

Relazione Progetto Programmazione ad Oggetti

LinQedIn

Giacomo Beltrame 1006153

15 aprile 2015

Credenziali d'accesso

Admin
keepahmu

Client
Basic: ilariab
Business: enzom
Executive: marior

Indirizzo e-mail referente

giacomo.beltrame@studenti.unipd.it



Indice

1	Analisi dei requisiti	3
1.1	Requisiti dell'interfaccia grafica	3
1.2	Requisiti funzionali	3
1.2.1	Admin	3
1.2.2	Client	3
1.3	Requisiti prestazionali	4
1.3.1	Tempo	4
1.3.2	Memoria	4
1.4	Vincoli di progettazione	4
1.5	Ambiente di sviluppo	4
2	Model	5
2.1	Database	5
2.2	Gerarchia	6
2.3	Rete/Collegamenti	7
3	Controller	8
3.1	Gerarchia	8
3.1.1	Admin	8
3.1.2	Client	8
4	View	9
4.1	Lato Client	9
4.1.1	ClientWindow	9
4.1.2	La gerarchia a livello grafico	12
4.2	Lato Admin	13
4.2.1	AdminWindow	13
4.2.2	La ricerca	13
5	Stati particolari e gestione delle eccezioni	14
6	Osservazioni	14
6.1	Ambiente di sviluppo	14
6.2	Compilazione dei sorgenti	14

Sommario

Lo scopo del progetto è lo sviluppo in C++/Qt di un sistema minimale per l'amministrazione ed utilizzo tramite interfaccia utente grafica di un database di contatti professionali ispirato a LinkedIn, il principale servizio web di rete sociale per contatti professionali, gratuito ma con servizi opzionali a pagamento.

1 Analisi dei requisiti

1.1 Requisiti dell'interfaccia grafica

L'interfaccia grafica (GUI) permette l'accesso in due diverse modalità: Amministratore (Admin) e Utente base (Client). Le due diverse modalità permettono di compiere le principali operazioni che saranno esplicitate nei paragrafi seguenti.

1.2 Requisiti funzionali

L'utilizzazione del software avviene nelle due modalità *Admin* e *Client*; entrambe permettono il salvataggio e la lettura su file del database degli utenti LinQedIn.

1.2.1 Admin

Questa modalità permette le seguenti funzionalità:

- inserimento di un nuovo utente nel database LinQedIn
- rimozione di un utente dal database LinQedIn
- cambio di tipologia di account (Basic, Business, Executive) per un utente LinkedIn
- ricerca nel database LinQedIn di un utente

1.2.2 Client

Questa modalità permette le seguenti funzionalità:

- aggiornamento del proprio profilo
- inserimento di un contatto nella propria rete
- rimozione di un contatto dalla propria rete
- ricerca nel database LinQedIn di un utente, diversa per ogni tipologia di account (specifiche a discrezione del progettista)

1.3 Requisiti prestazionali

1.3.1 Tempo

Non è stato tenuto conto di vincoli prestazionali riguardo al tempo di esecuzione date le piccole dimensioni del database.

In caso contrario il tipo di contenitore per il database influisce sul tempo di caricamento ed eliminazione e aggiunta di un nodo nel database, quindi la scelta del contenitore è da rivedere tenendo conto della scalabilità del progetto.

1.3.2 Memoria

Opportuna attenzione è stata data al requisito fondamentale della gestione della memoria allocata e quella da deallocare onde evitare la formazione di garbage (in C++ è assente un garbage collector a differenza di Java).

1.4 Vincoli di progettazione

Lo sviluppo di questo software avviene nell'ottica di soddisfare a pieno i seguenti vincoli di progettazione:

- **Correttezza:** il progetto deve compilare e funzionare correttamente, e raggiungere correttamente e pienamente gli scopi previsti.
- **Orientazione agli oggetti:** (A) progettazione ad oggetti, (B) modularità (in particolare, massima separazione tra il codice logico del progetto ed il codice della GUI del progetto), (C) estensibilità e (D) qualità del codice sviluppato.
- **Quantità e qualità:** quante e quali funzionalità il progetto rende disponibili, e la loro qualità.
- **GUI:** utilizzo corretto della libreria Qt, qualità ed usabilità della GUI.

1.5 Ambiente di sviluppo

Il software deve essere sviluppato in un ambiente Linux per la corretta compilazione ed esecuzione sulle macchine del laboratorio preposto per la correzione finale.

