Relazione per il progetto di Programmazione ad Oggetti

2014-2015

LinQedIn

Prelaz Marco
1047343

- Sistema di sviluppo: Linux (Ubuntu 14.04 LTS)
- Compilatore: GCC 4.8.2
- Versione Qt: 5.2.1
- Versione Qt Creator: 3.0.1
- **Nota per la compilazione: il progetto include un proprio file .pro**
- **Viene reso disponibile inoltre anche un file .xml con un database di esempio**

Introduzione

In questa relazione mi sono soffermato sulle classi, che a mio parere, sono le più importanti, poiché queste contengono le funzionalità di spicco di questo progetto. Oltre a fare un discorso generale su di esse, entrerò nei dettagli,ove necessario, per poter motivare le scelte che ho effettuato. Ci saranno quindi soltanto alcuni cenni alle classi **Profilo** e **Username**, visto che non sono altro che dei semplici contenitori dati.

Classe Utente

La classe **Utente** è una classe *base astratta polimorfa*, quindi non esistono e non potranno mai essere istanziati oggetti di tipo Utente. Appartiene ad una gerarchia che si dirama in tre diversi sottotipi, tutti derivati dalla classe base Utente, che si differenziano tra loro soltanto per una diversa implementazione nella funzionalità di ricerca che permette loro di avere diversi permessi per quanto riguarda la visualizzazione del profilo dell'utente cercato.
Nello specifico un **UtenteBasic**, categoria di utente base, potrà visualizzare soltanto le informazioni riguardanti i dati personali, quindi più precisamente dati anagrafici e luogo di residenza; salendo in ordine di importanza (ricordando però che il livello di derivazione resta lo stesso per tutte le categorie di utente) troviamo l'**UtenteBusiness** che avrà inoltre la possibilità di ottenere informazioni riguardo ai contatti personali dell'utente, quali telefono e e-mail, e sapere qual'è il suo titolo di studio. Per ultimo, ma primo in ordine di importanza, abbiamo l'**UtenteExecutive**, che avrà le stesse caratteristiche dell'UtenteBusiness, con in più la possibilità di visualizzare quali lingue conosce e quali sono le esperienza lavorative passate dell'utente del quale voleva avere informazioni.
Tutto ciò è stato possibile grazie all'implementazione, nelle sottoclassi, del metodo virtuale puro *userSearch()*, che utilizza i così detti funtori, oggetti della classe annidata alla classe Utente, **SearchFunctor**, che tramite la ridefinizione dell'operatore di "chiamata di funzione" (*operator()*) mi ha permesso di distinguere le tre tipologie di ricerca. Ciò rende l'applicazione anche estendibile a nuove categorie di utente che potrebbero essere aggiunte in futuro; basterà infatti aggiungere semplicemente un "*case 4*" con nuove e diverse funzionalità rispetto a quelle dei tre tipi già esistenti.
Tornando a parlare degli Utenti, la caratteristica che invece accomuna tutti i tipi descritti precedentemente è quella di essere definiti, oltre che da un **Profilo** personale contenente tutte le informazioni che abbiamo già visto, anche da un **Username** al quale sarà associata la password scelta; inoltre ogni utente possiede una **Rete** di amici che vedremo approfonditamente solo dopo aver parlato di come è strutturato il database.

