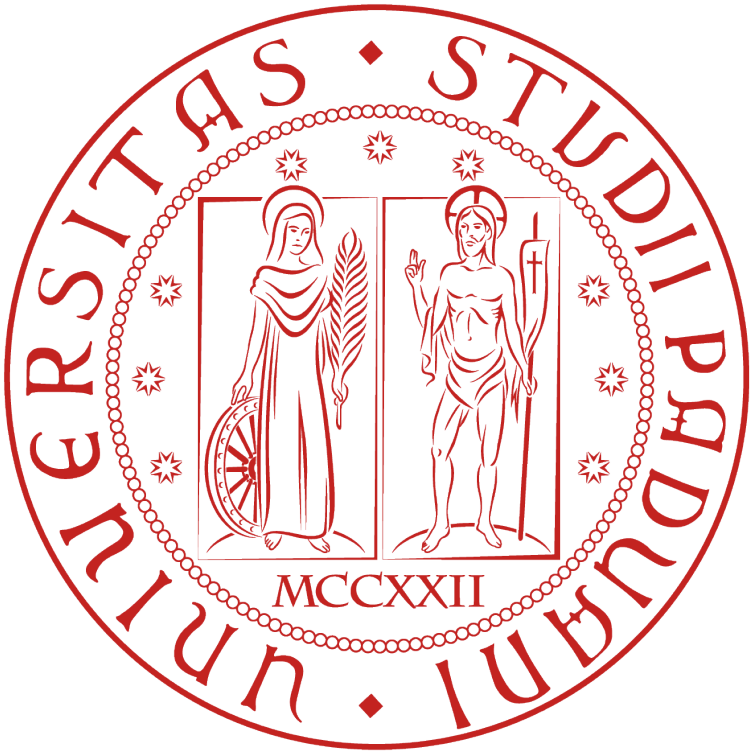


Università degli Studi di Padova
Dipartimento di Matematica
Corso di Laurea in Informatica

Relazione del progetto di Programmazione ad Oggetti: qDB

Zilio Matteo - 578200

Padova, 28 giugno 2014

Introduzione

Lo scopo del progetto qDB è lo sviluppo in C++/Qt di un sistema minimale per la gestione di un (piccolo) database tramite un'interfaccia utente grafica.
Si è scelto di creare un'applicazione che simuli il database del catasto, dentro al quale sono memorizzati i dati di tutti i beni immobili con la possibilità, per ogni oggetto inserito, di calcolare l'IMU secondo quanto stabilito dall'Agenzia delle Entrate.

Template di classe `Container<K>`

È stato definito un template di classe `Container<K>`, istanziabile a qualsiasi oggetto `K`, che svolge il ruolo vero e proprio del database. Durante la prima fase del progetto si è ritenuto di implementare il template con una struttura ad albero binario per gestire più velocemente la ricerca degli oggetti in esso contenuti. All'interno del template si trovano dunque, oltre i metodi propri, una classe `Item` che rappresenta i nodi dell'albero e una classe `Iterator` che permette di muoversi agevolmente sui nodi dell'albero.

`Item`
Il nodo dell'albero binario, rappresentato da questa classe, è costituito da un oggetto templatizzato di tipo `K` che, una volta istanziato, conterrà i dati e da due puntatori ad `Item` che puntano al figlio sinistro e destro dell'albero. L'unica richiesta che si impone all'oggetto di tipo `K` è che abbia definito l'`operator<`, ovvero che l'oggetto sia ordinabile. La classe è stata definita nella parte privata del template per questioni di information hiding.

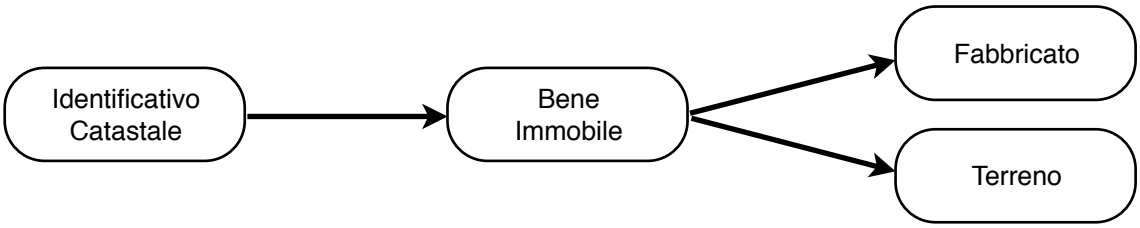
`Iterator`
Questa classe è stata definita nella parte pubblica del template per permettere anche alle classi esterne di usufruire di questi oggetti che implementano metodi di utilità per il template. Nel caso particolare, gli oggetti di tipo `Iterator` sono stati pensati per meglio muoversi all'interno dell'albero. Essi sono, infatti, dotati semplicemente di un puntatore ad un nodo dell'albero.

