UNIVERSITÀ DEGLI STUDI DI PARMA

Dipartimento di Matematica e Informatica

Corso di Laurea in Informatica

Tesi di Laurea

Object-Relational Mapping per applicazioni C++

Candidato:

Francesco Trombi

Relatore:

Prof. Enea Zaffanella

Anno Accademico 2013/2014

Indice

Alla mia famiglia,
che mi ha sempre sostenuto.

Introduzione

Un *database* è un insieme organizzato di dati. All'interno di un database le informazioni sono collegate da un modello logico. Un *Relational DataBase Management System* (RDBMS) è un sistema per la gestione di database basato sul modello logico relazionale. Tale modello deriva dal concetto di relazione matematica, ed è stato creato per semplificare l'interrogazione di un database, conservando l'indipendenza dei dati dalla loro rappresentazione fisica.

La *serializzazione* è la conversione dello stato di un oggetto in uno *stream* (una sequenza) di dati. Può essere utilizzata per implementare soluzioni a diversi problemi, tra cui la persistenza. La *persistenza* è la capacità di un oggetto di salvare il proprio stato su un dispositivo di memorizzazione che sopravviva alla terminazione del programma. Per implementare la persistenza è possibile utilizzare diverse tecniche, determinate dall'importanza che si conferisce alla struttura interna dell'oggetto. Se essa è secondaria, può essere implementata utilizzando la serializzazione: ad esempio, salvando lo stream all'interno di un file o all'interno di un database. Sarà poi compito dell'applicativo specifico gestire gli oggetti generando gli stream di dati ed interpretandoli. Se si desidera invece conservare la struttura interna dell'oggetto nel dispositivo di memorizzazione, è necessario utilizzare altri sistemi: per esempio, salvare lo stato dell'oggetto (e delle sue componenti) in un database relazionale, che oltre al permettere una semplice consultazione degli oggetti contenuti, consente la creazione di associazioni tra di essi. In questo caso, il problema si sposta sulla ricerca di meccanismi che garantiscano la persistenza degli oggetti salvandoli in un database, magari in automatico.

L'*Object-Relational Mapping* (ORM) è una tecnica di programmazione che permette di integrare programmi aderenti al paradigma della programmazione orientata agli oggetti e RDBMS. Le soluzioni ORM dunque forniscono interfacce per la persistenza degli oggetti del linguaggio di programmazione, garantendo l'astrazione dal tipo di DBMS utilizzato.

L'obiettivo del lavoro di tesi presentato è dunque quello di studiare e confrontare le soluzioni adottate in alcune piattaforme per l'ORM; in altre parole

la comprensione dei meccanismi che queste utilizzano per implementare la persistenza degli oggetti. Nello specifico, ci si è concentrati su soluzioni che supportano la programmazione nel linguaggio C++. Si effettuerà dunque un confronto tra alcune soluzioni disponibili, cercando di stilarne una classificazione in base a diverse caratteristiche; infine una di queste in particolare verrà approfondita. Le qualità utilizzate per categorizzare le diverse soluzioni sono state scelte coerentemente con gli obiettivi preposti: semplicità d'uso, legame con il linguaggio C++, estendibilità del prodotto, tipologia di licenza e di accesso al codice sorgente.

Alcune soluzioni utilizzano anche strumenti ad interfaccia grafica per mappare gli oggetti in un database relazionale. Questo per sottolineare che ogni soluzione avrà il proprio approccio per affrontare il problema; in questo lavoro verranno elencati i più interessanti, secondo i criteri prefissati.

Il primo capitolo si pone lo scopo di fare una panoramica sulle soluzioni disponibili per l'Object-Relational Mapping, cercando di individuare le categorie in cui si collocano ed elencando le differenze che le caratterizzano. Verrano descritte alcune soluzioni, che si sono distinte come esplicative ed utili per questo lavoro.

Nel secondo ci si concentrerà maggiormente sulla valutazione dei meccanismi che permettono la persistenza degli oggetti, rimanendo indipendenti da IDE o sovrastrutture. Sono state infatti scelte due soluzioni che nel primo capitolo si sono distinte maggiormente, che verranno confrontate sotto diversi aspetti, anche mediante alcuni semplici esempi pratici.

Il terzo capitolo si concentra sull'analisi dettagliata di una delle soluzioni, presentata nel primo capitolo e confrontata nel secondo; tale analisi mostrerà l'implementazione concreta dei meccanismi scelti per la gestione della persistenza degli oggetti e le possibili modalità di estensione per aumentare il supporto ai tipi del linguaggio.

Nelle conclusioni si parlerà dei risultati ottenuti e delle possibili estensioni future al lavoro svolto.

Capitolo 1

Soluzioni per l'ORM

Questo capitolo si concentra sulla ricerca di una suddivisione in categorie delle soluzioni ORM presenti sul mercato; da qui, si affronterà una panoramica delle soluzioni principali, mostrandone pregi e difetti.

1.1 Tipologie di soluzioni ORM

Le soluzioni ORM esistenti sul mercato sono numerose, ognuna con il proprio approccio al problema e con le proprie peculiarità. Per poter avere una visione chiara dello situazione odierna dei prodotti ORM, è necessario introdurre una loro categorizzazione, non dissimile dalle modalità di suddivisione dei prodotti software in generale. I prodotti ORM, infatti, devono essere considerati sotto diversi punti di vista, che vanno dal tipo di codice sorgente all'eventuale costo di una licenza. I criteri di categorizzazione delle soluzioni ORM si basano sulle seguenti caratteristiche:

- linguaggio di programmazione a cui si interfacciano e con cui sono state scritte;
- licenza di distribuzione del prodotto;
- accessibilità del codice sorgente;
- integrazione con altre componenti software.

Il linguaggio a cui si interfaccia una soluzione ORM è un fattore non secondario per la scelta del prodotto al quale affidarsi: esistono soluzioni per molti linguaggi, per esempio C++ (standard e dialetti), Java, linguaggi .NET, Objective-C, etc. Per alcuni utenti sarà per esempio meglio lavorare con alcuni dialetti C++, mentre altri vorranno uniformarsi allo standard.

Il codice sorgente può essere o *open source* o *closed source* (codice aperto o chiuso); il codice open è disponibile alla visione da parte di chiunque; quello closed non è visibile e tipicamente è proprietario dell'azienda produttrice. L'accessibilità del codice sorgente serve tipicamente per controllare la qualità del prodotto e per poter estendere il prodotto stesso. Nel caso del codice proprietario, non è solitamente possibile fare estensioni.

Le licenze di distribuzione del prodotto possono essere principalmente di due tipi: licenze proprietarie e licenze libere. Se si utilizza un software con licenza proprietaria per sviluppare un qualunque prodotto, tipicamente chi comprerà tale prodotto dovrà acquistare la licenza d'uso del software con cui è stato sviluppato. Un software proprietario, anche se distribuito con codice open source, spesso non è modificabile da parte dell'utente. Una licenza di distribuzione libera permette invece la modifica del prodotto e un'eventuale ridistribuzione sul mercato, in forma gratuita.

L'integrazione con altre componenti software è una caratteristica importante nell'analisi dei sistemi ORM: infatti alcune soluzioni ORM vengono distribuite insieme ad *Integrated Development Environment* (IDE, un software per l'ausilio allo sviluppo del codice sorgente) o come parte di *framework* (architettura logica di supporto allo sviluppo software). Alcune di queste possono poi lavorare in maniera indipendente, con gradi differenti di libertà.

Sul mercato sono presenti soluzioni che implementano tali caratteristiche nelle modalità più differenti, coprendo vari scenari. Per questo lavoro di tesi, sono state prese in considerazione diverse discriminanti, che hanno permesso di dividere il vasto insieme delle soluzioni ORM in quelle adatte e in quelle non.

La discriminante principale è stata la decisione iniziale di interfacciarsi con il linguaggio C++. Un altro fattore importante per la scelta è che la soluzione abbia un'elevata flessibilità, quindi limitate dipendenze a piattaforme specifiche o ad altro software; come conseguenza si ha che spesso sono stati scartati prodotti fortemente integrati con le componenti software con cui vengono distribuiti, nonostante siano quelli più avanzati sul mercato. Questo anche perché le soluzioni che offrono un'integrazione avanzata con altre componenti spesso non riescono a lavorare correttamente al di fuori di tale ambiente. Si prenda come esempio il software Apple Core Data, usato per la persistenza degli oggetti sui dispositivi iOS e Mac OS X, che supporta solamente il linguaggio proprietario Apple, Objective-C. Verranno comunque descritte alcune delle caratteristiche della soluzione per ORM disponibile sulla piattaforma .NET di Microsoft, per mostrare le peculiarità di una soluzione forte sul mercato, legata però indissolubilmente ad un IDE.

Le soluzioni indipendenti da altre componenti software sono spesso fornite in maniera libera, con codice consultabile e modificabile. Mostrano una

grande flessibilità, permettendo anche allo sviluppatore di non "sporcare" eccessivamente il codice con componenti diverse da quelle del C++ standard. Purtroppo non sempre sono supportate a livello di sviluppo e documentazione, in quanto vengono create nell'ambiente del software libero, spesso gratuito. Verranno però descritte nel dettaglio, in quanto più adatte a questo lavoro di tesi.

1.2 Soluzioni ORM sul mercato

Le soluzioni su mercato sono numerose e, come detto in precedenza, delle più diverse tipologie. Possono essere gratuite o a pagamento, con codice open o closed, e distribuite insieme ad altri prodotti. Nel seguito si farà una panoramica di alcune di esse, sottolineando in questo modo pregi e difetti dei diversi approcci all'ORM.

Microsoft Entity Framework

Entity Framework è un mapper relazionale a oggetti creato per gli sviluppatori .NET, open source e con licenza Apache (libera). Il prodotto è semplice ed efficiente, in grado di istanziare ed interrogare un database utilizzando poche righe di codice: la maggior parte del lavoro viene infatti svolta dal framework e da Visual Studio, l'IDE fornito dagli sviluppatori Microsoft. Visual Studio permette di osservare la struttura del database a diversi livelli di dettaglio, grazie ad un comodo menu a tendina; questa integrazione viene però garantita a scapito dell'indipendenza, in quanto Entity Framework non può essere utilizzato al di fuori di Visual Studio. Offre quattro tipologie di approccio al problema della persistenza dei dati:

- *Code First*: si scrive il codice sorgente e da esso si genera il database relazionale; tale database è aggiornabile dal codice in qualunque momento, poiché Visual Studio permette di comunicare le modifiche tramite un'apposita console testuale;
- *Code First to an existing database*: consiste nell'aprire un database già esistente, creando una connessione ad esso ed importando le tabelle desiderate; da esse viene infine generato il codice sorgente corrispondente;
- *Model First*: si progetta graficamente un modello entità-relazione tramite Entity Framework Designer, che viene tradotto in codice SQL; tale codice viene poi eseguito, creando il database;

- *Database First*: consiste nel partire da un database esistente e generare da esso il codice sorgente di interfacciamento tra il linguaggio e il database stesso, oltre che crearne il modello grafico.

In altre parole, gli approcci suggeriti da MS Entity Framework sono così suddivisi:

	Graphic Editor	**Code**
New Database	Model First	Code First
Existing Database	Database First	Code First (existing db)

QxOrm

Il supporto ORM fornito dagli sviluppatori del framework Qt viene rilasciato con licenza libera GNU GPL v3, oppure con la licenza proprietaria QxOrm Proprietary License. Qt è una libreria multipiattaforma per lo sviluppo di programmi con interfaccia grafica. Viene usata per il C++ standard, anche se lo arricchisce con componenti custom. QxOrm è scritta in C++ ed è costituita da due componenti principali:

- QxOrm: libreria ORM per sviluppatori C++/Qt;
- QxEntityEditor: editor grafico della libreria QxOrm.

Per entrambi l'IDE di Qt non è necessario, ma è fondamentale aver installato il framework Qt. QxOrm fornisce le seguenti funzionalità:

- persistenza: comunicazione con il database;
- serializzazione: binaria e XML;
- riflessione e introspezione: accesso alle informazioni delle classi.

L'editor QxEntityEditor è multipiattoforma: può essere utilizzata per i principali sistemi operativi desktop (Windows, Linux, Mac OS X), embedded (Raspberry Pi) e mobile (Android, iOS). Esso, oltre che permettere una generazione rapida di modelli E-R, permette di visualizzare graficamente le associazioni che intercorrono tra le varie entità: purtroppo la limitazione dovuta al mancato acquisto della licenza consiste nel non poter utilizzare modelli con più di cinque entità, limitando decisamente le potenzialità dello strumento. Il prodotto supporta i seguenti DBMS: SQLite, PostgreSQL, MySQL, Oracle e MS SQL Server. Di questa libreria si avrà una descrizione più dettagliata nel prossimo capitolo, poiché è stata scelta come rappresentante della sua categoria nel confronto diretto tra le soluzioni sul mercato. Necessita di alcune componenti delle librerie Boost, anch'esse elencate nel capitolo successivo.

ODB

Open source, cross-platform, cross-database. Viene rilasciato sotto la licenza GNU GPL v2. Questa libreria *stand alone* permette di mappare oggetti C++ in un database relazionale senza manipolare nemmeno una riga di SQL. Supporta i seguenti DBMS: MySQL, SQLite, PostgreSQL, Oracle e Microsoft SQL Server. La libreria può essere utilizzata sia con lo standard C++98/03, sia con quello C++11. Inoltre la libreria viene fornita con moduli appositi per l'editor Qt e la libreria Boost, in particolare per quanto riguarda *value types*, *containers*, e *smart pointers* (queste componenti verranno descritte nel capitolo successivo). Osservando il codice di esempio fornito dal sito del produttore, passare da una semplice classe C++ ad una comprensibile dalla libreria ODB è molto semplice: basta aggiungere poche righe di codice. Questa libreria è stata però scartata per i seguenti motivi:

- tutorial non funzionante presentato nel sito, con comandi di compilazione errati;
- uso di un "pre-compilatore" che legge le annotazioni nel linguaggio della libreria e le traduce in codice C++.

