettano la comunicazione con più database sono molte. NHIbernate, Entity Framework, Dapper, ECO, ecc… ma per quanto possano essere generiche, devono comunque essere configurate in fase di utilizzo. Dovendo realizzare una libreria, per quanto possibile, in grado di configurarsi automaticamente in base al database utilizzato, sfruttare un ORM in maniera diretta, gestendo manualmente la configurazione, non risulta la soluzione migliore.

Un sistema di comunicazione multi database con cui configurare gli ORM utilizzati però esiste, l'ODBC[11].

Utilizzando l'ODBC, previo aver installato i driver e settato i DSN[12] di sistema, è possibile effettuare tutte le comunicazioni verso i diversi database usando lo stesso linguaggio specifico, e considerando che la maggior parte degli ORM attualmente disponibili supportano comunicazioni ODBC, l'intero sistema potrebbe essere effettivamente di banale realizzazione.

Questo però, è purtroppo vero solo in merito alla macchina su cui si esegue il codice, in quanto l'ODBC richiede che i driver e le specifiche delle sorgenti dati siano specificati nella macchina host.

Si potrebbe dunque considerare la possibilità di inserire i driver all'interno della libreria, ma dal momento che i driver ODBC non sono programmati in .NET Framework, anche questa strada non è percorribile.

Pertanto, in quanto esistano strumenti che effettuano operazioni simili a quelle desiderate, non risulta esistere una libreria che permetta una comunicazione multi database a un così alto tasso di portabilità.

[11] ODBC: Open Database Connectivity. API Standard per connessioni client-server, indipendente dal linguaggio di programmazione, sistema di database e sistema operativo.
[12] DSN: Database Source Name: rappresenta le informazioni che indicano ad un programma come connettersi ad una determinata fonte dati.

5. Studio di una soluzione

Anche nell'ambito degli ORM esistono limiti. Uno dei più marcati (del quale NHIbernate soffre molto) è l'elaborazione di relazioni che presentano una quantità di tuple estremamente elevata. In questi casi affidarsi banalmente a una soluzione ORM rischierebbe di rallentare notevolmente le prestazioni, ma d'altro canto, l'utilizzo di un ORM con una quantità limitata di oggetti permette di godere di tutti i vantaggi relativi all'utilizzo del mapping relazionale.

Pertanto, per sviluppare una soluzione completa dal punto di vista logistico, si vogliono rendere disponibili due fondamentali punti di ingresso alla libreria.

Il primo permetterà un utilizzo diretto delle query SQL, nascondendo per semplicità l'implementazione dei comandi e richiedendo solo la query da eseguire e le informazioni di connessione al database scelto, mentre il secondo punto di ingresso si interfaccerà ad NHIbernate o a qualsiasi altro ORM sia stato implementato per la comunicazione al database.

Entrambi i punti di accesso si baseranno su un DataProvider che permetterà di ottenere dati fondamentali riguardo alla connessione utilizzata in un dato momento.

Per raggiungere l'obiettivo futuro dell'implementazione di più ORM, sarà inoltre necessario un intelligente wrap[13] su NHIbernate per mezzo di opportune interfacce, tale da rendere future implementazioni semplici e incapsulate all'interno del cuore funzionale del progetto.

Per quanto riguarda l'accesso diretto ai database per mezzo di query non transazionali, i vari sistemi di comunicazione ai vari database saranno sviluppati come progetti a se stanti, contenenti un driver custom che sarà ereditato da una classe driver base poi utilizzata a livello di configurazione della connessione.

Facendo questo sarà possibile non solo aggiungere eventualmente nuovi database al progetto, ma anche implementare la libreria su uno specifico database senza dover includere nel nuovo progetto anche i riferimenti a database non utilizzati.

Ad esempio, sviluppando un'applicazione comunicante con MySql sarà necessario importare le dipendenze per la libreria principale (HSystem.Data) e per il database specifico (HSystem.Data.MySql), senza essere costretti ad avere referenze anche agli atri sette database.

Per quanto riguarda la fase di implementazione dei test, invece, sarà creato un ulteriore progetto la cui configurazione sarà gestita per mezzo di un semplice file di configurazione.

Questo perché non avendo un punto d'accesso via codice alla configurazione delle sorgenti di comunicazione, come potrebbe essere il costruttore di un'applicazione windows, l'unica possibilità è predisporre il DataProvider dell'applicazione, in ambiente Unit Test, ad ottenere le informazioni di comunicazione da file.

[13] Per "wrap" si intende l'incapsulamento delle funzionalità di una certa soluzione software all'interno di una propria implementazione.

5.a Strumenti utilizzati

La soluzione proposta è stata sviluppata utilizzando Visual Studio 2013 come IDE[14], linguaggio di programmazione C# .NET Framework 4.5. È risultata inoltre indispensabile l'installazione del plugin di Visual Studio ReSharper, il quale rende disponibile una sessione di Unit Test eseguibile in maniera embedded[15] all'interno dell'applicazione. Il Framework utilizzato per l'esecuzione dei test è NUnit. Per facilitare inoltre l'installazione delle sorgenti assembly per la soluzione corrente è stato installato il plugin NuGet.

Per verificare l'effettivo funzionamento del codice si è preparata una macchina virtuale Windows 7 x64 a due processori e 1.5GB di memoria. All'interno sono stati installati secondo la *Tabella 1* i database e i relativi strumenti di amministrazione utilizzati per interfacciarsi ad essi.

Tabella 1: *Sistemi di Database utilizzati e relativa Suite*

Database	Suite
MySql Server 5.6.20.0	**Heidi SQL**
Oracle 11g Express	**SqlDeveloper**
SQL Server 2012 Service Pack 1	**Microsoft SQL Management Studio**
Firebird 2.5.3	**TurboBird**
PostgreSQL 9.3.5	**SQL Maestro for PostgreSQL**
DB2 10.5 Express-C	**EMS SQL Manager For DB2**
Ingres 10.0	**Razor SQL**

In ogni database verrà realizzato un piccolo ambiente di test composto da due tabelle, una tabella di testata (TEST_TABLE) collegata per mezzo di foreign key a una tabella di dettaglio (TEST_DETAIL) secondo lo schema rappresentato in *Figura 8*.

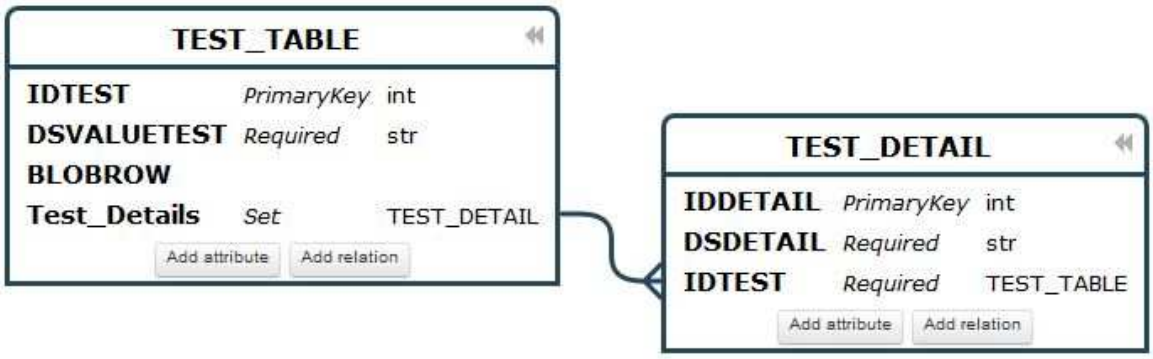


Figura 8: *Sistema relazionale implementato nei Database di riferimento*

[14] Per IDE: Integrated Development Environment, si intende un ambiente di sviluppo cha aiuti nella stesura e debug di codice sorgente per un'applicazione software.

[15] Per esecuzione "embedded" si intende l'esecuzione di codice effettuata direttamente all'interno dell'IDE Visual Studio.

5.b Note sui database e suite utilizzati

Si noti che ogni database ha una propria GUI[16] specifica tranne Ingres. In questo caso ho dovuto sfruttare una soluzione multi database (Razor SQL) opportunamente configurata per la connessione.

Per quanto riguarda SQLite, essendo un database locale salvato su singolo file, non è stato necessario installarlo in una macchina server, ma è bastato creare un database di test utilizzando l'IDE SQLite Expert ed includerlo come semplice file nel progetto.

6. Realizzazione

Si prenderanno ora in esame le varie sezioni del progetto per spiegarne l'utilizzo. Dopo aver esposto gli Enumeratori e le Interfacce utilizzate, verranno prese in considerazione le implementazioni di codice, partendo dal sistema centralizzato di accesso ai dati DbDataProvider, per poi esaminare i due punti d'ingresso dell'applicazione: l'accesso non transazionale ai database, per mezzo di query dirette DbCustom, e l'accesso ai database in maniera transazionale per mezzo di un ORM (attualmente solo NHIbernate) DbTransaction.

In seguito all'esposizione della tecnica utilizzata per la realizzazione di Unit Test sarà inoltre mostrato un caso di studio per effettuare statistica sui database utilizzati, oltre a un caso reale di utilizzo della libreria, utilizzando una piccola applicazione Windows Form in grado di eseguire semplici operazioni su tutti gli 8 database.

6.a Enumeratori

Figura 9: *Enumeratori utilizzati dalla libreria*

L'utilizzo degli enumeratori (*Figura 9*) in questo progetto è estremamente banale e si limita all'indicazione del tipo di database, selezionabile fra quelli indicati, e l'ORM utilizzato che, come accennato in precedenza, attualmente sarà solo NHIbernate, con possibilità di espansione futura.

[16] GUI: Graphic User Interface, o Interfaccia Grafica Utente. Permette all'utente di eseguire operazioni complesse su sistemi che altrimenti richiederebbero l'utilizzo della riga di comando (Quali, ad esempio, i DBMS).

6.b Interfacce

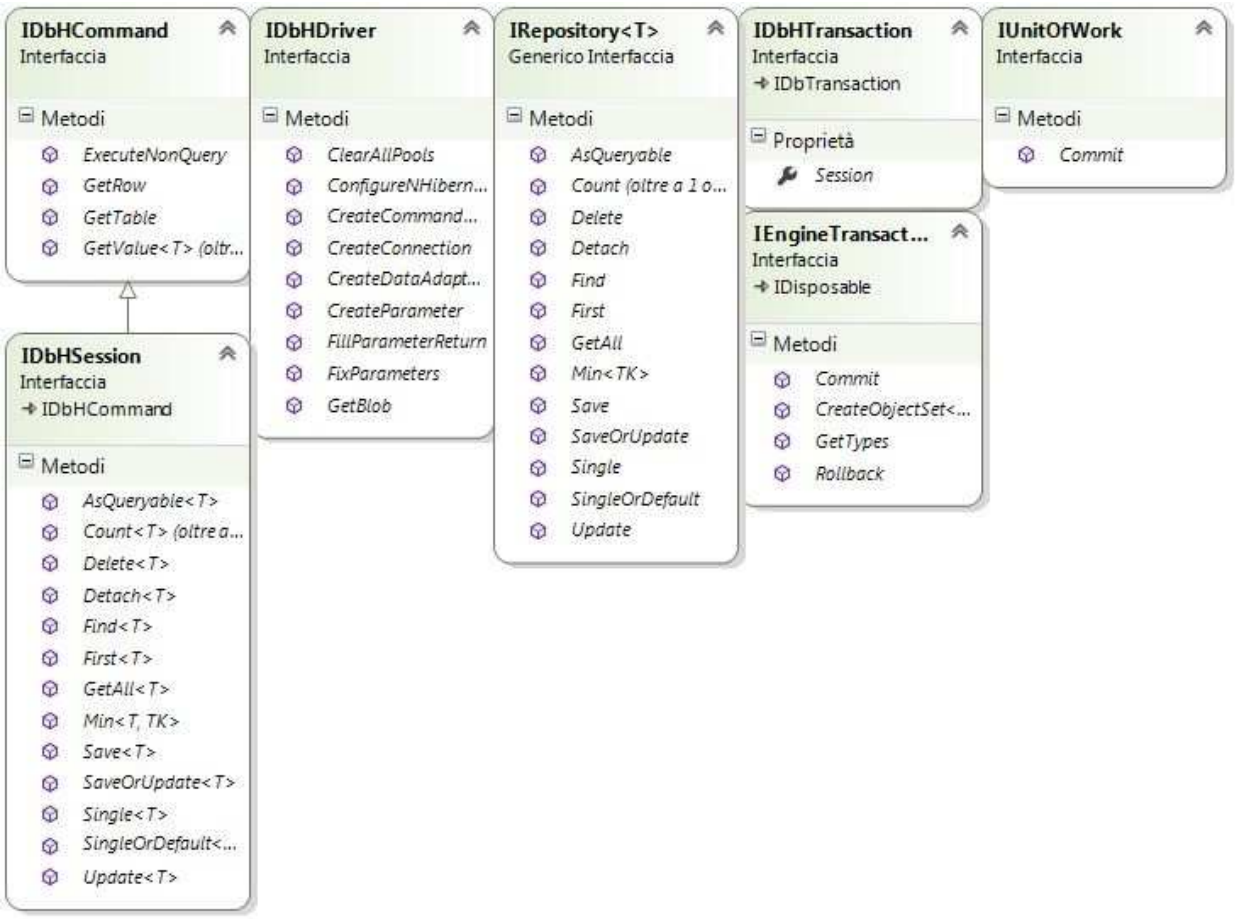


Figura 10: *Insieme delle interfacce utilizzate dalla libreria*

Il ruolo delle interfacce (*Figura 10*) di questo progetto è quello di rendere disponibile un ambiente comune dove le funzioni richiamate saranno redirezionate alle specifiche implementazioni, di ORM, o di database.