Il compilatore utilizzato è il GCC 4.6.1 e utilizza le librerie di Qt 5.3.2 (comando `qmake-qt532` nelle macchine del laboratorio).

2 Model

2.1 Database

Per ciò che riguarda il database, come contenitore della STL la scelta è ricaduta sul `map`, la cui implementazione è quella di un albero binario di ricerca;

inoltre, il `map` è un contenitore di tipo associativo, il cui template prevede un campo chiave e un campo info; questa peculiarità del `map` è stata sfruttata a pieno all'intero del progetto in quanto l'utilizzo di una chiave univoca (l'username degli utenti) permette la distinzione di ogni utente (non possono coesistere più utenti con lo stesso username); in questo modo la ricerca all'interno del database ha complessità $O(\log_2 n)$ dove n è il numero di elementi del contenitore.

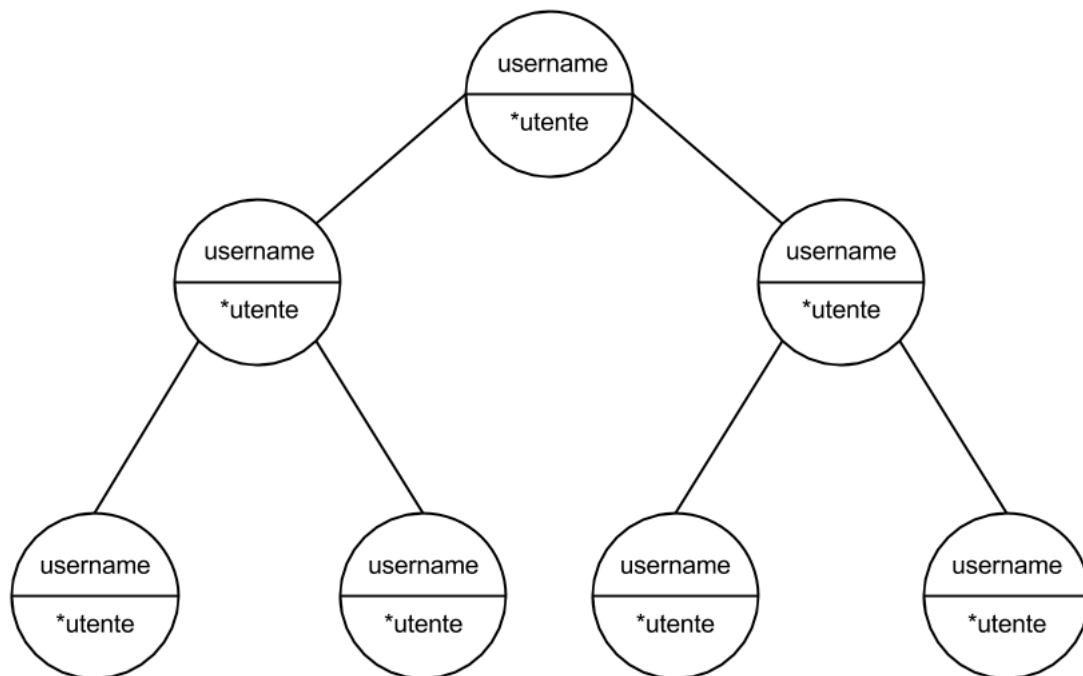


Figura 1: Implementazione del database

Come si può notare dalla *Figura 2* il campo info del `map` è un puntatore polimorfo alla classe base astratta *Utente* dato che gli oggetti di utenti istanziati possono avere il tipo dinamico di tre classi diverse (*UBasic*, *UBusiness*, *UExecutive*).

2.2 Gerarchia

La gerarchia di utenti prevede l'esistenza della classe base astratta *Utente*; la gerarchia è di tipo verticale: *Utente* -> *UBasic* -> *UBusiness* -> *UExecutive*.

La scelta di una gerarchia di tipo verticale è dettata dal fatto che ogni livello di utente ha un numero di privilegi, opzioni e funzionalità sempre maggiori; ad esempio ciò che caratterizza le varie classi della gerarchia è la diversa implementazione della funzione *find* che permette una ricerca con un numero sempre maggiore di filtri per un risultato sempre più accurato.

Una gerarchia così implementata permette, quindi, di richiamare sempre la funzione *find* della a partire da quella della classe *UBasic* che filtrerà gli utenti su tutto il database ritornando eventualmente il risultato alla *find* della classe successiva in gerarchia la quale effettuerà il filtraggio non più su tutto il database ma sul risultato restituito dalla *find* della sua superclasse (e così via fino ad arrivare alla *find* appartenente alla classe dell'oggetto chiamante).