Metodi del template
I metodi propri del template sono i seguenti:

- Costruttore: inizializza la struttura dati ad un albero vuoto;
- Costruttore di copia: crea un nuovo albero copiandolo nodo per nodo da un albero dato;
- Distruttore: invoca il distruttore sulla radice, se presente;
- `operator=`: se i due alberi non sono già gli stessi, l'albero di invocazione viene cancellato e viene copiato dall'albero passato come parametro;
- `Minimum`: ritorna un `Iterator` al nodo minimo che si trova, partendo dalla radice, scendendo l'albero sempre a sinistra;
- `Maximum`: ritorna un `Iterator` al nodo massimo che si trova, partendo dalla radice, scendendo l'albero sempre a destra;
- `Successor`: ritorna un `Iterator` al nodo che segue il nodo attuale, secondo l'ordine dato;
- `Predecessor`: ritorna un `Iterator` al nodo che precede il nodo attuale, secondo l'ordine dato;
- `Size`: ritorna un intero che indica il numero di nodi da cui è composto l'albero;
- `Empty`: ritorna vero se l'albero è vuoto, falso altrimenti;
- `AddItem`: inserisce, secondo l'ordine definito dalla classe `K`, l'oggetto all'interno dell'albero;
- `FindItem`: ritorna un `Iterator` che punta all'oggetto cercato, se non è presente punta a 0;
- `RemoveItem`: cerca l'oggetto da rimuovere e, se presente, lo rimuove mantenendo ordinato l'albero;
- `operator[]`: ritorna l'oggetto puntato da un `Iterator`.

Gerarchia di classi

La gerarchia di classi scelta rappresenta i beni immobili presenti sul territorio nazionale.



La classe `Error` è di utilità in quanto viene utilizzata dalla gerarchia per lanciare eccezioni e presenta un unico campo dati di tipo stringa che la classe predisposta a lanciare l'eccezione può utilizzare per descrivere l'errore.

Anche la classe `IdentificativoCatastale` risulta essere di utilità per la gerarchia in quanto raggruppa logicamente più campi dati che hanno senso solamente assieme, funziona da chiave nella nostra istanza del template in quanto, nella realtà, non possono esistere due diversi beni immobili con lo stesso identificativo. Inoltre, come esplicitamente richiesto dal template stesso, al fine di poter inserire ordinatamente gli oggetti nel `Container<K>`, la classe rende disponibili i classici operatori per il confronto e l'ordinamento. Il costruttore di questa classe può lanciare eccezioni tramite la classe `Error` nel caso si tenti di creare un oggetto con dei valori al di fuori del range stabilito dalle normative.

`Record` è la classe con la quale abbiamo istanziato il template `Container<K>`, implementa un puntatore smart a `BeneImmobile`.

La classe base `BeneImmobile` raccoglie al suo interno i campi dati privati comuni a tutti i tipi di beni.
Nel particolare troviamo: `IdentificativoCatastale` che identifica in modo univoco un bene, un campo dati `Proprietario` che memorizza l'attuale proprietario, il campo `RenditaCatastale` che descrive, appunto, la rendita del suddetto bene imposta dallo Stato. La classe definisce inoltre gli operatori di confronto ed ordinamento per aderire alle specifiche del template. È presente anche un campo dati che si è scelto di dichiarare pubblico perché rappresenta un'aliquota di base decisa dal governo e può risultare utile agli utenti della classe.
`BeneImmobile` è una classe astratta in quanto al suo interno definisce, oltre al distruttore virtuale, il metodo `calcoloImu` ed il metodo `clone`, entrambi definiti come virtuali puri. Questo comporta l'impossibilità di creare oggetti della classe base che funzionerà semplicemente da interfaccia comune per le classi derivate.

`Fabbricato` è una classe derivata pubblicamente da `BeneImmobile` ed è in grado di rappresentare tutti i tipi di fabbricati presenti sul territorio. Ha quattro campi dati privati, in aggiunta a quelli ereditati dalla classe base, che permettono di descrivere il fabbricato e anche in questa classe troviamo sette campi dati dichiarati pubblici perché rappresentano aliquote o moltiplicatori che potrebbero risultare di utilità. Implementa i metodi virtuali puri rendendola di fatto una classe concreta.

Anche `Terreno` deriva pubblicamente da `BeneImmobile` e rappresenta l'altra metà dei beni immobili presenti sul territorio.
Eredita anch'essa campi dati e metodi dalla classe base, definendone i metodi virtuali puri. Inoltre aggiunge due campi dati pubblici.