Classe DB

La classe **DB** ha il compito di raccogliere tutte le informazioni che contraddistinguono i vari utenti iscritti, compresa la loro **Rete** di amicizie. Dal punto di vista logico, il database è stato implementato tramite un **vettore** di puntatori a **Utente** (*vector<Utente*>*).
In particolare ho scelto di utilizzare come contenitore un vettore, della libreria *STL*, poiché gli *accessi in posizione arbitraria* avvengono in tempo ammortizzato costante, come per l'*inserimento*, visto che avviene sempre in coda (*push_back()*). È vero però che ciò non accade in fase di *eliminazione* (cosa che avrei invece se utilizzassi una **lista**) poiché ovviamente, a differenza dell'inserimento, questa avviene in posizione casuale. Vista la tipologia di applicazione, che si ispira al *social network* **LinkedIn**, a mio avviso è per l'appunto preferibile un vettore poiché le operazione di ricerca (che negli oggetti di tipo *list* è meno efficiente in quanto è necessario ogni volta percorre l'intera lista) verranno impiegate molto più frequentemente rispetto a quelle di cancellazione, che avverranno solamente in casi eccezionali.
Per quanto riguarda, invece, la scelta di utilizzare dei puntatori a Utente, c'è da dire che inizialmente avevo preso in considerazione l'alternativa **puntatori smart**, che però ho scartato quasi subito. Questo perché non sarebbe stata in alcun modo utile dato che, ci sarà sì condivisione di memoria, ma non sarà necessario gestirla in alcun modo; anzi l'eventuale decisione di aggiungere un campo dati *riferimenti* sarebbe stata addirittura scorretta, in quanto in LinQedIn non è vero che un oggetto Utente deve essere rimosso quando non ci sono più puntatori a lui, ma condizione necessaria e sufficiente è che l'**admin** decida di eliminarlo affinché questo non esista più. Infatti all'interno del metodo *elimina()* nella classe DB oltre alla rimozione del puntatore dal vector tramite *erase()* sarà presente anche un' invocazione al *distruttore* di Utente.
Un altra parte corposa della classe DB è sicuramente quella che riguarda il salvataggio e il caricamento su file *xml*. Per semplificare il più possibile la progettazione di queste due funzionalità ho fatto ricorso a due diverse librerie che *Qt* rende disponibili: *QXmlStreamWriter* e *QDomDocument*.
La prima è stata utile per la scrittura del file, in quanto questa libreria rende disponibile tutto ciò che serve per impaginare e strutturare al meglio i *tag* che compongono un file xml. Dopo aver associato il file, su cui andremo a scrivere, al *QXmlStreamWriter*, viene fatto scorrere l'intero database e ogni utente presente verrà identificato a *run-time* secondo il suo tipo, tramite *dynamic_cast*; verranno quindi memorizzate tutte le sue informazioni compresa sia la tipologia di utente, a seconda del risultato ottenuto dal *cast*, sia una stringa contenente l'elenco dei suoi amici (separati dall'identificatore ",") . Se l'utente è appena stato registrato o semplicemente non ha alcun altro utente presente nella propria rete, allora il tag "amici" sarà vuoto; stessa cosa vale per i campi non obbligatori della classe **Profilo**, ovvero quelli che non fanno parte del costruttore.
Per quanto riguarda la lettura, *QDomDocument* permette di creare una lista di *nodi*, corrispondenti agli utenti, attraverso i tag creati in fase di salvataggio. A questo punto è stato sufficiente scorrere questa lista e man mano prelevare tutti i dati dai vari tag ed infine controllare il tipo utente, sempre grazie al tag che opportunamente avevo memorizzato con

la funzione *salva()*, per creare un nuovo utente del tipo corretto. All'uscita del ciclo carico anche le reti di tutti i contatti tramite la funzione *caricaRete()* che essenzialmente funziona allo stesso modo con l'aggiunta del fatto che è necessario separare ogni amico e aggiungerlo uno per volta alla Rete dell'utente corrispondente (ricordo infatti che il tag era stato salvato come una singola stringa dove ogni utente era separato dall'altro tramite una “,”) .
È stato necessario dividere in queste due parti il caricamento (database e rete) poiché risultava impossibile caricare anche la rete congiuntamente a tutti gli altri elementi dato che quando inizio a caricare, ovviamente, non ho ancora tutti gli utenti presenti nel database.
La ragione per la quale ogni volta viene caricata la rete di ogni singolo utente e non solo quella dell'utente del quale è stato effettuato il *login* sta invece nel fatto che altrimenti, se non lo facessi, in fase di salvataggio perderei tutte le informazioni riguardante la rete di tutti gli altri utenti.

Classe Rete

La **Rete** rappresenta le amicizie di un utente ed è stata anch'essa realizzata tramite un *vector<Utente*>* con le medesime motivazioni che mi hanno spinto ad utilizzare lo stesso contenitore nella classe **DB**.
Di rilievo, a mio avviso, il metodo *getAmici* che, come già descritto nella sezione riguardante il salvataggio del database, serve a trasformare il vector rappresentante la rete in una semplice stringa in modo che questa possa essere salvata nel file xml. È importante però notare il fatto che questa funzione richieda come parametro un vettore contenente il database aggiornato, in modo da restituire solo gli utenti esistenti, e non anche quelli che potrebbero essere stati rimossi dall'*admin*, evitando così il rischio di avere dei *dangling pointer*.