Essendo parte integrante dell'implementazione del progetto, saranno ora esaminate una per volta e ne verranno spiegati con dovizia di particolari i metodi esposti.

6.b.1 IDbHCommand

Questa interfaccia espone i metodi SQL direttamente utilizzabili all'interno di una transazione ORM. Le implementazioni delle query, per rispettare la transazionalità dell'ORM sottostante, saranno opportunamente aggiunte alla transazione attiva.

Le firme pubbliche dei metodi consistono in:

-ExecuteNonQuery: Richiede due parametri: la stringa SQL da eseguire, e la lista dei parametri con cui parametrizzare la query. L'implementazione all'interno degli ORM si baserà sull'omonima operazione ExecuteNonQuery dell'interfaccia generica IDbCommand.

Come già specificato in precedenza, essendo questa una funzione richiamabile all'interno della sessione dell'ORM, dovrà essere gestito anche l'Enlist[17] del comando nella transazione attiva.

-GetTable: Richiede due parametri: la stringa SQL da eseguire, e la lista dei parametri con cui parametrizzare la query. Le varie implementazioni specifiche utilizzeranno un generico DataAdapter per restituire un DataTable. Si tratta di una funzione "comoda" più che funzionale, in quanto la sua implementazione, per natura, non può essere transazionale, trattandosi di una SELECT.

-GetRow: Richiede due parametri: la stringa SQL da eseguire, e la lista dei parametri con cui parametrizzare la query. Si appoggia sulla funzione GetTable e si aspetta un risultato contenente una sola DataRow, la quale sarà poi restituita come risultato.

-GetValue<T>: Richiede due parametri di cui il secondo opzionale: la stringa SQL da eseguire, e la lista dei parametri con cui parametrizzare la query. Questa funzione dovrà permettere la reimplementazione dell'ExecuteScalar, e il valore restituito sarà inoltre convertito a runtime e trasformato nel tipo T[18] specificato richiamando la funzione.

6.b.2 IDbHSession

Rappresenta una sessione generica di un database. Espone metodi in grado di effettuare operazioni con il repository interno, ed espone anche i metodi della IDbHCommand, esaminata precedentemente.

Le funzioni esposte da questa interfaccia sono le seguenti:

-Find<T>: Richiede un solo parametro, e cioè un'espressione del tipo Expression<Func<T, bool>> che permette di effettuare specifiche operazioni su un'entità di tipo T che restituiscano un booleano. Dove questo booleano è true, l'oggetto di tipo T considerato sarà un valido risultato dell'espressione.

La funzione permette di ottenere una lista di entità che rispettano il predicato. Per mantenere la massima genericità, la lista restituita sarà di tipo IEnumerable.

-Single<T>: Richiede come parametro un'espressione di controllo e restituisce il singolo elemento di tipo T che la verifica. In caso più elementi verifichino l'espressione sarà lanciata un'eccezione.

-SingleOrDefault<T>: Come la funzione Single<T> restituisce l'unica entità che verifica il predicato, in caso però non ne siano presenti restituisce il valore di default per l'oggetto T. In caso T sia una classe, il valore di default restituito sarà null[19].

-First<T>: In base al predicato passato come parametro, ottiene il primo elemento di un record set che lo verifichi.

[17] L'Enlist consiste nell'inserire un certo comando SQL all'interno di una transazione precedentemente creata. In seguito a questa operazione, l'istruzione SQL in oggetto si comporterà come se fosse parte della transazione.
[18] Per tipo "T" si intende un tipo di oggetto che può essere riconosciuto dal programma solamente a runtime, anche detto "Generic".
[19] Il valore "null" identifica un oggetto dichiarato, ma ancora inesistente a livello di memoria.

-Count<T>: Di questa funzione sono presenti due overload[20]. Il primo permette di ottenere il numero di elementi di tipo T che verificano un predicato passato come parametro, mentre il secondo permette di ottenere la quantità di tutti gli oggetti all'interno del repository di tipo T.

-Min<T, TK>: Data un espressione di controllo restituisce il valore minimo di un campo di tipo TK appartenente ad un oggetto di tipo T.

-Save<T>: Richiede come parametro un'entità di tipo T e in base all'implementazione si occuperà di salvarla nel Repository transazionale dell'ORM in uso.

-SaveOrUpdate<T>: Richiede come parametro un'entità di tipo T e in base all'implementazione si occuperà di salvarla nel Repository transazionale dell'ORM in uso, in caso l'oggetto fosse già presente effettua un aggiornamento.

-Detach<T>: Richiede come parametro un'entità di tipo T e in base all'implementazione si occuperà di escluderla dalla transazione attiva. Questo significa che operazioni di Save o Update effettuate su entità staccate dalla transazione non apporteranno alcuna modifica al database.

-Delete<T>: Richiede come parametro un'entità di tipo T e in base all'implementazione si occuperà di eliminarla dal repository. Il commit di una transazione con entità eliminate dal repository porterà alla cancellazione delle tuple corrispondenti.

-Update<T>: Richiede come parametro un'entità di tipo T e in base all'implementazione si occuperà di aggiornarne i valori all'interno del repository.

-AsQueryable<T>: Restituisce tutti gli elementi di tipo T che verificano il predicato passato come parametro convertendoli ad oggetti Queryable. Utilizzando un record set di questo tipo è possibile effettuare numerose query sui dati partendo direttamente dal record set.

-GetAll<T>: Non richiede alcun parametro e restituisce un oggetto IEnumerable[21] contenente tutte le entità di tipo T.

6.b.3 IDbHDriver

Espone i metodi relativi ai driver dei database implementati. Utilizzando un'interfaccia generica si ha la possibilità di eseguire a runtime il codice specifico per il singolo, o i singoli, database presi in considerazione.

I metodi che espone questa interfaccia saranno spiegati più nel dettaglio nel paragrafo *5.c*, pertanto si fornirà una semplice introduzione che sarà integrata in seguito:

-CreateConnection: Richiede una connection string come parametro e in base all'implementazione restituisce la relativa DbConnection.

[20] Per overload si intende un numero di definizioni dello stesso metodo superiore a 1 per cui i parametri con cui viene richiamato cambiano.
[21] Un oggetto IEnumerable è un oggetto generico rappresentante una lista di valori, al cui interno è implementato un sistema di accesso via indici.

-CreateCommand: Sono presenti due overload: il primo richiede un parametro di tipo DbConnection e restituisce un DbCommand generico, mentre il secondo richiede una stringa SQL ed una DbConnection e restituisce il DbCommand specifico per quell'operazione e quella connessione.

-CreateDataAdapter: Impostato ugualmente alla funzione CreateCommand permette di ottenere un DbDataAdapter in base a una DbConnection o alla coppia di parametri stringa SQL e DbConnection.

-CreateParameter: In base al generico DbHParameter (che sarà spiegato nel dettaglio in seguito) crea e restituisce un DbParameter specifico per l'implementazione sottostante.

-FixParameters: Effettua un controllo sulla stringa SQL per evitare errori nel riconoscimento dei nomi dei parametri per le varie implementazioni dei database.

-FillParameterReturn: Ottiene i parametri specifici restituiti dall'esecuzione del DbCommand passato come parametro e li trasforma in una lista di parametri generici DbHParameter.

-GetBlob: Richiede come parametri una DbConnection e una query SQL che restituisca un oggetto di database di tipo BLOB (Binary Large Object) e ne permette il fetch da database in base all'implementazione sottostante.

-ClearAllPools: Ripulisce i pool di connessione del database specificato.

6.b.4 IRepository<T>

Grazie a questa interfaccia è possibile accedere al repository generico di un'entità T. I metodi esposti sono gli stessi metodi esposti dalla interfaccia IDbHSession, in questo modo, chiunque la implementi sarà in grado di gestire i repository interni richiamando dalla sessione i metodi esposti dai singoli repository.

Differenza sostanziale con la IDbHSession è che un repository è specifico per oggetto di tipo T, dunque le firme dei metodi non specificheranno la classe degli oggetti su cui si sta lavorando, mentre all'interno della IDbHSession, essendo presenti più repository e più tipi di entità, è stato necessario sviluppare firme di metodi più generiche.

Ogni implementazione di ogni repository sarà specifica per ORM utilizzato. Quindi, ad esempio, nel caso di due ORM, NHIbernate ed Entity Framework, bisognerà creare due classi per gestire i due relativi repository, entrambi implementanti dall'interfaccia in oggetto.

6.b.5 IDbHTransaction

Interfaccia base al cui interno troviamo la sessione relativa all'ORM utilizzato. Ereditando da IDbTransaction permette automaticamente l'esposizione di Commit, Rollback, IsolationLevel e Connection. Tutto ciò che espone è una proprietà contenente la sessione relativa alla transazione attualmente in corso.

6.b.6 IEngineTransaction

Questa interfaccia rappresenta l'implementazione di una transazione per un ORM specifico, i metodi che espone sono:

-CreateObjectSet<T>: Non richiede parametri e permette di ottenere un Repository di entità di tipo T specifico per l'implementazione dell'ORM utilizzato.

-Commit: Effettua il Commit della transazione, applicando tutte le modifiche in sospeso al database. Per comodità, in seguito a un commit avvenuto con successo, la transazione verrà nuovamente aperta, in modo da permettere lato codice dei commit intermedi all'interno di una sola transazione.

-Rollback: Effettua il rollback della transazione. In questo caso tutte le modifiche vengono scartate e la transazione si considera fallita.

-GetTypes: Ottiene tutti i tipi degli oggetti contenuti nel repository. Le implementazioni di questo metodo dovranno utilizzare la classe statica EngineService per ottenere via reflection[22] tutti i tipi specificati come mappabili dall'implementazione ORM attuale. In *Figura 11* è presentata una sezione dell'implementazione della reflection all'interno di questa classe.

```
public static List<Type> GetType(string serviceName)
{
    switch (DbDataProvider.GetDbType(serviceName))
    {
        //Ottiene le classi di dominio in base all'ereditarietà di ClassMapping
        case DatabaseType.Oracle:
            return ReflectionUtil.GetAllTypes
                (t =>
                t.BaseType != null && t.BaseType.IsGenericType &&
                (t.BaseType.GetGenericTypeDefinition() == typeof(ClassMapping<>))
                && System.Attribute.GetCustomAttribute(t, typeof(NhiClassMapAttribute)) != null
                && ((NhiClassMapAttribute)System.Attribute.GetCustomAttribute(t, typeof(NhiClassMapAttribute)))
                .DbNameMap == DatabaseType.Oracle);

        case DatabaseType.SqlServer:
            return ReflectionUtil.GetAllTypes
                (t =>
                t.BaseType != null && t.BaseType.IsGenericType &&
                t.BaseType.GetGenericTypeDefinition() == typeof(ClassMapping<>)
                && System.Attribute.GetCustomAttribute(t, typeof(NhiClassMapAttribute)) != null
                && ((NhiClassMapAttribute)System.Attribute.GetCustomAttribute(t, typeof(NhiClassMapAttribute)))
                .DbNameMap == DatabaseType.SqlServer);
```

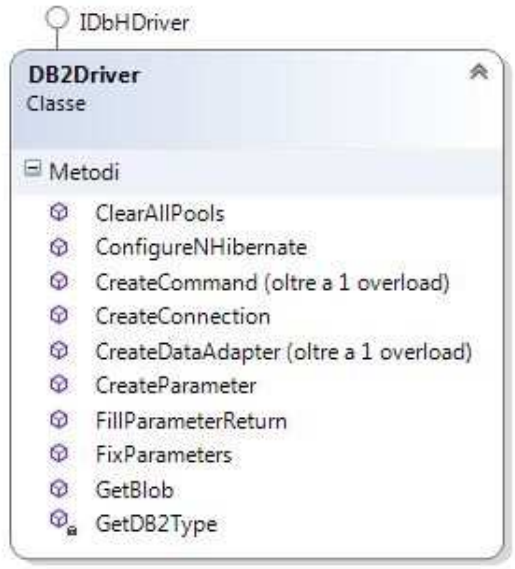
Figura 11: *Sezione dell'implementazione della classe EngineService*

6.b.7 IUnitOfWork

Interfaccia che rappresenta l'unità di lavoro sulla transazione creata e punto di ingresso da cui effettuare il Commit, unico metodo esposto. Il Rollback sarà implementato sul Dispose della classe che erediterà questa interfaccia.