Interfaccia Grafica

Per realizzare l'interfaccia grafica ci si è appoggiati alla libreria Qt che mette a disposizione una vasta quantità di codice e un'ottima documentazione. Sono state create tre classi: `MainWindow`, `CentralWidget` e `SearchDialog`.

La principale classe dell'interfaccia grafica è `MainWindow`, derivata pubblicamente dalla classe `QMainWindow`. Si è scelto di derivare `QMainWindow` in quanto fornisce la finestra principale per le applicazioni alla quale si possono aggiungere tools utili come `QMenuBar`, `QToolBar` e `QStatusBar`. Contiene anche lo spazio per un widget, nell'area centrale della finestra,.
Nella sua progettazione si è scelto di inserire un puntatore alla classe `Controller` per rendere agevole la comunicazione fra le due classi in relazione alla struttura definita dal design pattern MVC.
Lo scopo di questa classe è fornire i principali strumenti per l'interazione con l'utente.

Il widget, che trova posto al centro del `MainWindow`, è rappresentato dalla classe `CentralWidget` derivata pubblicamente da `QWidget`. Costituisce una maschera che viene utilizzata per l'inserimento e la modifica dei dati nel database e per mostrare i dati ottenuti dalla ricerca.
Affinché potesse essere utilizzabile per tutti questi scopi contemporaneamente si è scelto di agire sulla proprietà `readOnly` dei campi di testo. Ad esempio, quando viene mostrato il risultato di una ricerca, i campi vengono bloccati in modo che l'operazione di modifica sia voluta e non accidentale.
Al fine di limitare possibili errori durante l'inserimento di dati non conformi si è scelto di implementare, in questa classe, dei `QValidator` realizzati tramite `QRegExp` che possano effettuare il controllo prima

dell'immissione dei dati all'interno del `Container<K>`. Per agevolare l'utente ad effettuare un inserimento corretto e in linea coi diversi `QValidator`, si è fatto ricorso ai `placeholderText`.

Alle due classi precedenti si aggiunge la classe `SearchDialog`, che deriva pubblicamente da `QDialog`. `SearchDialog` riceve in ingresso dati, richiesti all'utente, per effettuare una ricerca. `QDialog` è stata studiata per gestire un compito breve; alcune delle sue peculiarità sono di essere sempre in primo piano e di interrompere le operazioni sulle altre finestre dell'applicazione se prima non viene fornita una risposta alla richiesta di informazioni da parte del programma.
Anche in questa classe, che si occupa di ricevere in ingresso dati dall'utente, sono stati implementati come nella classe precedente, i `QValidator` e i `placeholderText` opportuni.
Si è utilizzato un `QHash`, associando come chiave il tipo di campo e come valore il termine della ricerca, per agevolare il passaggio dei parametri di ricerca al `Controller`.

Ricerche permesse

I campi sui quali è permessa una ricerca sono quelli dell'oggetto `IdentificativoCatastale`.
Per poter effettuare la ricerca si richiede che vengano inseriti tutti i campi dati che lo compongono.
Se il bene cercato è presente vengono mostrati i dati dell'oggetto in `CentralWidget`, altrimenti viene mostrata una `QMessageBox` che avvisa l'utente della ricerca fallita.

Input/Output su file

Per l'I/O su file si è scelto di utilizzare il linguaggio XML perché standard internazionale di facile implementazione. È possibile utilizzare due classi messe a disposizione da Qt: `QXmlStreamReader` e `QXmlStreamWriter` rispettivamente per la lettura e per la scrittura di file formattati in XML.
Si allega al progetto un file di prova (dati.xml)

Design Pattern MVC

Il progetto è stato implementato seguendo il design pattern Model-View-Controller che realizza un disaccoppiamento tra: dati, rappresentazione grafica e reazione dell'interfaccia grafica agli input degli utenti.
Il model si occupa di gestire i dati e di fornire tutte le operazioni per poterli manipolare. Rientrano in questa categoria le classi `Container<K>`, `IdentificativoCatastale`, `BeneImmobile`, `Fabbricato`, `Terreno`, `Record`, `Error`.
La view, costituita da `MainWindow`, `CentralWidget` e `SearchDialog`, gestisce la logica di presentazione, cattura gli input utente e delega al controller l'elaborazione.
Il controller, rappresentato dalla classe `Controller`, trasforma le interazioni dell'utente, provenienti dalla view, in azioni sui dati contenuti nel model ed è per questo che, fra i campi dati privati della classe, troviamo un puntatore al `Container<K>` ed uno a `MainWindow`. Infatti per svolgere il proprio compito deve avere la possibilità di comunicare con entrambe le entità.

Piattaforma di sviluppo

Sistema Operativo:	Mac OS X 10.7.5
Compilatore C++:	GCC 4.2.1
Libreria Qt:	4.8.4