Andrea Mantovani 1070258

Sistema operativo di test: Ubuntu 14.04LTS 32-bit
Compilatore: GCC 4.6.1, 32 bit
Libreria Qt: Qt 5.3.2

Relazione progetto LinQdeIn

In questo documento, verranno esposti i principali componenti del progetto e le loro caratteristiche.
L'ordine con il quale si procede con l'esposizione segue la strutta del progetto come rappresentato nell'immagine sottostante, partendo dalla radice fino alle foglie andando da sinista a destra.
Il progetto aderisce al pattern di programmazione MVC(Model-View-Controller) per ogni modulo principale.

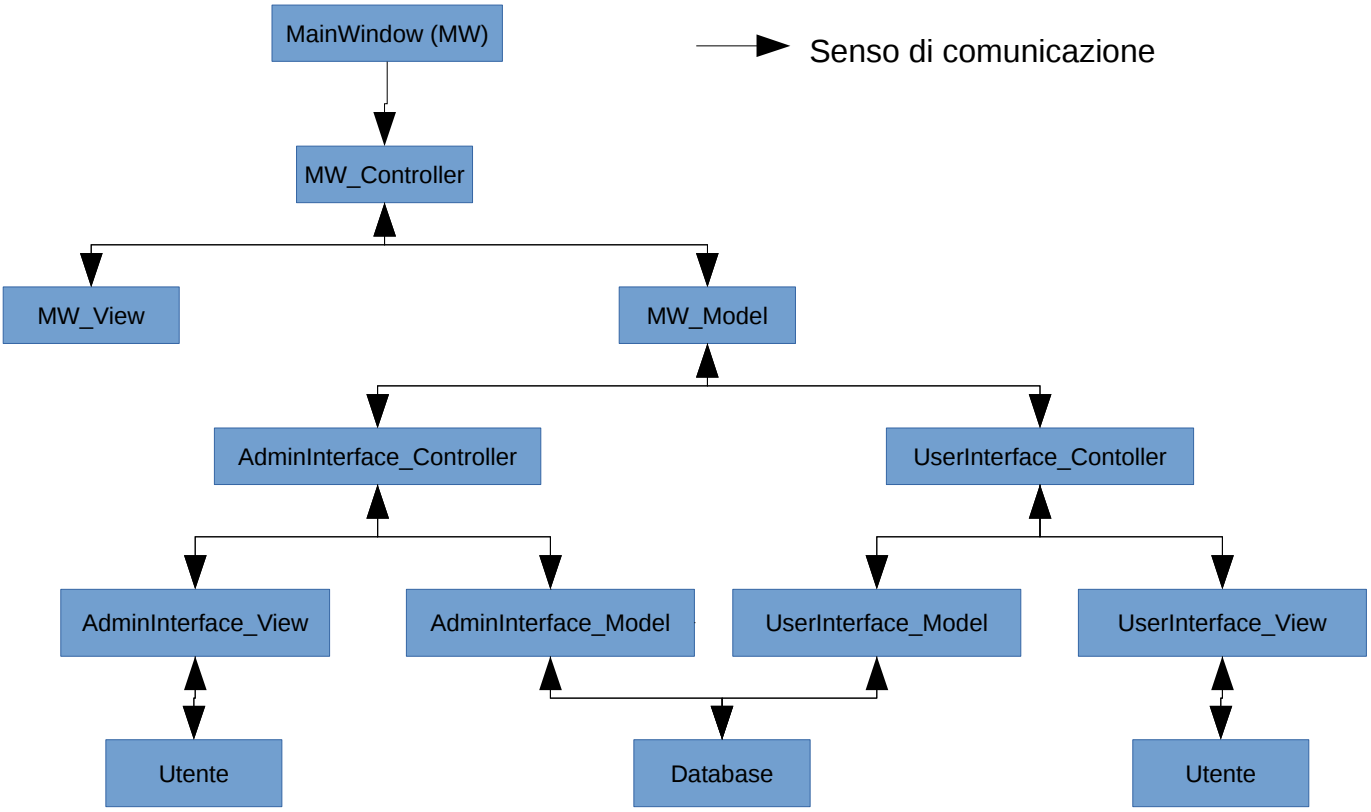


Figura 1

La maggior parte dei collegamenti dei nodi dell'albero ha due versi di percorrenza, ciò intende che è possibile, mediante l'uso di slot, signal e metodi appositi poter ricevere ed inviare informazioni.