[22] La reflection è la capacità di un programma di eseguire elaborazioni che hanno per oggetto il programma stesso, e in particolare la struttura del suo codice sorgente.

6.c Driver di database



I Driver dei database sono le implementazioni reali dei sistemi di connessione e comunicazione, realizzati per ognuno dei database implementati.

L'esempio mostrato in *Figura 12* mostra l'implementazione del Driver per la comunicazione verso DB2. Ogni Driver eredita dall'interfaccia IDbDriver presentata precedentemente e contiene metodi che permettono la creazione delle strutture dati atte alla comunicazione col database, CreateConnection e CreateCommand, e alcune funzioni di supporto.

Figura 12: Esempio di Driver di Database

-FillParameterReturn: permette di ottenere i parametri risultanti da una query SQL, trasformarli in DbHParamter in modo da mantenere i risultati dei vari database standardizzati, e restituirli al metodo chiamante.

-GetBlob: permette di ottenere un campo BLOB[23] dal database. Essendo il procedimento non standard è stato necessario differenziarlo per ogni implementazione.

-FixParameters: fa da supporto alla sintassi SQL dei parametri aggiungendo rispettivamente ":" o "@" di fronte al nome del parametro, poiché in base al database lo standard potrebbe variare.

-Get{Database}Type: effettua un mapping dal tipo specifico del database al tipo equivalente in .NET, permettendo dunque una conversione fra {Database} [24] Parameter al generico DbHParameter e viceversa.

-ClearAllPools: serve semplicemente ad effettuare il refresh dei pool di connessione per fare in modo che alla chiusura dell'oggetto Connection, tutte le connessioni legate ad esso siano correttamente marcate per la distruzione alla chiusura della stessa.

-ConfigureNHibernate: permette di lanciare i metodi di configurazione dell'ambiente di NHibernate. Saranno impostati rispettivamente Driver e Dialect.

[23] BLOB: Binary Large Object. A livello fisico, nei database è un array di byte, al cui interno è possibile salvare qualsiasi tipo di file, a patto che le dimensioni non eccedano le dimensioni massime per un campo di questo tipo del database utilizzato.
[24] Per "{Database}" Si intende il nome specifico del database utilizzato. Essendo diverso in base all'implementazione del driver si è deciso di rappresentarlo con un segnaposto.

Per quanto riguarda il Driver, esso è sempre presente e corretto all'interno della classe Driver messa a disposizione da NHibernate. Per quanto riguarda il Dialect, invece, di alcuni database viene fornita solo un'implementazione incompleta, che non supporta l'approccio Code-First[25] utilizzato per questo progetto.

Pertanto, seguitamente ad uno studio approfondito, si è rivelato necessario scrivere Dialect customizzati basandosi su altri Dialect funzionanti, in modo che supportassero l'approccio scelto. Di seguito un esempio riguardante DB2. La stessa procedura è stata eseguita anche per Ingres.

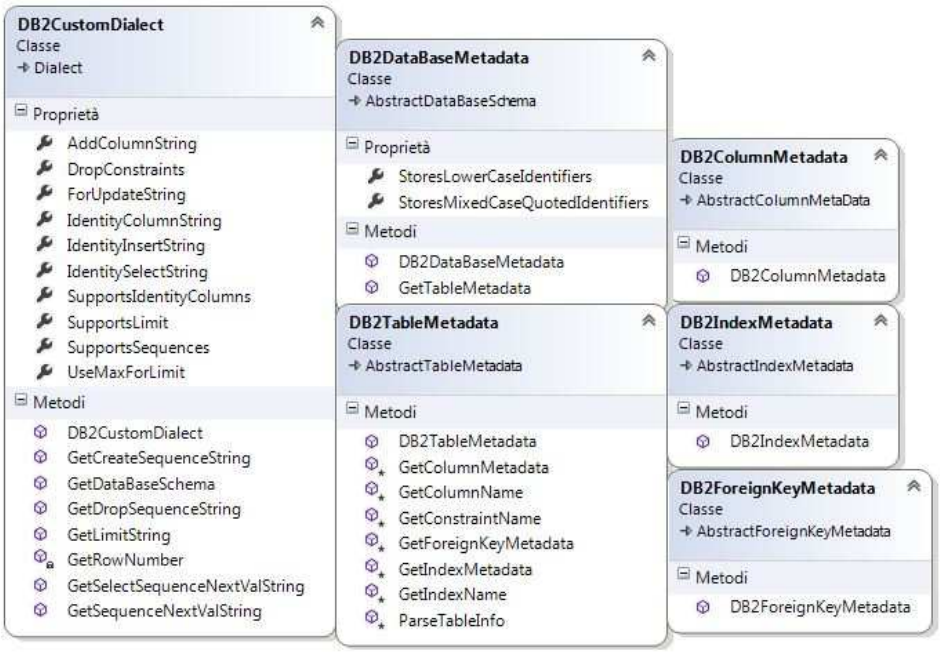


Figura 13: ***Schema di un Dialect NHIbernate e delle classi da cui dipende***

Com'è possibile notare dalla *Figura 13*, il Dialect mette a disposizione operazioni fondamentali per la comunicazione col database, ad esempio il recupero di informazioni sullo schema o addirittura il codice SQL necessario ad ottenere il valore successivo di una sequenza (GetSequenceNextValString e GetSelectSequenceNextValString).

Nel caso di DB2 la maggior parte di questi metodi era già implementata (per Ingres è stato necessario riscrivere tutto, a parte il costruttore), ma il metodo GetDatabaseSchema non era presente.

Questo metodo permette l'estrapolazione di metadati relativi al database utilizzato, creando e restituendo il DataBaseMetadata opportuno. Tale classe è un'implementazione dell'interfaccia AbstractDataBaseSchema contenuta in NHibernate e contiene informazioni sul funzionamento del database, quali ad esempio i booleani StoresLowerCaseIdentifiers e

[25] Per "Approccio Code-First" si intende un approccio allo sviluppo di ORM in cui i dati di mapping delle entità vengono scritte a mano piuttosto che generate dal database. È dunque possibile avviare la stesura del codice senza avere ancora effettuato il setup del database desiderato.

StoresMixedCaseQuotedIdentifiers, oltre al metodo GetTableMetadata, che restituisce un oggetto ITableMetadata.

Questo tipo di oggetto fornisce un dizionario in grado di interfacciare i dati ottenuti e i loro corrispettivi metadati, consentendo di impostare i nomi delle colonne nelle tabelle di metadati relativi a vari oggetti del database, quali Indici, ForeignKey, ecc..

Le implementazioni DB2ColumnMetaData, DB2ForeignKeyMetaData e DB2IndexMetadata sono dunque state effettuate in base a questo criterio, e una volta completata l'implementazione, è stato possibile utilizzare questo Dialect al posto di quello già presente.

Esposto dunque il cuore implementativo del progetto si può iniziare a considerare la vera e propria implementazione funzionale, aprendo una rapida appendice su un'importante classe di utility del progetto, per poi passare alla classe DbDataProvider, il sistema d'accesso ai dati.

6.d Classe di supporto: ReflectionUtil

Figura 14: *Class Diagram per la classe ReflectionUtil*

Per facilitare le operazioni di configurazione degli ORM implementati ed evitare la necessità di gestire via codice le liste di tipi utilizzate dalle implementazioni ORM si è deciso di realizzare una classe che permettesse di accedere ai tipi desiderati via reflection.

L'implementazione rappresentata in *Figura 14* consiste di due metodi:

-IsNUnit: Effettua dei controlli sui tipi caricati per capire se l'ambiente di esecuzione è di tipo NUnit. Grazie a questo metodo è dunque possibile sapere se sono all'interno dell'ambiente di test.

-GetAllTypes: Permette di ottenere tutti i tipi di tutti gli Assembly caricati in memoria, filtrandoli in base al predicato passato come parametro.

In *Figura 15* è possibile visualizzare l'intera implementazione della classe in oggetto:

```
/// <summary>
/// Classe di supporto per l'utilizzo della reflection
/// </summary>
internal class ReflectionUtil
{
    /// <summary>
    /// In base agli attributi specificati sull'ambiente di esecuzione, verifico se l'ambiente è NUnit
    /// </summary>
    /// <returns></returns>
    internal static bool IsNUnit()
    {
        var trace = new StackTrace();

        for (int i = 0; i < trace.FrameCount; i++)
        {
            MethodBase methodBase = trace.GetFrame(i).GetMethod();
            if (methodBase.IsDefined(typeof(TestAttribute), true))
                return true;
            if (methodBase.IsDefined(typeof(TestFixtureSetUpAttribute), true))
                return true;
        }
        return false;
    }

    /// <summary>
    /// Trova tutti i tipi in base al predicato specificato
    /// </summary>
    /// <param name="predicate"></param>
    /// <returns></returns>
    internal static List<Type> GetAllTypes(Func<Type, bool> predicate)
    {
        var types = new List<Type>();

        List<Assembly> assemblies = AppDomain.CurrentDomain.GetAssemblies().ToList();

        foreach (var assembly in assemblies)
        {
            IEnumerable<Type> t = assembly.GetTypes().Where(predicate);
            types.AddRange(t);
        }
        return types;
    }
}
```

Figura 15: *Implementazione classe ReflectionUtil*

6.e Centro di controllo: DbDataProvider

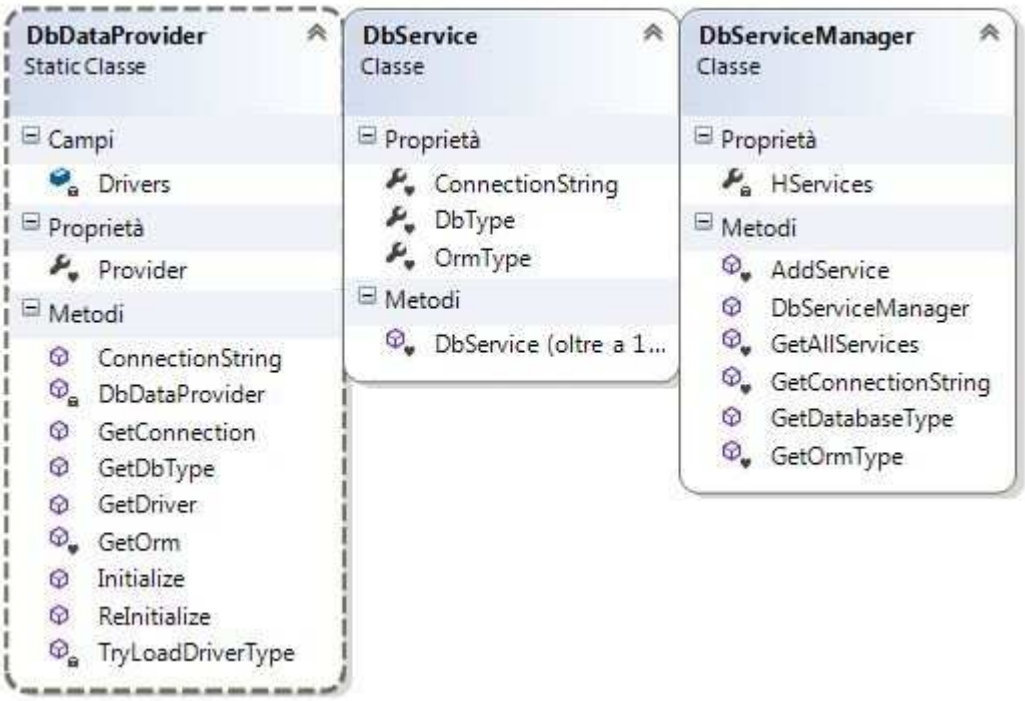
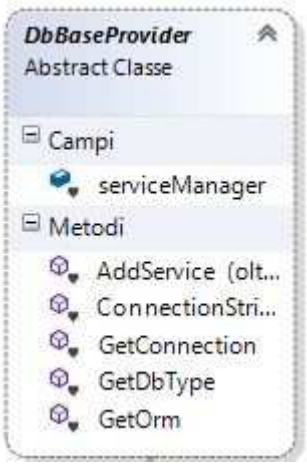


Figura 16: *Class Diagram in riferimento al sistema DbDataProvider*

I servizi principali di comunicazione sono forniti dalla classe DbDataProvider, la cui implementazione è rappresentata in *Figura 16*. Essa consente di salvare i dati relativi alle connessioni ed espone metodi in grado di ottenere informazioni fondamentali per la connessione. Essendo Singleton implementato via classe statica [26] , il costruttore verrà richiamato automaticamente al primo accesso runtime ad essa.

