Relazione progetto LinQedIn

Sistema operativo: Ubuntu 14.04 LTS

Compilatore: g++ 4.8.2

Versione Qt: Qt 5.3.2

Lo scopo del progetto LinQuedIn è quello di realizzare un piccolo sistema per la gestione di un database di contatti professionali ispirato alla piattaforma LinkedIn. Il progetto è stato sviluppato in C++/Qt, seguendo il pattern Model-View-Controller. In prima istanza è stato definito il model, ovvero l'insieme di classi e funzioni utili per poter accedere ed operare sui dati. Successivamente, assieme alla view, è stato definito il controller. Il controller è un'interfaccia che opera sul model e restituisce i risultati alla view. La view invece legge i dati dal model e quando è necessario modificarli, invia un segnale al controller.

1.Model

1.1 Classe Utente

Un utente è composto principalmente dalle sue informazioni di profilo, dal suo username e password e dalla sua rete di contatti.

È stata creata quindi la classe “Profilo” che contiene al suo interno tramite una relazione “has-a” le seguenti classi: “Anagrafica”, “EsperienzeProfessionali”, “EsperinezeScolastiche”, “AbilitaPossedute” e “LingueParlate”.

È stata definita un'ulteriore classe “Luogo” che rappresenta l'insieme di informazioni che fanno riferimento ad un luogo, quali nazione, provincia, comune e via.

Nella classe “Anagrafica” sono stati quindi definite le seguenti variabili: sesso, booleana, nome e cognome come stringhe, data di nascita e luogo di nascita che è di tipo Luogo.

È stata definita poi una classe “Esperienza” che rappresenta un'esperienza generica, la quale contiene una data di inizio, di fine, il luogo in cui si è svolta ed una stringa descrizione che rappresenta la descrizione dell'esperienza. Da questa classe sono state derivate le classi “EsperienzaProfessionale” ed “EsperienzaScolastica”. La classe “EsperienzaProfessionale” oltre a contenere tutti i membri della classe padre, contiene anche due stringhe che rappresentano rispettivamente il nome dell'azienda e la qualifica che l'utente possedeva o possiede in quella determinata azienda. La classe “EsperienzaScolastica” che deriva anche lei dalla classe “Esperienza” contiene invece tre stringhe che rappresentano il nome della scuola, il grado della scuola ed il titolo della scuola nella quale l'utente ha avuto l'esperienza. Per distinguere se un'esperienza è ancora in corso si è deciso di settare una data flag (2100.1.1), che quando letta nella data di fine di un'esperienza indica se questa è ancora in corso.

Poiché un'utente possiede un numero indefinito di esperienze professionali e scolastiche, sono state create due classi ad-hoc per la gestione di un'insieme di esperienze.

È stata quindi definita la classe “EsperienzeProfessionali” la quale al suo interno possiede un campo dati di tipo “std::list<EsperienzaProfessionale>” che contiene un'insieme di oggetti di tipo “EsperienzaProfessionale”. È stata scelta una lista poiché il tipo di accesso che viene effettuato alle esperienze è prevalentemente di tipo sequenziale. Poi è stato definito nella parte pubblica di questa classe un iteratore di tipo costante e due metodi che restituiscono un iteratore che punta rispettivamente al begin e all'end della lista. Questa classe è stata dotata di un metodo per poter inserire nuove esperienze professionali all'interno della lista, di poterle modificare una volta dato l'iteratore che punta all'esperienza da modificare e l'esperienza modificata ed un metodo per poter eliminare un'esperienza attraverso l'iteratore che punta alla

