Matteo Di Pirro

Matricola 1074041

Programmazione ad Oggetti

AA 2014/2015

Relazione sul progetto

LinQedIn

Versioni usate nello sviluppo

Lo sviluppo è avvenuto in ambiente Windows, con Qt 5.3.2 e compilatore MinGW 4.8.2.

Precisazioni

Si è scelto di utilizzare le classi rese disponibili dalla libreria Qt anche per la parte di modellazione della realtà. Questa scelta può sembrare svantaggiosa, perché un cambio di libreria grafica comporterebbe la riscrittura e l'aggiornamento di tutto il codice. Tuttavia, il fatto che Qt mette a disposizione funzionalità per qualsiasi bisogno, unito alla sua grande diffusione, rendono poco probabile un cambio di libreria. Inoltre uno degli scopi del progetto è l'acquisizione di familiarità con questa libreria, perciò utilizzarla è sembrata una buona scelta. Inoltre, l'integrazione tra STL e Qt rende necessari moltissimi cast (ad esempio, da stl::string a QString) per la gestione della GUI. L'uso di Qt anche nella modellazione evita questi problemi.

Controller tra GUI e utenti

Si è scelto di aderire al design pattern MVC. La parte di "controller" è rappresentata dalle tre classi *LinQedInAdmin*, *LinQedInClient* e *LinQedInUser*. Di seguito una descrizione di ciascuna di esse:

- *LinQedInAdmin* rappresenta l'amministratore del database, che viene caricato ed inizializzato in lettura/scrittura. Le operazioni di inserimento e rimozione utilizzano l'interfaccia della classe *Database*, analizzata più avanti. L'inserimento, in particolare, richiede solamente le informazioni di base (nome, cognome, indirizzo email, sesso e tipologia di account). Si è scelto di non richiedere la password: l'utente infatti non avrebbe modo di conoscerla (l'inserimento è effettuato dall'amministratore). La procedura viene completata utilizzando come default la stringa "password". Questo è ovviamente poco sicuro, ma l'utente può cambiare la propria password in ogni momento.
  L'operazione di cambio di tipologia di account funziona rimovendo e reinserendo lo stesso utente, ma con una tipologia diversa: l'abbonamento viene rinnovato usando la data corrente come data di pagamento.
  L'operazione di ricerca nel database viene eseguita con l'aiuto di una gerarchia di funtori, con classe base *SearchFunctor*. Ad ogni tipo diverso di ricerca corrisponde un funtore nuovo, che estende *SearchFunctor* con un override dell'operator().
- *LinQedInClient* rappresenta un utente loggato al servizio LinQedIn. È un'astrazione di *SmartUser*, usata per rappresentare gli utenti nel database e permette un numero ridotto di operazioni sugli utenti. L'operazione di ricerca è eseguita con gli stessi funtori già menzionati, ma viene sollevata un'eccezione nel caso in cui l'utente loggato non sia abilitato a cercare per username. La ricerca per nome e cognome, disponibile per tutti, viene eseguita sempre. Viene ritornato un QVector di *SearchResults*, classe che sarà analizzata in seguito.
- *LinQuedInUser* rappresenta un generico utente dell'applicazione, ma non l'utente loggato. È un'astrazione di *SmartUser* che permette di accedere ad informazioni selezionate sugli utenti nella GUI senza aver accesso effettivamente al database.