Il setup delle informazioni di comunicazione avviene attraverso l'impostazione di opportuni Service, gestiti da un ServiceManager che permette di effettuare operazioni sui Servizi presenti e al contempo estrapolarne informazioni.

La classe DbDataProvider comunicherà dunque con il ServiceManager che farà da tramite con la lista dei Servizi.



Per quanto riguarda il setup effettivo dei Servizi, viene utilizzata la classe DbBaseProvider, rappresentata in *Figura 14*.

Dopo averla ereditata nella classe provider della propria applicazione sarà possibile richiamare il metodo AddService per aggiungere nuovi servizi al Service Manager, dopodiché basterà richiamare l'Initialize del DataProvider impostando il provider appena creato per registrare le informazioni di connessione della propria applicazione.

Figura 17: *Class Diagram relativo alla classe DbBaseProvider*

[26] In riferimento al Paragrafo3.b

6.f Diagramma delle classi. Punto di ingresso DbCustom

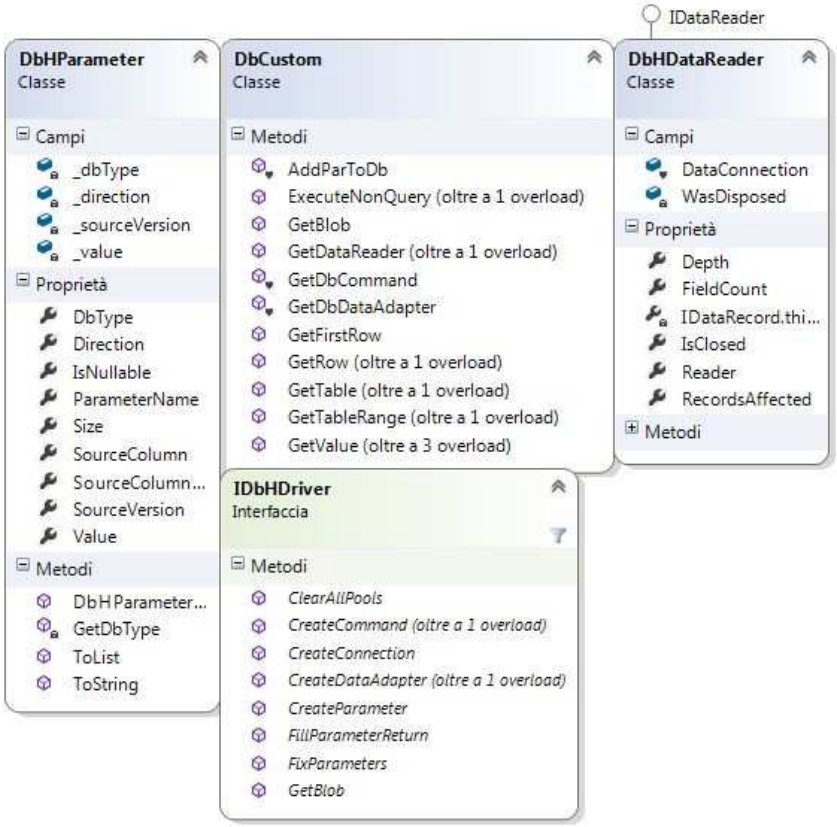


Figura 18: *Class Diagram in riferimento al sistema DbCustom*

Il primo punto di ingresso alla libreria, e senza dubbio il più semplice da utilizzare, è la classe statica DbCustom, di cui è presentata una panoramica in *Figura 17*.

Grazie ad essa è possibile effettuare query in standard SQL specificando il servizio di connessione desiderato senza doversi preoccupare del database su cui sarà eseguita la query. Il nome del servizio, infatti, sarà sufficiente alla libreria per capire il tipo di database interrogato e porterà alla creazione delle opportune strutture dati di comunicazione.

Il cuore del funzionamento si basa sulle implementazione dei DbDriver esposti in precedenza, infatti, utilizzando l'interfaccia da cui tutti i Driver implementano, è possibile sfruttare il metamorfismo del paradigma ad oggetti per creare codice responsivo ai vari cambiamenti di configurazione, senza doversi preoccupare del tipo di driver attualmente utilizzato.

I metodi pubblici resi disponibili dalla DbCustom sono un'estensione di quelli disponibili per mezzo della classe generica DbCommand, nello specifico si tratta di:

-ExecuteNonQuery
-GetRow
-GetFirstRow
-GetTable
-GetTableRange
-GetValue
-GetDataReader

Ognuno di questi metodi supporta l'SQL utilizzando parametri custom DbHParameter che permettono un mapping diretto fra .NET Type, DbType e {Database}Type

Per supportare l'esecuzione di questi metodi sono stati create delle funzioni private di supporto:

-GetDbCommand: In base al nome del servizio, interroga il DbDataProvider per ottenere il comando relativo al database collegato al servizio, restituendo, ad esempio, nel caso di un database MySql, un oggetto MySqlConnection.

-AddParToDb: Permette di esaminare i DbHParameter passati come parametri, generare i relativi {Database}Parameter, collegarli al comando in esecuzione, ed eseguire la query desiderata

-GetDbDataAdapter: Permette di ottenere un {Database}DataAdapter con cui effettuare operazioni di Fill() su tabelle o set di tabelle.

Alla classe DbCustom sono affiancate due importanti strutture dati. Il già accennato DbHParameter e il DbHDataReader.

Il DbHParameter non è altro che una re implementazione dei classici DbParameter, ma con alcune funzionalità in più; non è infatti obbligatorio specificare un DbType, in quanto sarà effettuata una conversione automatica a runtime fra tipi di dato .NET, a DbType, in base alla configurazione in *Figura 19*:

```
private static DbType GetDbType(Type type)
{
    if (type == typeof(object))     return DbType.Object;
    if (type == typeof(Int16))      return DbType.Int16;
    if (type == typeof(Int32))      return DbType.Int32;
    if (type == typeof(Int64))      return DbType.Int64;
    if (type == typeof(UInt16))     return DbType.UInt16;
    if (type == typeof(UInt32))     return DbType.UInt32;
    if (type == typeof(UInt64))     return DbType.UInt64;
    if (type == typeof(Boolean))    return DbType.Boolean;
    if (type == typeof(Byte))       return DbType.Byte;
    if (type == typeof(SByte))      return DbType.SByte;
    if (type == typeof(Decimal))    return DbType.Decimal;
    if (type == typeof(Single))     return DbType.Single;
    if (type == typeof(Double))     return DbType.Double;
    if (type == typeof(String))     return DbType.String;
    if (type == typeof(DateTime))   return DbType.DateTime;
    if (type == typeof(Byte[]))     return DbType.Object;
    if (type == typeof(TimeSpan))   return DbType.Time;
    if (type == typeof(Guid))       return DbType.Guid;
    if (type == typeof(DateTimeOffset)) return DbType.DateTimeOffset;

    throw new Exception("Il DataType non può essere ricavato autonomamente. È necessario specificarlo.");
}
```

Figura 19: Configurazione fra .NET Type e DbType

Vi è poi implementato un supporto più esteso nel confronto del valore passato alla proprietà Value, che permette di evitare alcuni errori di conversione e arrotondamento, soprattutto nel momento in cui vengono valorizzati i parametri di output.

Sono inoltre presenti due funzioni di utility:

-ToList: Trasforma il parametro in una lista di parametri, con esso alla posizione 0.
-ToString: Formatta il nome, il tipo e il valore del parametro, restituendolo sotto forma di stringa, utile per effettuare operazioni di debug sul contenuto dell'oggetto.

```
public class DbHDataReader: IDataReader
{
    internal DbConnection DataConnection;
    private bool WasDisposed;

    /// <summary> ...
    public DbHDataReader(DbCommand cmd)
    {
        DataConnection = cmd.Connection;
        DataConnection.Open();
        Reader = cmd.ExecuteReader();
    }

    /// <summary> ...
    public DbDataReader Reader { get; set; }

    /// <summary> ...
    public void Dispose()
    {
        if (WasDisposed)
            return;

        if (!Reader.IsClosed)
            Reader.Close();

        if (DataConnection.State != ConnectionState.Closed)
            DataConnection.Close();

        Reader.Dispose();

        DataConnection.Dispose();
        WasDisposed = true;
    }

    /// <summary> ...
    public void Close()
    {
        if (!Reader.IsClosed)
            Reader.Close();
        if (DataConnection.State != ConnectionState.Closed)
            DataConnection.Close();
    }

    /// <summary>
    /// Restituisce il nome della colonna all'indice i
    /// </summary>
    /// <param name="i"></param>
    /// <returns></returns>
    public string   GetName(int i)          {return Reader.GetName(i);}
    ...
    ...
```

Figura 20: Implementazione classe DbHDataReader

Un caso di studio interessante, è invece, il DbHDataReader, di cui una sezione significativa della sua implementazione è presentata in *Figura 20*. Basandosi sulla implementazione della DbCustom, per mezzo del comando GetDataReader, è infatti possibile restituire ed utilizzare un DataReader all'esterno della connessione che l'ha generato.

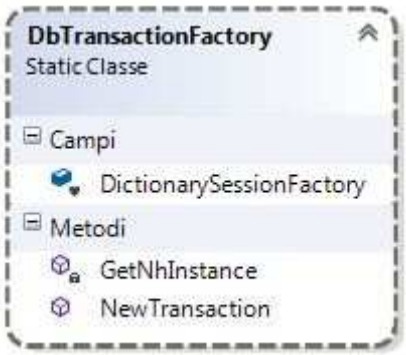
Utilizzando un'implementazione standard, questo porta chiaramente ad accedere ad un oggetto DataReader che nel momento in cui viene restituito ha già internamente una connessione chiusa.

Per questo motivo è stato necessario implementare una classe contenente il DbDataReader e tutte le informazioni sulla connessione che esso usa, passando da un costruttore per mezzo del quale effettuare l'apertura della connessione, e re implementando i metodi Dispose e Close resi disponibile dall'interfaccia IDataReader, in modo da gestire sia il DbDataReader che la connessione.

Dopodiché sono stati re-implementati tutti gli altri metodi del DbDataReader semplicemente richiamando i metodi della proprietà Reader (l'effettivo DbDataReader) del DbHDataReader.

6.g Diagramma delle classi. Punto di ingresso DbTransaction

Figura 21: Classe Factory di creazione Istanze di ORM



Per utilizzare la libreria partendo dalla classe DbTransaction, e quindi accedendo ai metodi di SQL transazionale, è necessario ottenere un'istanza dell'ORM con cui si desidera lavorare (per ora solo NHIbernate) dalla classe statica DbTransactionFactory presentata in *Figura 21*.

