Relazione Programmazione ad oggetti 2014-2015

Andrea Giacomo Baldan 579117

February 18, 2015

1. `Username:` Casey `Password:` rayback
2. `Username:` root `Password:` toor

INTRODUZIONE

Il progetto LinQedin tratta lo sviluppo di un applicazione desktop atta a gestire una rete di contatti professionali e interazioni lavorative tra utenti basata sul famoso *social network* Linkedin©, da cui eredita alcune funzionalità.

SPECIFICHE PROGETTUALI

Il programma è stato sviluppato in C++/Qt 5.3.2 con compilatore gcc ver. 4.8.2 e testato su ambiente Linux Ubuntu 12.10 e 14.04, inoltre compila ed esegue con successo in ambiente Windows, testato alla versione 7 e nei computer del laboratorio Paolotti, tuttavia essendo stato implementato utilizzando alcune funzionalità della libreria Qt rilasciate solo dalla versione 5 in poi, per la compilazione basta lanciare il comando *make*, nel caso fosse necessario ricreare il file .pro va utilizzato il comando qmake-qt532 - project, non e' stata utilizzata alcuna funzionalita di C++11. Per lo sviluppo del codice, sia logico che grafico, è stato utilizzato esclusivamente l'editor Sublime Text 3, non è stato utilizzato QtCreator ne QtDesigner.

ORGANIZZAZIONE DIRECTORY E INSTALLAZIONE

La radice contiene il file *linqedin.pro* generato con il comando qmake-qt532 -project e *relazione.pdf*. I sorgenti e gli headers della parte logica si trovano all'interno del percorso /logic, mentre /gui contiene tutti i sorgenti Qt adibiti all'interfaccia grafica. Alcune direttive grafiche sono state inserite in un file *style.qss* nella cartella /style, la directory /img contiene invece tutte le immagini utilizzate nell'interfaccia grafica. Nel caso si renda necessario rigenerare il file *.pro* con il comando qmake-qt532 -project si dovrà aggiungere la seguente istruzione all'interno del *.pro* generato: *QT += widgets*. E' stato inserito un database di prova, *database.json*, con le credenziali *Baldan* e *ciao* come username e password oppure *Casey* e *rayback* si accede ad un profilo di immagine completo.

LOGICA

La parte logica del progetto LinQedin è stata implementata perlopiù seguendo le linee guida offerte dal Prof. Ranzato salvo alcune personalizzazioni. La memoria è interamente gestita mediante l'uso di puntatori smart ove necessario, negli altri casi la deallocazione avviene nei distruttori, tutti i puntatori che popolano i contenitori utilizzati, se non associati all'uso di smart pointer, vengono prima deallocati e poi rimossi con i metodi della libreria *std erase()* e *clear()*. Ho scelto di creare una classe *SmartPtr<T>*, che costituisce un puntatore smart templatizzato, per poterlo utilizzare liberamente in ogni parte del programma che richieda gestione automatica di memoria allocata e/o per futuri scopi.

GERARCHIA UTENTI E RICERCA

Un utente è formato da un account, da una rete di collegamenti e da un campo intero che funge da contavisite, possiede inoltre due classi interne incapsulate nella parte privata di esclusivo utilizzo dell'oggetto *User*, esse rappresentano due funtori utilizzati per funzionalità di ricerca e utilità, infine è formato da due vettori contenti dei puntatori smart alla classe *Messagge*, rappresentanti posta in entrata e posta in uscita;di quest'ultima, la capacità è limitata dal grado di privilegio dell'utente. Il campo dati account è un puntatore ad un istanza di *Account* che rappresenta l'account associato all'utente, comprensivo di informazioni personali, livello di privilegio e pagamenti dell'utente associato; mentre il campo dati *net* è un puntatore ad un oggetto *LinqNet*, costituito da una lista di puntatori smart alla classe base *User* con metodi di aggiunta e rimozione. *User* è astratta in quanto possiede alcuni metodi virtuali puri, in particolare il metodo di ricerca, rappresenta è uno degli aspetti su cui verte maggiormente il concetto di polimorfismo, implementato mediante funtore, ogni livello privilegio ha accesso ad un diverso grado di precisione nella ricerca. è estesa "a cascata "da altre 3 classi, che rappresentano i livelli di privilegio all'interno di LinQedin.