LiteSQL

LiteSQL è una libreria C++ che mappa oggetti del linguaggio in un database relazionale. L'ultima versione del software è però la 0.3.5, il che suggerisce una certa instabilità del prodotto. LiteSQL garantisce supporto ai principali DBMS: SQLite3, PostgreSQL e MySQL. LiteSQL è in grado di gestire il database, per esempio creando tabelle ed aggiornando lo schema. Oltre alla persistenza degli oggetti, LiteSQL fornisce relazioni tra oggetti, che possono essere usati per modellare ogni struttura dati del C++. Gli oggetti possono essere selezionati, filtrati ed ordinati utilizzando API basate sulle classi e sui template, con il *type checking* a tempo di compilazione.

Wt::Dbo

La libreria Wt::Dbo è la componente per la persistenza dei dati presente nel framework Wt, creata per lo sviluppo di applicazioni web in C++. Supporta i seguenti DBMS: SQLite, MySQL, Firebird e PostgreSQL. Viene rilasciata con licenza libera (GNU GPL) e proprietaria. La descrizione effettiva del funzionamento di questa libreria verrà espansa nei capitoli seguenti, in quanto Wt::Dbo è stata reputata, in base ai criteri esposti precedentemente, tra le più adeguate al raggiungimento degli obiettivi prefissati. I pregi principali di

questa libreria stanno nella semplicità: semplicità del codice sorgente, semplicità d'uso, semplicità di estensione. È infatti in grado di rendere gli oggetti persistenti con poche righe di codice, semplici da comprendere e replicare; a questa caratteristica unisce la semplicità del proprio codice sorgente, che ha permesso un'ampia comprensione dei meccanismi interni che regolano la gestione del salvataggio degli oggetti del C++ all'interno di un database.

Capitolo 2

Confronto tra soluzioni

Mentre nel capitolo precedente si discuteva sulle tipologie di soluzioni da utilizzare, in questo si cercherà di definire con precisione quali di esse rispondono meglio alle caratteristiche da noi ricercate, che sono:

- codice sorgente open source;
- licenza di distribuzione libera;
- maggior indipendenza possibile da un IDE, o da altre componenti software.

Si vedranno dunque in dettaglio due delle librerie accennate in precedenza, Wt::Dbo e QxOrm, prima separatemente, valutando compatibilità e caratteristiche generali; poi confrontandole direttamente con l'ausilio di un esempio pratico.

2.1 Descrizione delle soluzioni scelte

Prima di mostrare l'esempio scelto per sottolineare le differenze tra le due soluzioni, è necessario descriverle entrambe, aggiungendo profondità alla descrizione fatta nel capitolo precedente.

2.1.1 Caratteristiche di Wt::Dbo

Wt::Dbo è una libreria ORM C++ distribuita come parte di Wt (*Web Toolkit*) per la costruzione di applicazioni web basate su database; funziona però anche al di fuori di questo contesto. Wt, infatti, viene principalmente utilizzata per applicazioni web che necessitano di comunicare con librerie e/o applicazioni desktop C++; Wt::Dbo ne è una componente importante, che

fornisce supporto ORM anche ad applicazioni che non hanno nulla a che fare con il web. Wt, e dunque Wt::Dbo, viene fornita in due modalità: con licenza GNU GPL oppure con licenza commerciale. La licenza commerciale permette l'utilizzo di driver per il DBMS Oracle, che nella versione gratuita non è supportato.

Wt garantisce l'utilizzo della componente ORM al di fuori del resto del framework; quindi Wt::Dbo può essere utilizzata per qualunque tipologia di applicazione C++ a cui si voglia aggiungere supporto alla persistenza degli oggetti.

Wt::Dbo garantisce una visione del database basata sulle classi che la generano; le tabelle del database mantengono infatti la gerarchia delle classi C++. La libreria permette di gestire anche la sincronizzazione tra codice sorgente e database, permettendo di inserire, aggiornare e cancellare record. Le classi vengono mappate in tabelle; i dati membro di una classe, se sono di uno dei tipi di dato supportati di default dalla libreria, sono mappati in corrispondenti colonne nella tabella corrispondente alla classe; i puntatori e le collezioni di puntatori sono mappati in associazioni logiche, la cui integrità all'interno del database è controllata tramite vincoli di chiave esterna. Nel caso si presenti un tipo non supportato, le alternative sono due: estendere il supporto al nuovo tipo di dato, affinché venga mappato in una sola colonna, oppure mappare tale tipo in una nuova tabella, utilizzando un puntatore per collegarlo alla classe che lo contiene. Un oggetto di una classe mappata nel database viene chiamato *database object* (da qua il nome della libreria, *Dbo*). I risultati delle query possono essere definiti in termini di database object, primitivi o tuple di essi. Non si usa l'XML come rappresentazione intermedia nella traduzione da C++ a database e viceversa. Wt::Dbo, nella versione gratuita supporta i seguenti DBMS:

- Firebird;
- SQLite3;
- MySQL;
- PostgreSQL.

Per familiarizzare con il prodotto si è deciso di creare alcune classi C++ elementari, che contenessero costrutti supportati dalla libreria, ma che ci permettessero di comprendere come tale supporto venisse affrontato. I test effettuati rigurdano la gestione dei seguenti tipi di dato del C++:

1. interi di macchina;

2. enumerazioni `enum`;
3. `bool`;
4. `float` e `double`, verificando la consistenza delle approssimazioni tra C++ e database.

Interi di macchina

Con i tipi interi di macchina non ci sono stati problemi: il numero `INT_MAX` viene memorizzato correttamente nel database. Il difetto maggiore si è riscontrato sui tipi `unsigned`, non supportati dalla libreria. Il motivo è dovuto al fatto che molti DBMS non supportato i tipi interi senza segno; spesso si punta ad implementare vincoli (check) di non negatività su attributi di tipo intero con segno. Chi ha sviluppato la libreria ha quindi reputato che, nel caso, l'utente può convertire gli interi senza segno in interi con segno. Per questo il lavoro di tesi si è sviluppato anche in quella direzione, cioè all'estensione del supporto ad altri tipi di dato ritenuti interessanti.

Enumerazioni

La libreria supporta correttamente le enumerazioni, trasformandole in un campo intero all'interno del database.

Booleani

La libreria si comporta bene con i tipi di dato booleani; il booleano in SQLite segue la convenzione del C di interpretare `0` come `false` e tutto il resto come `true`. Per verificare che qualunque cosa venga presa come vera se non l'intero `0`, si è inserito nel campo booleano sia `INT_MAX`, sia una stringa, sia un carattere. Il dato rimane consistente sul database, dove viene sempre salvato come numero intero `1`.

Numeri in virgola mobile

I numeri in virgola mobile presentano un problema che nasce dall'approssimazione durante la stampa a video: nel database i valori vengono salvati nel formato corretto, mentre nella stampa a video vengono troncati. Prendendo come test un numero periodico, si nota come il database salvi correttamente sia il `float`, che il `double`, stampandoli a video tramite le funzionalità offerte dalle librerie standard del C++. I numeri in virgola mobile del C++ hanno cinque valori speciali:

1. `inf`, ottenibile da `1.0/0.0`;
2. `-inf`, ottenibile da `-1.0/0.0`;
3. `nan`, ottenibile da `0.0/0.0`;
4. `0+`, ottenibile da `1.0/+inf`;
5. `0-`, ottenibile da `1.0/-inf`.

Per quanto riguarda i primi due valori, `inf` e `-inf`, la libreria riesce a memorizzarli e ad estrarli correttamente dal database, mantenendo la nomenclatura `inf` e `-inf`. Con i valori `0+` e `0-` invece, viene sempre memorizzato all'interno del database il numero `0` (senza alcun segno), rappresentando di fatto questi due valori differenti con una sola costante numerica. Il valore speciale `nan` viene invece salvato come `NULL`; se il campo corrispondente però ha il vincolo `NOT NULL`, durante il tentativo di inserimento viene lanciata un'eccezione per la violazione di tale vincolo.

Come detto in precedenza, Wt::Dbo necessita di alcune componenti delle librerie Boost, in particolare:

- Algorithm;
- `any`;
- `enable_if` e `disable_if`;
- `function`;
- `iterator_range`;
- `lexical_cast`;
- `multi_index_container`;
- `optional`;
- Phoenix;
- Posix;
- Spirit;
- Thread;
- `tuple`;
- `type_traits`.

Tali componenti verranno illustrate in seguito.

2.1.2 Caratteristiche di QxOrm

Il supporto ORM fornito dagli sviluppatori del framework Qt si basa su due componenti principali: QxOrm, la libreria ORM vera e propria, nucleo delle operazioni cardine per garantire la persistenza degli oggetti; e QxEntityEditor, un editor grafico che permette la progettazione grafica di modelli E-R, traducibili automaticamente in database relazionali e codice C++ grazie alle funzionalità di QxOrm. Qt è un framework per lo sviluppo di applicazioni C++, basato su quattro componenti:

1. GUI;
2. network;
3. XML;
4. database.

La libreria QxOrm si colloca nella quarta componente, fornendo le seguenti funzionalità:

- persistenza: comunicazione con il database, con costruzione di tabelle e associazioni (1-1, 1-N, N-N);
- serializzazione: binaria e XML;
- riflessione e introspezione: accesso alle definizioni delle classi, per ricavarne proprietà e chiamarne i metodi.

QxEntityEditor è un editor grafico per QxOrm: fornisce una modalità grafica di creazione e gestione del modello dei dati. Tale editor è multipiattaforma, in grado di generare codice per le principali famiglie di sistemi operativi desktop (Windows, Linux, Mac OS X), per i maggiori sistemi operativi mobili (Android e iOS) ed infine per alcuni sistemi embedded (come Raspberry Pi, su cui vengono istallate distribuzioni Linux particolari).

QxOrm e QxEntityEditor offrono una gamma migliore di DBMS supportati rispetto a Wt::Dbo, permettendo l'interazione con due prodotti importanti come quello di Miscrosoft e Oracle. Supporta dunque: SQLite, MySQL, PostgreSQL, Oracle e MS SQL Server.

QxEntityEditor è basato su plugin e fornisce diverse modalità per importare/esportare i modelli dei dati:

- generazione di classi C++ persistenti;
- generazione di script DDL SQL per i DBMS supportati;

- gestione dell'evoluzione dello schema per ogni versione del progetto;
- trasferimento del modello attraverso la rete, con la creazione di applicazioni client-server;
- importazione della struttura di database esistenti, tramite una connessione ODBC, per i DBMS supportati.

L'uso di QxOrm presenta diversi vantaggi: innanzitutto è una libreria non intrusiva, che dunque può essere utilizzata per progetti già esistenti, entrando nel processo di sviluppo anche durante le fasi intermedie. Un altro vantaggio è l'assenza del meta-linguaggio XML per garantire la persistenza dei dati. Come spesso accade in altre meccaniche di programmazione, gli oggetti persistenti non devono derivare da un *"super-oggetto"*, che vincola ulteriori inclusioni per la creazione di un progetto, rallentando il lavoro dello sviluppatore. È stato inoltre implementato con il *template meta-programming*, una tecnica di meta-programmazione che consiste nel far utilizzare i template ad un compilatore per generare codice sorgente temporaneo, che viene fuso dal compilatore stesso con il resto del codice e poi compilato insieme. Infine, QxOrm offre le seguenti compatibilità:

- Visual C++, dalla versione 2008 (Windows);
- `gcc` (Linux);
- `clang` (Mac OS X);
- MinGW (Windows).

QxOrm e QxEntityEditor necessitano di alcune componenti aggiuntive: parti della libreria Boost vengono infatti utilizzate, come in Wt::Dbo, per le funzionalità più diverse; inoltre si basano fortemente sul framework Qt. Le componenti aggiuntive sono fondamentali, ed inoltre rappresentano una grossa fetta del tempo e dello spazio necessari all'installazione dei prodotti.

Le librerie Boost contengono un'ampia gamma di utility importanti per sviluppare in C++, tanto che alcune di esse sono state incluse nello standard C++11. Nel caso di QxOrm, sono necessarie per le seguenti funzionalità:

- `smart_pointer`;
- componenti per la serializzazione;
- libreria `type_traits`;
- `multi_index_container`;

- contenitori non ordinati;
- `any`;
- `tuple`;
- `foreach`;
- `function`.

Tali componenti verranno spiegate nel dettaglio successivamente. QxEntityEditor permette di avere due approcci al problema della persistenza: il primo si basa sulla generazione grafica di un modello E-R grafico, che poi viene esportato in codice C++ e SQL. Durante l'esportazione in codice SQL permette all'utente di scegliere come associare i tipi del C++ in SQL. Il secondo approccio consiste nell'importare un database all'interno dell'editor grafico, per poterlo manipolare più facilmente, e generare da esso codice C++ e modello grafico.