La funzione NewTransaction, in base all'ORM specificato nella configurazione del provider, relativo al servizio scelto, restituirà l'istanza opportuna di tipo DbTransaction, tale oggetto sarà internamente implementato in base al diagramma delle classi in *Figura 22*, e conterrà al suo interno alcune proprietà generalmente attribuibili alle transazioni, quali Connection e IsolationLevel, oltre al cuore dell'implementazione della libreria, e cioè:

Session: Oggetto di tipo DbSession. In essa è implementata l'interfaccia IDbHSession e permette all'utente di accedere ai dati per mezzo di Repository. Tali repository vengono generati a runtime, uno per ogni entità mappata dal sistema ORM. Per accedere a un repository specifico partendo dalla sessione, sarà necessario richiamare la funzione GetRepository<T>() dove T è l'entità di cui si vuole ottenere il Repository.

UnitOfWork: Questo oggetto re-implementa Commit e Rollback delle transazioni, pertanto le chiamate del tipo Transaction.Commit() e Transaction.Rollback() richiameranno essenzialmente le relative funzioni della UnitOfWork, dove il metodo Rollback è stato implementato come Dispose

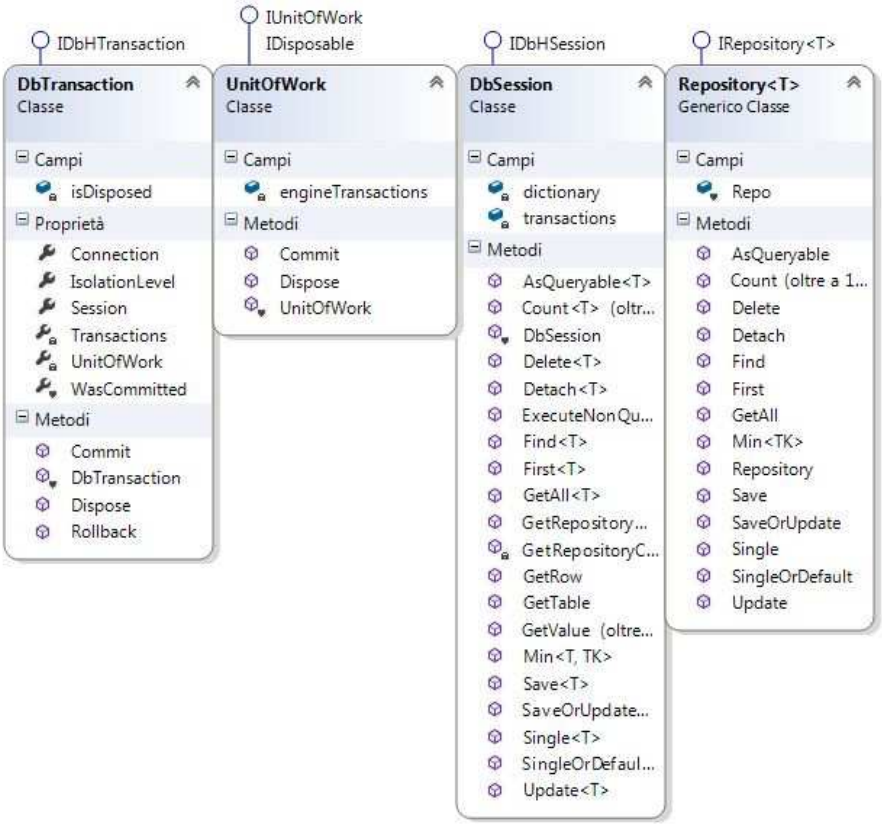


Figura 22: Insieme di classi che gestiscono una transazione

Compreso il funzionamento delle classe generiche atte alla creazione di una nuova transazione possiamo dunque concentrare l'attenzione sull'implementazione effettiva di NHibernate, rappresentata via diagrammi di classe in *Figura 23.*

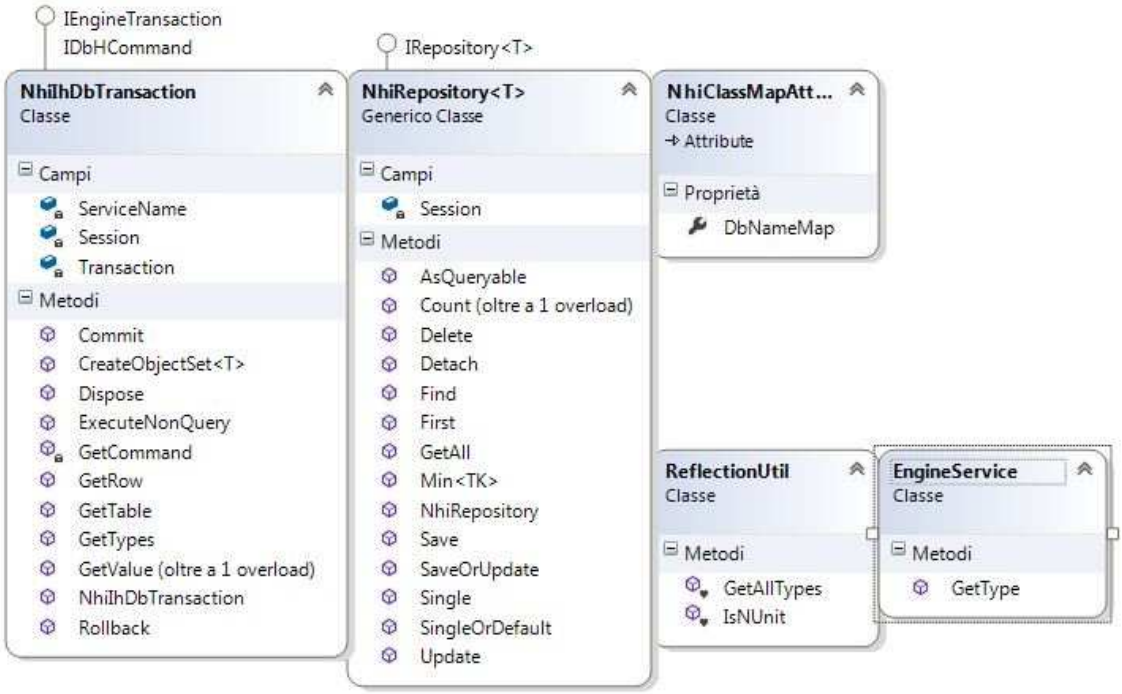


Figura 23: Rappresentazione delle classi relative a una transazione NHIbernate

Utilizzando l'approccio Code-First, il flusso dei dati partirà dal dominio, per poi passare dalla mappatura specifica di NHIbernate. In fase di configurazione, è necessario specificare la lista di tipi da cui estrapolare le informazioni sugli oggetti, la quale viene fornita via reflection utilizzando il seguente approccio:

Per prima cosa, a livello di mapping in Fluent NHIbernate è necessario specificare il tipo di database a cui il mapping fa riferimento per mezzo dell'attributo NhiClassMap, come mostrato in *Figura 24*, relativamente ad un database DB2.

```
[NhiClassMap(DbNameMap = DatabaseType.DB2)]
public class TestDetailMapDb2 : ClassMapping<DbDetail>
{
    public TestDetailMapDb2()
    {
        Table("TEST_DETAIL");
        Lazy(true);
        Id(x => x.Iddetail, map => map.Generator(Generators.Identity));
        Property(x => x.Dsdetail);
        ManyToOne(x => x.Testata, map =>
        {
            map.Column("IDTEST");
            map.NotNullable(true);
            map.Cascade(Cascade.None);
        });
    }
}
```

Figura 24: Implementazione class mapping NHIbernate

A questo punto, a runtime, la classe statica EngineService si occuperà di ottenere tutti i tipi dell'Assembly il cui valore DbNameMap dell'attributo NhiClassMap indichi il database specificato dalle informazioni di connessione specificate nei servizi. Il tutto è facilitato dall'utilizzo della classe statica ReflectionUtil esposta in precedenza nel paragrafo *5.d*.

Utilizzando questi dati per effettuare la configurazione di NHIbernate verrà utilizzata la ISessionFactory risultante per costruire l'oggetto DbTransaction che sarà utilizzato dall'esterno.

L'unica classe che rimane da esaminare è NhiReposiory, che non è altro che un'implementazione dell'interfaccia IRepository<T> che all'interno effettua le operazioni sulla Session già rese disponibili da NHIbernate. In *Figura 25* è presente una piccola parte del codice dell'implementazione del Repository NHIbernate.

```
class NhiRepository<T> : IRepository<T> where T : class
{
    private readonly ISession Session;
    public NhiRepository(ISession session)
    {
        Session = session;
    }

    #region IRepository<T> Members

    /// <summary>
    /// Ottiene una lista di oggetti di tipo T in base a una condizione
    /// </summary>
    /// <param name="where"></param>
    /// <returns></returns>
    public IEnumerable<T> Find(Expression<Func<T, bool>> @where)
    {
        return Session.Query<T>().Where(where);
    }

    /// <summary>
    /// Ottiene l'unico oggetto di tipo T in base alla condizione
    /// </summary>
    /// <param name="where"></param>
    /// <returns></returns>
    public T Single(Expression<Func<T, bool>> @where)
    {
        return Session.Query<T>().Single(where);
    }

    /// <summary>
    /// Ottiene il primo oggetto di tipo T in base alla condizione
    /// </summary>
    /// <param name="where"></param>
    /// <returns></returns>
    public T First(Expression<Func<T, bool>> @where)
    {
        return Session.Query<T>().First(where);
    }
```

Figura 25: Esempio di implementazione del repository NHIbernate

6.h Unit Testing

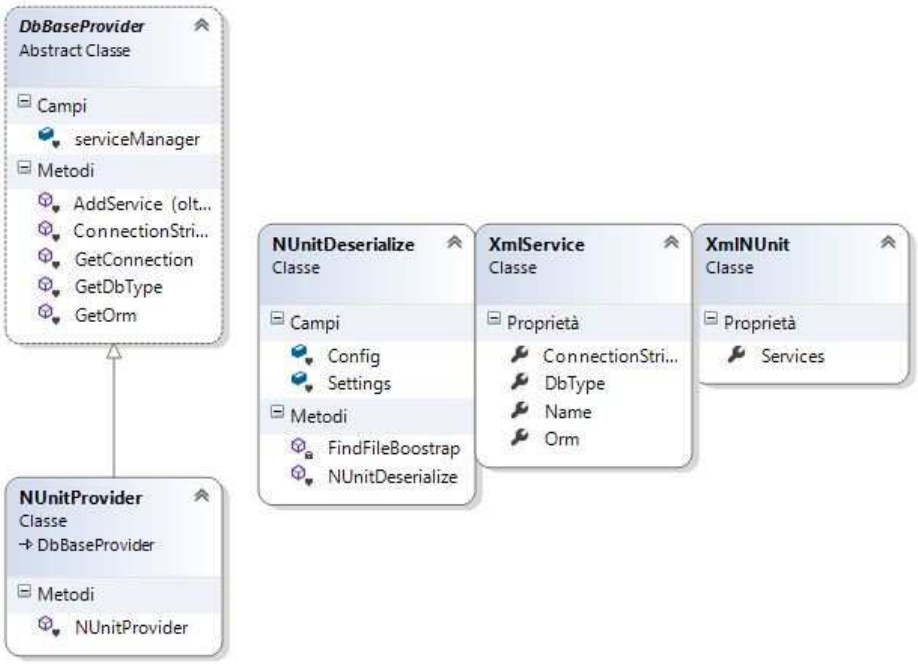


Figura 26: Diagramma della classi relativo a sessione Unit Test

Per poter utilizzare i metodi di Unit Testing, non essendo eseguiti da un applicazione in particolare, e quindi mancando di inizializzazione opportuna del Driver, è stato creato un costruttore ad hoc all'interno della DbDataProvider. Nel momento in cui verrà richiamata, in quanto classe statica, si attiverà il costruttore che via reflection capirà se l'ambiente in cui è eseguito è di Unit Test grazie all'utilizzo della classe ReflectionUtil.

La risposta affermativa a questa domanda genererà la deserializzazione del file XML di configurazione "nunit.config" che genererà a sua volta una lista di servizi aggiunti al DataProvider via AddService, passando dall'implementazione della DbBaseProvider per gli Unit Test: la NUnitProvider.

Le operazioni specificate in questi paragrafi sono rese possibili dalle strutture dati di cui si offre una panoramica in *Figura 26*.