Basic User

Utente di livello base, derivato pubblicamente da *User*, possiede funzionalità comuni a tutti gli utenti ereditate da *User* come aggiunta e rimozione collegamenti dalla propria rete, *getters* e *setters* e i metodi di utilita' implementati in *Userc.cpp*. l'override del metodo di ricerca permette un massimo di 50 risultati e non restituisce alcuna informazione aggiuntiva sui target di ricerca se non l' effettiva presenza all'interno del database LinQedin, l'invio dei messaggi mediante override del metodo *BasicUser::sendMessage(const Username&)* è limitato al valore del campo dati statico *static int basicMailLimit*, fissato a 10.

Business User

Un utente business possiede tutte le funzionalità di un utente basic, inoltre ha la possibilità di effettuare una ricerca più accurata ottenendo informazioni sui target visualizzando il profilo completo, il limite è aumentato a 100 risultati, ha inoltre la facoltà di iscriversi a dei gruppi, sezioni di LinQedin amministrate da account Executive dove è possibile creare discussioni o semplicemente dei post informativi / proposte lavorative. Implementa inoltre una funzionalità "passiva ", ossia una lista di utenti suggeriti da un algoritmo di confronto che valuta la somiglianza tra account mediante alcuni criteri, quali la media pesata delle competenze, interessi e lingue, riassunta in una percentuale intera. Quest'ultima feature e' stata creata mediante l'utilizzo di un metodo di calcolo somiglianza, *int similarity(const SmartPtr<User> >&)*, che mediante medie pesate calcola una percentuale di somiglianza tra i profili in base a competenze, lingue e interessi, accoppiato ad un funtore che scorre tutto il database di LinQedin e restituisce un vettore popolato da puntatori smart ad *User* che superano una certa soglia di somiglianza.

Executive User

L'utente executive possiede le massime funzionalità all'interno di LinQedin, messaggistica in uscita illimitata, possibilità di creare ed amministrare gruppi e può visualizzare gli ultimi 10 utenti che hanno visitato il suo profilo con allegate percentuali di somiglianza. Può inoltre ricercare mediante filtri attivabili con il carattere ':'e unificare chiavi di ricerca multiple separandole da una virgola, per esempio ricercando: ':c++,perl,java'restituisce una lista di utenti con competenze / interessi in c++ perl e java, fino ad un massimo di 400 risultati, ogni risultato mostra tutte le informazioni dell'utente in questione, quali gruppi di adesione e lista collegamenti.

ACCOUNT ED INFO

Ho scelto di implementare l'oggetto *User* come un entità rappresentante un utente a livello astratto, estendibile mediante aggiunta di funzionalità nella gerarchia ed override di metodi virtuali gia' presenti, ciò che lo delinea

sono l'account e le info associate.