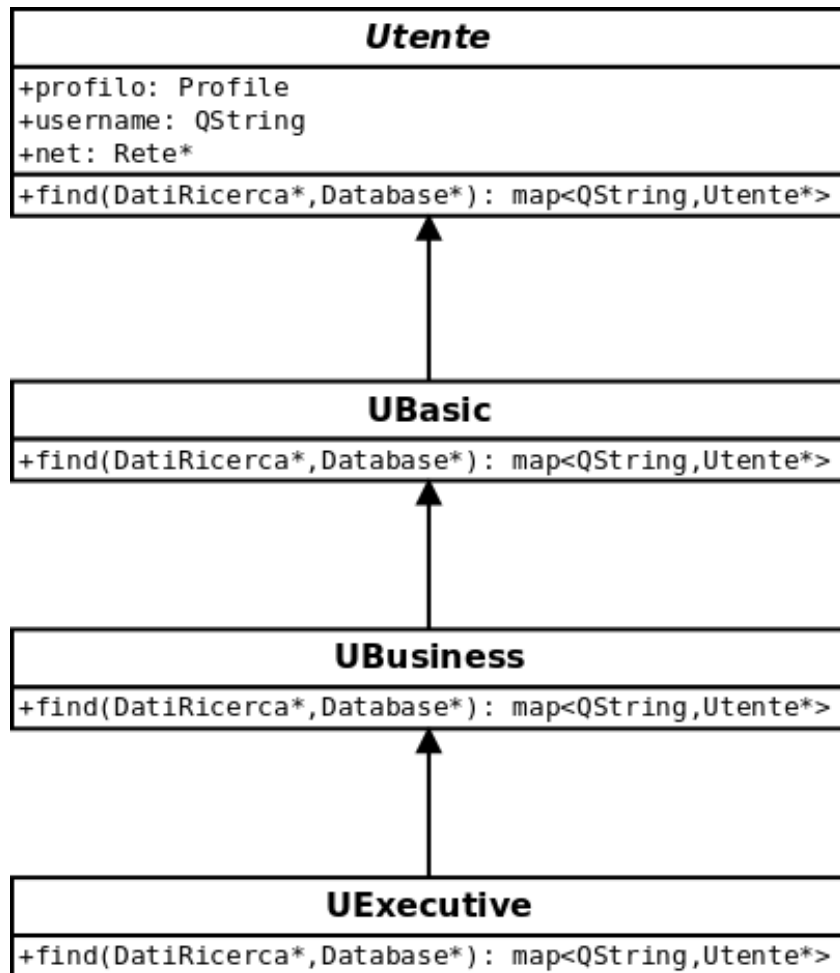


Figura 2: Gerarchia utenti

2.3 Rete/Collegamenti

Anche per l'implementazione della rete viene utilizzato il `map` per sfruttare l'associatività di questo contenitore.

Per quanto riguarda il collegamento tra gli utenti attraverso le reti, la scelta è stata quella di far in modo che, quando l'utente A decide di collegarsi all'utente B, quest'ultimo è automaticamente collegato all'utente A senza la segnalazione con notifiche o richiesta di conferma prima di applicare l'associazione tra i due utenti (lo stesso vale per la rimozione); non è escluso che in futuro questa peculiarità venga implementata.