Terminata la de-serializzazione dei servizi e l'aggiunta degli stessi alle configurazioni di comunicazione, i test potranno funzionare come se fossero effettivamente richiamati in un'applicazione.

Trattandosi di un progetto il cui scopo è la generalizzazione dell'esecuzione di query per i database implementati, la sintassi dei test sarà equivalente a prescindere dal database sul quale sarà eseguito, la cui identificazione a runtime sarà gestita dal nome del servizio relativo. Cambiando dunque specifica del servizio, sarà possibile vedere come i test funzionino su tutte le piattaforme mantenendo la stessa semantica.

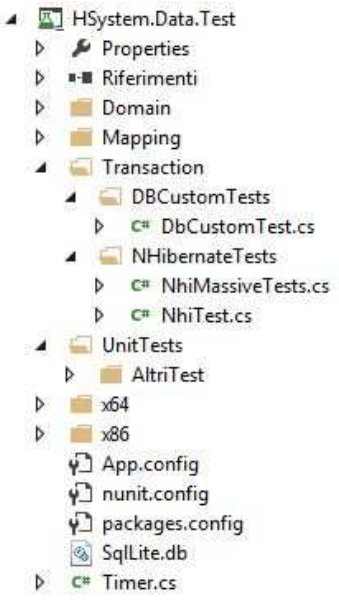
6.h.1 Rapporto sui test eseguiti

Per effettuare i test di seguito, è stato prima di tutto necessario impostare la lista dei servizi impostando il file XML di configurazione nunit.config all'interno del progetto come mostrato in *Figura 27*.

```
<?xml version="1.0" encoding="utf-8" ?>
<NUnitConfig>
  <ListService>
    <Service Name="mys" DbType="MySQL" Orm="NHI"
             ConnectionString="Server=192.168.1.249;Database=TESTDB;Uid=ema;Pwd=tesiema;" />

    <Service Name="ora" DbType="Oracle" Orm="NHI"
             ConnectionString="Data Source=(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(HOST=192.168.1.249)
             (PORT=1521))(CONNECT_DATA=(SERVICE_NAME=xe)));User Id=ema;Password=tesiema;" />

    <Service Name="sqlite" DbType="SQLite" Orm="NHI"
             ConnectionString="Data Source=SqlLite.db;Version=3;foreign keys=true;" />

    <Service Name="sql" DbType="SqlServer" Orm="NHI"
             ConnectionString="Server=192.168.1.249;Database=dbtest;User Id=ema;Password=tesiema;" />

    <Service Name="postgre" DbType="PostgreSQL" Orm="NHI"
             ConnectionString="Server=192.168.1.249;Port=5432;Database=testdb;User Id=ema;Password=tesiema;" />

    <Service Name="firebird" DbType="Firebird" Orm="NHI"
             ConnectionString="User=ema;Password=tesiema;Database=C:\FirebirdServer\testdb.fdp;DataSource=192.168.1.249;Port=3050;
             Dialect=3;Charset=NONE;Role=;Connection lifetime=15;Pooling=true;MinPoolSize=0;MaxPoolSize=50;Packet Size=8192;ServerType=0;" />

    <Service Name="ingres" DbType="Ingres" Orm="NHI"
             ConnectionString="Host=192.168.1.249;Database=dbtest;Uid=emaserver;Pwd=macmanu;" />

    <Service Name="db2" DbType="DB2" Orm="NHI"
             ConnectionString="Server=192.168.1.249:50000;Database=TEST;UID=Ema;PWD=macmanu;CurrentSchema=EMA;" />
  </ListService>
</NUnitConfig>
```

Figura 27: Contenuto file di configurazione Unit Test

Specificato il file di configurazione che la classe DbDataProvider deserializzerà si può specificare la struttura del progetto di test, rappresentata in *Figura 28*.



All'interno di questo progetto si possono distinguere:
-Cartella Domain: questa cartella contiene tutte le classi di dominio che verranno mappate per i vari ORM implementati.

-Cartella Mapping: questa cartella contiene i mapping degli oggetti specificati nella cartella Domain per ogni ORM implementato.

-Transaction: questa cartella contiene tutti i test che effettuano delle operazioni all'interno dei database, nello specifico troveremo all'interno della cartella DBCustomTests tutti i test relativi alla classe DbCustom, mentre all'interno della cartella NHIbernateTests saranno presenti tutti i test relativi ad NHIbernate.

-Cartella UnitTests: questa cartella contiene test generici sull'ambiente di progetto, quali ad esempio controlli selle stringhe di connessione specificate nel file di configurazione.

Figura 28: Struttura progetto di Test

-Cartelle x64 e x86: contengono i driver di SQLite, in quanto inserito come file di progetto (SqlLite.db) necessita dei driver all'interno della cartella in cui si troveranno i binari a compilazione eseguita.

-nunit.config: file di configurazione presentato all'inizio di questo paragrafo.

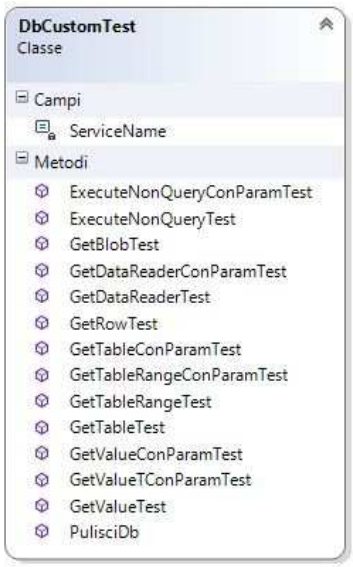
-Classe Timer.cs: classe di utility che permette di calcolare il tempo di esecuzione di alcuni test.

6.h.2 Test sulla classe DbCustom

Tutti i test presenti all'interno della classe DbCustomTests sono pensati per sottoporre a test in modo completo la relativa classe DbCustom.

Una panoramica dei test implementati è disponibile in *Figura 29*.

Figura 29: Panoramica dei Test sulla DbCustom implementati



-Test sull'istruzione ExecuteNonQuery: questi test, resi disponibili con o senza parametri, effettuano la pulizia della tabella considerata, inseriscono una tupla e dopodiché verificano ne verificano il corretto inserimento a database per mezzo di opportune istruzioni messe a disposizione del framework NUnit definite Assert che verificano la veridicità di un'espressione. Un esempio di questo tipo di test è presentato in *Figura 30*.

```
[Test]
public void ExecuteNonQueryTest()
{
    DbCustom.ExecuteNonQuery("DELETE FROM TEST_TABLE", ServiceName);
    DbCustom.ExecuteNonQuery("Insert into TEST_TABLE(DSVALUETEST) VALUES ('Test')", ServiceName);
    Assert.AreEqual(1, DbCustom.GetValue<int>("SELECT COUNT(*) FROM TEST_TABLE WHERE DSVALUETEST = 'Test'", ServiceName));
}
```

Figura 30: Esempio di esecuzione Test sull'operazione ExecuteNonQuery

-GetBlobTest: questo test genera un vettore contenente 400 byte inizializzati per mezzo di un banale ciclo, il vettore generato sarà poi inserito nel campo opportuno di un record della tabella TEST_TABLE e sarà poi riottenuto via Query. Una volta restituito sarà controllato che il vettore sia uguale a quello creato precedentemente.

-Test sull'istruzione GetDataReader: analogamente ai test sull'ExecuteNonQuery, permette di inserire valori all'interno del database che saranno poi letti per mezzo della classe DbHDataReader esposta nel paragrafo *5.f.* In *Figura 31* è disponibile un esempio di questa tipologia di Test.

```
[Test]
public void GetDataReaderTest()
{
    DbCustom.ExecuteNonQuery("DELETE FROM TEST_TABLE", ServiceName);
    DbCustom.ExecuteNonQuery("Insert into TEST_TABLE(DSVALUETEST) VALUES ('Test')", ServiceName);

    using (DbHDataReader reader = DbCustom.GetDataReader("SELECT DSVALUETEST FROM TEST_TABLE WHERE DSVALUETEST = 'Test'", ServiceName))
    {
        reader.Read();
        Assert.AreEqual("Test", reader.GetString(0));
        reader.Dispose();
        Assert.IsTrue(reader.IsClosed);
    }
}
```

Figura 31: Implementazione Test su istruzione GetDataReader

-Test sull’istruzione GetTable e GetTable Range: effettua l’inserimento di tuple che verranno poi lette dalla tabella ottenuta dai metodi GetTable e GetTableRange. Verificato l’effettivo inserimento, il test si considera effettuato con successo. Il test sull’istruzione GetRow si basa su un’implementazione interna che si appoggia al metodo GetTable, pertanto non vi è alcun dettaglio significativo da aggiungere sul suo funzionamento.

-Test sull’istruzione GetValue: questi test effettuano l’esecuzione di classiche query scalari (COUNT) sui record set inseriti, e si assicurano che il risultato restituito sia giusto o che, nel caso di utilizzo dell’istruzione GetValue<T>, il risultato sia convertito opportunamente.

-PulisciDb: Questo test, essendo l’ultimo ad essere eseguito, permette di lasciare sempre il database in uno stato consistente, ripulendo le tabelle su cui son stati effettuati i test.

I risultati dei test relativi alla DbCustom sono rappresentati di seguito in *Tabella 2.*

Service Name: mys Server: MySql	**Service Name: db2 Server: DB2**	**Service Name: ora Server: Oracle**	**Service Name: postgre Server: PostgreSQL**
DbCustomTest *(14 tests)* ExecuteNonQueryConParamTest ExecuteNonQueryTest GetBlobTest GetDataReaderConParamTest GetDataReaderTest GetRowTest GetTableConParamTest GetTableRangeConParamTest GetTableRangeTest GetTableTest GetValueConParamTest GetValueTConParamTest GetValueTest PulisciDb	DbCustomTest *(14 tests)* ExecuteNonQueryConParamTest ExecuteNonQueryTest GetBlobTest GetDataReaderConParamTest GetDataReaderTest GetRowTest GetTableConParamTest GetTableRangeConParamTest GetTableRangeTest GetTableTest GetValueConParamTest GetValueTConParamTest GetValueTest PulisciDb	DbCustomTest *(14 tests)* ExecuteNonQueryConParamTest ExecuteNonQueryTest GetBlobTest GetDataReaderConParamTest GetDataReaderTest GetRowTest GetTableConParamTest GetTableRangeConParamTest GetTableRangeTest GetTableTest GetValueConParamTest GetValueTConParamTest GetValueTest PulisciDb	DbCustomTest *(14 tests)* ExecuteNonQueryConParamTest ExecuteNonQueryTest GetBlobTest GetDataReaderConParamTest GetDataReaderTest GetRowTest GetTableConParamTest GetTableRangeConParamTest GetTableRangeTest GetTableTest GetValueConParamTest GetValueTConParamTest GetValueTest PulisciDb
Service Name: firebird Server: Firebird	**Service Name: sqlite Server: SQLite**	**Service Name: sql Server: SQLServer**	**Service Name: ingres Server: Ingres**
DbCustomTest *(14 tests)* ExecuteNonQueryConParamTest ExecuteNonQueryTest GetBlobTest GetDataReaderConParamTest GetDataReaderTest GetRowTest GetTableConParamTest GetTableRangeConParamTest GetTableRangeTest GetTableTest GetValueConParamTest GetValueTConParamTest GetValueTest PulisciDb	DbCustomTest *(14 tests)* ExecuteNonQueryConParamTest ExecuteNonQueryTest GetBlobTest GetDataReaderConParamTest GetDataReaderTest GetRowTest GetTableConParamTest GetTableRangeConParamTest GetTableRangeTest GetTableTest GetValueConParamTest GetValueTConParamTest GetValueTest PulisciDb	DbCustomTest *(14 tests)* ExecuteNonQueryConParamTest ExecuteNonQueryTest GetBlobTest GetDataReaderConParamTest GetDataReaderTest GetRowTest GetTableConParamTest GetTableRangeConParamTest GetTableRangeTest GetTableTest GetValueConParamTest GetValueTConParamTest GetValueTest PulisciDb	DbCustomTest *(14 tests)* ExecuteNonQueryConParamTest ExecuteNonQueryTest GetBlobTest GetDataReaderConParamTest GetDataReaderTest GetRowTest GetTableConParamTest GetTableRangeConParamTest GetTableRangeTest GetTableTest GetValueConParamTest GetValueTConParamTest GetValueTest PulisciDb

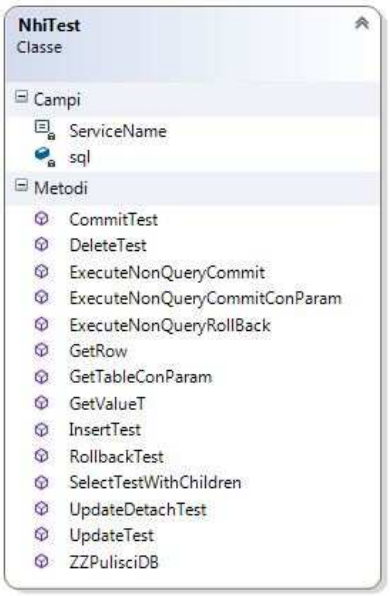
Tabella 2: risultato dell'esecuzione dei test effettuati sulla classe DbCustom divisi per database

6.h.3 Test sulle transazioni con NHIbernate

Per quanto riguarda i test sulla struttura della DbTransaction con ORM specifico NHIbernate, ne son stati realizzati di due tipi: il primo consiste in un test simile a quello effettuato per la classe DbCustom, dove di tutti gli esempi di utilizzo ne viene meticolosamente controllato il funzionamento, mentre il secondo tipo di test è un inserimento massivo di entità effettuato su tutti i database implementati.