Account

La classe *Account* è formata da credienziali d'accesso, dalle info personali dell'utente a cui è associato e da un campo che indica il livello di privilegio del profilo utente all'interno di LinQedin, infine di un vector di puntatori alla classe *Payment*. Mantenere traccia del livello di privilegio dell'utente LinQedin all'interno del suo account semplifica alcune operazioni che avrebbero richiesto ulteriori controlli in fase di memorizzazione e lettura dei dati su file, inoltre semplifica la gestione a livelli di astrazione successivi(es: *GUI*) in quanto evita numerose operazioni di RTTI da effettuarsi mediante dei *dynamic_cast* (es: generazione di differenti *Widgets* in base al livello privilegio). Le credenziali d'accesso sono rappresentate dalla classe *Username*, altro non è che un oggetto formato da due campi privati di tipo *string*, username e password con relativi metodi d'accesso; mentre le informazioni utente sono rappresentate da un puntatore alla classe base polimorfa astratta *Info*, estesa dalle classi *UserInfo* e *Bio* ed estendibile in futuro per un eventuale modifica o aggiunta di altre informazioni utili ad un profilo LinQedin. La classe *Payment* è formata da un puntatore allo username del richiedente, un puntatore alla classe *Subscription* che altro non è che una formalizzazione dei piani utente proposti da LinQedin con i costi dell'offerta fissati mediante campi statici, un puntatore alla classe base *BIllMethod* che contiene al momento solamento il pagamento mediante carta di credito, ma estensibile in futuro ad altre modalità di pagamento elettronico, infine un campo booleano che indica l'approvazione o meno del pagamento e un campo QDate che tiene traccia della data di richiesta del pagamento.

account.h

```
class Account {
private:
    Info* _info;
    Username _user;
    privLevel _privilege;
    vector<SmartPtr<Payment> > _history;
    Avatar _avatar;
public:
    ...
};
```

Sistema di rappresentazione delle informazioni

All'interno delle classi informative *Info UserInfo* e *Bio* sono presenti i metodi d'accesso e modifica dei campi dati ma la logica di output di questi ultimi è lasciata ad una classe interfaccia d'appoggio, *Dispatcher*, implementata seguendo il pattern *MVC*. La classe *Dispatcher* è un oggetto senza campi dati con distruttore virtuale i cui unici metodi virtuali puri che possiede ritornano una *string* che rappresenta l'informazione "formattata", pronta per l'output grafico e prendono come parametro un riferimento costante ad ogni sottotipo della classe *Info*, in questo caso sono *string Dispatcher::dispatch(const UserInfo&) const = 0;* e *string Dispatcher::dispatch(const Bio&) const = 0;*. All'interno della classe base *info*, oltre ad un metodo di clonazione è stato dichiarato un metodo di stampa virtuale puro *Info::dispatch* che prende come parametro un riferimento costante ad un oggetto *Dispatcher*, e nel corpo del metodo di stampa viene richiamato il metodo *dispatch* presente in *Dispatcher* con parametro attuale l'oggetto puntato dal puntatore *this*, viene cosi automaticamente risolto il metodo da richiamare mediante overriding e overloading utilizzando il tipo statico passato al chiamante. In questo modo è stato possibile separare al massimo la logica di "dispatching"dal modello, e mediante ereditarietà è possibile creare il proprio sistema di rappresentazione dei dati; in questo caso è stata scelta una rappresentazione in *HTML* mediante la sottoclasse *DispatcherHTML*.

UserInfo.cpp

```
string UserInfo::dispatch(const Dispatcher& d) const {
    return d.dispatch(*this);
}
```

Dispatcher.h

```
class Dispatcher {
public:
    virtual ~Dispatcher();
    virtual string dispatch(const UserInfo&) const = 0;
    virtual string dispatch(const Bio&) const = 0;
};

class DispatcherHtml : public Dispatcher{
public:
    virtual string dispatch(const UserInfo&) const;
    virtual string dispatch(const Bio&) const;
};
```

GESTIONE DATI

A livello logico la gestione dei dati è prerogativa esclusiva della classe *LinqDB*, essa si occupa della lettura e scrittura dei dati su file permanente in memoria, è il primo oggetto ad essere allocato all'avvio dell'applicazione e si occupa di popolare la struttura di LinQedin in ram, rappresentata da una lista di puntatori smart alla classe base polimorfa *User*, mediante metodi privati di lettura richiamati dal metodo pubblico *LinqDB::load()* e al salvataggio di eventuali modifiche mediante metodi privati di scrittura richiamati dal metodo pubblico costante *LinqDB::save()*. La scelta di una lista come contenitore è dettata dal fatto che sia l'inserimento in coda che un eventuale rimozione risultano costanti in tempo O(1), rispetto piuttosto ad un vector che richiederebbe il ridimensionamento per l'eliminazione di un oggetto, mentre l'utilizzo di contenitori associativi quali set o map sarebbe risultato superfluo e inadatto allo scopo. In caricamento o salvataggio dei dati, se non presente, un file viene automaticamente generato, l'operazione di salvataggio si occupa di sovrascrivere i dati ad ogni chiamata del metodo, a prescindere che vi siano state eventuali modifiche o meno ai dati in ram.