La generazione del codice C++ avviene grazie a QxOrm, il nucleo della suite: le classi vengono localizzate all'interno della directory `include/`, con *getter*, *setter*, costruttore di default, costruttore che prende come parametro la chiave primaria dell'entità associata alla classe e distruttore virtuale. Un difetto del prodotto è che si basa fortemente sui tipi di dato presenti nelle librerie di Qt, come per esempio `QString`, utilizzati diffusamente al posto dei tipi standard del C++.

2.1.3 Componenti della libreria Boost

Boost è un insieme di librerie C++ contenenti diversi tool per gli utilizzi più differenti. Alcune delle sue componenti sono entrate a far parte delle versioni più recenti della libreria standard del C++.

Algorithm Library

È una libreria che contiene una collezione di algoritmi di ogni tipo. Le componenti presenti in questa libreria utilizzate dalle soluzioni ORM prese in esame sono:

- String: questa libreria fornisce un'implementazione generica agli algoritmi sulle stringhe che mancano nella STL;
- `any_of`: contiene quattro varianti dell'algoritmo `any_of`, che testa gli elementi di una sequenza e ritorna `true` se uno qualunque degli elementi ha una particolare proprietà;

- `find`: trova il primo elemento di un dato tipo all'interno di una sequenza.

`any`

Nella programmazione basata sui template a volte potrebbe essere utile un tipo "generico": variabili che sono solamente variabili, accettanti valori differenti da quelli offerti dai soli tipi C++. Esistono tre categorie di tipi "generici":

1. tipi generici per convertire liberamente un valore, per esempio un intero in stringa; per supportare tale modalità di conversione la libreria fornisce il `lexical_cast`;

2. tipi che contengono valori di differenti tipi, ma non permettono conversioni tra di essi; di fatto un modo sicuro per garantire la conservazione del valore, senza che venga sporcato da operazioni che introducono errore;

3. tipi indiscriminati che posso riferirsi a qualunque cosa, mantenendo comunque un riferimento verso il tipo sottostante, assicurando ogni forma di accesso ed interpretazione al programmatore.

La classe `any` supporta la copia di ogni tipologia di valore e la lettura sicura dei valori. Vengono inoltre fornite altre componenti ausiliarie, quali `any_function`, un adattatore generalizzato per funzioni, ed `any_iterator`, un adattatore generalizzato per iteratori.

Contenitori non ordinati

La libreria Boost fornisce classi di contenitori non ordinati: `unordered_set` e `unordered_multiset`. Queste classi sono utili per avere contenitori rapidi da scorrere.

`enable_if` e `disable_if`

`enable_if` e `disable_if` sono costrutti templatici utilizzati nella metaprogrammazione di template, per controllare (cioè abilitare o disabilitare) l'istanziazione di funzioni e classi templatiche, a seconda del verificarsi o meno di alcune condizioni sui parametri del template, controllate a tempo di compilazione. Per esempio è possibile definire funzioni templatiche che vengono abilitate se e solo se il tipo templatico corrisponde ad un insieme di tipi designati.

`foreach`

Il costrutto `foreach` della libreria Boost permette di automatizzare le funzionalità dell'algoritmo `std::for_each()`: dunque scorre le sequenze liberando il programmatore dall'utilizzo di iteratori e dalla scrittura di predicati appositi. Il costrutto `foreach` non usa allocazione dinamica, puntatori a funzione, funzioni virtuali; non effettua chiamate non riconosciute dall'ottimizzatore del compilatore. Garantisce ottime prestazioni, separandosi di poco da quelle dei cicli scritti a mano.

`function`

La libreria Boost.Function contiene una famiglia di classi templatiche, che sono di fatto *wrapper* per oggetti funzione. Condivide funzionalità con i puntatori a funzione. Ogni oggetto funzione (o puntatore a funzione) può essere compatibile: cioè ogni argomento dell'interfaccia designata da Boost.Function può essere convertita negli argomenti dell'oggetto funzione bersaglio.

`iterator_range`

È una classe che implementa l'incapsulamento di due iteratori, affinché corrispondano al concetto di *Forward Range*. Se l'argomento templatico non è un modello di *Forward Traversal Iterator*, è possibile comunque usare un sottoinsieme dell'interfaccia.

`lexical_cast`

A volte viene utile poter convertire tipi in stringhe e viceversa, soprattutto quando c'è una differenza tra la rappresentazione del valore all'interno del programma e quella all'esterno di esso (per esempio, come viene presentato un valore nel codice sorgente e come viene scritto nell'interfaccia grafica del programma). Le librerie standard C++ offrono un buon numero di modalità per compiere questa conversione, che differiscono in semplicità d'uso, estendibilità e sicurezza. Le funzioni templatiche di `lexical_cast` offrono una modalità comoda e coerente per gestire conversioni comuni da e per tipi arbitrari, quando vengono rappresentati sotto forma testuale. Per conversioni più complesse gli sviluppatori della libreria Boost suggeriscono di utilizzare `stringstream`; per conversioni numeriche suggeriscono invece un altro componente della libreria, `numeric_cast`.

`multi_index_container`

È una classe che permette la costruzione di contenitori in grado di mantenere più indici, con ordinamento e semantiche di accesso diverse. Gli indici forniscono interfacce simili a quelle dei contenitori STL, rendendo il loro utilizzo familiare. Il concetto di "multi-indice" sulla stessa collezione di elementi è preso dai database relazionali, e permette l'implementazione di strutture dati complesse nello spirito delle tabelle relazionali ad indici multipli, dove insiemi e mappe non bastano. Viene fornita un'ampia gamma di indici, modellati come i loro corrispondenti nella STL: `std::set`, `std::list` e insiemi hash.

`optional`

Questa classe templatica è un *wrapper* per rappresentare oggetti opzionali (o "*nullable*") che fino ad un certo momento possono non contenere un valore valido. Gli oggetti opzionali offrono molte funzionalità: possono essere utilizzati per il passaggio per valore e all'interno dei contenitori della STL.

Phoenix

Phoenix è una libreria che permette di utilizzare teniche aderenti al paradigma della programmazione funzionale.

Posix Time

Questa porzione della libreria Boost definisce un sistema di tempo non uniforme con risoluzione a nano/micro-secondi e proprietà di calcolo stabile. Questo sistema di tempo usa il calendario gregoriano per implementare la rappresentazione del tempo. Tramite funzioni apposite, è possibile convertire un tipo di dato temporale in stringhe di diverso formato. Le componenti principali sono:

- `ptime`: interfaccia principale per la manipolazione delle porzioni di tempo;
- `time_duration`: tipo base per la rappresentazione di una lunghezza di tempo;
- `time_period`: fornisce una rappresentazione diretta per la rappresentazione di un range tra due tempi;
- `time_iterator`: fornisce un meccanismo per iterare attraverso il tempo; assomigliano agli iteratori bidirezionali.

`smart_pointers`

Sono oggetti che memorizzano puntatori ad oggetti allocati dinamicamente (nell'heap). Si comportano come puntatori C++ classici, tranne per il fatto che eliminano automaticamente l'oggetto puntato al momento opportuno. Sono particolarmente utili, poiché in caso di eccezioni assicurano una distruzione appropriata degli oggetti allocati dinamicamente. Possono essere usati anche per tenere traccia degli oggetti allocati dinamicamente, nel caso siano condivisi tra più proprietari.

Spirit

Spirit è una libreria C++ che fornisce funzionalità per il parsing. È object-oriented e ricorsiva discendente. Permette di scrivere grammatiche usando un formato simile a quello EBFN (*Extended Backus Naur Form*) direttamente in C++. Si basa sul tool di parsing Qi.

Thread

Questa libreria permette l'utilizzo di diversi thread di esecuzione, con dati condivisi. Fornisce metodi e classi per la gestione dei thread stessi, della loro interazione con gli altri per la sincronizzazione dei dati o per la gestione di copie separate di dati. Contiene diverse classi, tra le quali compare `mutex`, usata per l'accesso sincronizzato a risorse condivise.

`tuple`

Una tupla o (n-tupla) è una collezione di elementi a dimensione fissa. In un linguaggio di programmazione una tupla è un oggetto che contiene altri oggetti come elementi. Tali elementi possono essere di tipo uguale o diverso. Le tuple mostrano la loro importanza in diverse circostanze. Per esempio sono utili per definire funzioni che ritornano più di un valore. Per compensare l'assenza delle tuple nel C++, la Boost Tuple Library le implementa utilizzando i template.

`type_traits`

È una parte della libreria Boost contenente un insieme di specifiche classi *traits*, ognuna delle quali incapsula una singola caratteristica dal sistema dei tipi C++. Permettono per esempio di capire se un tipo è un puntatore, se ha un costruttore base, etc.

Queste classi contengono un modello unico: ogni classe eredita dal tipo `true_type` se il tipo ha la proprietà specifica; altrimenti eredita da `false_type`. `type_traits` contiene inoltre un insieme di classi capaci di modificare i qualificatori: per esempio è possibile rimuovere il qualificatore `volatile` da un tipo. Ogni classe che esegue tali trasformazioni definisce un unico membro `typedef` che rappresenta il risultato della trasformazione. Al suo interno contiene costrutti che vengono utilizzati esplicitamente nelle librerie prese in esame:

- `is_base_of <A, B>`: eredita da `true_type` se `A` compare in un qualunque punto dell'albero di derivazione di `B`; altrimenti, eredita da `false_type`;
- `is_enum <T>`: se `T` è un tipo enumerazione, eredita da `true_type`; altrimenti da `false_type`;

2.2 Wt::Dbo vs QxOrm

L'esempio che si è deciso di costruire è classico ma esplicativo, basato su tre classi: `Studente`, `Iscrizione`, `Corso`. Si è deciso di utilizzare come backend il DBMS SQLite. Per mostrare al meglio il funzionamento delle due librerie, i passi per lo svolgimento della prova sono stati così pensati:

1. scrittura del codice C++ con l'ausilio della libreria Wt::Dbo;
2. esecuzione del programma generato da tale codice, con conseguente creazione del database;
3. estrazione del codice SQL dal database creato, utilizzando il comando di SQLite3 `.schema`;
4. creazione di un progetto QxEntityEditor tramite importazione da database SQLite3;
5. generazione di codice SQL e C++ tramite i tool offerti da QxEntityEditor e QxOrm.

2.2.1 Esempio: Studenti

L'esempio si basa dunque su tre classi. Le classi scelte vogliono essere un modello che coinvolga uno studente, un corso di laurea. Questi due oggetti saranno legati da un'iscrizione, che conterrà i dati dello studente e del corso di laurea. Sono dunque presenti i seguenti vincoli:

- uno studente può avere diverse iscrizioni, ma non allo stesso corso nello stesso anno accademico;
- un'iscrizione deve essere relativa ad un solo studente e ad un solo corso di laurea;
- un corso può ricevere più iscrizioni.

La classe `Studente` ha una matricola intera, un nome e un cognome testuali; inoltre possiede una "collezione" di iscrizioni. La classe `Corso` ha un codice intero, un nome testuale ed una tipologia, implementata tramite un'enumerazione: il numero `0` rappresenta la "Laurea Triennale", il numero `1` la "Laurea Magistrale" ed il numero `2` il "Dottorato"; possiede inoltre una "collezione" di iscrizioni. La classe `Iscrizione` ha un anno accademico intero (per semplicità); contiene inoltre un "puntatore" ad uno studente ed un "puntatore" ad un corso. Nel codice di esempio si è deciso di effettuare un inserimento ed una modifica di un record del database, mostrando come tali operazioni vengono implementate con l'ausilio della libreria Wt::Dbo.

```
1  #include <iostream>
2  #include <string>
3  #include <iostream>
4  #include <Wt/Dbo/Dbo>
5  #include <Wt/Dbo/backend/Sqlite3>
6  #include <string>
7
8  namespace dbo = Wt::Dbo;
9
10 class Iscrizione;
11 class Studente;
12
13 class Corso {
14   public:
15   int codice;
16   enum Tipo {
17     Triennale = 0,
18     Magistrale = 1,
19     Dottorato = 2
20   };
21   Tipo tipo;
22   std::string nome_corso;
23   dbo::collection< dbo::ptr<Iscrizione> > iscrizione;
24
25   template<class Action>
26   void persist(Action& a){
27     dbo::id(a, codice, "codice");
28     dbo::field(a, tipo, "tipo_corso");
```

```
29      dbo::field(a, nome_corso, "nome_corso");
30      dbo::hasMany(a, iscrizione, dbo::ManyToOne, "codice_corso
        ");
31    }
32  };
33
34  // codice per disabilitare il campo version nel db
35  namespace Wt {
36    namespace Dbo {
37      template<>
38      struct dbo_traits<Corso> : public dbo_default_traits {
39        static const char *versionField() {
40          return 0;
41        }
42        // cambio di chiave primaria
43        typedef int IdType;
44        static IdType invalidId() {
45          return int();
46        }
47        static const char *surrogateIdField() {
48          return 0;
49        }
50      };
51    }
52  }
53  namespace Wt {
54    namespace Dbo {
55      template<>
56      struct dbo_traits<Iscrizione> : public dbo_default_traits
        {
57        static const char *versionField() {
58          return 0;
59        }
60      };
61    }
62  }
63  namespace Wt {
64    namespace Dbo {
65      template<>
66      struct dbo_traits<Studente> : public dbo_default_traits {
67        static const char *versionField() {
68          return 0;
69        }
70        // cambio di chiave primaria
71        typedef int IdType;
72        static IdType invalidId() {
73          return int();
74        }
75        static const char *surrogateIdField() {
```

```
76          return 0;
77        }
78      };
79    }
80  }
81
82  class Iscrizione {
83    public:
84    int anno_accademico;
85    dbo::ptr<Studente> studente;
86    dbo::ptr<Corso> corso;
87
88    template<class Action>
89    void persist(Action& a){
90      dbo::field(a, anno_accademico, "anno_accademico");
91      dbo::belongsTo(a, studente, "matricola", dbo::NotNull);
92      dbo::belongsTo(a, corso, "codice_corso", dbo::NotNull);
93    }
94  };
95
96  class Studente {
97    public:
98    int matricola;
99    std::string cognome;
100   std::string nome;
101
102   dbo::collection< dbo::ptr<Iscrizione> > iscrizioni;
103
104   template <class Action>
105   void persist(Action& a) {
106     dbo::id(a, matricola, "matricola");
107     dbo::field(a, cognome, "cognome");
108     dbo::field(a, nome, "nome");
109     dbo::hasMany(a, iscrizioni, dbo::ManyToOne, "matricola");
110   }
111 };
112
113 void run () {
114   dbo::backend::Sqlite3 sqlite3("studenti.db");
115   dbo::Session session;
116   session.setConnection(sqlite3);
117
118   session.mapClass<Studente> ("studenti");
119   session.mapClass<Iscrizione> ("iscrizioni");
120   session.mapClass<Corso> ("corsi");
121   session.createTables();
122
123   dbo::Transaction transaction(session);
124
```

```
125   // Creazione del primo studente
126   Studente* s = new Studente();
127   s->matricola = 231452;
128   s->cognome = "Trombi";
129   s->nome = "Francesco";
130   dbo::ptr<Studente> francesco = session.add(s);
131
132   // Creazione della sua iscrizione
133   Iscrizione* i = new Iscrizione();
134   i->anno_accademico = 2013;
135   dbo::ptr<Iscrizione> iPtr = session.add(i);
136   iPtr.modify()->studente = francesco;
137
138   // Aggiunta del corso a tale iscrizione
139   Corso* c = new Corso();
140   c->codice = 31;
141   c->tipo = Corso::Triennale;
142   c->nome_corso = "Informatica";
143   dbo::ptr<Corso> informatica = session.add(c);
144   iPtr.modify()->corso = informatica;
145
146   transaction.commit();
147 }
148
149 int main () {
150   run();
151   return 0;
152 }
```

