Relazione sul progetto di Programmazione ad Oggetti a.a. 2014/2015

LinQedIn

Nome: Alberto Ferrara
Matricola: 1049378

- Compilatore: GCC
- Versione Qt Creator: 3.2.1
- Versione Qt: 5.3.2
- Per la compilazione: è presente un file .pro
- Tempo compilazione: 12,84s
- Sistema Operativo: OS X Yosemite (10.10.2)
- Versione Macchina: MacBook Pro QuadCore i7 2.3 GHz

Indice

Capitolo 1

Panoramica

1.1 Compilazione

Dopo essersi posizionati nella cartella contenente i file di progetto, eseguire da terminale il comando qmake che genera il Makefile collegando tra loro i file .h, .cpp e .ui(interfaccia grafica).
Dopodiché sarà necessario semplicemente eseguire il comando make per compilare il progetto. Questo comando creerà nella cartella di compilazione un file eseguibile **LinQedIn** che potrà essere avviato mediante un doppio click.

1.2 Introduzione al progetto

In questo progetto si è cercato di sviluppare un software[1] che includesse le principali funzionalità del social network LinKedin. Sono rese disponibili funzionalità di amministratore del sistema e di utente. L'amministratore ha la possibilità di inserire, eliminare, cambiare tipologia di account (Basic, Business, Executive) all'utente.
L'utente ha la possibilità di modificare il proprio profilo, visualizzarlo e ricercare nel database gli utenti. A seconda del tipo di account sottoscritto vedrà il profilo degli utenti in modalità differenti. Potrà inoltre aggiungere un utente alla sua rete contatti e toglierlo.

1.3 Informazioni generali sul progetto

Il database che si è scelto di utilizzare è un vettore (vector<T> della STL) di puntatori ad utenti. In seguito verrà motivata questa decisione. La classe SmartUtente non è stata implementata perché la gestione della memoria in maniera condivisa non è stata presa in considerazione dal momento che, a mio parere, non sarebbero mai avvenute copie di Utente. Allo stesso tempo

[1] LinQedIn

non era necessario inserire un campo dati riferimento per il conteggio dei puntatori ad un oggetto Utente. Questo perché, anche se un oggetto Utente non sarebbe stato più *puntato* da nessuno, non doveva essere eliminato. Inoltre, SmartUtente sarebbe stata potenzialmente utile per la gestione dell'eventuale garbage provocato dall'eliminazione di un utente, ma potendo avvenire solo al momento della rimozione da parte dell'amministratore, la distruzione di un utente era racchiusa in un'unica funzionalità e quindi facile da gestire mediante una chiamata al distruttore virtuale di utente PRIMA della rimozione mediante erase() dal vector tramite l'apposito iteratore.

Capitolo 2

Struttura

2.1 Elenco delle classi modellate

La gerarchia di classi principale è la gerarchia degli utenti. Classe base astratta, Utente da cui derivano 3 sottoclassi: UtenteBasic, UtenteBusiness, UtenteExecutive. Abbiamo poi la classe LinQedInAdmin e LinQuedInClient come collegamento tra parte logica e grafica (in pratica il *controller*). Come database, abbiamo la classe DB che contiene un vettore di puntatori ad utente (polimorfi) che avranno differente tipo dinamico. Anche la classe Rete, i cui oggetti sono contenuti dentro la classe Utente, è caratterizzata da un vettore di puntatori ad utente. Per incapsulare l'username e la password di ogni utente è stata definita una classe Username contenuta sempre dentro utente. Infine per la visualizzazione del proprio profilo, del profilo della persona cercata, per la modifica del proprio profilo e per la registrazione di un nuovo utente sono presenti le seguenti classi: ShowProfile, ShowProfileFind, ModifyProfile, registrationwindow;