MainWindow

Come suggerisce il nome è la finestra principale, dove viene visualizzata la GUI del programma.
La sua natura è di wrapper al puntatore del **MainWindow_Contoller**.

Il **MainWindow_Contoller** gestisce il cambio di interfacce tra lato amministratore/utente e gli eventi della finestra principale, come la visualizzazione e la chiusura. Ad ogni cambio di interfaccia si invoca una reset su quella da visualizzare, che riaggiorna la pagina corrente in modo da avere informazioni consistenti all'eventuali modifiche fatte sull'altro lato del programma.

La View, descritta nella **MainWindow_View,** e' un QWidget con all'interno una **QTabWidget,** scelta per la sua comodita di cambiare interfaccia senza ulteriori gestioni.

I puntatori ad ogni interfaccia sono contenuti nella **MainWindow_Model** e serve per avere accesso al loro lato controller.

AdminInterface

E' la parte lato amministratore, con le seguenti funzionalità:

- INSERIMENTO UTENTE
- RIMOZIONE UTENTE
- MODIFICA UTENTE (anche per tipologia)
- RICERCA (limitata per nome e cognome)
- CARICAMENTO DATABASE
- SALVATAGGIO DATABASE

La parte contoller fa da ponte tra la view e il model. Gli input dell'utente possono modificare la view o i dati relativi ad un utente. Questi ultimi vengono controllati, se hanno significato vengono passati al contoller che gli immette nel database, altimenti viene lanciato una SIGNAL con l'errore accurato.

La **MainWindow_View** è divisa in tre parti:

- la LabelTools con tutti i bottoni per avviare i servizi offerti dall'AdminInterface
- La TableUsers, dove sono elencati tutti gli utenti
- La utily, una QscrollArea che usando lo slot pubblico **setFrameUtiliy(QWidget *)** è possibile visualizzare qualunque interfaccia grafica derivata da QWidget. Le connessioni con il widget vengo risolte nell'AdminInterface_Controller prima di invocare lo slot e permette una totale liberta d'uso.

La view è pensata in modo che l'amministratore possa avere un'immediata visione della pagina dell'utente.
Il patter MVC si rivela comodo in questo caso, perchè è possibile riusare la view dell'utente semplicemente cambiando l'interpretazione dei segnali emessi.
L'esempio principale è la richiesta di modifica nell'interfaccia utente la quale carica il modulo di modifica. La differenza sta nel livello di accesso fornito. Se all'utente consente la modifica di tutte le sue aree informative, al lato amministratore in più vi è la scelta del tipo di account.

La parte model, ovvero **MainWindow_Model**, contiene un puntatore al Database ed uno all'utente selezionato, necessario per operare sul database.

UserInterface

Parte lato utente per la gestione del suo account. Qui è possibile modificare i propri dati, cercare, togliere e aggiungere collaboratori. Come mostrato in figura 1, anch'esso è diviso nella parte Model, View e Controller.

La **UserInterface_Controller** rappresenta la parte Controller, gestisce le richieste della View e le modifiche al database limitate all'utente registrato.
Il Model è contenuto nella **UserInterface_Model**, dove avviene l'aggancio al database, al puntatore dell'utente registrato e all'utente attuale.
La distinzione tra registrato e attuale, nasce quando l'utente registrato naviga nelle pagine di altri utenti e quindi è necessaria per poter decidere cosa visualizzare nella View. L'utente registrato può accedere a tutte le funzionalità della UserInterface indipendentemente dalla sua tipologia di account.
Al contrario, la view dell'utente attuale varia a seconda dei permessi concessi dalla tipologia dell'account.