Descrizione del codice

Poiché nel capitolo successivo verrà descritto nel dettaglio il funzionamento della libreria, in questo caso si limiterà la descrizione, permettendo comunque di comprendere i costrutti, i metodi e gli oggetti utilizzati in questa porzione di codice. Innanzitutto è necessario includere le librerie `wtdbo` e `wtdbosqlite3` (che è la libreria apposita per il backend SQLite3). Per rendere una classe persistente bisogna aggiungerle il metodo `persist()`: al suo interno, ogni campo della classe che si desidera mappare in una colonna della tabella del database verrà mandato come parametro al metodo `field()`. L'ordine delle chiamate al metodo `field()` non devono essere per forza nello stesso ordine testuale in cui vengono dichiarati i campi. Poiché di default la libreria fissa come chiave primaria di una entità un campo intero con il flag `AUTOINCREMENT` attivo, per fissarne una diversa è necessario specializzare `Wt::Dbo::dbo_traits<C>` (al di fuori della definizione della classe interessata). Il campo scelto per diventare chiave primaria non verrà passato come gli

altri al metodo `field()`, ma al metodo `id()`. Per avere una chiave primaria composta invece, è necessario avere il tipo C++ corrispondente. Tale tipo deve avere dei requisiti di base, quali il costruttore di default, gli operatori di confronto (`==` e `<`) e un operatore di flusso. Questa tipologia di chiave primaria può contenere anche chiavi esterne.

La libreria inserisce inoltre di default un campo ulteriore, chiamato `version` (usato per il controllo della concorrenza); per disabilitarlo è necessario specializzare ulteriormente il costrutto `Wt::Dbo::dbo_traits<C>`.

Per rappresentare le associazioni tra entità, si utilizza la seguente modalità. Innanzitutto si stabilisce se la relazione è una Molti-a-Molti o una Uno-a-Molti; le associazioni Uno-a-Uno non sono supportate, ma possono essere implementate tramite associazioni Uno-a-Molti. Nel caso di questo esempio, si hanno due associazioni Uno-a-Molti: un corso può avere molte iscrizioni, ma ogni iscrizione è relativa ad un solo corso; uno studente ha molte iscrizioni, ma un'iscrizione vale solo per uno studente. Questo tipo di associazione viene mappato in Wt::Dbo nel seguente modo: prendiamo per esempio la relazione Studente-Iscrizione (quella Corso-Iscrizione viene implementata analogamente). All'interno della classe `Studente` (ovvero, sul lato "Molti" dell'associazione da codificare) viene aggiunto un campo, di tipo `dbo::collection< dbo::ptr<Iscrizione> >`, cioè una collezione di iscrizioni. Il metodo `persist()` di `Studente` dovrà mappare anche questo campo, ma non con il metodo `field()`, bensì con il metodo `hasMany()`; tale metodo prende come parametri dall'associazione il tipo, che è `dbo::ManyToOne`, e il nome, in questo caso `matricola`. La classe `Iscrizione` dovrà contenere un campo di tipo `dbo::ptr<Studente>`, che nel metodo `persist()` non verrà mappato con `field()`, ma con `belongsTo()`, che prende come parametro il nome dell'associazione, che è `matricola`. Come si è detto per l'associazione Corso-Iscrizione l'implementazione è analoga. Di default le chiavi esterne sono settate come `NULLABLE`; per ovviare a questa cosa, basta aggiungere `dbo::NotNull` come ultimo parametro della chiamata a `belongsTo()`.

Nel `main` viene poi chiamata una funzione, per comodità nominata `run()`, che contiene la parte del codice che si occupa della comunicazione con il database. Per prima cosa viene creato un oggetto di tipo `Session`, che tramite il metodo `setConnection()` permette di instaurare una connessione con il database. Seccessivamente si crea una mappa delle classi: con il metodo `mapClass<T>()` viene creato un modello della tabella che rappresenta la classe `T`, in base al codice contenuto nel metodo `persist()`. Infine, con il metodo `createTables()`, vengono effettivamente create le tabelle che mappano le classi designate dal metodo `mapClass<T>()`. Il passo successivo è la creazione di un oggetto di tipo `Transaction`, che nella chiamata al costruttore passa come parametro la `Session` usata fino a quel momento.

Nel codice della funzione `run()` si mostrano poi alcuni esempi di modifica del database. Per inserire un oggetto nel database, per esempio uno `Studente`, si crea un'istanza della classe in C++; viene poi creato un `dbo::ptr<Studente>`, al quale viene assegnato il risultato della chiamata del metodo `add()` che prende come parametro l'oggetto C++. Per modificare un oggetto è sufficiente chiamare il metodo `modify()` sul `dbo::ptr<>` corrispondente all'oggetto da modificare.

La chiamata del metodo `commit()` da parte della transazione rende definitive le modifiche apportate fino a quel momento (a meno di errori di serializzazione causati da modifiche concorrenti).

2.2.2 Codice SQL generato da Wt::Dbo

Dal codice C++ visto nel paragrafo precedente viene generato un database SQLite3. Da questo database è stato estratto il file SQL che contiene tutti i comandi generati dalla libreria. Il file SQL è il seguente:

```
1  PRAGMA foreign_keys=OFF;
2  BEGIN TRANSACTION;
3  CREATE TABLE "iscrizioni" (
4    "id" integer primary key autoincrement,
5    "anno_accademico" integer not null,
6    "matricola_matricola" integer not null,
7    "codice_corso_codice" integer not null,
8    constraint "fk_iscrizioni_matricola" foreign key (
9      "matricola_matricola"
10   ) references "studenti" ("matricola"),
11   constraint "fk_iscrizioni_codice_corso" foreign key (
12   "codice_corso_codice"
13   ) references "corsi" ("codice")
14 );
15 INSERT INTO "iscrizioni" VALUES(1, 2013, 231452, 31);
16 CREATE TABLE "corsi" (
17   "codice" integer not null,
18   "tipo_corso" integer not null,
19   "nome_corso" text not null,
20   primary key ("codice")
21 );
22 INSERT INTO "corsi" VALUES(31, 0, 'Informatica');
23 CREATE TABLE "studenti" (
24   "matricola" integer not null,
25   "cognome" text not null,
26   "nome" text not null,
27   primary key ("matricola")
28 );
29 INSERT INTO "studenti" VALUES(231452,'Trombi','Francesco');
```

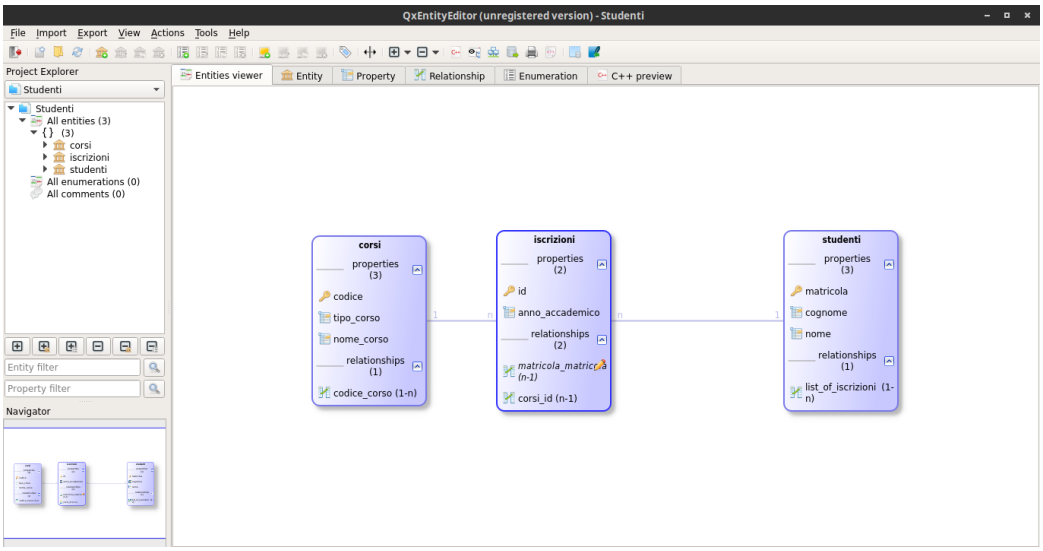

Figura 2.1: Schermata di QxEntityEditor

```
30 DELETE FROM sqlite_sequence;
31 INSERT INTO "sqlite_sequence" VALUES('iscrizioni',1);
32 COMMIT;
```

2.2.3 Importazione del progetto in QxEntityEditor

Dopo aver considerato la generazione del database da parte di Wt::Dbo, si passa al tool grafico offerto da Qt: QxEntityEditor. Tramite l'opzione di importazione, è possibile generare un nuovo progetto direttamente dal database, decidendo quali entità e relazioni importare. A causa di qualche errore durante questa operazione, l'associazione tra `Corso` ed `Iscrizione` è stata aggiunta a mano. Nella figura 2.1, che rappresenta la schermata principale del progetto, si nota come QxEntityEditor fornisca una visione completa delle entità. Da questa schermata, eseguendo un doppio click su una entità, è possibile entrare in una schermata di modifica, presentata nella figura 2.2; tale parte dell'editor fornisce un'interfaccia grafica per manipolare i campi della tabella corrispondente; è inoltre possibile creare relazioni tra entità, aggiungendole all'elenco presente in fondo alla schermata.

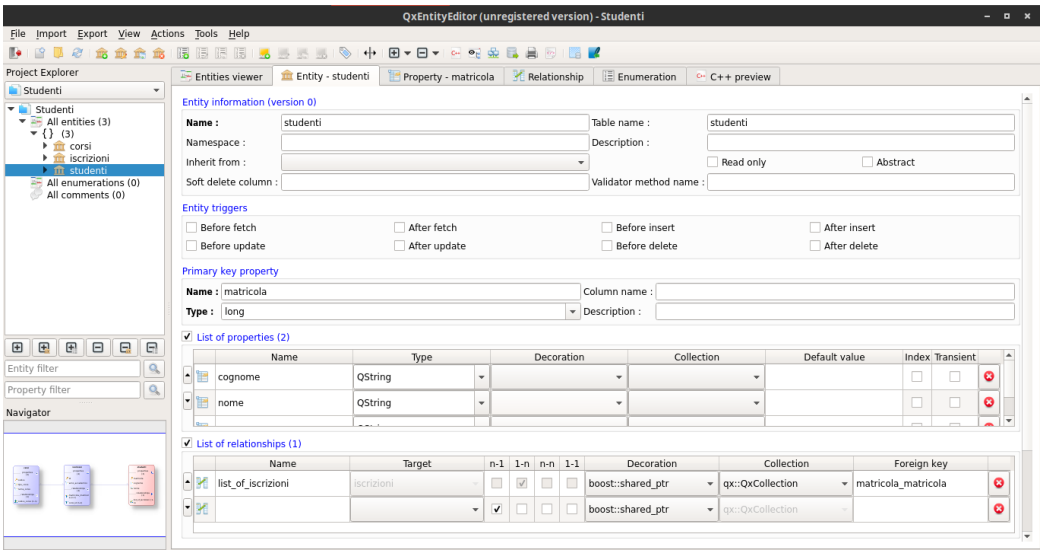

Figura 2.2: Schermata di modifica entità

2.2.4 Codice SQL generato da QxOrm e QxEntityEditor

Il codice generato da QxOrm e QxEntityEditor è più leggero di quello generato da Wt::Dbo, poiché non è in grado di leggere i record già presenti. La soluzione offerta da Qt infatti crea la struttura del database, ma non lo popola. Inoltre, in questo specifico caso, mancano le chiavi esterne: questo perché QxOrm le inserisce dopo aver creato lo script SQL, e in SQLite non è possibile aggiungere un vincolo di chiave esterna nel database in un secondo momento, come si può notare dall'errore scritto nel commento presente nel seguente codice SQL.

```
1  CREATE TABLE corsi (
2    codice INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
3    tipo_corso INTEGER,
4    nome_corso TEXT
5  );
6  CREATE TABLE iscrizioni (
7    id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
8    anno_accademico INTEGER,
9    matricola_matricola INTEGER,
10   corsi_id INTEGER
11 );
12 CREATE TABLE studenti (
13   matricola INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
```

```
14    cognome TEXT,
15    nome TEXT
16  );
17  --
18  -- WARNING : SQLite database doesn't support ALTER TABLE
        my_table ADD --
19  --  CONSTRAINT DDL script ! --
20  --             Cannot ADD FOREIGN KEY '
        fk_iscrizioni_matricola_matricola'--
21  --             for table 'iscrizioni'
22  --
23  --
24  -- WARNING : SQLite database doesn't support ALTER TABLE
        my_table ADD --
25  --  CONSTRAINT DDL script ! --
26  --             Cannot ADD FOREIGN KEY 'fk_iscrizioni_corsi_id
        '--
27  --               for table 'iscrizioni'
28  --
```