Il database

Per il salvataggio permanente dei dati in LinQedin è stato scelto di utilizzare il formato JSON, piuttosto semplice da gestire grazie alle nuove librerie inserite nel framework Qt dalla versione 5 in poi, risulta inoltre più intuitivo e meno verboso di XML, più semplice e leggero di un database SQlite e di facile interfacciamento per un eventuale integrazione online mediante javascript e altri linguaggi web.

LinqClient e LinqAdmin

Due classi di "livello intermedio ", nel senso che si trovano proprio a metà strada tra la parte logica dell'applicazione e la parte grafica. Esse gestiscono rispettivamente la parte utente e la parte amministrativa in LinQedin, forniscono un collegamente tra i metodi logici e la rappresentazione grafica dei risultati richiesti. Sono formati da un campo dati puntatore alla classe *LinqDB*, mediante cui possono apportare qualsiasi modifica al database; la classe *LinqClient* inoltre possiede un campo puntatore ad *User*, che viene inizializzato grazie alle credenziali passate al costruttore, utilizzate per ricavare l'utente già in ram dopo il caricamento dei dati dal DB.

GUI

Per l'interfaccia grafica dell'applicazione sono state utilizzate le librerie Qt alla versione 5.3.2, il front-end è strutturato come una pila di *QGridLayout* contenuti all'interno di un *Widget* padre "globale "mediante un *QStackedLayout* che richiamando lo slot appropriato carica la sezione richiesta. Un menù orizzontale è stato creato utilizzando un *QHboxLayout* contente i pulsanti necessari a spostarsi tra le sezioni, ogni sezione è in effetti un *QGridLayout* ed al suo interno contiene i *widget* necessari alle interazioni dell'utente con la base di dati di LinQedin. Per i contenuti prettamente testuali ho scelto di utilizzare *QTextEdit* e *QTextBrowser* per poter usufruire delle funzionalità HTML supportate ed ottenere cosi una migliore presentazione. All'avvio compare una finestra di login, con due pulsanti, adibiti all'ingresso in LinQedin o alla registrazione di un nuovo profilo mediante un form, mentre utilizzando le credenziali "root "e "toor ", rispettivamente come username e password, viene lanciata la finestra di amministrazione. La classe *Loader* si occupa di restituire il puntatore all'oggetto *LinqClient* allocato sullo heap e inizializzato usando le credenziali d'accesso inserite. Le eventuali situazioni di errore sono state gestite mediante *QMessageBox* catturando le eccezioni sollevate dai vari metodi, ogni sezione possiede un puntatore all'oggetto *LinqClient* che fa da interfaccia tra la parte logica e la parte grafica. Tutto il codice è stato scritto interamente a mano, senza alcun utilizzo di *QtCreator* o dello strumento *QtDesigner*.

Sezioni

1. `Gui_Userwindow:` *Widget* padre contenitore dei layout, deriva pubblicamente da *QWidget* e genera il menù orizzontale formato da *QPushButton*, possiede inoltre l'overload di alcuni eventi del mouse, come ad esempio il *dragging* o l'ingrandimento mediante doppio click, in quanto trattandosi di una finestra *frameless* si distacca completamente dalle direttive del sistema operativo per le operazioni di spostamento o ingrandimento / rimpicciolimento. Tutte le sezioni presentano un avatar, generato mediante la classe *Gui_Avatar*.