L'ultima parte, la View, è gestita dalla **UserInterface_View** divisa in due momenti:

- Il loggin dell'utente

- La view dell'utente

La prima viene richiesto di inserire l'username creato in fase di inserimento nel database. Una volta essersi loggati con successo la View carica una QscrollArea dove viene inserita una **ViewBase**.
Quest'ultima è la base astratta della gerarchia di classi, mostrata in figura 2, che rappresentano le pagine dell'utente.

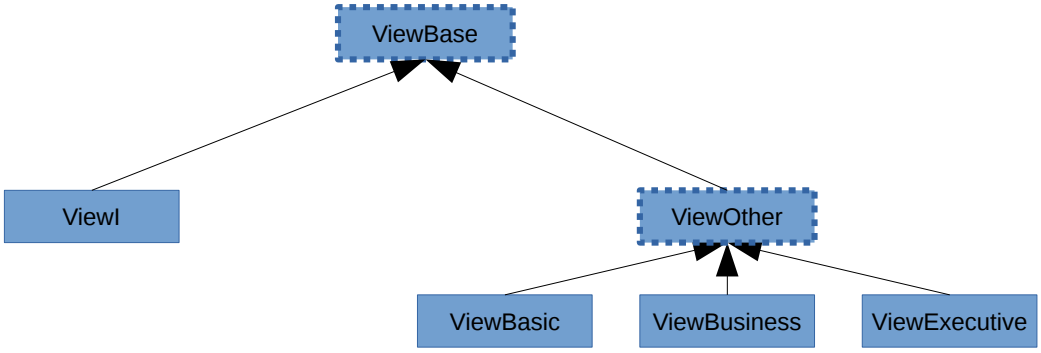
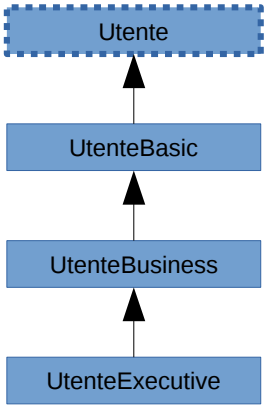


Figura 2

Ogni ViewBase richiede l'implementazione del metodo virtuale **loadMainPage(const smartptr_utente &)** dove viene creato il layout della pagina con dentro le informazioni dell'utente.
La classe **ViewOther** è specifica per trattare le richieste verso altri utenti, come quella di collaborazione, mentre la **ViewI** è per i segnali di gestione del proprio account.
L'estendibilità della classe ViewOther avviene orizzontalmente, lasciando indipendente ogni classe figla tra di loro.
Il maggior vantaggio è la completa liberta nel disegnare i layout dei diversi tipi di account, senza dover preoccuparsi di come saranno rappresentati gli altri. I segnali sono accorpati nella ViewBase, eccetto quelli per la richiesta di collaborazione che sono nella ViewOther, e ogni classe figlia decide quali usare.

Utente

La rappresentazione di un utente di LinQedIn è manipolata nell'oggetto **Utente**, classe base virtuale della gerarchia che definisci le varie tipologie di utente, come mostrato in figura 3. Lo scopo di usare questo tipo di gerchizzazione è di dare ad ogni classe derivata maggiori servizi, metodi più efficaci o eliminare alcune limitazioni, rispetto alla classe genitore.



Figuta 3

I metodi virtuali puri della classe Utente sono tre e servono per: definire il livello di accesso, leggere e scrivere i

proprio dati su un file xml.
Il livelli di accesso sono quattro, Master, Basic, Business e Executive, tutti definiti nell' **enum Type** dentro al namespace **LevelAccess**. La prima è per poter accedere al settagio della tipologia di account, risevata all'amministratore, le altre servono per unire le diverse view con i tipi di account messi a disposizione.