Database

La classe *Database* fa uso del Singleton Design Pattern. L'uso di questo modello è motivato dal fatto che ci deve essere solamente un'istanza del db in memoria. Tutte le richieste di load successive alla prima ottengono un puntatore al db già in memoria. In questo modo la gestione risulta molto semplificata. Costruttori, distruttore e operato= sono dichiarati privati, in modo tale che si possa creare (o ottenere) un'istanza del db solo attraverso il metodo load (lettura/scrittura) o constLoad (sola lettura). La memorizzazione del db su file utilizza un file .xml, in cui ogni utente (nodo *user*) ha nodi per le informazioni di login (*login*, con username e password), per il pagamento (*payment*, solo se non è basic) e per le informazioni generali (*information*, con un figlio *info* per ogni categoria di informazione disponibile). I nodi *value* rappresentano le foglie dell'albero XML, e contengono valori che saranno interpretati in modo diverso a seconda del nodo padre. Infine, l'attributo *type* di *user* indica il tipo di utente (1=basic, 2=business, 3=executive); il *type* di *informations* rappresenta la stringa identificativa di una categoria di informazioni, che sarà analizzata in seguito. Per la rete di contatti viene memorizzato un file separato, in modo da garantire una minimale separazione tra utenti e contatti. La scelta di modellare il db di utenti in modo diverso, ovvero con file diversi per ogni tipo di informazione, sarebbe stata possibile, ma sarebbe stata molto più complessa: la gestione delle informazioni è stata strutturata in modo che nessuno, al di fuori delle classi che estendono *Information* conosca le stringhe identificative di ogni categoria. Inoltre ogni utente può inserire o meno le proprie informazioni (ad eccezione di quelle personali, specificate in fase di inserimento nel db). Quindi, per caricare tutti dati di un utente sarebbe stato necessario mantenere aperti tutti i file relativi alle categorie, verificando, di volta in volta, la presenza o meno di informazioni relative ad uno specifico utente. Sarebbe stato possibile farlo in modo sequenziale (ovvero senza ricerche continue su file) solo mantenendo ordinato il db, il che avrebbe richiesto O(n*logn) operazioni per ogni modifica (dove n è il numero totale di utenti). Si è preferito quindi usare un file unico. Al contrario, la rete dei contatti è memorizzata in un file separato: questo non ha comportato troppe operazioni aggiuntive o un grosso aumento di complessità. Si è scelto l'uso di una QList, al posto di QVector, per memorizzare il db in memoria perché un'operazione possibile è la cancellazione di un generico utente. La cancellazione in una posizione arbitraria ha reso necessario avvalersi di una lista.
I metodi insert e remove, non sono dichiarati const per una scelta precisa: se lo fossero sarebbero richiamabili anche da oggetti ritornati da constLoad(), ma questo non sarebbe corretto.
La classe *SmartUser*, puntatore smart di *User*, dichiara l'amicizia a *Database*. Il motivo di questa scelta è il seguente: *SmartUser* non è dichiarata interno a *Database* perché è utilizzata in molte altre classi (ad esempio *LinQedInClient* o *ContactsNetwork*). Tuttavia il legame con *Database* è molto più forte, perché alla chiusura del db devono essere azzerati tutti i riferimenti agli utenti per poterli cancellare (gli utenti sono memorizzati con condivisione di memoria). Un singolo utente però potrebbe avere moltissimi riferimenti (ad esempio tutti quelli nella rete di contatti), perciò quella di un ciclo sull'intero db per l'azzeramento delle liste di contatti è sembrata una scelta troppo pesante. Si è scelto invece di scorrere gli utenti una sola volta, e di impostare i loro riferimenti ad 1 in modo che possa essere richiamato il distruttore elemento per elemento. All'uscita di questo ciclo tutti i riferimenti sono a 0, e quindi tutti gli utenti sono deallocati. Un metodo per settare esplicitamente i riferimenti in un puntatore smart è però una scelta pericolosa. Per questo motivo è marcato private ed inaccessibile all'esterno. L'unica classe a potervi accedere è *Database*, grazie alla dichiarazione di amicizia.

Informazioni del profilo

Per gestire le possibili informazioni di ciascun utente è stata creata una gerarchia con classe base *Information*. L'unico campo dati di questa classe è un puntatore ad una QString, che rappresenta la stringa