2.1.1 Classe utente e sue derivate

Tre sono le tipologie di utenti presenti in questo progetto e che, come da specifica, offrono funzionalità diverse. Ricordiamo che la classe base Utente è una classe base *polimorfa* e *astratta*, ovvero non è possibile istanziare oggetti di questa classe.
I diversi tipi di utenti hanno, come già menzionato, funzionalità e permessi diversi. Concretamente l'unica cosa che cambia è la funzionalità di ricerca (*find*) nel database degli utenti, che restituirà informazioni diverse a seconda del tipo di utente. Partendo dal basso, l'utente di base è l'UtenteBasic. Questo utente ha la possibilità di effettuare una ricerca del database e di ottenere le seguenti informazioni: nome e cognome. Ha inoltre, giustamente, a disposizione la funzionalità di inserimento di un utente nella sua rete dei

contatti. Altro tipo di utente, di livello intermedio, è l'UtenteBusiness. Questo, a differenza del basic, ha la possibilità di visualizzare anche tutto ciò che riguarda i contatti di un'altro utente, ovvero email, numero di telefono e indirizzo. La categoria di utente rimanente, ovvero l'UtenteExecutive ha la possibilità, oltre a quelle offerte dall'utente business, di visualizzare tutte le esperienze di curriculum fatte da un utente, ovvero esperienze didattiche, lavorative, personali e le lingue conosciute.
Tutto ciò è stato implementato tramite i cosiddetti Funtori. I funtori sono degli oggetti di una classe (definita appunto Functor) nella quale è stato fatto l'overloading dell'operatore -chiamata a funzione- (ovvero l'operatore operator()). Dunque, al momento dell'invocazione della find, a seconda del tipo (dinamico) dell'utente che la invoca, la find restituirà informazioni diverse (tramite un apposito vettore gestito diversamente nei vari casi). Queste informazioni verranno poi passate, tramite la classe controller LinQedInClient ad una view dedicata (ShowProfileFind) che le stamperà a video.

2.1.2 Gestione database

La classe DB che ha il compito di immagazzinare le informazioni dei vari utenti, compresa la rete, è strutturata nel seguente modo. Nella parte privata è presente un contenitore (vector) in cui vengono salvati dei puntatori ad Utente. E'stato scelto un vettore perchè, nel corso dell'utilizzo del programma, sarà necessario accedere a elementi casuali del contenitore, dovendo accedere in molti metodi ai dati dei vari utenti, e questa operazione nei vettore è decisamente più veloce. Il vettore tuttavia non è migliore della lista per quanto riguarda l'eliminazione di un oggetto, poichè deve ricompattarsi ogni volta che viene eliminato un oggetto. Nelle liste non è così, visto che semplicemente sposta il puntatore del nodo precedente a quello successivo del nodo eliminato. Tuttavia ragionando sui vari casi d'uso, le operazioni di accesso casuale saranno *notevolmente* maggiori di quelle di eliminazione di un oggetto, quindi per questo vector è migliore.

2.1.3 Gestione Rete utenti

La rete dei contatti invece è salvata in una classe a parte, Rete, che è formata da un array di puntatori ad Utenti. E' stato scelto un vector per lo stesso motivo del DB spiegato sopra, anche se, secondo me, una lista sarebbe andata bene ugualmente per quanto riguarda l'efficiente, visto che il numero di operazioni di ricerca casuale nel contenitore e di eliminazione di un utente dalla propria rete saranno in media le stesse.

2.2 Gestione del salvataggio dati in XML

Per poter rendere solide le informazioni contenute nell'applicazione, queste vengono salvate in un file XML.
La scelta di XML è stata fatta principalmente per il fatto che questo linguaggio è fortemente supportato da Qt, infatti nelle librerie sono presenti dei parser per la lettura/scrittura piuttosto semplici da utilizzare. Per la scrittura è stato utilizzato lo XMLStreamWriter. Dopo aver associato il file (esistente o meno) allo StreamWriter, viene fatto scorrere l'intero database per salvare, ordinatamente utente per utente, i vari campi dati. Per il salvataggio del Tipo Utente è stato effettuato un dynamic-cast e salvato un numero (1,2,3) a seconda del risultato ottenuto.
Per quanto riguarda la lettura invece, si è scelto di utilizzare il QDomDocument, che dopo varie prove, è risultato essere di più semplice gestione rispetto all'XMLStreamWriter. Il QDomDocument crea una lista, dove ogni nodo corrisponde ad un utente con i suoi campi dati. Una volta finito di caricare tutti i tag di un nodo, ossia tutti i campi dati di un utente, viene eseguito un check sul tipo utente e a seconda del risultato viene creato e inserito nel DB un tipo utente diverso. Così per tutti gli utenti.
Il caricamento della rete, invece, viene effettuato al momento del login dell'utente, unitamente al caricamente delle reti per tutti gli utenti (per poter mantenere corretto il file XML al momento del salvataggio). Questa dilazione nel caricamento della rete avviene per 2 principali ragioni: Dato che la rete contiene dei puntatori ad Utente, al momento del caricamento di un utente non è detto (anzi è fortemente probabile) che non tutti i suoi amici siano già stati aggiunti, rendendo impossibile l'operazione. Il secondo motivo è conseguenza della possibilità di un utente di potersi eliminare. Per questo, al momento del caricamento della rete, viene cercato l'utente nel DB, se questo risulta ancora presente viene caricato, altrimenti si passa al successivo. Se non si procedesse in questa maniera, ci sarebbe il rischio di avere dei dangling pointer ad utenti che si sono eliminati.