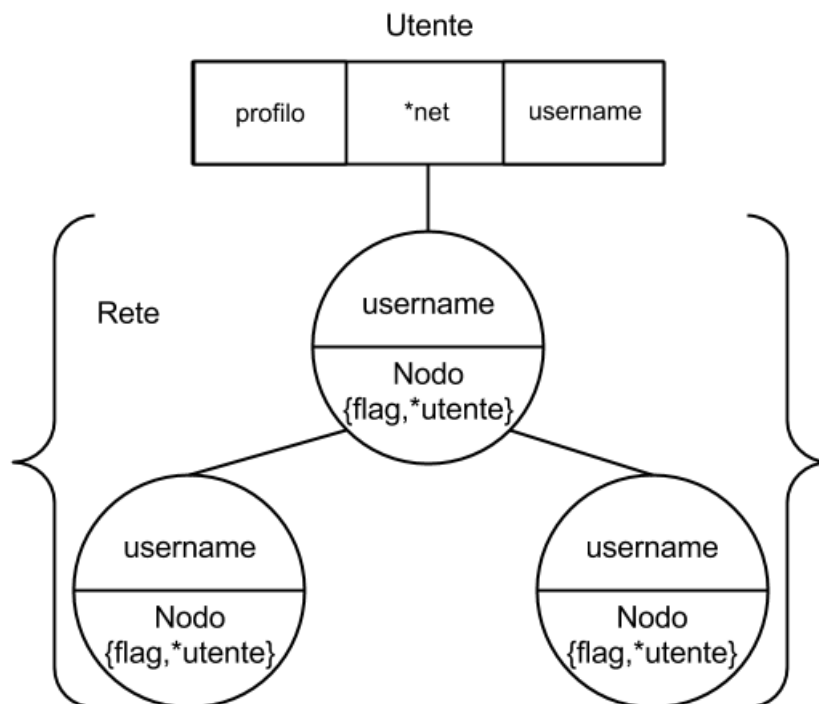


Figura 3: Implementazione di un utente e della sua rete

Come si può notare in *Figura 3*, il campo info della rete è un oggetto della classe *Nodo*, il quale contiene un campo dati *bool* *flag* (utilizzato in fase di *load* e *save* del database da/su file) ¹. La scelta di avere un puntatore diretto agli utenti del database piuttosto che semplicemente l'username serve ad evitare che ogni volta venga invocata la *find* per il recupero degli utenti della rete dell'utente in questione.

¹Dato che l'appartenenza di A nella rete di B implica l'appartenenza di B nella rete di A, per evitare la ridondanza su file dell'associazione [A-B,B-A], nella costruzione delle reti, un nodo viene settato con *flag true* mentre l'altro *false*; così facendo, nel salvataggio del database su file viene considerata solo l'associazione con *flag true* evitando di scrivere due volte la stessa coppia.

3 Controller

La parte Controller del MVC è costituita da classi che permettono la comunicazione tra Model e View in modo da mantenere separate la parte di comportamento da quella dell'aspetto del software; aderendo, infatti, al pattern Model/View/Controller avremo come risultato un codice facilmente manutenibile permettendo la modifica del codice con il minimo sforzo.

3.1 Gerarchia

Le classi del Controller sono organizzate in una gerarchia orizzontale; tale scelta è dettata dalla necessità di gestire le chiamate ai metodi del Model automaticamente a prescindere se esse vengano effettuate dall'admin o dal client.

Inoltre questa gerarchia permette di richiamare le stesse classi della View adattandone l'aspetto in base al tipo della classe del Controller utilizzata.

3.1.1 Admin

Costituita da un campo dati statico di tipo QString e da uno di tipo oggetto Database, la classe *Admin* è costituita da metodi propri del lato Amministratore.

Questa classe è sprovvista di un campo dati Utente in quanto l'amministratore è una entità astratta che può agire su tutti gli utenti e non è previsto che la password d'accesso possa essere modificata.

3.1.2 Client

La classe *Client* invece, oltre al campo dati di tipo oggetto Database, è provvista di un campo dati Utente, più precisamente un puntatore polimorfo alla classe base astratta Utente in quanto qui abbiamo il concetto di entità concreta di un utente che può modellare i propri dati e la propria rete (oltre ad effettuare ricerche).

Un campo dati risultati della libreria standart STL di tipo `map<QString,Utente*>` (stesso tipo del campo dati della classe *Database*), lo ritroviamo in entrambe le classi della gerarchia perchè utilizzato per raccogliere i risultati di eventuali ricerche effettuate.

4 View

La parte View del MVC ha alla sua base la creazione di un oggetto della classe *MainWindow* il quale si compone di una menubar ², un centralwidget ³ ed una statusbar ⁴.

In particolare, il centralwidget viene inizializzato con le classi *AdminWindow* o *ClientWindow* le quali ereditano entrambe la classe *QTabWidget* per permettere la suddivisione delle informazioni in sezioni distinte ed ordinate.

4.1 Lato Client

4.1.1 ClientWindow

La classe *ClientWindow* istanzia 4 classi proprie del lato client (*ViewDatiAnagrafici*, *ViewTitoliStudio*, *ViewImpieghi*, *ViewReti*) ed una in comune anche al lato admin (*ViewRicerca*).

La classe *ViewDatiAnagrafici* è collegata alla classe *DAnagrafici* del Model; una volta effettuata una modifica, con il bottone Applica si applicano le modifiche al database caricato in RAM; solo una volta premuto il tasto Salva nel menu File questi cambiamenti saranno permanenti anche su disco.