I Controller

I *controller* non sono altro che delle classi che fungono da connessione tra parte *logica*, che abbiamo analizzato fino al punto precedente, e parte *grafica*, che vedremo in seguito.
Nello specifico la classe **LinQedInAdmin** ha il compito di interrogare la classe **DB**, in modo da ottenere le informazioni e metodi necessari a fornire tutte le risposte alle richieste effettuate dall'*interfaccia grafica* dedicata alla gestione del database.
Per quanto riguarda la parte dedicata all'utente la strutturazione è pressoché identica in quanto la classe **LinQedInClient** per rispondere all'*interfaccia grafica* corrispondente, dovrà interpellare le classi **Utente**, **Rete** e **DB**.

Interfaccia Grafica

Per lo sviluppo dell'*interfaccia grafica* ho deciso di sfruttare il *tool QtDesigner*, mentre ho definito le varie funzionalità, soprattutto per quanto riguarda gli effetti che si devono avere al momento di un *click* di un particolare *QPushButton*, nelle corrispondenti classi.
Quando verrà lanciato questo *software* sullo schermo comparirà l'**InterfacciPrincipale**, da qui sarà possibile effettuare il *login* tramite *username* e *password* e accedere quindi all'**InterfacciaUtente**; altrimenti, se si vuole gestire il database e i gli utenti che lo compongono, sarà sufficiente entrare nell'**IterfacciaAdmin** tramite l'apposito pulsante.
Nel primo caso si potrà modificare le informazioni che compongono il proprio profilo entrando nella finestra **InterfacciaModificaProfilo**, visualizzarlo (**InterfacciaProfiloUtente**) o cercare un utente, tramite username, per poter visualizzare il suo in **InterfacciaProfiloAmico**. Sarà inoltre possibile aggiungere o togliere amici dalla propria rete, sempre attraverso l'username, e infine visualizzare tutti questi o, in alternativa, tutti gli utenti presenti nel database in una *QTableWidget* che conterrà solo le informazioni principali, tra le quali l'username in modo che l'utente loggato abbia la possibilità di aggiungere agli amici un utente di cui non conosceva quel campo.
L'amministratore in più avrà la possibilità, come definito nella specifica del progetto, di registrare un nuovo utente nel database attraverso l'**InterfacciaRegistrazione** o di cambiare la tipologia di account di uno già esistente.
Poiché inoltre era richiesta una ricerca, oltre che per username, anche tramite nome e cognome ho stabilito che questi due campi dovessero essere identificativi per un utente, come lo è anche il campo username. È stato quindi necessario impedire, sia in fase di registrazione che di modifica profilo, l'inserimento nel database di più utenti con la coppia nome,cognome identica (anche se in realtà, per fare un esempio, esiste più di un “Mario Rossi” in Italia) oltre al fatto che sono state definite delle espressioni regolari per controllare l'input di campi che necessitano o soltanto lettere (come nome, cognome, città di nascita/residenza e provincia) o solo numeri, caratteristica tipica di ogni numero di telefono, con per entrambi i tipi ulteriori restrizioni per quanto riguarda il numero di caratteri che è possibile inserire.
In ogni *QlineEdit* presente nelle varie interfacce non verranno considerati eventuali spazi prima e dopo il testo che solitamente vengono inseriti per errore (tranne che per la password dove questi potrebbero essere voluti). Ciò serve, oltre che ad avere un aspetto grafico più gradevole quando si va a visualizzare un profilo, anche per evitare eventuali inconsistenze tra fase di registrazione/modifica e quelle di ricerca. Per lo stesso motivo in quest'ultima, in particolare solo in quella per nome e cognome dell'admin, non verranno considerati diversi nomi/cognomi che hanno le lettere scritte con caratteri maiuscoli o minuscoli da una parte e viceversa dall'altro; quindi per esempio “Luca” sarà uguale a “luca”.
Infine ho cercato di gestire tutte le possibili eccezioni che potrebbero verificarsi tramite alcuni *QMessaggeBox* che avvertono dell'errore commesso.