Essendo LinQedIn un programma orientato al social, ogni utente dispone di una lista di contatti rappresentata dalla classe **Rete** derivata pubblicamente da una QList di puntatori ad utente. Si è scelta una lista per la sua caratteristica di rimuovere un nodo qualsiasi in tempo costante, dato che è improbabile che si attui la rimozione sempre o in testa o in coda. C'è da tenere presente che la rimozione dovrebbe essere un evento raro ma considerando il carico globale questa scelta dovrebbe portare maggior efficienza al programma.

Entity e SmartClass

Prima di parlare del database bisogna definire quali entità esistono in esso. La classe **Entity** definisce che ogni entità del programma ha un username e un campo dati **Info**, dove si raccolgono le informazioni personali dell'utente.
Alla forma attuale del progetto le entità sono due: il **Frankenstein** e l'Utente.

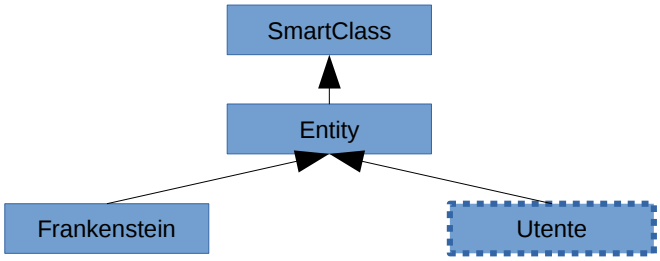


Figura 4

L'Utente è la forma visibile della Entity, un individuo completo che rimane memorizzato nel database e viene visualizzata dalle pagine utente.
Il Frankestein invece è nascosto e definisce un modello di Utente che può essere incompleto.
Il motivo per cui non deriva da Utente è per via della sua definizione di identità. Un Frankestein può essere uguale anche ad un'altro Frankestein, con alcune caratteristiche aggiuntive. Invece un Utente è uguale solo a se stesso.
Questo evento nasce da come è definito l'operator==(const Frankestein &), che restituisce true quando l'oggetto chiamante ha un insieme di informazioni che sono un sottoinsieme di quelle del parametro formale.
Inoltre è possibile vedere un Utente come un Frankenstein ma non viceversa, altrimenti sarebbe possibile inserirlo nel database.

La classe **SmartClass** offre il servizio di memoria condivisa, gestita con il reference counting e di smart pointer polimorfici adatti all'uso, chiamati **smartptr**. La sua costruzione è limitata alle sole classi derivate, data la natura puramente implementativa e la mancata necessita di un metodo virtuale puro.
Dato che Utente deriva implicitamente da SmartClass, anchesso usa la gestione controllata della memoria e questo aspetto è utile per mantenere sempre aggiornato lo stato dei contatti di un utente che vengono modificati o cancellati.

Database

L'ultimo argomento esposto riguarda la struttura e il funzionamento del database.
Inanzitutto, nel database vengono inseriti solo i puntatori **smartptr_utente**, derivati da smartptr per togliere continui down cast, ad un unico oggetto Utente allocato in memoria.
Il puntatore viene infine duplicato in tante copie quante sono le entry del database. Le entry sono una struttura