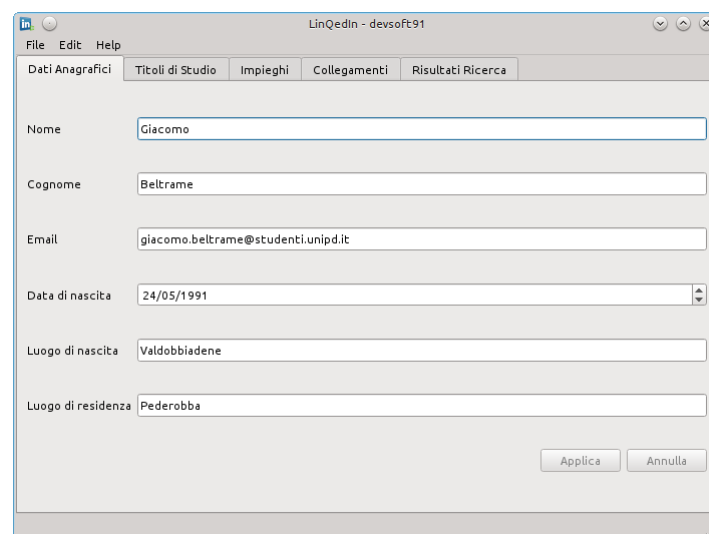
The image shows a screenshot of a Qt application window titled 'LinQedin - devsoft91'. The window has a menu bar with 'File', 'Edit', and 'Help'. Below the menu bar is a tab bar with five tabs: 'Dati Anagrafici', 'Titoli di Studio', 'Impieghi', 'Collegamenti', and 'Risultati Ricerca'. The 'Dati Anagrafici' tab is active. The form contains several input fields: 'Nome' with the value 'Giacomo', 'Cognome' with 'Beltrame', 'Email' with 'giacomo.beltrame@studenti.unipd.it', 'Data di nascita' with '24/05/1991' and a date picker icon, 'Luogo di nascita' with 'Valdobbiadene', and 'Luogo di residenza' with 'Pederobba'. At the bottom right of the form are two buttons: 'Applica' and 'Annulla'.

Figura 4: Classe *ViewDatiAnagrafici*

²Questa è la classica barra degli strumenti dove poter gestire la sessione, lanciare una ricerca o accedere alle informazioni del software

³Qui vengono mostrate le classi grafiche centrali per la visualizzazione e manipolazione delle informazioni nel database

⁴Si tratta della barra in fondo all'applicazione dove appaiono messaggi di successo o errore in seguito a determinate azioni da parte dell'utente

La classe `ViewTitoliStudio` è collegata alla classe `TitoliStudio` del Model; qui si possono visualizzare e modificare diploma ed eventuali lauree e di quest'ultime aggiungerne ed eliminarne con gli appositi tasti.

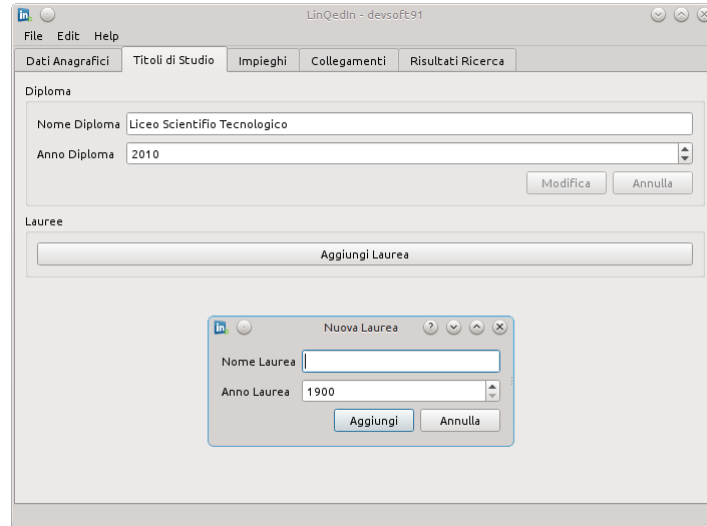


Figura 5: Classe `ViewTitoliStudio` e `WidgetNewLaurea`

La classe `ViewImpieghi` è collegata alla classe `Impieghi` del Model; qui si possono visualizzare, modificare, aggiungere ed eliminare gli impieghi dell'utente. In `WidgetNewImpiego` spuntando l'apposita checkbox si può posizionare l'impiego in testa alla lista che di default viene aggiunto in coda.