stessa.
La stessa cosa è stata fatta per le esperienze scolastiche, le quali sono contenute all'interno della classe "EsperienzeScolastiche" che ricalca la classe "EsperienzeProfessionali".
È stata poi definita la classe abilità i cui campi dati sono le stringhe "nomeAbilita" e "descrizioneAbilita" che rappresenta una particolare abilità posseduta da un'utente. L'insieme di queste abilità è contenuto nella classe "AbilitaPossedute" nella cui parte privata è stato definito un "std::map<std::string,Abilita>" la cui chiave è il nome dell'abilità. Nella parte pubblica di questa classe è stato definito un iteratore costante e due metodi che restituisco un iteratore che punta al begin e all'end del map. La classe è stata dotato di un metodo di inserimento di una nuova abilità, di modifica di un'abilità dato il nome dell'abilità da modificare e l'abilità modificata ed un metodo per eliminare un abilità dato il suo nome.
Per le lingue parlate da un'utente è stata creata la classe "Lingua" che contiene nei suoi campi dati due stringhe: "nomeLingua" e "livelloCompetenzaLinguista" i cui valori rappresentano i livelli europei delle lingue. Per gestire l'insieme delle lingue è stato seguito lo stesso approccio seguito per le abilità, creando una classe "LingueParlate" il cui campo dati è un map che contiene un insieme di oggetti di tipo "Lingua".
La classe "Profilo" è stata poi incapsulata nella classe Utente con una relazione "has-a".
Per quanto riguarda lo username e la password, è stato deciso di creare una classe login che permettesse la gestione di questi campi dati, a sua volta incapsulati all'interno della classe "Profilo".
La terza componente che un utente possiede è la sua rete di contatti. Si è pensato quindi di creare una classe dedicata alla gestione della rete, chiamata "Rete". Nella parte privata di questa classe è stato dichiarato un campo dati di tipo std::map<std::string, const SmartUtente*>. La chiave di questo map è lo username di un utente mentre SmartUtente rappresenta una classe contenente un puntatore ad un oggetto di tipo "Utente" la cui funzione verrà spiegata al punto 1.7. Nella parte pubblica della classe "Rete" è stato dichiarato un iteratore costante e due metodi per potere ottenere un iteratore che punta rispettivamente al begin e all'end della rete. Inoltre la classe è stata dotata di un metodo per poter aggiungere un nuovo utente alla propria rete ed un metodo per poterlo eliminare. Infine è stato creato un metodo che una volta passato un iteratore che punta ad uno "SmartUtente", questo restituisce lo "SmartUtente" stesso.
Nella parte privata della classe "Utente" sono quindi presenti tre puntatori: "Profilo*" che punta ad un oggetto di tipo "Profilo", "Login*" che punta ad un oggetto di tipo "Login" ed infine "Rete*", un puntatore che punta ad un oggetto di tipo "Rete".
Nella parte pubblica di "Utente" ritroviamo tutti i metodi "set e get" che consentono di andare ad operare sugli oggetti che punta.
La classe "Utente" è una classe astratta pura grazie alla presenta dei metodi virtuali puri "std::map<std::string, Utente*>cercaUtente(Database*, const std::string&) const=0" ed al metodo "std::vector<std::string> getDescrizioneRicerca() const=0" le cui funzioni verranno spiegate al punto 1.4.
Nella parte protetta della classe "Utente" è stato definito un oggetto "FuntoreRicerca" la cui funzione sarà spiegata anch'essa al punto 1.4.

1.2 Classi "UtenteGratis" e "UtenteAPagamento"

Poiché gli utenti di LinkedIn si suddividono in utenti paganti e non paganti, per essere il più possibile coerenti con questo modello e per ragioni di estensibilità del codice, si è deciso di derivare da utente due classi: "UtenteGratis" ed "UtenteAPagamento". Anche se queste classi non implementano nuove funzionalità, non è da escludere che in futuro possano includere nuovi campi dati e nuovi metodi.

