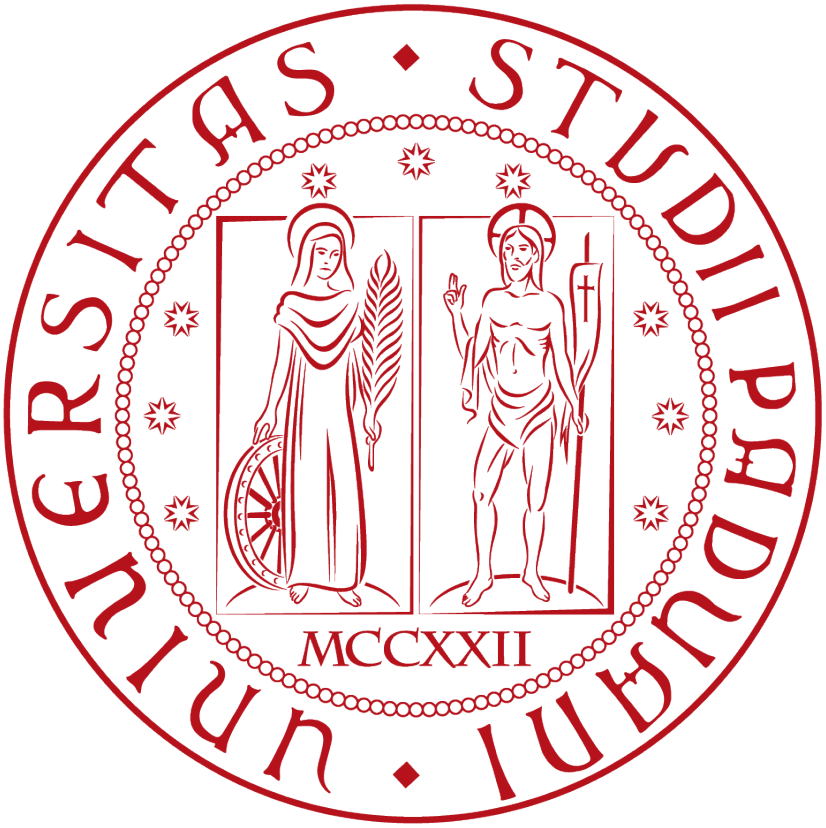
Davide Trevisan **Matricola: 1070686**

LinQedIn



Progetto del corso di programmazione ad oggetti

A.A. 2014/2015

Sistema operativo di sviluppo

il sistema operativo di sviluppo è stato Ubuntu 14.04 "Trusty Tahr" , nella sua versione a 64 bit (il codice è stato anche testato su Windows 8.1 a 32 bit, con la sola aggiunta della stringa "INCLUDEPATH += .\" al file .pro del progetto).

Ambiente di sviluppo:

il software è stato sviluppato mediante il famoso IDE Qt Creator 3.2.1 (opensource) basato su Qt 5.3.2 (GCC 4.6.1, 64 bit).

Compilatore

il compilatore utilizzato è gcc nella versione 4.8.2 (Ubuntu 4.8.2-19ubuntu1).

Scopo e descrizione del progetto

Lo scopo del progetto è quello di sviluppare in C++, con l'aiuto di Qt, un piccolo programma per l' amministrazione ed utilizzo tramite interfaccia utente grafica di un database di contatti professionali, in questo caso salvati in un file XML, ispirato al celebre LinkedIn.

Prendendo spunto da LinkedIn, gli utenti possono essere di tre distinte tipologie, basic, business ed executive, dotate di crescenti e più esaustive capacità di ricerca man mano che la tipologia è di livello più elevato.

Nello specifico il programma è costituito da una schermata di login unificata per amministratore e utente "normale" ; la schermata dell'amministratore permette di svolgere le funzioni di inserire un utente nel database, rimuovere un contatto dal database, mostrare l'XML del database, cercare un utente specifico, cambiare la tipologia di iscrizione di un utente e cambiare la propria password ;

Un utente invece ha disponibili le funzioni di inserimento e rimozione di un contatto nella propria rete, per mostrare il profilo o modificarlo, per vedere l'intero database o un contatto specifico o per compiere una ricerca all'interno di esso (ricerca che fornisce più informazioni al "crescere" della tipologia di iscrizione).

È incluso tra i file necessari al funzionamento del programma anche uno di tipo .qrc e l'icona corrispondente, al fine di poter definire un'icona all'applicazione quando è in esecuzione.