identificativa di una categoria di informazioni (ad esempio, “Personali”, “Di Studio” e così via). Tutti i metodi forniti sono virtuali puri, e dovranno quindi essere “ridefiniti” dalle classi che la estendono. Il metodo getInformationList ha un parametro bool usato per determinare se la lista di informazioni verrà successivamente scritta su file (valore=false) o se dovrà essere stampata con una formattazione user friendly (valore=true).
La struttura di *Information* è del tutto generale: l’aggiunta di una nuova informazione risulta molto semplice, perché basta estendere la classe base. Per quanto riguarda il salvataggio nel db, l’attributo *type* di *informations* contiene proprio la stringa puntata dal campo di *Informations*. In questo modo la scrittura su file diventa semplicissima: è sufficiente richiamare getIDString() per la stringa identificativa e getInformationList() per ottenere un QVector contenente tutti i campi di una certa categoria di informazioni. La lettura da file risulta però più complicata, perché il costruttore non può essere virtuale. Si è scelto di applicare il Factory Method Design pattern, che permette di creare una classe *InformationCreator* con un unico metodo virtuale createInfo, che ritorna un *Information**, sfruttando il tipo di ritorno covariante di C++. createInfo decide, in base alla stringa initType, che classe di informazioni inizializzare e richiama il costruttore di quella classe, passando le informazioni necessarie all’inizializzazione in una QStringList. Sarà poi compito di ciascuna classe interpretare queste informazioni nel modo corretto. Questo assicura un buon livello di estensibilità, in quando è sufficiente estendere *InformationCreator* per aggiungere categorie di informazioni nuove.

Ricerca

Ogni tipologia di account ha associati dei permessi; in particolare: numero di risultati di una ricerca visibili, possibilità di vedere il profilo di un altro utente e possibilità di vedere la sua rete di contatti. Questi permessi sono gestiti dalla classe *Permissions*. La ricerca avviene utilizzando, come già detto, una gerarchia di funtori con *SearchFunctor* come base. Ogni classe derivata deve fare l’override di operator(), che verrà usato per decidere se uno *SmartUser* debba essere incluso o no nella lista di risultati. Un esempio è *SearchByNameSurname*, che viene inizializzato con due QString (nome e cognome) e inserisce nei risultati solo gli utenti che hanno nome e cognome uguali a quelli definiti. *SearchbyNameSurname* deriva da *SearchByName* e da *SearchBySurname*, che a loro volta estendono, come detto, *SearchFunctor*. Questo causa un’ereditarietà a diamante, gestita attraverso derivazione virtuale.
I risultati sono gestiti attraverso *SearchResults*, un’astrazione ai contenitori di Qt che permette di iterare, senza modificare, sui risultati di una ricerca. Per impedire l’accesso ad informazioni vietate (ovvero non pertinenti alla ricerca) viene usata una classe interna a *SearchResults*: *Result*, che permette di leggere solo nome, cognome, sesso, username, lavoro attuale e permessi di un certo utente. Per memorizzare i risultati si utilizza un QVector, in quanto non c’è necessità di rimozione, ma solo di inserimento in coda. La classe iteratore è *ScanResults*.

Utenti

La parte fondamentale dell’applicazione sono gli utenti, modellati attraverso la classe *User*. Ogni categoria di utente ha dei permessi specifici che verranno usati per differenziare la rappresentazione dei dati risultanti dalla ricerca. I permessi sono mostrati nella seguente tabella:

	Utente Basic	**Utente Business**	**Utente Executive**
Limite Risultati	10	50	Illimitati
Visualizzazione Profili	Dei Contatti	Dei Contatti	Tutti
Visualizzazione Contatti	No	Si	Si
Ricerca per Nome e Cognome	Si	Si	Si
Ricerca per solo Nome	No	Si	Si

Ricerca per solo Cognome	No	No	Si
Ricerca per Username	No	Si	Si

I permessi per le ricerche sono modellati usando un enum *SearchTypes*. Ogni tipologia di abbonamento ha associato un numero compreso nell'intervallo [0,7], con un totale di 2^3=8 stati possibili risultanti dall'interpretazione bit a bit del numero binario:

username surnameOnly nameOnly

Il valore 0, caratteristico degli utenti Basic indica la possibilità di ricerca solo per nome e cognome. Il valore 1 (=*nameOnly*) indica che è possibile anche fare una ricerca solo per nome, ottenendo tutti gli iscritti con il nome specificato (e cognome qualsiasi). Viceversa per *surnameOnly* (valore=2). *username* (valore=4) specifica la possibilità di cercare per username. Combinazioni di questi 3 bit danno luogo a 8 possibilità diverse.
La ricerca è virtuale pura, ed è soddisfatta da *BasicUser* e *FeeUser*. Successiamente, l'astrazione di *User* rappresentante un client (*LinQedInClient*) si occuperà di rappresentare i risultati in modo consistente ai permessi. In questo modo è sufficiente modificare i permessi per ogni classe di utente, senza dover cambiare l'intero metodo per la ricerca. Gli utenti sono inseriti nel db con l'aiuto di una classe smart pointer: *SmartUser*. La memoria riguardante ogni utente è condivisa. Per *User* vale quanto detto per *information* riguardo al salvataggio/lettura su file. Anche qui è dunque presente una classe *UserCreator*, interfaccia per il Factory Method Design Pattern. La classe *Profile* memorizza tutte le informazioni del profilo di un utente. Contiene una QMap che usa come chiavi le stringhe identificative delle categorie di informazioni e come valore un puntatore all'informazione stessa. Il puntatore è un puntatore smart, modellato attraverso la classe *SmartInfo* che opera con gestione profonda della memoria. La condivisione non sarebbe stata sensata, in quanto appare poco probabile che due utenti condividano le stesse informazioni. getInformationBySectionName() permette di ottenere una categoria specifica di informazioni sapendo la stringa identificativa; queste possono anche non essere presenti: in tal caso è sollevata un'eccezione. Le informazioni personali, tuttavia, ci sono sicuramente. Il metodo getPersonalInformation() permette di accedervi velocemente.
Le categorie di utenti sono modellate attraverso una gerarchia. *BasicUser* è l'utente base, con abbonamento gratuito. *BusinessUser* e *ExecutiveUser* derivano da una classe intermedia, *FeeUser*, che raccoglie informazioni sulla data di scadenza del piano.
Per la rete di contatti (classe *ContactsNetwork*) è stata scelta una lista. Il motivo è, anche qui, la possibilità di rimozione in una posizione arbitraria.

GUI

La GUI fa uso di tre finestre principali. *LoginWindow* è la finestra di login, dalla quale è possibile effettuare l'accesso come client o come admin. *ClientWindow* permette di utilizzare l'applicazione lato utente. La pagina principale, *ClientProfileWidget*, permette di visualizzare il proprio profilo, e di navigare tra i profili dei propri contatti (se i permessi lo consentono). Cliccando sul logo si trona al proprio profilo. È inoltre possibile, attraverso un menù, modificare le informazioni del proprio profilo e quelle di login, ottenere informazioni su LinQedIn, sul proprio abbonamento e di effettuare delle ricerche nel db. Le modalità di ricerca, e i risultati, differiscono in base ai permessi, secondo quanto detto sopra. Per i form di modifica delle informazioni e di ricerca sono state previste due classi base, *EditInfosWidget* e *SearchWidget*, che mettono a disposizione layout, segnali e slot utili per evitare ripetizioni e per mantenere una disposizione grafica uniforme. *AdminWindow* permette di utilizzare l'applicazione lato admin. All'apertura della finestra vengono mostrati due form per la ricerca (nome/cognome e username) e una lista contenente tutti gli utenti del db. Se vengono effettuate ricerche la lista si aggiorna per mostrare gli utenti che soddisfano i requisiti. Per visualizzare nuovamente tutto il db è sufficiente cercare "*" in "Cerca per username", oppure avviare qualsiasi ricerca senza parametri nel form. Sia *AdminWindow* sia *ClientWindow* utilizzano un pannello a schede per permettere di eseguire più operazioni insieme. La scheda principale, ovvero quella iniziale, non può essere chiusa. Infine, è stata prevista un'interfaccia, *GUIStyle*, che raccoglie colori è stile

(CSS) comuni a tutta l'applicazione. In questo modo si può ottenere una grafica uniforme, senza ripetizioni nel codice.