1.3 Classi “UtenteBase”, “UtenteBusiness” ed “UtenteExecutive”

La principale differenza presente tra i diversi tipi di utente di LinkedIn consta nella capacità di ricerca che il singolo utente possiede. Questa differenza è stata implementata nel progetto creando tre nuove classi.
La classe “UtenteBase”, che deriva dalla classe “UtenteGratis”, permette ad un utente di cercare nuove persone solamente attraverso i dati anagrafici delle stesse.
La classe “UtenteBusiness”, che deriva da “UtenteAPagamento”, permette ad un utente di cercare altri utenti non solamente attraverso i loro dati anagrafici, ma anche attraverso i titoli di studio e le lingue parlate.
La classe “UtenteExecutive” è quella che rappresenta la miglior capacità di ricerca ricalcando le potenzialità di ricerca della classe “UtenteBusiness”, a cui viene aggiunta la possibilità di cercare un utente anche per le esperienze professionali e le abilità personali. Anche questa classe deriva da “UtenteAPagamento”.

1.4 Ricerca

Il principale obbiettivo di LinkedIn è quello di poter cercare altri utenti all'interno di LinkedIn stesso. Per poter dare questa possibilità anche agli utenti di LinQedIn, nella classe base astratta “Utente” è stato dichiarato il metodo virtuale puro “std::map<std::string, Utente*>cercaUtente(Database*, const std::string&) const=0”. Questo metodo virtuale permette di poter lanciare una ricerca in modo completamente polimorfo, che a seconda del tipo dinamico del puntatore ad “Utente”, restituirà un insieme diverso di risultati. In “Utente” è poi stata definita nella parte protetta, la classe “FuntoreRicerca”. Il costruttore di questa classe è un costruttore ad un solo parametro che si aspetta una stringa. Una volta che è stato invocato, questa stringa viene inserita all'interno di un “std::vector<std::string>” i cui elementi sono le singole parole della stessa.
Nella parte privata della classe “UtenteBase” è stata derivata da “FuntoreRicerca”, la classe “FuntoreBase” (per questo motivo era stata dichiarata nella parte protetta di “Utente”). La stessa cosa è stata fatta anche per le classi “UtenteBusiness” ed “UtenteExecutive”, andando a chiamare le classi derivate da “FuntoreRicerca” rispettivamente “FuntoreBusiness” e “FuntoreExecutive”.
La ricerca funzionerà quindi nel seguente modo:
1)Un “Utente*” chiamerà il metodo “cercaUtente” passandogli come parametri un puntatore al Database (illustrato nel punto 1.5) e la stringa da cercare.
2)A seconda del tipo dinamico di “Utente*” verrà invocato il metodo “cercaUtente” re-implementato nella classe che corrisponde al suo tipo dinamico.
3)Supponendo che il tipo dinamico fosse stato “UtenteBase*”, il metodo “cercaUtente” costruirà un “FuntoreBase”, passandogli come parametro al costruttore di “FuntoreBase” la stringa da cercare. Successivamente il metodo “cercaUtente” inizierà ad iterare sul database di utenti. Ad ogni iterata, verrà invocato il “FuntoreBase” a cui è stato fatto l'overloading dell'operatore parentesi passandogli l'utente a cui punta l'iteratore al database. Il funtore a questo punto andrà a controllare se la stringa cercata è presente all'interno dell'utente e se lo è restituirà true. A questo punto il metodo “cercaUtente”, se il funtore ha restituito true, prenderà il puntatore all'utente e lo metterà all'interno di un vector di oggetti di tipo “Utente*”.
4)Il metodo “cercaUtente”, dopo aver scansionato tutto il database, restituirà al chiamante un vector contenete tutti gli utenti che corrispondono alla stringa cercata.
Il punto 3 sarebbe stato uguale anche nel caso il tipo dinamico di “Utente” fosse stato “UtenteBusiness” o “UtenteExecutive” con la sola differenza che sarebbero stati invocati rispettivamente i funtori “Business” ed “Executive” che sarebbero andati a controllare anche altri campi dell'utente su cui venivano invocati.