Classi della parte logica e loro descrizione

il programma è composto dalle seguenti classi:

- **Utente**, che è la classe base (non astratta perché utilizzata per allocare nella ram i contatti del database), che contiene nel suo campo privato l'informazione relativa alla tipologia dell'utente, e che implementa i tre metodi virtuali di ricerca nella loro versione

base;

da questa classe derivano:

- **Utente Basic,** che è la tipologia di utente di base e non fa l'overriding del metodo "Ricercaincampo" , in quanto non viene utilizzato;
- **Utente Business,** che è la tipologia di utente intermedia e contiene l'overriding di tutti i metodi di ricerca
- **Utente executive,** che è la tipologia di utente superiore e che contiene anch'essa l'overriding di tutti i metodi di ricerca;

notare come le classi derivate siano dotate di un opportuno costruttore da Utente* in maniera da poter collegare le info e la rete degli oggetti del database, creando così condivisione di memoria e side effect voluti, ad esempio nella parte di modifica del profilo o di inserimento e rimozione di contatti nella rete;

la classe utente quindi fa uso nei suoi campi anche delle classi:

- **Username,** semplice classe contenitore della QString username;
- **Info**, che contiene le varie informazioni dell'utente e i metodi per accedervi e modificarle;
- **Rete**, che è un semplice vector di Qstring, dotato anch'esso di metodi appositi al fine di accedervi e modificarlo e poterne verificare il contenuto;

Sono state implementate inoltre le classi:

- **DB**, che è il database del programma, implementato tramite una map<QString(lo username),Utente*>, dotata degli opportuni metodi e del distruttore ridefinito per distruggere anche i puntatori interni alla classe utente; è dotata anche di opportune funzioni per salvare e caricare il database su e da file XML. Da notare come qui vengano allocati elementi della classe base, in quanto la distinzione tra l'una e l'altra tipologia di utente qui non è necessaria, poiché l'informazione sulla tipologia è già contenuta nell'apposito campo della classe Utente.
- **LinQuedInClient**, che rappresenta il client per un utente di LinQuedIn e dotato di un opportuno costruttore e distruttore;
- **LinQuedInAdmin**, che rappresenta l'account di amministratore di LinQuedIn; la classe è amica di Utente e DB, in quanto nel primo caso non volevo dotare Utente di un metodo set_tipologia (motivo per cui Utente è anche amica di loadxml), nel secondo perché trovo necessario per l'amministratore il completo accesso al DB.

I metodi di input-output *loadxml* e *savexml*, che si avvalgono delle librerie Qt per la gestione di

file in XML, che sono in grado anche di sopperire alla mancanza del file XML del database creandone uno vuoto in caso non sia già presente, oltre ad *adminlogged* e a *save_admin_password* , che invece si avvalgono delle libreria generiche di Qt per la scrittura e lettura della password da un file .txt, sono tutti contenuti nella sorgente "io.cpp" e quanto più possibili indipendenti dalle classi, che hanno richiamato semplicemente il loro prototipo quando necessitano di una di queste funzioni.

Classi della parte grafica e loro descrizione

L'interfaccia è stata realizzata interamente *senza* utilizzare il designer integrato nell'IDE.

L'interfaccia è composta dalle seguenti classi:

- vi è la classe **LoginDialog** che è derivata da QDialog e funge da punto di snodo tra la schermata per l'utente "normale" e l'amministratore, in quanto permette di accedere a entrambi; inoltre segnala se il login è fallito oppure se la password dell'amministratore è sbagliata ed è in grado in caso di assenza del file "password.txt" , presente nella medesima directory di esecuzione del programma, che contiene la password dell'amministratore, di creare il file e di dare alla password il valore di default "Admin" ;

vi sono poi le finestre dedicate ad utenti e amministratore:

- vi è la classe **MyWidget** derivata da QWidget che è la finestra principale dell'utente; essa svolge alcune funzionalità all'interno della finestra generata e le funzioni scrivono le loro risposte nei campi QtextEdit della stessa, segnalando anche le operazioni non consentite, come ad esempio la ricerca di un utente inesistente o l'aggiunta o rimozione di un utente che non esiste o non presente nella lista amici; per alcune altre funzionalità sono state aggiunte, mediante puntatori smart all'interno della classe, delle classi che permettono la realizzazione di finestre aggiuntive:
  - la classe **Profilo**, derivata anch'essa da QWidget, che permette la modifica dei campi del profilo; autonomamente riempie i campi vuoti con la stringa "non specificato"
  - solo per gli utenti non basic o derivati da esso, la classe **Ricerca** che permette la ricerca tra i vari campi della classe Info degli elementi del database; non è consentita la ricerca vuota e quella di " " e ciò viene opportunamente segnalato tramite un messaggio;
- vi è poi la la classe **AdminGUI**, derivata da QWidget, che è quasi speculare nella sua struttura alla classe MyWidget, che rappresenta la finestra principale dell'interfaccia grafica dell'amministratore; anche in AdminGUI sono presenti avvertimenti a schermo e

sono state implementate con il medesimo metodo di MyWidget altre finestre:

- ✔ la classe **NewUtente**, derivata da QWidget, che permette l'inserimento nel database di un nuovo utente (almeno lo username è richiesto per procedere all'inserimento, se non c'è viene segnalato tramite un opportuno messaggio); segnala se l'utente che si tenta di inserire esiste già e si occupa autonomamente di inserire al posto dei campi vuoti la stringa "non specificato"; provvede autonomamente a "ripulire" i campi una volta inserito con successo un utente nel database;
- ✔ la classe **ChangeSubType**, derivata da QWidget, che permette di cambiare la tipologia di iscrizione di un utente nel database; è segnalato il caso in cui si tenti di settare come tipologia quella che già l'utente aveva in origine e il caso in cui l'utente a cui si cerca di cambiare la tipologia di iscrizione non sia presente nel database.
- ✔ la classe **ChangePassword**, derivata da QDialog, che permette di cambiare la password dell'amministratore, che è salvata nel file "password.txt"; è segnalato il tentativo di settare una password vuota.

NB: tutte le classi, a parte Ricerca, sono in grado di segnalare tramite un QMessage il completamento dell'operazione effettuata; inoltre tutte, a parte NewUtente mantengono il memoria il testo presente al momento della chiusura, al fine di poter ritornare allo stato in cui si era nel caso di chiusure accidentali; la chiusura di AdminGUI o MyWidget inoltre comporta la chiusura dell'applicazione.

Scelte progettuali di maggior rilevanza

➔ ***Gestione degli errori***

Gli errori sono stati gestiti esclusivamente mediante controlli preventivi per evitare il sollevamento di eccezioni, e nella maggior parte dei casi viene comunicato a schermo da un QtextEditor. Ho preferito evitare quanto più possibile le QMessageBox per gli avvisi in quanto a mio parere invasive. Sono comunque utilizzate alla chiusura di alcune finestre per evidenziare il completamento di una operazione e renderlo trasparente all'utente.

➔ ***Overriding***

Per le classi derivate dall'utente base è stato fatto un opportuno overriding delle funzioni di ricerca in modo che fornissero informazioni crescenti al crescere della tipologia: ad esempio usersearch nel caso di utente base fornisce solo il nome e il cognome, in un utente business fornisce tutte le info e in un utente executive fornisce anche lo username e la rete di contatti.

➔ ***Identificazione di tipi a run-time***

L' identificazione a run-time dei tipi è stata effettuata esclusivamente con l' utilizzo del

costrutto dynamic_cast<T1>(T2), nello specifico per la creazione del pulsante che permette la ricerca per campi, riservata agli utenti che non sono basic o derivati da esso.

- ➔ ***Gestione della memoria***

È stata implementata una politica simile al "write-back" per quanto riguarda le modifiche al database, e quindi ogni modifica del database comporta l'effettiva scrittura nel file XML. L'applicazione fa un uso minimo della memoria, in quanto ho cercato il più possibile di ricorrere all'utilizzo di puntatori e riferimenti. Inoltre le varie finestre da cui è composto il programma sono implementate nella classe "padre" mediante l'utilizzo di puntatori alle stesse, allocati nello heap una sola volta e solo se c'è dall'utilizzatore la richiesta effettiva della specifica finestra. Come puntatori per le finestre della parte grafica si è fatto largo uso dei puntatori smart offerti dalle librerie Qt, nello specifico di QsharedPointer, che garantisce l'eliminazione dell'oggetto nel caso non vi siano più puntatori che si riferiscono ad esso. Per il metodo di rimozione dal database e per il distruttore dello stesso è stato necessario definire anche la distruzione dei sotto oggetti, così come per le classi LinQuedInClient e LinQuedInAdmin.
Per la parte grafica non è stato necessario ridefinire il distruttore standard, in quanto tutti i widget che vengono inseriti in una finestra sono figli della stessa, e quindi, come da comportamento del distruttore standard di Qt, non possono sopravvivere alla distruzione del padre e viene quindi invocata la delete anche sui figli quando viene invocata sul padre.

- ➔ ***Divisione tra parte logica e grafica ed estensibilità***

il programma mantiene una buona divisione tra parte logica e grafica, ed è utilizzabile tramite piccole modifiche anche solamente da riga di comando. È aperto anche all'implementazione di nuove tipologie di utenti con un minimo cambiamento nel codice.
Il codice della GUI è poi largamente riutilizzabile in quanto si è cercato di ridurre al minimo le dipendenze tra una classe e l'altra.