dati templetizzate, chiamata **HashListUtente<typename HashFunction>**, di contenitori associativi chiave valore, dove la chiave è un intero che viene determinata dalla HashFunction che come parametro richiede un puntatore ad Entity. Il gruppo di HashFunction create risiedono nel namespace **HashGroupUtente**.
A seconda di cosa la HashFunction esamina nell'Entity si vengo a creare tipi di entry diverse. Lo scopo di questa gestione è diminuire i tempi di ricerca che si avrebbero scandendo linearmente un database composto da un'unico vector o una list.
L'HashListUtente è derivata per implementazione da un Qvector< QList<smartptr_utente> >, l'intero restituito dall'HashFunction non è altro che la posizione nella cella dell'array. Nel caso ci siano più elementi che hanno la stessa chiave vengono aggiunti in testa alla lista, percorsa tutta durante la ricerca. Tuttavia la sua dimensione è ridotta perciò in termini di tempo c'è un guadagno.
La scelta di una lista è sempre dovuta all'eliminazione; l'aggiunta in testa ha un tempo equivalente all'inserimento in coda di un'array, perciò non è un elemento di scelta.
L'uso di un oggetto Entity, per determinare la chiave, è stato scelto per permettere l'uso del Frankenstein.
Il metodo che effetua la ricerca è **getUser(const Query &) const** e ha come contratto di restituire un QVector di smartptr_utete degli utenti che sono risultati adeguati. La classe **Query** è la base astratta di tutti i funtori che hanno il fine di definire quali sono le caratteristiche dell'utente da cercare.
Ogni classe derivata da Query deve implementare il metodo **void compose(Frankenstein &) const** la quale inserisce nel Frankestein le caratteristiche richieste. Con l'uso di questi oggetti si ottengono due vantaggi:

1. Ricerche più rapide, grazie alla compatibilità con la funzione hash;
2. Possibilità di definire ricerche personalizzate come sarebbe possibile solo mediante funtori;

Una volta composto il Frankenstein, sarà il database ha determinare a quale entry appartiente. Al momento ce ne sono di tre tipi: nome, cognome e username. Nel caso il Frankestein non appartenesse a nessuno di questi tre gruppi, si avvierebbe una ricerca lineare su una QList chiamata *general* per ottenere comunque un risultato. Ovviamente questa operazione è dispendiosa, ma basterebbe creare nuovi tipi di entry per risolvere il problema. Se al contrario appartenesse a più entry, si sceglierebbe la prima.

L'altro compito richiesto era la scrittura e la lettura su un file xml del database. Per attuare questo compito si è formata la gerachia di figura 5.

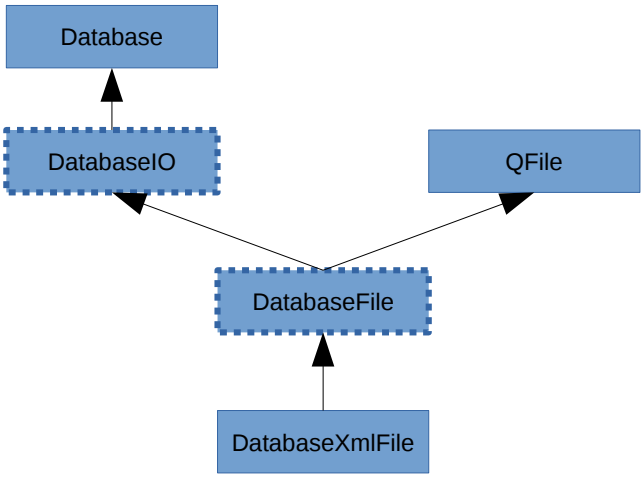


Figura 5

La classe **Database** provvedere a tutte le funzionalità previste da un DBMS. Oltre a questo, **DatabaseIO** ha il compito di gestire il salvataggio e il caricamento del DB su un device di I/O. Le classi derivate, si specializzano su quale mezzo si desidera immagazzinare le informazione raccolte e il formato voluto, attraverso i metodi **bool save()** e **bool load()**. Per la gestione degli errori, il metodo **error()** ha come contratto di restituire, in formato di QString, il messaggio dell'errore commesso. La stringa era l'unico plausibile oggetto che potessere essere usato per specificare l'errore, accurato in fase di salvataggio e caricamento, su un qualsiasi dispositivo.

DatabaseFile è la classe base da dove deriverrano tutti i database che necessiterrano di essere gestiti su un qualsiasi tipo di file memorizzabile in un archivio di massa.
La prima concretizzazione di DatabaseIO viene su **DatabaseXmlFile**, che gestisce il contenuto del database su un file xml usando le classi offerte da Qt, QStreamXmlReader per caricare e QStreamXmlWriter per salvare.