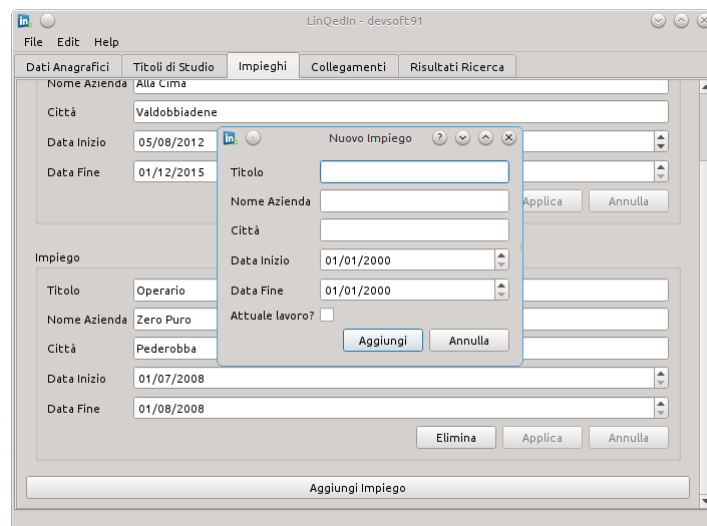


Figura 6: Classe `ViewImpieghi` e `WidgetNewImpiego`

La classe `ViewReti` è collegata alla classe `Rete` del Model; qui si può visualizzare la lista degli utenti che appartengono alla propria rete; inoltre si possono eliminare dalla rete e visualizzarne il profilo completo.

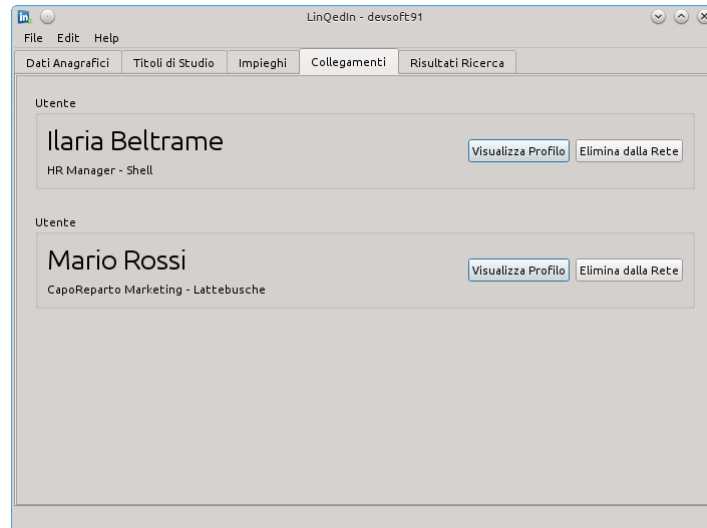


Figura 7: Classe ViewReti

La ricerca viene effettuata su tutto il database; si può però notare in *Figura 8* la differenza tra un utente che è già nella propria rete o meno in base alla presenza o meno del tasto per aggiungerlo ai propri collegamenti.

Inoltre la visualizzazione del profilo è più o meno dettagliata in base al tipo di account dell'utente che effettua la ricerca senza eccezioni per gli utenti che già sono nei collegamenti.

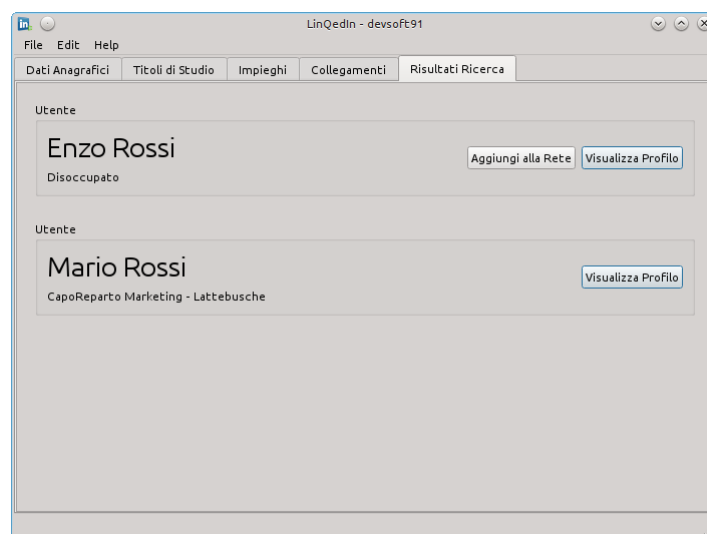


Figura 8: Classe ViewRicerca

4.1.2 La gerarchia a livello grafico

Ciò che cambia a livello grafico nella gerarchia di utenti sono:

- profondità nella visualizzazione delle informazioni del profilo di un utente
- presenza di un numero diverso di filtri per una ricerca più o meno dettagliata

Per quanto riguarda il primo punto, scendendo nella gerarchia, la visualizzazione delle informazioni del profilo sono sempre maggiori ⁵; in particolare l'utente di tipo Basic può visualizzare solo i dati anagrafici, quello di tipo Business anche i titoli di studio mentre quello di tipo Executive anche gli impieghi.

Per quanto riguarda, invece, i filtri di ricerca possiamo osservare le differenze nell'immagine che segue:

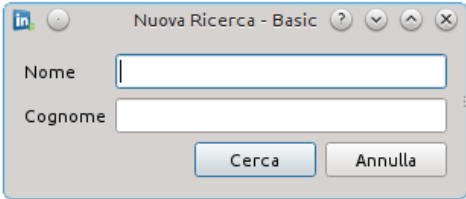
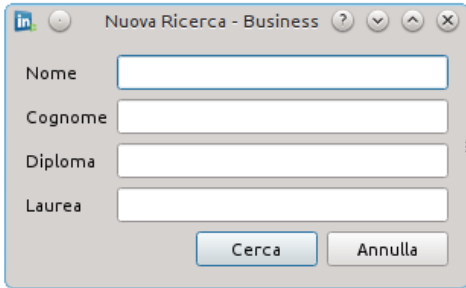
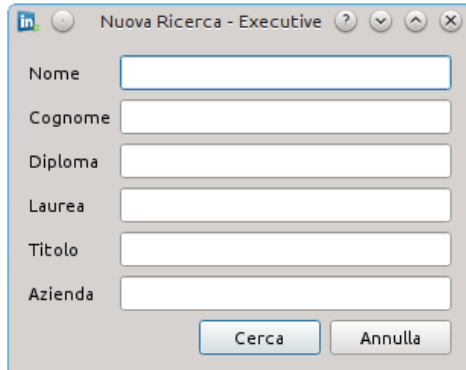
	find utente Basic
	find utente Business
	find utente Executive

Figura 9: WidgetNewRicerca della gerarchia di utenti a confronto

⁵I dettagli del profilo non cambiano a prescindere dal fatto che l'utente di cui si visualizza il profilo appartenga o meno alla rete dell'utente loggato nell'applicazione

4.2 Lato Admin

4.2.1 AdminWindow

Come detto poc'anzi la classe **ViewRicerca** è in comune sia al lato Admin che al lato Client (ovviamente con visualizzazioni diverse), quindi l'unica classe che caratterizza il lato Admin è **ViewGestioneUtenti**.

Questa classe permette di creare un nuovo utente, eliminarne uno già esistente oppure cambiare il tipo di account di un utente già presente nel database.

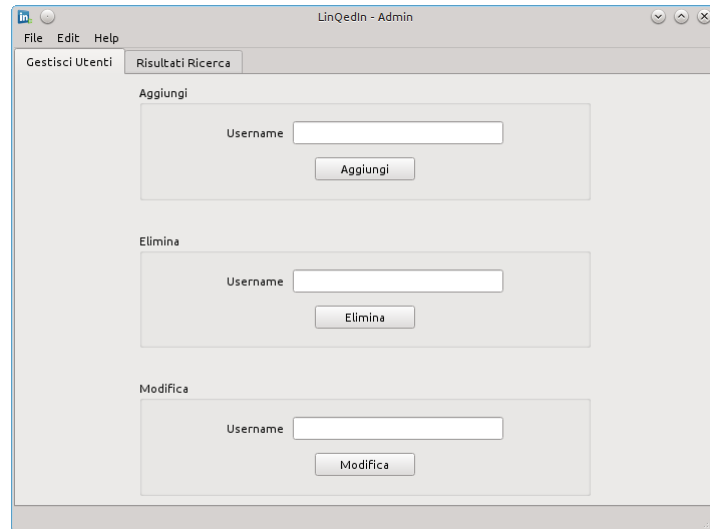


Figura 10: Classe ViewGestioneUtenti

4.2.2 La ricerca

Dal lato amministratore la ricerca può essere effettuata solamente tramite username oppure si può ricavare tutti gli utenti del database come si può osservare nell'immagine che segue:

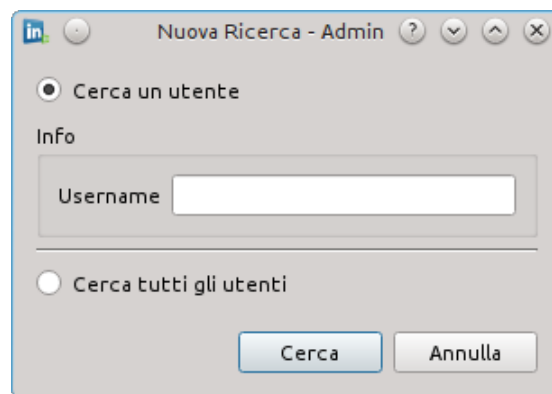


Figura 11: Classe WidgetNewRicerca

5 Stati particolari e gestione delle eccezioni

Particolare attenzione è stata posta riguardo allo stato in cui si trova l'applicazione in base al tipo di operazioni effettuate dall'utente.

Sono presenti due livelli: lo stato del database in RAM (temporaneo) e lo stato del database su disco (permanente anche dopo la chiusura dell'applicazione).

In questa applicazione è prevista una determinata sequenza di stati: prima di tutto l'utente modifica l'interfaccia grafica; una volta fatto ciò non è ancora stato applicato nessun cambiamento al database in RAM finchè non viene data la conferma di ciò con l'apposito tasto se previsto nella classe grafica dove si trova l'utente (infatti nel menu *File* il tasto per il salvataggio su disco è ancora disabilitato); una volta che viene data conferma per applicare i cambiamenti in RAM, il tasto per il salvataggio su disco viene abilitato dal sistema per abilitare il salvataggio dello stato corrente anche permanentemente su disco.

Nel caso in cui si effettui *logout* o l'uscita dall'applicazione senza aver apportato modifiche al database o dopo aver salvato quest'ultime su disco non viene segnalato nulla.

Nel caso contrario, se il sistema si accorge che lo stato del database in RAM è stato modificato ma non salvato su disco, l'applicazione chiede all'utente se è sicuro di uscire senza salvare, dandogli la possibilità di scegliere come comportarsi.

6 Osservazioni

6.1 Ambiente di sviluppo

Questo software è stato sviluppato nella versione KDE di Linux con ambiente grafico Plasma 4; le immagini stesse inserite in questo documento dell'applicazione in esecuzione sono state catturate in questo ambiente.

Detto ciò, la visualizzazione dello stesso software nelle macchine del laboratorio (versione Ubuntu di Linux con ambiente grafico Gnome) differisce leggermente dalla vera implementazione finale che si è voluto raggiungere.

Ciò, comunque, non influisce sulle funzionalità del software e sul suo comportamento che rimane integro a prescindere dalla distribuzione Linux utilizzata.

6.2 Compilazione dei sorgenti

Per la corretta compilazione dei sorgenti è stato incluso il file *resources.qrc* che permette la corretta localizzazione delle immagini all'interno della struttura del progetto.

Inoltre, viene fornito anche il file di progetto .pro al quale, oltre alla stringa QT += widgets, sono state aggiunte le seguenti righe di codice ⁶:

```
DEFINES += BDAY=\\\\" $$system(date +%d)\\\\"
DEFINES += BMONTH=\\\\" $$system(date +%b)\\\\"
DEFINES += BYEAR=\\\\" $$system(date +%Y)\\\\"
DEFINES += BHOUR=\\\\" $$system(date +%H)\\\\"
DEFINES += BMINUTE=\\\\" $$system(date +%M)\\\\"
DEFINES += BSECOND=\\\\" $$system(date +%S)\\\\"
```

Di conseguenza il file .pro verrà fornito assieme ai file sorgenti e sarà sufficiente lanciare in sequenza i comandi *qmake-qt532* e *make*.

Nel caso si volesse ignorare il file .pro fornito, aggiungere le righe qui sopra (oltre alla stringa QT += widgets) nel file .pro generato con il comando *qmake-qt532 -project* prima di eseguire *qmake-qt532* e *make*.

⁶Queste righe di codice permettono di visualizzare nel widget delle informazioni l'ultima data e ora di build del progetto