Una panoramica dei test implementati è disponibile in *Figura 29*.

Figura 32: Panoramica dei Test implementati per la classe DbTransaction utilizzando NHIbernate



Effettuando il test di inserimento massivo sarà inoltre possibile, per mezzo della classe Timer, farsi un'idea di quale database sia più veloce in rapporto al numero di entità inserite.

I test implementati per verificare il corretto funzionamento del repository e della sessione di NHIbernate sono i seguenti:

-Test che verificano il corretto funzionamento della transazione: sono i test CommitTest e RollbackTest, il cui scopo è verificare la corretta implementazione delle funzioni omonime Commit e Rollback.

I test di questa tipologia sono realizzati generando un esempio tipico di transazione e, nel caso di test sull'operazione Commit, si controllerà che i risultati siano effettivamente riportati su DB, mentre per quanto riguarda il test sull'operazione Rollback ci si assicurerà che nessuna modifica al database sia effettuata.

Nel caso specifico presentato in *Figura 33*, ad esempio, viene dapprima creata una nuova entità di testata e salvata all'interno del repository. Prima del termine dello scope [27] della variabile dbTransaction, però non viene richiamata alcuna operazione di commit. Al termine della using, l'oggetto dbTransaction subirà una operazione di Dispose, al cui comportamento è stato applicato un override che permetta di effettuare il Rollback della transazione. Il risultato da verificare è che all'interno della tabella TEST_TABLE non sia inserito alcun risultato.

```
[Test]
public void RollbackTest()
{
    DbCustom.ExecuteNonQuery("DELETE FROM TEST_TABLE", ServiceName);

    using (IDbHTransaction dbTransaction = DbTransactionFactory.NewTransaction(ServiceName))
    {
        var tt = new DbHeader { Value = "aaaa" };
        dbTransaction.Session.Save(tt);
    }

    var c = DbCustom.GetValue<int>("SELECT COUNT(*) FROM TEST_TABLE", ServiceName);
    Assert.AreEqual(0, c);
}
```

Figura 33: Esempio di Test sull'istruzione Rollback di una transazione

[27] Per "scope" si intende il blocco al cui interno la variabile è riconosciuta. All'esterno di tale scope, la variabile non viene vista dal programma.

-Test che lavorano con le entità: si tratta dei test DeleteTest, InsertTest, UpdateTest ed UpdateDetachTest. Effettuano le operazioni deducibili dai loro nomi sulle entità accedendovi da repository NHIbernate, verificandone il corretto funzionamento.

In *Figura 34* un esempio di test sulle funzioni Delete e Save: per prima cosa sarà creata e salvata un'entità all'interno del repository, in seguito al commit l'entità sarà persistente nel database. Al termine della prima fase sarà poi creata un'altra transazione con lo scopo di ottenere ed eliminare l'entità precedentemente inserita nel repository conseguentemente all'operazione di Commit nel database.

Effettuando un Commit in questo scope non sarà più presente alcuna tupla all'interno della tabella, in quanto l'unica presente è stata appena eliminata.

```
[Test]
public void DeleteTest()
{
    DbCustom.ExecuteNonQuery("DELETE FROM TEST_TABLE", ServiceName);
    using (IDbHTransaction dbTransaction = DbTransactionFactory.NewTransaction(ServiceName))
    {
        var tt = new DbHeader { Value = "aaaa" };
        dbTransaction.Session.Save(tt);
        dbTransaction.Commit();
    }

    using (IDbHTransaction dbTransaction = DbTransactionFactory.NewTransaction(ServiceName))
    {
        DbHeader tt = dbTransaction.Session.Single<DbHeader>(k => k.Value == "aaaa");
        dbTransaction.Session.Delete(tt);
        dbTransaction.Commit();
    }

    var c = DbCustom.GetValue<int>("SELECT COUNT(*) FROM TEST_TABLE", ServiceName);
    Assert.AreEqual(0, c);
}
```

Figura 34: Implementazione Test sulle operazioni di Save e Delete di entità dal repository NHIbernate

-Test che lavorano sulle query transazionali: sono le stesse tipologie di test presentate all'interno dei test per la classe DbCustom. La differenza sostanziale è che le query SQL eseguite sono inserite all'interno della transazione attiva, e dunque sarà necessario un commit prima di apprezzarne le effettive modifiche.

In *Figura 35* viene mostrato un esempio di istruzione INSERT transazionale, eseguita all'interno di una transazione NHIbernate.

```
[Test]
public void ExecuteNonQueryCommit()
{
    DbCustom.ExecuteNonQuery("DELETE FROM TEST_TABLE", ServiceName);
    using (IDbHTransaction dbTransaction = DbTransactionFactory.NewTransaction(ServiceName))
    {
        sql = "INSERT INTO TEST_TABLE ( DSVALUETEST) VALUES ('/* DSVALUETEST */' )";
        dbTransaction.Session.ExecuteNonQuery(sql);
        dbTransaction.Commit();
    }
    Assert.AreEqual(1, DbCustom.GetValue<int>("SELECT COUNT(*) FROM TEST_TABLE WHERE DSVALUETEST ='/* DSVALUETEST */' ", ServiceName));
}
```

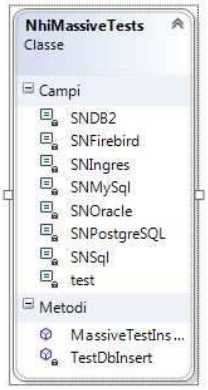
Figura 35: Esempio di query SQL transazionale eseguita all'interno del repository NHIbernate

I risultati dei test relativi alla transazionalità con NHIbernate sono rappresentati di seguito in *Tabella 3*.

Service Name: mys Server: MySql	**Service Name: db2 Server: DB2**	**Service Name: ora Server: Oracle**	**Service Name: postgre Server: PostgreSQL**
NhiTest *(14 tests)* CommitTest DeleteTest ExecuteNonQueryCommit ExecuteNonQueryCommitConParam ExecuteNonQueryRollBack GetRow GetTableConParam GetValueT InsertTest RollbackTest SelectTestWithChildren UpdateDetachTest UpdateTest ZZPulisciDB	NhiTest *(14 tests)* CommitTest DeleteTest ExecuteNonQueryCommit ExecuteNonQueryCommitConParam ExecuteNonQueryRollBack GetRow GetTableConParam GetValueT InsertTest RollbackTest SelectTestWithChildren UpdateDetachTest UpdateTest ZZPulisciDB	NhiTest *(14 tests)* CommitTest DeleteTest ExecuteNonQueryCommit ExecuteNonQueryCommitConParam ExecuteNonQueryRollBack GetRow GetTableConParam GetValueT InsertTest RollbackTest SelectTestWithChildren UpdateDetachTest UpdateTest ZZPulisciDB	NhiTest *(14 tests)* CommitTest DeleteTest ExecuteNonQueryCommit ExecuteNonQueryCommitConParam ExecuteNonQueryRollBack GetRow GetTableConParam GetValueT InsertTest RollbackTest SelectTestWithChildren UpdateDetachTest UpdateTest ZZPulisciDB
Service Name: firebird Server: Firebird	**Service Name: sqlite Server: SQLite**	**Service Name: sql Server: SQLServer**	**Service Name: ingres Server: Ingres**
NhiTest *(14 tests)* CommitTest DeleteTest ExecuteNonQueryCommit ExecuteNonQueryCommitConParam ExecuteNonQueryRollBack GetRow GetTableConParam GetValueT InsertTest RollbackTest SelectTestWithChildren UpdateDetachTest UpdateTest ZZPulisciDB	NhiTest *(14 tests)* CommitTest DeleteTest ExecuteNonQueryCommit ExecuteNonQueryCommitConParam ExecuteNonQueryRollBack GetRow GetTableConParam GetValueT InsertTest RollbackTest SelectTestWithChildren UpdateDetachTest UpdateTest ZZPulisciDB	NhiTest *(14 tests)* CommitTest DeleteTest ExecuteNonQueryCommit ExecuteNonQueryCommitConParam ExecuteNonQueryRollBack GetRow GetTableConParam GetValueT InsertTest RollbackTest SelectTestWithChildren UpdateDetachTest UpdateTest ZZPulisciDB	NhiTest *(14 tests)* CommitTest DeleteTest ExecuteNonQueryCommit ExecuteNonQueryCommitConParam ExecuteNonQueryRollBack GetRow GetTableConParam GetValueT InsertTest RollbackTest SelectTestWithChildren UpdateDetachTest UpdateTest ZZPulisciDB

Tabella 3: risultato dell'esecuzione dei test effettuati sulla classe DbTransaction e su NHIbernate divisi per database

6.h.4 Test sulle transazioni massive con NHIbernate



Come accennato poco fa, grazie all'implementazione generica data ad NHIbernate, è possibile effettuare operazioni di statistica basandosi sulla velocità dei server implementati effettuando l'inserimento di un numero sempre crescente di entità.

Il test presentato in *Figura 36* è stato sviluppato salvando dapprima tutti i nomi di servizio dei server disponibili ed eseguendo in seguito operazioni massive di inserimento su ognuno di loro, utilizzando la classe Timer.cs per monitorare il tempo impiegato per ogni database.

```
[Test]
public void MassiveTestInsert()
{
    string logTime = TestDbInsert(SNOracle);

    logTime += TestDbInsert(SNMySql);

    logTime += TestDbInsert(SNSql);

    logTime += TestDbInsert(SNFirebird);

    logTime += TestDbInsert(SNIngres);

    logTime += TestDbInsert(SNDB2);

    logTime += TestDbInsert(SNPostgreSQL);

    Console.WriteLine(logTime);
}
```

Figura 36: Test di inserimento massivo utilizzando NHIbernate

Com'è possibile notare l'intera esecuzione si basa sulla stringa "logTime" che sarà consecutivamente integrata dall'esecuzione della funzione TestDbInsert la quale, basandosi su una variabile *n* specificante il numero di entità da inserire, procederà a creare in memoria un'istanza di testata, cioè un record della tabella TEST_TABLE, alla quale collegherà via foreign key *n* entità di dettaglio, cioè record della tabella TEST_DETAIL.

Effettuare un'operazione di Commit in questa situazione avrà il compito di trasferire tutti gli oggetti presenti nella memoria all'interno dei rispettivi database. Il fatto che le implementazioni siano identiche permette di effettuare statistica in modo sicuro, sapendo a priori che le uniche differenze di tempistica fra l'inserimento di record in un database dall'altro non può essere imputabile ad altro che ai database stessi.

Si noti inoltre che nella raccolta di risultati non è presente il database SQLite, in quanto essendo un database locale non è stato considerato un caso di studio importante ai fini di questo tipo di statistica.

