Relazione sul progetto di Programmazione ad oggetti

Giovanni Mazzocchin, mat. 1071619

January 29, 2015

Abstract

Il software è stato sviluppato in linea con il design pattern MV (ModelView),mantenendo quasi sempre la separazione tra il codice logico e il codice dell'interfaccia grafica. Particolare attenzione e' stata prestata alla ricerca incrementale, che e' stata resa particolarmente estensibile tramite funtori polimorfi.

1 Gerarchie di classi

- Gerarchia degli utenti (parte del Model)
  - Le tre tipologie di account LinQedIn (Basic,Business ed Executive) sono modellate tramite una gerarchia di quattro classi secondo il seguente schema:
    Utente (classe base polimorfa e astratta, dichiarata in `"headers/utente.h"`);
    UtenteBasic (classe concreta, eredita da Utente, dichiarata in `"headers/utente_basic.h"`);
    UtenteBusiness (classe concreta, eredita da UtenteBasic, dichiarata in `"headers/utente_business.h"`);
    UtenteExecutive (classe concreta, eredita da UtenteBusiness, dichiarata in `"headers/utente_executive.h"`).
- Gerarchia dei funtori (parte del Model)
  - La ricerca incrementale richiesta dalle specifiche è stata implementata con l'ausilio di una gerarchia di classi le cui istanze vengono utilizzate come funtori.
    Anche in questo caso la gerarchia è composta da quattro classi, in modo da generare una corrispondenza uno ad uno con le tipologie di account, secondo il seguente schema:
    OttieniInfos (classe base polimorfa e astratta, logicamente collegata alla classe 'Utente', dichiarata in `"headers/ottieni_infos.h"`);
    EsisteUtente (classe concreta, eredita da OttieniInfos, logicamente

collegata alla classe 'UtenteBasic', dichiarata in `"headers/esiste_utente.h"`);
ProfiloCompletoDi (classe concreta, eredita da EsisteUtente, logicamente collegata alla classe 'UtenteBusiness', dichiarata in `"headers/profilo_completo_di.h"`);
OttieniAncheReteDi (classe concreta, eredita da ProfiloCompletoDi, logicamente collegata alla classe 'UtenteExecutive', dichiarata in `"headers/ottieni_anche_rete_di")`.

- Gerarchia delle informazioni da visualizzare (parte della View)
  - Nell'implementazione della ricerca incrementale è stata cruciale la realizzazione di una gerarchia che modella le informazioni da visualizzare in modo appunto 'incrementale'.
    La gerarchia è composta da tre classi:
    InfoVisBasic (classe base polimorfa e concreta, dichiarata in `"headers/infovis_basic.h")`, permette di visualizzare l'informazione 'l'utente cercato esiste/non esiste';
    InfoVisBusiness (classe concreta, eredita da InfoVisBasic, dichiarata `in "headers/infovis_business.h")`, permette di visualizzare l'intero profilo (eccetto la rete di contatti) dell'utente cercato.
    InfoVisExecutive (classe concreta, eredita da InfoVisBusiness, dichiarata in `"headers/infovis_executive.h")`, permette di visualizzare l'intero profilo (compresa la rete di contatti) dell'utente cercato.

- Incapsulamento delle informazioni logiche (parte del Model)
  - Le classi **Utente**, **Profilo** (dichiarata in `"headers/profilo.h"`) e **Info** (dichiarata in `"headers/info.h"`), le quali si occupano di gestire le informazioni sugli utenti, sono collegate tra loro dalla relazione *has a*, infatti **Utente** funge da *wrapper* per **Profilo**, la quale a sua volta si comporta da wrapper per **Info**.

- Caricamento dei dati in RAM (parte del Model)
  - La classe **DB** (dichiarata nel file `"headers/db.h"`) si occupa del caricamento dei dati in memoria tramite il metodo *load()*. Gli utenti vengono caricati con il tipo adatto tramite il controllo sul tag `tipoacc` nel file XML.

2 Ricerca incrementale: descrizione dettagliata

2.1 Contratto da soddisfare

L'utente di tipologia *Basic* può ottenere l'informazione *'l'utente cercato esiste/non esiste'*;
l'utente di tipologia *Business* può visualizzare il profilo completo (eccetto la rete

di contatti) dell'utente cercato;
l'utente di tipologia *Executive* può visualizzare il profilo completo (compresa la rete di contatti) dell'utente cercato.