2.2.5 Codice C++ generato da QxEE

Il codice C++ generato dalla libreria QxOrm, partendo dal modello dei dati creato dall'editor grafico QxEntityEditor è suddiviso in due directory: `include/` e `src/`. Oltre a queste sono presenti le directory `custom/`, per aggiungere codice al progetto senza intaccare quello generato da QxOrm, e la directory `bin/`, atta a contenere gli eseguibili del progetto. Nella prima sono contenuti i file di header; di questi sei, tre sono stati ritenuti interessanti da analizzare in questo lavoro di tesi: `corsi.gen.h`, `iscrizioni.gen.h` e `studenti.gen.h`.

`corsi.gen.h`

```
1   #ifndef _STUDENTI_CORSI_H_
2   #define _STUDENTI_CORSI_H_
3
4   class iscrizioni;
5
6   class STUDENTI_EXPORT corsi
7   {
8
9     QX_REGISTER_FRIEND_CLASS(corsi)
10
11  public:
12
```

```
13    typedef qx::QxCollection<long, boost::shared_ptr<iscrizioni
       > >
14      type_codice_corso;
15
16  protected:
17
18    long m_codice;
19    long m_tipo_corso;
20    QString m_nome_corso;
21    type_codice_corso m_codice_corso;
22
23  public:
24
25    corsi();
26    corsi(const long & id);
27    virtual ~corsi();
28
29    long getcodice() const;
30    long gettipo_corso() const;
31    QString getnome_corso() const;
32    type_codice_corso getcodice_corso() const;
33    type_codice_corso & codice_corso();
34    const type_codice_corso & codice_corso() const;
35
36    void setcodice(const long & val);
37    void settipo_corso(const long & val);
38    void setnome_corso(const QString & val);
39    void setcodice_corso(const type_codice_corso & val);
40
41    type_codice_corso getcodice_corso(
42      bool bLoadFromDatabase,
43      const QString & sAppendRelations = QString(),
44      QSqlDatabase * pDatabase = NULL, QSqlError * pDaoError =
       NULL
45      );
46    type_codice_corso & codice_corso(
47      bool bLoadFromDatabase, const QString & sAppendRelations
       = QString(),
48      QSqlDatabase * pDatabase = NULL, QSqlError * pDaoError =
       NULL
49    );
50
51  public:
52
53    static QString relation_codice_corso() { return "
       codice_corso"; }
54
55  public:
56
```

```
57    static QString column_codice() { return "codice"; }
58    static QString column_tipo_corso() { return "tipo_corso"; }
59    static QString column_nome_corso() { return "nome_corso"; }
60
61  public:
62
63    static QString table_name() { return "corsi"; }
64
65  };
66
67  typedef boost::shared_ptr<corsi> corsi_ptr;
68  typedef qx::QxCollection<long, corsi_ptr> list_of_corsi;
69  typedef boost::shared_ptr<list_of_corsi> list_of_corsi_ptr;
70
71  QX_REGISTER_COMPLEX_CLASS_NAME_HPP_STUDENTI(
72    corsi, qx::trait::no_base_class_defined, 0, corsi
73  )
74
75  #include "../include/iscrizioni.gen.h"
76
77  #include "../custom/include/corsi.h"
78
79  #endif // _STUDENTI_CORSI_H_
```

iscrizioni.gen.h

```
1   #ifndef _STUDENTI_ISCRIZIONI_H_
2   #define _STUDENTI_ISCRIZIONI_H_
3
4   class studenti;
5   class corsi;
6
7   class STUDENTI_EXPORT iscrizioni
8   {
9
10    QX_REGISTER_FRIEND_CLASS(iscrizioni)
11
12  public:
13
14    typedef boost::shared_ptr<studenti>
       type_matricola_matricola;
15    typedef boost::shared_ptr<corsi> type_corsi_id;
16
17  protected:
18
19    long m_id;
20    long m_anno_accademico;
21    type_matricola_matricola m_matricola_matricola;
```

```
22    type_corsi_id m_corsi_id;
23
24  public:
25
26    iscrizioni();
27    iscrizioni(const long & id);
28    virtual ~iscrizioni();
29
30    long getid() const;
31    long getanno_accademico() const;
32    type_matricola_matricola getmatricola_matricola() const;
33    type_corsi_id getcorsi_id() const;
34
35    void setid(const long & val);
36    void setanno_accademico(const long & val);
37    void setmatricola_matricola(const type_matricola_matricola
       & val);
38    void setcorsi_id(const type_corsi_id & val);
39
40    type_matricola_matricola getmatricola_matricola(
41      bool bLoadFromDatabase, const QString & sAppendRelations
       = QString(),
42      QSqlDatabase * pDatabase = NULL, QSqlError * pDaoError =
       NULL
43    );
44    type_corsi_id getcorsi_id(
45      bool bLoadFromDatabase, const QString & sAppendRelations
       = QString(),
46      QSqlDatabase * pDatabase = NULL, QSqlError * pDaoError =
       NULL
47    );
48
49  public:
50
51    static QString relation_matricola_matricola() {
52      return "matricola_matricola";
53    }
54    static QString relation_corsi_id() { return "corsi_id"; }
55
56  public:
57
58    static QString column_id() { return "id"; }
59    static QString column_anno_accademico() { return "
       anno_accademico"; }
60
61  public:
62
63    static QString table_name() { return "iscrizioni"; }
64
```

```
65 };
66
67 typedef boost::shared_ptr<iscrizioni> iscrizioni_ptr;
68 typedef qx::QxCollection<long, iscrizioni_ptr>
       list_of_iscrizioni;
69 typedef boost::shared_ptr<list_of_iscrizioni>
       list_of_iscrizioni_ptr;
70
71 QX_REGISTER_COMPLEX_CLASS_NAME_HPP_STUDENTI(
72   iscrizioni, qx::trait::no_base_class_defined, 0, iscrizioni
73 )
74
75 #include "../include/studenti.gen.h"
76 #include "../include/corsi.gen.h"
77
78 #include "../custom/include/iscrizioni.h"
79
80 #endif // _STUDENTI_ISCRIZIONI_H_
```

studenti.gen.h

```
1  #ifndef _STUDENTI_STUDENTI_H_
2  #define _STUDENTI_STUDENTI_H_
3
4  class iscrizioni;
5
6  class STUDENTI_EXPORT studenti
7  {
8
9    QX_REGISTER_FRIEND_CLASS(studenti)
10
11 public:
12
13   typedef qx::QxCollection<long, boost::shared_ptr<iscrizioni
       > >
14     type_list_of_iscrizioni;
15
16 protected:
17
18   long m_matricola;
19   QString m_cognome;
20   QString m_nome;
21   type_list_of_iscrizioni m_list_of_iscrizioni;
22
23 public:
24
25   studenti();
26   studenti(const long & id);
```

```
27    virtual ~studenti();
28
29    long getmatricola() const;
30    QString getcognome() const;
31    QString getnome() const;
32    type_list_of_iscrizioni getlist_of_iscrizioni() const;
33    type_list_of_iscrizioni & list_of_iscrizioni();
34    const type_list_of_iscrizioni & list_of_iscrizioni() const;
35
36    void setmatricola(const long & val);
37    void setcognome(const QString & val);
38    void setnome(const QString & val);
39    void setlist_of_iscrizioni(const type_list_of_iscrizioni &
       val);
40
41    type_list_of_iscrizioni getlist_of_iscrizioni(
42      bool bLoadFromDatabase, const QString & sAppendRelations
       = QString(),
43      QSqlDatabase * pDatabase = NULL, QSqlError * pDaoError =
       NULL
44    );
45    type_list_of_iscrizioni & list_of_iscrizioni(
46      bool bLoadFromDatabase, const QString & sAppendRelations
       = QString(),
47      QSqlDatabase * pDatabase = NULL, QSqlError * pDaoError =
       NULL
48    );
49
50  public:
51
52    static QString relation_list_of_iscrizioni() {
53      return "list_of_iscrizioni";
54    }
55
56  public:
57
58    static QString column_matricola() { return "matricola"; }
59    static QString column_cognome() { return "cognome"; }
60    static QString column_nome() { return "nome"; }
61
62  public:
63
64    static QString table_name() { return "studenti"; }
65
66  };
67
68  typedef boost::shared_ptr<studenti> studenti_ptr;
69  typedef qx::QxCollection<long, studenti_ptr> list_of_studenti
       ;
```

```
70 typedef boost::shared_ptr<list_of_studenti>
       list_of_studenti_ptr;
71
72 QX_REGISTER_COMPLEX_CLASS_NAME_HPP_STUDENTI(
73   studenti, qx::trait::no_base_class_defined, 0, studenti
74 )
75
76 #include "../include/iscrizioni.gen.h"
77
78 #include "../custom/include/studenti.h"
79
80 #endif // _STUDENTI_STUDENTI_H_
```

`iscrizioni.gen.cpp`

Per completezza, verrà mostrato anche un file contenuto nella directory `src/`; è evidente come la complessità sia di gran lunga maggiore di quella necessaria a Wt::Dbo.

```
1  #include "../include/Studente_precompiled_header.gen.h"
2
3  #include "../include/iscrizioni.gen.h"
4  #include "../include/studenti.gen.h"
5  #include "../include/corsi.gen.h"
6
7  #include <QxMemLeak.h>
8
9  QX_REGISTER_COMPLEX_CLASS_NAME_CPP_STUDENTE(iscrizioni,
       iscrizioni)
10
11 namespace qx {
12
13 template <>
14 void register_class(QxClass<iscrizioni> & t)
15 {
16    qx::IxDataMember * pData = NULL; Q_UNUSED(pData);
17    qx::IxSqlRelation * pRelation = NULL; Q_UNUSED(pRelation);
18    qx::IxFunction * pFct = NULL; Q_UNUSED(pFct);
19    qx::IxValidator * pValidator = NULL; Q_UNUSED(pValidator);
20
21    t.setName("iscrizioni");
22    t.setPropertyBag("QX_EE_GENERATED_BY_SQLITE_IMPORT_PLUGIN"
       , "1");
23
24    pData = t.id(& iscrizioni::m_id, "id", 0);
25    pData->setPropertyBag("
       QX_EE_GENERATED_BY_SQLITE_IMPORT_PLUGIN", "1");
26
```

```
27    pData = t.data(& iscrizioni::m_anno_accademico, "
      anno_accademico", 0, true, true);
28    pData->setPropertyBag("
      QX_EE_GENERATED_BY_SQLITE_IMPORT_PLUGIN", "1");
29
30    pRelation = t.relationManyToOne(& iscrizioni::
      m_matricola_matricola, "matricola_matricola", 0);
31    pRelation->setPropertyBag("
      QX_EE_GENERATED_BY_SQLITE_IMPORT_PLUGIN", "1");
32    pRelation = t.relationManyToOne(& iscrizioni::m_corsi_id,
      "corsi_id", 0);
33
34    qx::QxValidatorX<iscrizioni> * pAllValidator = t.
      getAllValidator(); Q_UNUSED(pAllValidator);
35 }
36
37 } // namespace qx
38
39 iscrizioni::iscrizioni() : m_id(0), m_anno_accademico(0) { ;
      }
40
41 iscrizioni::iscrizioni(const long & id) : m_id(id),
      m_anno_accademico(0) { ; }
42
43 iscrizioni::~iscrizioni() { ; }
44
45 long iscrizioni::getid() const { return m_id; }
46
47 long iscrizioni::getanno_accademico() const { return
      m_anno_accademico; }
48
49 iscrizioni::type_matricola_matricola iscrizioni::
      getmatricola_matricola() const { return
      m_matricola_matricola; }
50
51 iscrizioni::type_corsi_id iscrizioni::getcorsi_id() const {
      return m_corsi_id; }
52
53 void iscrizioni::setid(const long & val) { m_id = val; }
54
55 void iscrizioni::setanno_accademico(const long & val) {
      m_anno_accademico = val; }
56
57 void iscrizioni::setmatricola_matricola(const iscrizioni::
      type_matricola_matricola & val) { m_matricola_matricola =
      val; }
58
59 void iscrizioni::setcorsi_id(const iscrizioni::type_corsi_id
      & val) { m_corsi_id = val; }
```

```
60
61 iscrizioni::type_matricola_matricola iscrizioni::
       getmatricola_matricola(bool bLoadFromDatabase, const
       QString & sAppendRelations /* = QString() */, QSqlDatabase
        * pDatabase /* = NULL */, QSqlError * pDaoError /* = NULL
        */)
62 {
63     if (pDaoError) { (* pDaoError) = QSqlError(); }
64     if (! bLoadFromDatabase) { return getmatricola_matricola()
       ; }
65     QString sRelation = "matricola_matricola";
66     if (! sAppendRelations.isEmpty() && ! sAppendRelations.
       startsWith("->") && ! sAppendRelations.startsWith(">>")) {
        sRelation += "->" + sAppendRelations; }
67     else if (! sAppendRelations.isEmpty()) { sRelation +=
       sAppendRelations; }
68     iscrizioni tmp;
69     tmp.m_id = this->m_id;
70     QSqlError daoError = qx::dao::fetch_by_id_with_relation(
       sRelation, tmp, pDatabase);
71     if (! daoError.isValid()) { this->m_matricola_matricola =
       tmp.m_matricola_matricola; }
72     if (pDaoError) { (* pDaoError) = daoError; }
73     return m_matricola_matricola;
74 }
75
76 iscrizioni::type_corsi_id iscrizioni::getcorsi_id(bool
       bLoadFromDatabase, const QString & sAppendRelations /* =
       QString() */, QSqlDatabase * pDatabase /* = NULL */,
       QSqlError * pDaoError /* = NULL */)
77 {
78     if (pDaoError) { (* pDaoError) = QSqlError(); }
79     if (! bLoadFromDatabase) { return getcorsi_id(); }
80     QString sRelation = "corsi_id";
81     if (! sAppendRelations.isEmpty() && ! sAppendRelations.
       startsWith("->") && ! sAppendRelations.startsWith(">>")) {
        sRelation += "->" + sAppendRelations; }
82     else if (! sAppendRelations.isEmpty()) { sRelation +=
       sAppendRelations; }
83     iscrizioni tmp;
84     tmp.m_id = this->m_id;
85     QSqlError daoError = qx::dao::fetch_by_id_with_relation(
       sRelation, tmp, pDatabase);
86     if (! daoError.isValid()) { this->m_corsi_id = tmp.
       m_corsi_id; }
87     if (pDaoError) { (* pDaoError) = daoError; }
88     return m_corsi_id;
89 }
```