Capitolo 3

Divisione grafica/logica

In questo progetto si è cercato di seguire almeno in parte, come consigliato, il pattern MVC. Questo pattern consiste nel suddividere la parte logica (model), la parte grafica (View) e la parte che le collega (Controller). Nello specifico, nella gestione dell'amministratore ad esempio, è presente la classe AdminMainWindow che corrisponde alla view, la quale effettua delle richieste al controller a seconda delle esigente dell'utilizzatore. Il controller è identificato dalla classe LinQedInAdmin che ha il compito di interrogare la parte logica per ottenere le informazioni (oltre a fornire le risposte alla view). La parte logica è identificata principalmente dalla classe DB che contiene tutte le informazioni e i metodi per gestirle, modificale e restituirle con determinati vincoli richiesti dal controller. Per la parte utente la strutta è pressoché la stessa, in cui la View è identificata dalla classe UserMainWindow, il controller da LinQedInClient e la parte logica da Utente e sempre DB. Questa è la struttura alla base del progetto. Inoltre sono presenti ulteriori classi (riguardanti la parte logica e la grafica) che sono servite per organizzare al meglio le funzionalità, come per esempio la finestra di registrazione (RegistrationWindow), la finestra di modifica dati personali (ModifyProfile) e la finestra di visualizzazione profilo (ShowProfile). All'avvio dell'applicazione ci si trova nella MainWindow che ha lo scopo di prendere la decisione di chi utilizza il programma di entrare come utente o come amministratore, aprendo le relative view.
Per lo sviluppo dell'interfaccia grafica si è sfruttato il tool messo a disposizione dal FrameWork QtCreator, ossia QtDesigner. Con questo è stata sviluppata SOLAMENTE la parte grafica, mentre tutte le funzionalità (ad esempio il comportamento da tenere in seguito al click di un determinato pulsante) è stato fatto tutto a mano. Si è scelta questa via per concentrarsi maggiormente sulla parte logica e perchè, una volta compilato il codice generato da QtDesigner, si è notato che l'output generato automaticamente non era poi così degenere e che non erano presenti particolari problemi tali da dover rendere necessario l'intervento a mano.

Per quanto riguarda la finestra di registrazione nuovo utente (registratiowindow) sono state utilizzate delle espressioni regolari per controllare l'input inserito. Più precisamente per quanto riguarda il nome e cognome si è limitato l'inserimento alle sole lettere maiuscole (A-Z) e minuscole (a-z). Per il numero di telefono, invece, si è limitato l'input a solo caratteri numerici (0-9) con una lunghezza di massimo 10 cifre, considerando che vengano inseriti solamente numeri di telefono fissi/mobile di utenti italiani, e quindi senza prefisso internazionale (l'eventuale modifica per rendere disponibile questa flessione sui numeri di telefono è praticamente immediata).

Capitolo 4

Considerazioni Finali

In questo progetto si è cercato di utilizzare la maggior parte dei concetti introdotti dal corso di Programmazione ad Oggetti. Una parte importante, non utilizzata nel progetto, è la cosiddetta derivazione virtuale e gerarchia a diamante. Si sarebbe potuto, per rendere più estensibile il codice, far ereditare virtualmente i vari utenti dalla classe base Utente, in modo tale da permettere un giorno l'introduzione di una classe derivata che derivi da due classi tra quelle concrete. Allo stesso tempo, però, si è pensato che non avrebbe senso un utente derivato da due utenti diversi, poiché le funzionalità di base sono uguali per tutti. Per questo si è scelto di ereditare normalmente dalla classe base Utente.