2.2 Funzionamento

La catena di invocazioni parte dalla classe **ClientWidget** (parte della View,dichiarata in `"headers/client_widget.h"`), nello slot *mostraProfilo()*: tale slot invoca il metodo **LinqEdInClient**::*ricercaNelDB()* (parte del Model). A questo punto entrano in gioco i funtori:
infatti **LinqEdInClient**::*ricercaNelDB()* crea un puntatore polimorfo con tipo statico **OttieniInfos*** (base astratta della gerarchia di funtori) e tipo dinamico determinato dalla chiamata al metodo **Utente**::*getFuntore()* (metodo virtuale di utente che ritorna un puntatore a funtore adatto alla tipologia di utente).Questo puntatore polimorfo viene così passato come parametro al metodo **DB**::*ricerca(OttieniInfos*...)* il quale finalmente utilizza il funtore. Sara' proprio l'overloading dell'operatore di chiamata a funzione (e conseguente overriding nelle sottoclassi, in quanto esso e' stato dichiarato `virtual` nella classe **OttieniInfos**) a causare la visualizzazione dei profili cercati tramite una chiamata ai costruttori di **InfoVisBasic**, **InfoVisBusiness** oppure di **InfoVisExecutive**.

3 Modalità di utilizzo del software

3.1 Modalità 'cliente'

Tramite questa modalità l'utente del software agisce appunto da *cliente LinQedIn*: se il login ha successo egli può visualizzare il proprio profilo, modificarne alcune parti e interrogare il database di *LinQedIn*.

3.2 Modalità 'amministratore'

L'utente in modalità *amministratore* può modificare il database *LinQedIn* aggiungendo nuovi utenti, rimuovendone, modificando la tipologia di account di un utente. Può ovviamente interrogare il database *LinQedIn*.

3.3 Modalità 'simulazione cliente@amministratore'

Questa modalità implementa una sorta di interazione *client-server* in quanto l'utente del software può agire sia da cliente sia da amministratore *LinQedIn*. Ovviamente tutte le modifiche apportate dal cliente verranno rese pubbliche all'amministratore, e viceversa.

4 Gestione dei dati persitenti

4.1 Struttura del database

I dati persistenti vengono salvati nel file `"data/db.xml"` (nella directory del progetto). Il file è in formato XML e viene letto e scritto da programma tramite le librerie Qt *QXmlStreamReader* e *QXmlStreamWriter*.

5 Interfaccia grafica

5.1 Finestra di *start*

La finestra di *start* si apre all'avvio del programma e dà all'utente la possibilità di scegliere la modalità di utilizzo del software.

5.2 Finestra di *login*

La finestra di login appare solamente se precedentemente sono state scelte le modalità *cliente* o *interazione cliente@amministratore*.

5.3 Interfaccia cliente

L'interfaccia cliente permette all'utente che ha effettuato il login di visualizzare il proprio profilo (tramite delle *QLineEdit*), di aggiungere contatti alla propria rete e di visualizzare i profili degli altri utenti *LinQedIn* (secondo i privilegi della propria tipologia di account). Essa permette inoltre all'utente di modificare alcune parti del proprio profilo(precisamente i campi 'nome', 'cognome', 'skills', 'studi' e 'lingue', tramite le stesse *QLineEdit*).

5.4 Interfaccia amministratore

Questa interfaccia permette all'amministratore *LinQedIn* di visualizzare i profili completi degli utenti cercati, di cambiare la tipologia di account di un utente e di aggiungere nuovi utenti al database.

6 Compilare ed eseguire *LinQedIn*

6.1

Spostarsi nella directory `sources_makefile` (subdirectory della cartella *LinQedInConsegna*) e lanciare da shell i comandi `make` e `make clean`. L'eseguibile generato avrà come nome `linqedin`. I login corretti sono i seguenti:
username::::::::**password**
`fbianchi`::::::::::`qwe2`
`mverdi`::::::::::::::`qwe3`
`aqwrt`::::::::::::::::`qwe4`

gmazzocc::::::::::qwe5.