Descrizione del codice

I file di header del progetto generato da QxOrm contengono le definizioni delle classi, che, come si può notare, sono simili a quelle utilizzate da Wt::Dbo. Infatti contengono:

- alcuni `typedef` per semplificare la gestione di collezioni templatiche, simili a quelle viste in Wt::Dbo;
- costruttore senza parametri;
- costruttore che prende come parametro il campo corrispondente alla chiave primaria;
- distruttore virtuale;
- *getter* e *setter* dei campi della classe;
- *getter* e *setter* per le stringhe contenenti i nomi della tabella, delle colonne e delle associazioni relativi alla classe specifica.

I file con estensione `.cpp` contenuti nella directory `src/` contengono, oltre all'implementazione dei metodi definiti nell'header tutto il codice che garantisce la persistenza della classe a cui sono associati. Contengono la funzione templatica `register_class()`, che prende parametro la `QxClass` relativa alla tabella che vogliono mappare. Al contrario di Wt::Dbo, dove il codice necessario al mapping dei tipi e alla gestione delle query veniva lasciato all'interno della libreria, QxOrm genera all'interno del progetto tali componenti, come si può notare dal codice allegato.

2.3 Codice SQL generato per PostgreSQL

L'esempio precedente era basato sul backend SQLite; per completezza si è deciso di estenderlo ad un altro DBMS, cioè PostgreSQL. Il codice sorgente dell'esempio è stato naturalmente modificato. Il primo cambiamento è nelle direttive di inclusione per il preprocessore: alla riga 5 l'inclusione alla libreria di backend è stata modificata per supportare PostgreSQL, quindi è diventata:

```
#include<Wt/Dbo/backend/Postgres>
```

Il secondo cambiamento è stato fatto nella sezione di codice iniziale della funzione `run()` (righe 114, 115 e 116), per creare una connessione al database PostgreSQL. Le righe diventano dunque:

```
dbo::backend::Postgres postgres(
host=localhost user=postgres dbname=studenti
);
dbo::Session session;
session.setConnection(postgres);
```

Come si può notare dai codici seguenti, entrambe le librerie aggiungono le chiavi esterne in un secondo momento: questo indica una migliore flessibilità da parte di Wt::Dbo, che riesce meglio ad adattarsi al DBMS a cui si interfaccia. In ogni caso, anche QxOrm riesce ad aggiungere i vincoli di chiave esterna, non possibile con DBMS SQLite. Anche in questo caso il codice SQL generato da QxOrm non ha i vincoli `NOT NULL` sugli attributi.

QxOrm

```
1  CREATE TABLE corsi (
2    codice BIGSERIAL NOT NULL PRIMARY KEY,
3    tipo_corso INTEGER, nome_corso TEXT
4  ) WITH (OIDS = FALSE);
5  CREATE TABLE iscrizioni (
6    id BIGSERIAL NOT NULL PRIMARY KEY,
7    anno_accademico INTEGER,
8    matricola_matricola INTEGER,
9    corsi_id INTEGER
10 ) WITH (OIDS = FALSE);
11 CREATE TABLE studenti (
12   matricola BIGSERIAL NOT NULL PRIMARY KEY,
13   cognome TEXT, nome TEXT
14 ) WITH (OIDS = FALSE);
15 ALTER TABLE iscrizioni ADD CONSTRAINT
       fk_iscrizioni_matricola_matricola
16   FOREIGN KEY (matricola_matricola) REFERENCES studenti(
       matricola);
17 ALTER TABLE iscrizioni ADD CONSTRAINT fk_iscrizioni_corsi_id
18   FOREIGN KEY (corsi_id) REFERENCES corsi(codice);
```

Wt::Dbo

```
1  CREATE TABLE corsi (
2      codice integer NOT NULL,
3      tipo_corso integer NOT NULL,
4      nome_corso text NOT NULL
5  );
6  CREATE TABLE iscrizioni (
```

```
7     id integer NOT NULL,
8     anno_accademico integer NOT NULL,
9     matricola_matricola integer NOT NULL,
10    codice_corso_codice integer NOT NULL
11 );
12 CREATE SEQUENCE iscrizioni_id_seq
13    START WITH 1
14    INCREMENT BY 1
15    NO MINVALUE
16    NO MAXVALUE
17    CACHE 1;
18 ALTER SEQUENCE iscrizioni_id_seq OWNED BY iscrizioni.id;
19 CREATE TABLE studenti (
20    matricola integer NOT NULL,
21    cognome text NOT NULL,
22    nome text NOT NULL
23 );
24 ALTER TABLE ONLY iscrizioni ALTER COLUMN id SET DEFAULT
      nextval(
25   'iscrizioni_id_seq'::regclass
26 );
27
28 COPY corsi (codice, tipo_corso, nome_corso) FROM stdin;
29 31      0       Informatica
30 \.
31
32 COPY iscrizioni (
33   id, anno_accademico, matricola_matricola,
      codice_corso_codice
34 ) FROM stdin;
35 1       2013    231452  31
36 \.
37
38 SELECT pg_catalog.setval('iscrizioni_id_seq', 1, true);
39
40 COPY studenti (matricola, cognome, nome) FROM stdin;
41 231452  Trombi  Francesco
42 \.
43
44 ALTER TABLE ONLY corsi
45    ADD CONSTRAINT corsi_pkey
46    PRIMARY KEY (codice);
47
48 ALTER TABLE ONLY iscrizioni
49    ADD CONSTRAINT iscrizioni_pkey
50    PRIMARY KEY (id);
51
52 ALTER TABLE ONLY studenti
53    ADD CONSTRAINT studenti_pkey
```

```
54      PRIMARY KEY (matricola);
55
56  ALTER TABLE ONLY iscrizioni
57      ADD CONSTRAINT fk_iscrizioni_codice_corso
58        FOREIGN KEY (codice_corso_codice)
59        REFERENCES corsi(codice);
60
61  ALTER TABLE ONLY iscrizioni
62      ADD CONSTRAINT fk_iscrizioni_matricola
63        FOREIGN KEY (matricola_matricola)
64        REFERENCES studenti(matricola);
```

2.4 Scelta di Wt::Dbo

L'analisi delle soluzioni ha permesso, in prima battuta, di evidenziare i prodotti migliori presenti sul mercato; in secondo luogo ha consentito la scelta di una soluzione in grado di rispondere alle necessità proposte da questo lavoro: i criteri discriminanti sono stati diversi, per poter trovare un oggetto semplice, maneggevole, dalle performance relativamente buone e con una gamma di compatibilità più ampia possibile. Per valutare le due soluzioni si sono dunque usati i seguenti criteri, al cui elenco seguirà una descrizione della soluzione che si adatta meglio al requisito corrispondente:

- indipendenza da altre componenti software;
- aderenza al C++ standard;
- possibilità di estensione della soluzione stessa;
- tipologia di licenza e limitazioni conseguenti;
- semplicità d'uso;
- supporti ad altre componenti;
- codici sorgenti generati;
- componenti aggiuntive necessarie;
- libertà d'azione, per esempio nel mapping dei tipi o nelle modifiche per cambiare chiavi primarie, etc.

Indipendenza da altre componenti

Mentre QxOrm e QxEntityEditor sono fortemente legate al resto del framework Qt, Wt::Dbo riesce a svincolarsi dal contesto in cui viene distribuito, dimostrandosi più indipendente della soluzione alternativa.

Aderenza al C++ standard

Anche in questo caso Wt::Dbo si comporta meglio della concorrente, adattandosi meglio al codice del C++ standard; infatti non necessita in abbondanza di tipi custom del framework Wt (come ad esempio `WString`); QxOrm invece è fortemente basato sui tipi custom del framework Qt (come ad esempio `QString`), complice anche il fatto che il codice C++ viene generato da QxEntityEditor.

Possibili estensioni

A prima vista il codice della libreria Wt::Dbo fornisce maggior spazio per le possibili estensioni, per esempio al supporto dei tipi C++. Tali informazioni vengono anche fornite nel tutorial della libreria. QxOrm ha invece una struttura più complessa di quella della concorrente, rendendo le modifiche più complesse da applicare.

Licenze

Entrambe le soluzioni vengono fornite con due licenze: gratuita o commerciale. La differenza tra le due sta nelle limitazioni che le licenze gratuite impongono allo sviluppatore: nel caso di QxEntityEditor non è possibile creare esempi con più di cinque entità, mantenendo però il supporto ai DBMS; nel caso di Wt::Dbo non è possibile utilizzare i driver per Oracle, avendo però pieno supporto nel caso di progetti che coinvolgono più di cinque entità. Nel caso specifico di questo lavoro di tesi, la limitazione imposta da Wt::Dbo è più leggera; in altri contesti potrebbe però essere vero il contrario.

Semplicità d'uso

Rispetto a questa condizione, entrambe le librerie hanno punti di forza e punti deboli. Wt::Dbo necessita di poche righe di codice per garantire la persistenza degli oggetti C++, ma nella sua semplicità offre una visione d'insieme del progetto inferiore a quella offerta dalla concorrente. La soluzione proposta da Qt infatti, grazie a QxEntityEditor, permette di avere una visione grafica del progetto nella sua interezza; inoltre è possibile apportare ad

esso modifiche da una finestra apposita dell'editor grafico, senza toccare una sola riga di codice.

Supporti

Come detto prima nella sezione dedicata alle licenze, i supporti delle librerie nella loro accezione gratuita sono differenti; la più completa è sicuramente QxOrm, poiché garantisce supporto a MS SQL Server e Oracle, sacrificando quello a Firebird (comunque poco utilizzato a livello commerciale).

Codici sorgenti generati

Come si può facilmente vedere, il codice sorgente generato dalle soluzioni è differente. Partendo dal presupposto che Wt::Dbo non genera codice C++, QxOrm è in grado di creare una struttura di directory completa. Il codice è però pieno di costrutti del framework Qt, quali le `QString` o le `QxClass`. Dal punto di vista del codice SQL invece, le limitazioni di QxOrm sono evidenti. Wt::Dbo è in grado di creare correttamente il database (in questo caso utilizzando SQLite3), mentre QxEntityEditor restituisce un errore mentre importa il database (probabilmente perché utilizza SQLite invece che SQLite3), facendo chiudere inaspettatamente il programma. Il codice che poi prova a generare dalle tabelle (da correggere, perché importate male) è lacunoso, e per colpa dell'ordine delle operazioni nella generazione del codice, non è in grado di aggiungere chiavi esterne in SQLite.

Libertà d'azione

QxEntityEditor, non permettendo la scrittura, la modifica e l'eliminazione di record nel database, offre naturalmente meno funzionalità di Wt::Dbo; il problema della mancata aggiunta delle chiavi esterne nello script per SQLite è un'altro segnale delle limitazioni di QxOrm. In entrambi in casi non è stato trovato il modo di generare vicoli di integrità quali i `CHECK`, in quanto la filosofia che sta dietro a queste soluzioni lascia il controllo dei valori allo sviluppatore C++. QxEntityEditor permette però di scegliere come mappare i tipi da C++ a SQL e viceversa, ovviando in parte al problema delle dipendenze ai tipi del framework citati in precedenza; l'editor grafico inoltre rappresenta un notevole vantaggio, permettendo di creare nuove entità ed associazioni con pochi click.