2. `Gui_Overview:` E' la prima sezione caricata una volta effettuato il login e rappresenta una sorta di homepage del proprio profilo LinQedin, con le informazioni personali, competenze, interessi, gruppi etc visualizzabili all'interno di *Gui_DispInfo*,classe personalizzata che deriva pubblicamente *QTextBrowser* costituita da un campo edit in modalità *readonly* e due campi dati *QString* utili a mantenere traccia di alcune informazioni in base al contenuto da visualizzare. In questa sezione si trova inoltre la lista delle connessioni e un campo *QLineEdit* adibito alla ricerca, parametrizzata e non; sia i risultati della ricerca che le connessioni sono visualizzate sempre attraverso l'oggetto *Gui_DispInfo*, inoltre vengono generati alcuni pulsanti contenuti in una *QToolBar* appena sotto l'area di output, utili per le operazioni comuni agli utenti quali eliminazione o aggiunta di connessioni, esplorazione dei risulati di ricerce qualora fossero più d'uno. Infine per account di livello Business o superiore vi è inoltre una lista di profili "somiglianti ", che potrebbero rappresentare delle possibili connessioni per offerte lavorative o altro. Deriva da *QGridLayout*.

3. `Gui_Statistics:` Qui troviamo alcune statistiche di profilo, quali lo storico dei pagamenti, un conta visite, il numero di messaggi inviati e rimanenti nel mese corrent e in base ai privilegi dell'account è possibile vedere gli ultimi 10 visitatori, il loro grado di somiglianza con il proprio profilo e la percentuale di keywords che rimandano al proprio account utilizzate nella ricerca dagli altri profili. Deriva da *QGridLayout*.

4. `Gui_Groups:` Area accessibile solo a profili di livello Business o superiore, qui è possibile gestire l'interazione con i gruppi all'interno di LinQedin, o, previa qualifica di livello Executive, crearne e amministrarne di propri. Deriva da *QGridLayout*.

5. `Gui_Messages:` Sezione adibita alla gestione della posta interna di LinQedin, è possibile leggere, inviare messaggi ad altri utenti LinQedin o eliminare la posta più vecchia, ogni account è limitato ad un determinato numero di messaggi in uscita, che, una volta raggiunta la quota, viene resettato automaticamente al primo avvio del mese successivo. Deriva da *QGridLayout*.

6. `Gui_Settings:` Qui viene gestita “l’apparenza ”del profilo, viene generata una serie di *QLineEdit* per la modifica o inserimento di informazioni personali, clickando sul pulsante in basso a destra “UNLOCK ”è possibile sbloccare questi campi e modificarli, una volta terminato clickando il pulsante “SAVE ”le modifiche vengono apportate anche al database e rese permanenti mediante salvataggio. è inoltre possibile aggiungere competenze, lingue o interessi con lo stesso sistema, oppure eliminarne utilizzando un *ContextMenu* accessibile mediante click destro sulle liste. Sempre utilizzando menu a tendina è infine possibile aggiungere, o eliminare esperienze, lavorative o scolastiche.

Ad ogni modifica effettuata viene sempre richiamato il metodo di salvataggio dall’interfaccia *LinqClient*, in ogni caso eventuali modifiche vengono rese permanenti al logout dell’applicazione. I restati layout, *Gui_Login*, *Gui_AdminWindow* e *Gui_Registration* si occupano delle sezioni precedentemente introdotte, login, pannello amministrativo e registrazione nuovo utente.

NOTE

Le eventuali situazioni di errore sono gestite tutte a livello logico mediante la classe *Error*, essa contiene una collezione di codici errore ed un costruttore che inizializza il messaggio, una volta catturata l’eccezione a livello grafico, viene utilizzata la classe *QMessageBox* per stampare a video i problemi rilevati. Per mancanza di tempo non sono riuscito a commentare adeguatamente il codice, e la compilazione presenta un paio di warning innocui.