I risultati dell'esecuzione di questo test con numero di entità crescente da 100 a 1000 con incrementi di 100 viene rappresentato di seguito per mezzo del grafico in *Figura 37.*

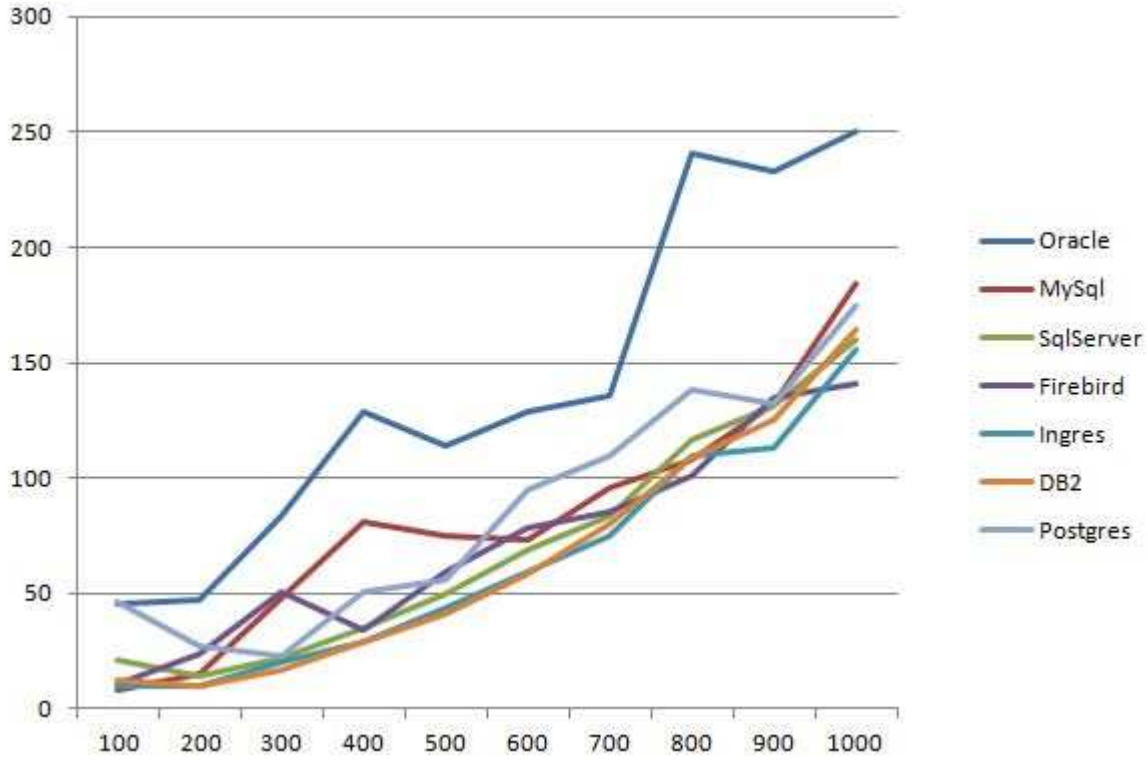


Figura 37: Grafico di prestazioni sotto inserimento massivo dei database implementati

La tendenza, come si può notare, è più o meno comune a tutti i database esaminati. Bisogna però considerare che le versioni dei database utilizzati sono spesso versioni limitate a livello di memoria o numero di processori, dunque i risultati di un test simile sono estremamente variabili in base all'ambiente su cui si effettuano.

Nondimeno, le capacità che ha questa libreria di effettuare statistica sono state confermate.

6.i Esempio di utilizzo

Viene presentato ora un esempio tipico dell'implementazione di questa libreria all'interno di un'applicazione Windows Form. Per implementarla è sufficiente aggiungere ai riferimenti la libreria di classe base (HSystem.Data.dll) e ogni libreria relativa ai database a cui si vuole comunicare come accennato precedentemente nel paragrafo *4*, "Studio di una soluzione".

Per prima cosa è necessario creare una propria classe driver che erediti dalla classe DbBaseProvider, secondo lo schema rappresentato dal diagramma di classe in *Figura 38.*

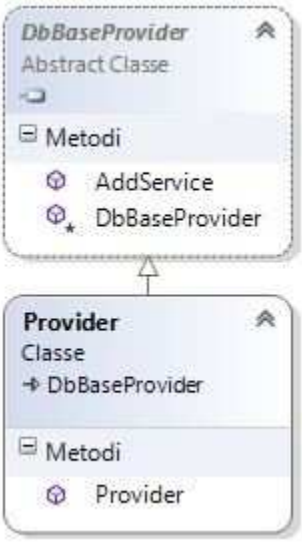


Figura 38: Diagramma di classe dell'implementazione di un Provider Custom

All'interno del costruttore della classe sarà possibile richiamare la funzione della classe base AddService per impostare tutte le informazioni di comunicazione desiderate (*Figura 39*).

```
class Provider : DbBaseProvider
{

    public Provider()
    {
        AddService("mys", DatabaseType.MySql, @"Server=10.201.233.223;Database=TESTDB;Uid=ema;Pwd=tesiema;", OrmType.NHibernate);
    }
}
```

Figura 39: Esempio di costruttore di Provider Custom

Impostato il Provider e le informazioni di comunicazione, nel costruttore del Form dell'applicazione o, più in generale, nel punto di ingresso della nostra applicazione, sarà necessario richiamare il metodo DbDataProvider.Initialize(new Provider()).

In seguito all'inizializzazione del provider sarà possibile utilizzare DbCustom senza altre impostazioni.

Specificando semplicemente il nome del servizio specificato nel provider, infatti, la classe DbDataProvider si occuperà di estrapolare le informazioni di connessione utilizzate per la comunicazione.

Le informazioni di configurazione presentate in questi paragrafi si traducono in un'impostazione di progetto simile a quella mostrata in *Figura 40.*

```
public Form1()
{
    InitializeComponent();
    DbDataProvider.Initialize(new Provider());
}

private void btnInserisci_Click(object sender, EventArgs e)
{
    DbCustom.ExecuteNonQuery("INSERT INTO TEST_TABLE(DSVALUETEST) VALUES('" + txtValue.Text + "')", "mys");
}
```

Figura 40: Esempio di setup di Provider custom e utilizzo

Per quanto riguarda l'utilizzo della DbTransaction, però, non basta impostare le informazioni di comunicazione via servizi. È inoltre necessario crearsi un dominio contenente le informazioni delle tabelle e, nel caso di NHIbernate, un mapping specifico per quella tabella.

La classe di dominio (*Figura 41*) viene realizzata specificando l'oggetto che rappresenterà ogni singolo record della tabella di dettaglio (TEST_DETAIL) mentre la mappatura di NHibernate indicherà in che modo i campi dell'oggetto in dominio debbano essere considerati.

```
namespace UnitTest.Domain
{
    public class DbDetail
    {
        public virtual int Iddetail { get; set; }
        public virtual int Idtest { get; set; }
        public virtual string Dsdetail { get; set; }
        public virtual DbHeader Testata { get; set; }
    }
}
```

Figura 41: Esempio di classe di dominio

Per creare una classe di mapping fruibile dalla libreria occorre un'impostazione del codice che tenga conto del fatto che:

-La classe di mapping deve ereditare dalla classe astratta ClassMapping<T> messa a disposizione di NHibernate.
-Table() è utilizzato per definire il nome della tabella relativo al mapping che si sta realizzando.
-Lazy() indica l'utilizzo del lazy loading. Se true, l'accesso ai dati delle liste di entità collegate all'oggetto considerato saranno effettuati solo a runtime, risparmiando dunque risorse.
-Id() indica il tipo di chiave primaria della tabella. In questo caso generato da identità del database.
-Property() si usa per mappare un semplice attributo. In caso il nome della colonna non fosse specificato, NHIbernate utilizzerebbe automaticamente il nome dell'attributo durante la costruzione delle Query.

-ManyToOne() è utilizzato per indicare una foreign key molti a uno, in questo caso nei confronti della tabella di testata (TEST_TABLE).

Un esempio di mapping NHIbernate per la classe di dominio rappresentata prima in *Figura 41* è mostrato di seguito in *Figura 42*.

```
[NhiClassMap(DbNameMap = DatabaseType.DB2)]
public class TestDetailMapDb2 : ClassMapping<DbDetail>
{
    public TestDetailMapDb2()
    {
        Table("TEST_DETAIL");
        Lazy(true);
        Id(x => x.Iddetail, map => map.Generator(Generators.Identity));
        Property(x => x.Dsdetail);
        ManyToOne(x => x.Testata, map =>
        {
            map.Column("IDTEST");
            map.NotNullable(true);
            map.Cascade(Cascade.None);
        });
    }
}
```

Figura 42: Mapping NHIbernate di una classe di dominio

7. Spunti per il futuro

La complessità e la lungimiranza nella realizzazione di questo progetto ha permesso di lasciare molte possibilità implementative disponibili per il futuro. Come già accennato, una delle tante espansioni possibili all'interno del progetto è quella dell'aggiunta di altri sistemi ORM, quali ad esempio Entity Framework.

Si potrebbe inoltre pensare anche all'aggiunta di nuovi database appoggiandosi o alle implementazioni di Dialect e Driver di NHibernate, o creandole basandosi su implementazioni già esistenti.

Altre modifiche importanti potrebbero essere l'aggiunta di un sistema di esecuzione di stored procedure che permetta di non appoggiarsi all'utilizzo diretto dell'SQL via DbCustom, o l'implementazione di file di configurazione per il setup ed esecuzione degli Unit Test in modo estremamente personalizzabile, o addirittura l'implementazione di un Client esterno a console che permetta la completa gestione dei test implementati.

8. Conclusioni

È stato dunque possibile realizzare una libreria implementata in modo tale da rendere la comunicazione con sistemi di database il più generale possibile. L'implementazione di Repository e il corretto utilizzo delle interfacce ha permesso anche la realizzazione di un progetto scalabile, in grado di rispondere a nuove esigenze di comunicazione senza che il codice funzionale debba essere sensibilmente modificato ogni volta.

La presenza di due punti di ingresso, uno transazionale, ed uno basato sui sistemi di connessione implementato da Microsoft permette inoltre di non essere costretti nell'utilizzo di oggetti sottoperformanti per la propria applicazione. Se sarà necessario gestire una piccola quantità di oggetti in modo sicuro, sarà possibile utilizzare la DbTransaction, mentre in caso di grandi quantità di dati, tali da creare problemi ai moderni sistemi ORM, sarà comunque possibile ricorrere alla DbCustom sfruttando query dirette.

Particolarmente importante, inoltre, è stata la capacità della libreria di sostenere un sistema di analisi prestazionale in grado di mettere a confronto i database utilizzati.

Nel complesso il sistema nato da queste osservazioni è solido e compatto, e sono convinto che nel tempo si rivelerà un importante strumento di sviluppo.

9. Bibliografia

[1] R.Johnson J. Vlissides E. Gamma, R. Helm. *Design Patterns: Elements of Reusable Object Oriented Software.* Addison-Wesley Masson, 1995.

[2] Alex Miller. *Patterns I hate #1: Singleton*, Luglio 2007

[3] Martin Fowler. *Data Mapper*

[4] Beck, K. *Test-Driven Development by Example*, Addison Wesley - Vaseem, 2003

[5] Eric F. Elisabeth F. Kathy S. Bert B. *Head First Design Patterns*, 2004

10. Sitografia

-Data Mapper: *http://martinfowler.com/eaaCatalog/dataMapper.html*

-Resharper: https://www.jetbrains.com/resharper/download/download_thanks.jsp?os=rs

-NuGet: https://www.nuget.org/

MySql :
-Server: http://dev.mysql.com/downloads/file.php?id=453397
-GUI: http://www.heidisql.com/download.php
Oracle :
-Server: http://www.oracle.com/technetwork/database/database-technologies/express-edition/downloads/index.html
-GUI: http://www.oracle.com/technetwork/developer-tools/sql-developer/overview/index.html
SQLite :
-Server+GUI: http://www.sqliteexpert.com/download.html
Microsoft SQL Server:
-Server+GUI: http://www.microsoft.com/it-it/download/details.aspx?id=29062
Firebird:
-Server: http://sourceforge.net/projects/firebird/files/firebird-win64/2.5.3-Release/Firebird-2.5.3.26778_0_x64.exe/download
-GUI: http://code-sd.com/products/turbobird/
PostgreSQL:
-Server: http://www.postgresql.org/download/windows/
-GUI: http://www.sqlmaestro.com/download/
DB2:
-Server: http://www-01.ibm.com/software/data/db2/linux-unix-windows/downloads.html
-GUI: http://www.sqlmanager.net/products/db2/manager
Ingres:
-Server: http://esd.actian.com/#
-GUI: http://razorsql.com/download.html