Componenti aggiuntive

Entrambe le soluzioni necessitano di componenti aggiuntive, tra le quali, in entrambi i casi, compare un sottoinsieme delle librerie Boost. Inoltre, Wt::Dbo viene fornita con il framework Wt, mentre QxEntityEditor e QxOrm si basano sul framework Qt (ma non sull'IDE). La caratteristica che fa pendere l'ago della bilancia a favore di Wt::Dbo è quella riguardante la compilazione; la soluzione di Wt permette infatti la compilazione solo tramite il compilatore installato sul sistema operativo, mentre il progetto QxEntityEditor deve passare per il framework Qt.

Conclusione

La libreria QxOrm ha presentato molti punti di forza, primo tra tutti l'utilizzo di un editor grafico per la creazione di un modello dei dati. Nonostante questo grande pregio, sono state però individuate alcune carenze di grande peso per la decisione:

1. sembra avere dei problemi a leggere da database: nell'importare il database generato da Wt::Dbo si è verificato un fallimento, e non riesce mai a leggere la relazione 1-N tra corsi e iscrizioni;

2. necessità di acquisto licenza per database con più di cinque entità (ovvero, qualunque esempio reale);

3. non sembra essere in grado di generare chiavi primarie senza autoincremento;

4. durante la generazione del codice DDL SQL non è in grado di aggiungere chiavi esterne (con SQLite), poiché non è possibile farlo in quel DBMS in un secondo momento rispetto alla creazione dello schema;

5. non sembra in grado di generare query di inserimento, modifica ed eliminazioni;

6. genera un gran numero di file, con numerose dipendenze da altre librerie del framework Qt;

7. l'editor grafico è comunque molto limitato.

Questi difetti non annullano il fatto che QxOrm e QxEntityEditor siano ottimi prodotti per la persistenza dei dati, ma le necessità di questo lavoro di tesi erano diverse; dunque la scelta è ricaduta sulla libreria Wt::Dbo, principalmente per i seguenti motivi:

- semplice da installare;
- indipendente dall'IDE;
- meno invasiva;
- tutorial ben documentato;
- licenza GNU General Public License (GPL);
- scritta in C++.

Wt::Dbo non è un prodotto perfetto, ha molte limitazioni; la semplicità del codice sorgente permette però di ovviare ad esse estendendo la libreria.

Capitolo 3

Struttura di Wt::Dbo

La libreria Wt::Dbo è completamente scritta in C++, e rilasciata con licenza GNU GPL v2; dunque è possibile visionare il codice sorgente e modificarlo a proprio piacimento. Il lavoro è cominciato con lo studio dei singoli file che compongono la libreria; successivamente si è approfondito sui file utili per estendere la libreria stessa, ed infine si sono apportate le modifiche. Per effettuare le estensioni richieste durante il lavoro di tesi è stata considerata l'ultima versione stabile della libreria Wt, cioè Wt 3.3.3. Il codice della libreria Wt::Dbo è strutturato in due macro-sezioni: la prima è quella che contiene le classi vere e proprie della libreria; la seconda è la directory `backend/`, che contiene le classi per interfacciarsi ai DBMS supportati.

3.1 Analisi del codice

Si parlerà successivamente dei backend e del loro funzionamento; prima è necessario descrivere le classi di base che sono contenute nella directory principale del codice sorgente di Wt::Dbo (`src/Wt/Dbo/`).

`Call`

I file `Call`, `Call.C` e `Call_impl.h` contengono classi e metodi per eseguire comandi sul database.

`collection`

I file `collection` `collection_impl.h` implementano un contenitore STL-compatibile che permette di effettuare due azioni:

- iterare il risultato di una query;

- mappare il lato "molti" di un'associazione molti-a-uno o di una molti-a-molti.

L'iteratore implementato soddisfa i requisiti di un `InputIterator`, dunque è possibile scorrere i risultati da `begin()` a `end()` unicamente alternando la lettura di un elemento all'incremento dell'iteratore. Quando una `collection` rappresenta il risultato di una query, è possibile iterarne il risultato solamente una volta.

`DbAction`

I file `DbAction`, `DbAction.C` e `DbAction_impl.h` contengono classi e metodi per la gestione dello schema del database.

`Exception`

I file `Exception` e `Exception.C` contengono classi e metodi per la gestione delle eccezioni in Wt::Dbo.

`Field`

I file `Field` e `Field_impl.h` contengono le classi e metodi per la gestione dei campi delle classi da mappare nelle tabelle del database.

`FixedSqlConnectionPool`

I file `FixedSqlConnectionPool` e `FixedSqlConnectionPool.C` contengono le classi e i metodi per la creazione e la gestione di un insieme di connessione di dimensione fissata. La dimensione viene fissata all'inizio. Questo va bene quando il numero di thread disponibile è limitata. Notare che il numero di connessioni non equivale al numero di sessioni, in quanto la sessione usa una connessione solo quando deve eseguire una transazione. Dal punto di vista dell'ORM queste componenti non sono molto interessanti; la libreria infatti è stata inizialmente pensata per gestire applicazioni web e questo è un tipico caso di utilizzo di un pool di connessioni per gestire efficientemente le numerose connessioni client.

`Json`

I file `Json` e `Json.C` contengono le classi e i metodi per la gestione della serializzazione Json.

`ptr`

I file `ptr`, `ptr.C`, `ptr_impl.h` e `ptr_tuple` contengono classi e metodi per la creazione e la gestione dei puntatori (e delle tuple di puntatori) ad oggetti dbo. Tali puntatori permettono di gestire le associazioni tra tabelle e tra classi; inoltre vengono usati per la lettura dei risultati di una query.

`Query`

I file `Query`, `Query.C`, `QueryColumn`, `QueryColumn.C`, `Query_impl.h`, `QueryModel` e `QueryModel_impl.h` contengono classi e metodi per la gestione delle query da C++ a SQL. In particolare `QueryModel` permette di gestire la visione e la modifica dei risultati delle query tramite il modello MVC (*Model-View-Controller*).

`Session`

I file `Session`, `Session.C` e `Session_impl.h` contengono classi e metodi per la creazione e la gestione di sessioni del database. Una sessione del database gestisce i metadati che riguardano il mapping delle classi C++ nelle tabelle del database. Gestisce inoltre una transazione attiva, che necessita di accedere agli oggetti del database. È possibile fornire una sessione con una connessione al database dedicata usando il metodo `setConnection()`, oppure un insieme di connessioni usando `setConnectionPool()`. In ogni caso la sessione non prende il controllo della connessione (o dell'insieme di connessioni).

`SqlConnection`

I file `SqlConnection` e `SqlConnection.C` contengono metodi e classi per la creazione e la gestione di una connessione con un database. Una connessione SQL gestisce una singola connessione al database. Gestisce inoltre una mappa dei precendenti *prepared statement* ordinata dall'identificatore. La classe `SqlConnection` è una classe astratta, che viene specializzata in base al DBMS dai file contenuti nella directory `backend/`.

`SqlConnectionPool`

I file `SqlConnectionPool` e `SqlConnectionPool.C` contengono classi e metodi per la gestione di insiemi di connessioni al database. Un insieme di connessioni SQL è condiviso tra diverse sessioni, per permettere alle sessioni stesse di ottenere la connessione mentre eseguono una transazione. Notare che una

sessione necessita di una connessione solo mentre gestisce una transazione, e un numero di connessioni uguale al numero di transazioni concorrenti.

`SqlQueryParse`

Il file `SqlQueryParse.C` contiene classi e metodi per la gestione sicura del parsing delle query.

`SqlStatement`

I file `SqlStatement` e `SqlStatement.C` contengono classi e metodi per la gestione dei prepared statement SQL. La classe `SqlStatement` è una classe astratta, che viene specializzata in base al DBMS dai file contenuti nella directory `backend/`. Uno statement può essere usato più volte, ma non in modo concorrente. Non può nemmeno essere copiato.

`SqlTraits`

I file `SqlTraits`, `SqlTraits.C` e `SqlTraits_impl.h` contengono classi e metodi per la gestione dei tipi. La classe `sql_value_traits` serve per la risoluzione dei tipi SQL partendo dai tipi C++. Questa classe può essere specializzata per un tipo `T`, al fine di estendere il supporto a tipi definiti dall'utente o tipi predefiniti che non sono direttamente supportati dalla libreria (ad esempio, i tipi unsigned). I tipi di dato direttamente supportati sono::

- `std::string`;
- `char const*`;
- `short`, `int`, `long`, `long long`;
- `float`, `double`;
- tipi `enum`;
- `bool`;
- `std::vector<unsigned char>`: per memorizzare stream di dati binari;
- `boost::optional<T>`: per rendere il tipo opzionale, permettendo un valore `NULL` all'interno del database;

- `boost::posix_time::ptime`: per mappare un tipo di dato temporale, come un timestamp; un valore invalido viene mappato come `NULL`;
- `boost::posix_time::time_duration`: intervallo di tempo.

Inoltre, vengono fornite delle classi traits per i tipi della libreria Wt, come `WDate`, `WDateTime`, `WTime` e `WString`. Contiene anche la classe `FieldInfo`, necessaria per estrarre e gestire le informazioni sui campi (per esempio, se essi sono chiavi primarie, chiavi esterne, ...). L'ultima classe da segnalare è quella `query_result_traits`, che gestisce i tipi risultanti dalle query. La libreria fornisce supporto ai tipi primitivi, utilizzando `sql_value_traits`, agli oggetti mappati, utilizzando i `ptr`, e alle `boost::tuple<>` di ogni loro combinazione.

`StdSqlTraits`

I file `StdSqlTraits` e `StdSqlTraits.C` contengono le specializzazioni per i `sql_value_traits`, una per ogni tipo supportato dalla libreria. Di tali classi templatiche si devono fornire nuove specializzazioni per estendere il supporto a tipi di dato supplementari.

`StringStream`

I file `StringStream.C` e `StringStream.h` contengono funzioni per la trasformazione di tipi in flussi di stringhe.

`Transaction`

I file `Transaction` e `Transaction.C` contengono la classe per l'implementazione e la gestione di una transazione conforme all'idioma RAII. La maggior parte delle manipolazioni sul database necessitano di una transazione attiva; le modifiche non vengono salvate nel database finché la transazione non viene conclusa tramite l'esecuzione (con successo, cioè senza lancio di eccezioni) del metodo `commit()`.

Una transazione può essere attiva finché non viene chiamato un commit o un roll-back. Se viene lanciata un'eccezione, viene automaticamente invocato un roll-back. È possibile creare più transazioni annidate, che agiscono all'unisono e si riferiscono alla stessa transazione logica: una transazione logica fallisce se una delle sue transazioni fallisce e conclude (commit) se e solo se tutte le transazioni hanno eseguito il commit.

`weak_ptr`

I file `weak_ptr` e `weak_ptr_impl.h` contengono classi e metodi per la creazione e la gestione di puntatori `ptr` deboli ad oggetti del database. Un `weak_ptr` ha API e funzionalità simili a quelle di un `ptr`, ma non contiene un riferimento all'oggetto; bensì esegue una query per ottenere l'oggetto. Viene usato per implementare le relazioni 1-1, non supportate direttamente dalla libreria:

- la classe A possiede (*hasOne*) un oggetto della classe B, con un `weak_ptr`;
- la classe B appartiene (*belongsTo*) ad un oggetto della classe A, con un `ptr`.

Questo non può essere implementato con due `ptr`, perché altrimenti si creerebbe un ciclo che causerebbe un'interferenza con l'implementazione di `ptr`, costruita su un puntatore condiviso basato su un contatore. Allo stesso modo, un `weak_ptr` non può essere usato al di fuori di una relazione 1-1. Un `weak_ptr` ha le stesse capacità di un `ptr`, se non che non possiede l'operatore di dereferenziazione.

`WtSqlTraits`

I file `WtSqlTraits` contengono il supporto ai tipi della libreria Wt, specializzando la classe `sql_value_traits`. I tipi supportati da questa porzione della libreria sono:

- `WDate`;
- `WDateTime`;
- `WTime`;
- `WString`.

3.2 Analisi dei backend

Come si è detto, la libreria Wt::Dbo fornisce il supporto per i seguenti DBMS:

- Firebird;
- SQLite3;
- MySQL;

- PostgreSQL.

I vari file di backend si differenziano, in quanto devono implementare i diversi dialetti del SQL. I file di backend contengono la gestione di tutte quelle porzioni di codice che permettono un collegamento effettivo ad un DBMS; mentre nel codice visto in precedenza l'implementazione era completamente slegata dal DBMS desiderato.

Prendendo come esempio il backend SQLite3, la visione del codice ha mostrato che vengono specializzate le classi "standard" presenti nel codice generale: `SqlConnection`, `SqlException` e `SqlStatement` diventano, rispettivamente, `Sqlite3Connection`, `Sqlite3Exception` e `Sqlite3Statement`. Sono presenti, in override, anche le funzioni descritte in precedenza: il metodo `bind()`, atto alla mappatura dei tipi C++ in tipi SQL; il metodo `getResult()`, che restituisce i risultati delle query in relazione al tipo richiesto. Questo meccanismo permette al programmatore di disinteressarsi del backend durante la scrittura del codice sorgente, dovendo solamente specificarlo durante la creazione della connessione con il database: è la libreria che pensa alla specializzazione degli oggetti e dei metodi.

Naturalmente la descrizione presentata per il backend SQLite è valida per i backend di tutti gli altri DBMS supportati.

3.3 Modalità di estensione

La libreria può essere estesa per quanto riguarda il supporto dei tipi: i file che contengono tale funzionalità sono `StdSqlTraits` e `StdSqlTraits.C`; in particolare per estendere il supporto ad un nuovo tipo di dato è necessario aggiungere in entrambi una specializzazione della struct `sql_value_trais<T>` per il tipo desiderato. In questo lavoro di tesi si è deciso di estendere la libreria Wt::Dbo in modo da supportare il tipo `unsigned long`.

In particolare, la porzione di codice aggiunta nel file `StdSqlStatement` è la seguente:

```
1  template<>
2  struct WTDBO_API sql_value_traits<unsigned long, void> {
3    static const bool specialized = true;
4    static const char* type(
5      SqlConnection* conn, int size
6    );
7    static void bind(
8      unsigned long v,
9      SqlStatement *statement,
10     int column,
11     int size
```

```
12    );
13    static bool read(
14      unsigned long& v,
15      SqlStatement *statement,
16      int column, int size
17    );
18  };
```

Al file `StdSqlTraits.C` è stata invece aggiunta la seguente porzione di codice:

```
1   const char* sql_value_traits<unsigned long>
2   ::type(
3     SqlConnection *conn,
4     int size
5   ) {
6     return "unsigned not null";
7   }
8
9   void
10  sql_value_traits<unsigned long>
11  ::bind(
12    unsigned long v,
13    SqlStatement* statement,
14    int column,
15    int size
16  ) {
17    statement->bind(column, static_cast<long long>(v));
18  }
19
20  bool
21  sql_value_traits<unsigned long>
22  ::read(
23    unsigned long& v,
24    SqlStatement* statement,
25    int column,
26    int size
27  ) {
28    bool result = statement->getResult (column, &value) ;
29    v = static_cast <unsigned long>(value) ;
30    return result ;
31  }
```

La libreria Wt::Dbo è dunque estendibile, per quanto riguarda il supporto ai tipi del C++. Estendere la libreria non è sempre semplice: le struct templatiche `sql_value_traits` non permettono di prendere due argomenti templatici, il che impedisce, per esempio, di estendere il supporto ad un qualunque tipo di contenitore di `char` (tramite due puntatori `unsigned char*`

che indicano inizio e fine del contenitore). Cercare di aggiungere una seconda struct templatica che permetta tale estensione genera una serie di errori a cascata che coinvolgono file più complessi della libreria. Un altro problema, dovuto alle limitazioni del linguaggio C++, è l'impossibilità di utilizzare metodi virtuali templatici.

Superando tali difficoltà si è voluto mostrare un esempio che, per quanto semplice, fosse in grado di spiegare come aggiungere il supporto ad un tipo primitivo del C++: si è deciso di includere il supporto per il tipo `unsigned long`. Per aggiungere altri supporti, la modalità di lavoro è la medesima.

3.3.1 Mapping dei tipi per SQLite

Come detto in precedenza, il mapping dei tipi dal C++ in SQL viene fatto grazie alla funzione `bind()`, che, noto il tipo da mappare e il backend a cui connettersi, attua la conversione. Nel caso specifico, per fare un esempio, SQLite3 accetta numerosi tipi di dato (circa una ventina), che vengono poi mappati dal DBMS in cinque tipi "base", seguendo le seguenti regole:

1. se il tipo dichiarato contiene la stringa `INT`, allora viene assegnato al tipo `INTEGER`;
2. se il tipo dichiarato contiene una delle stringhe `CHAR`, `CLOB` o `TEXT`, la colonna prende tipo `TEXT`;
3. se il tipo dichiarato contiene la stringa `BLOB` o non viene specificato un tipo, allora la colonna prende tipo `NONE`;
4. se il tipo dichiarato contiene una stringa qualunque tra `REAL`, `FLOA` o `DOUB`, allora la colonna prende tipo `REAL`;
5. in ogni altro caso, la colonna prende tipo `NUMERIC`.

Considerando questo "mapping interno" di SQLite, la libreria effettua le seguenti conversioni:

- `std::string` in `TEXT`;
- `short` in `INTEGER`;
- `int` in `INTEGER`;
- `float` in `REAL`;
- `double` in `DOUBLE PRECISION`;

- `boost::data` in `INT64`;
- `std::vector<unsigned char>` in `BLOB`.

3.4 Approcci all'ORM

Nell'estendere la libreria si è presentata la necessità di scegliere l'approccio di estensione. Nell'ORM infatti, esistono due approcci per fornire la persistenza degli oggetti C++: il primo si concentra sulla "O" (*Object*), il secondo sulla "R" (*Relational*).

Per comprendere entrambi gli approcci bisogna pensare a come memorizzare un oggetto, considerando quali informazioni dell'oggetto stesso sono interessanti.

3.4.1 Approccio basato sulle associazioni

Questo approccio vede l'oggetto come un insieme di componenti separate in relazione tra loro. Un oggetto è, in altre parole, una collezione di sotto-oggetti: a livello di RDBMS questa visione viene implementata mappando i campi dell'oggetto come chiavi esterne, collegando tramite associazioni le varie tabelle. I casi d'uso di questa modalità sono quelli in cui lo sviluppatore vuole avere la possibilità di effettuare interrogazioni che coinvolgono più (porzioni di) oggetti, memorizzate in una o più tabelle; lo sviluppatore è quindi interessato ad avere (anche) una visione più completa del database, non limitata ad una collezione di oggetti monolitici. L'esempio visto in precedenza, con gli studenti, le iscrizioni e i corsi, rientra in questo ambito: chi consulta la base di dati ha la necessità di interrogare separatamente la tabella dei corsi, quella delle iscrizioni e quella degli studenti. Seguendo tale approccio nell'estensione della libreria, i passi da fare sono nella direzione di ampliare il supporto ad un numero sempre maggiore di tipi, per garantire il più alto numero di associazioni tra tabelle del database.

3.4.2 Approccio basato sugli oggetti

Questo approccio si basa su una visione "monolitica" dell'oggetto da rendere persistente. Al contrario dell'approccio precedente, l'utente non è interessato alla base di dati dal punto di vista "relazionale", ma al puro storage. Questa tecnica viene utile nel caso ci siano oggetti complessi; tali oggetti vengono serializzati (in un qualunque formato), e salvati in una tabella del database che per esempio contiene un `id` e un campo di tipo `blob` in cui viene memorizzata la serializzazione dell'oggetto. Per esempio, questo approccio può essere

utilizzato per memorizzare in un database un insieme di numeri a precisione arbitraria della libreria GMP. Il valore del numero a precisione arbitraria, pur essendo tipicamente composto da un certo numero di componenti, viene serializzato come un unico blocco dati, tipicamente in formato binario (o in formato stringa, se si preferisce una maggiore leggibilità e portabilità, a scapito delle prestazioni). Estendere la libreria utilizzando tale approccio vuol dire modificare gli `std_sql_traits` con chiamate a funzioni di serializzazione e deserializzazione. Dunque, mentre nell'approccio precedente la gestione degli oggetti avviene tramite il database, ora avviene a livello di C++. Per il momento non si parlerà della serializzazione di array o altre struttre dati simili, poiché non tutti i DBMS supportano tali costrutti.

Esempio di serializzazione: `mpz_t`

Un numero intero a lunghezza arbitraria della libreria GMP viene gestito dal tipo `mpz_t`. Tali numeri vengono inizializzati con due componenti: una stringa contenente la scrittura del numero e la base in cui va letto. A questo punto la serializzazione può essere gestita tramite funzioni custom, oppure utilizzando quelle fornite dagli stessi sviluppatori di GMP. Per rendere persistente una classe che contiene un `mpz_t` seguendo l'approccio basato sugli oggetti è possibile sfruttare il seguente esempio.

```
1  namespace dbo = Wt::Dbo;
2
3  class MPZ_T {
4    public:
5    mpz_t number;
6
7    private:
8    std::string s_number;
9
10   void set_s_number(mpz_t n, int base) {
11     s_number = mpz_get_str(NULL, base, n);
12   }
13
14   public:
15   void set_number (mpz_t n, int base) {
16     mpz_set(number, n);
17     set_s_number(number, base);
18   }
19   template<class Action>
20   void persist(Action& a){
21     dbo::field(a, s_number, "s_number");
22   }
23 };
```

Nella parte `private` della classe viene definita una stringa, `s_number` in cui viene inserita la serializzazione del campo `number`, eseguita in automatico alla chiamata del *setter* del campo stesso. Tale meccanismo può essere implementato all'interno della libreria Wt::Dbo: nei `std_sql_traits` è possibile creare una funzione `bind()` specializzata per gli `mpz_t`, che chiama un'apposita funzione di serializzazione; allo stesso modo una specializzazione della funzione `getResult()`, che chiamerà una funzione di deserializzazione. Rendendo persistente solamente la stringa, la chiave primaria è un id con flag `AUTOINCREMENT` attivo; la tabella conterrà dunque un identificatore e un campo testuale.

Per la serializzazione binaria si rimanda al lavoro di Fabio Trabucchi sulla PPL, creatore di funzioni di serializzazione binaria per gli oggetti della PPL. Tali funzioni implementano anche la compressione dello stream.

Conclusioni

In questo lavoro di tesi si è fornita una categorizzazione di alcune soluzioni ORM, scelte in base a criteri precisi, al fine di poter confrontare le soluzioni utili e comprendere meglio alcuni meccanismi che regolano tale tecnica di programmazione. Si sono scelte due soluzioni in particolare, che sono state confrontate anche tramite esempi pratici, visionando il corrispondente codice C++ e SQL per capire al meglio come i diversi prodotti si interfacciano a questo problema. Si è presa una soluzione, la più aderente alle caratteristiche desiderate, quali la flessibilità, il linguaggio supportato e la tipologia di licenza e di accessibilità del codice sorgente; tale soluzione è stata esaminata approfonditamente, studiandone il codice sorgente e il supporto ai backend. Si è infine visto come estendere tale soluzione, proponendo due diverse tipologie di approccio.

Si tenga presente che le soluzioni scartate nei diversi passi del lavoro non sono necessariamente peggiori di quelle mantenute fino alla fine: semplicemente non rispondevano alle caratteristiche fissate inizialmente. Spesso si sono escluse soluzioni ORM estremamente performanti ed interessanti; purtroppo sotto punti di vista differenti.

Il frutto di questo lavoro è una panoramica sulle soluzioni ORM più o meno aderenti alle proprietà volute: quello che se ne può dedurre è che il mondo delle soluzioni ORM è estremamente ampio, con prodotti estremamente differenti gli uni dagli altri. Probabilmente, cambiando anche leggermente le variabili usate per la discriminazione delle soluzioni, si sarebbero scelti prodotti differenti.

Purtroppo, anche per motivi di tempo limitato, il lavoro non ha potuto esplorare tutte gli approcci proposti per risolvere il problema in questione. Questo lavoro vuole essere anche il punto di partenza per diversi sviluppi futuri, offrendo diverse strade da seguire. Per esempio: all'interno delle tabelle generate da Wt::Dbo è presente un campo di default che indica la versione della tabella a cui si sta accedendo. Sarebbe interessante studiare come le diverse soluzioni sul mercato gestiscano il controllo delle diverse versioni del database, in modo tale da aggiornare il database ad ogni variazione del codice

dell'applicazione. È anche possibile stilare una sorta di classifica che indichi quale soluzione affronti meglio il problema. Un altro ambito che meriterebbe forse di essere maggiormente approfondito è stato quello legato all'estensione delle librerie utilizzando l'approccio "monolitico" indicato nell'ultimo capitolo. Sarebbe interessante concentrarsi sulla serializzazione legata agli approcci ORM, per verificare anche quale soluzione abbia performance migliori in termini di spazio e tempo.

Bibliografia

[1] Free Software Foundation, Inc. *GNU General Public License, version 2*
http://www.gnu.org/licenses/gpl-2.0.html

[2] Free Software Foundation, Inc. *GNU General Public License, version 3*
http://www.gnu.org/licenses/gpl-3.0.html

[3] Microsoft Developers
Microsoft Entity Framework
http://msdn.microsoft.com/en-us/data/ef.aspx

[4] The Apache Software Foundation
Apache License
http://www.apache.org/licenses/LICENSE-2.0

[5] The Qt Company
QxOrm 1.3.2 and QxEntityEditor 1.1.8
http://www.qxorm.com/qxorm_en/home.html

[6] The Qt Company
QxOrm 1.3.2: C++ Object Relational Mapping library
http://www.qxorm.com/doxygen/html/index.html

[7] Code Synthesis Developers
ODB Manual
http://www.codesynthesis.com/products/odb/doc/manual.xhtml

[8] Emweb Developers
Wt::Dbo Tutorial
http://www.webtoolkit.eu/wt/doc/tutorial/dbo/tutorial.html

[9] Emweb Developers
Wt::Dbo Documentation
`http://www.webtoolkit.eu:3080/wt/doc/reference/html/group__dbo.html`

[10] SQLite Developers
SQLite
`http://www.sqlite.org/`

[11] The PostgreSQL Global Development Group
PostgreSQL
`http://www.postgresql.org/`

[12] Boost Developers
Boost: C++ Libraries
`http://www.boost.org/`

[13] Free Software Foundation, Inc.
The GNU Multiple Precision Arithmetic Library
`http://gmplib.org/`

[14] Roberto Bagnara, Patricia M. Hill, Enea Zaffanella, Abramo Bagnara
Parma Polyhedra Library Credits
`http://bugseng.com/it/prodotti/ppl/credits`