Per semplificare la ricerca si è pensato di dotare le classi “Anagrafica”, “EsperienzeProfessionali”, “EsperienzeScolastiche”, “LingueParlate” ed “AbilitaPossedute” di un metodo “bool contieneStringa(const std::string)” che una volta passata una stringa, restituisco true se la stringa è presente all'interno dei loro campi dati.
Poiché in c++ non è stato trovato alcun metodo all'interno della libreria standar che cercasse una stringa dentro ad un'altra in modo del tutto case-insesitive, si è ritenuto opportuno creare una classe “CercaStringa” per ovviare a questo problema. Questa classe infatti fornisce dei metodi che consentono il confronto case-insesitive tra stringhe.

1.5 Database

Per poter salvare gli utenti e poterli ritrovare alla successiva riapertura dell'applicazione è stato necessario creare un database che salvasse i dati su file. Il database salva i dati in formato XML su file e per la sua implementazione sono state usate le classi della libreria Qt QFile, QXmlStreamWriter e QXmlStreamReader.
È stata dunque creata la classe “Database” che contiene nella sua parte privata un map di oggetti di tipo “SmartUtente&” e come campo chiave lo username dei singoli utenti. È stato poi definito nella parte pubblica un iteratore e 2 metodi per potere ottenere rispettivamente un iteratore che punta al begin ed un uno all'end del map. Questa classe è poi stata dotata di un metodo per poter inserire un nuovo utente all'interno del map una volta passato un puntatore allo stesso, ed un metodo per poter eliminare un utente data una stringa contente il suo username. L'eliminazione di un utente viene gestita in modo profondo invocando la delete su “SmartUtente” il quale invocherà la delete profonda su “Utente”. È stato poi definito un ulteriore metodo che consente di ottenere un puntatore ad un utente una volta data la stringa contenete il suo username.
Il metodo “salvaDatabase” legge il contenuto del map e salva i singoli utenti all'interno di un file. Utilizzando QXmlStreamWriter, per ogni campo dati di utente, viene creato un nodo XML contente il valore del dato e viene scritto su file. La rete di un utente viene invece scritta su un file differente, per evitare problemi al caricamento del database, quando potrebbero non essere ancora stati inizializzati i puntatori ai contatti della propria rete. Per il salvataggio è stato necessario fare “typechecking” per andare a controllare il tipo dell'utente e salvarlo nel file in modo che al successivo caricamento del database gli utenti mantenessero lo stesso tipo.
Il caricamento su file avviene leggendo il file creato dopo aver salvato il database e andando popolare il map. La rete viene caricata dopo aver creato tutti gli utenti.
Il distruttore di questa classe è stato ridefinito, andando ad invocare una distruzione profonda su oggetti di tipo “SmartUtente&” allocati nello heap, i quali a loro volta invocano la distruzione profonda sul puntatore ad “Utente” a cui essi puntano.
Nella classe “Utente” è stato ridefinito l'operatore delete, il quale una volta invocato, va a distruggere in modo profondo gli oggetti puntati dalla classe “Utente”.

1.6 Amministratore

Per gestire il lato amministratore ovvero l'inserimento, l'eliminazione ed il cambio di tipo di iscrizione di un utente, è stata creata la classe “Amministratore”. Essa contiene nella sua parte privata un puntatore alla classe Database. Il metodo “caricaDatabase” alloca nello heap un nuovo database e successivamente invoca il metodo “caricaDatabase” che richiama dal puntatore al database, il metodo “caricaDatabase” definito dentro “Database”. Il metodo “salvaDatabase” invece invoca il metodo “salvaDatabase” sul puntatore al database, definito dentro la classe “Database”. L'inserimento di un nuovo utente prende un puntatore “Utente*” e lo passa al database, il quale lo aggiunge al map. Il metodo “eliminaUtente” si aspetta una stringa con lo

username dell'utente da eliminare ed invoca la rispettiva funziona dal puntatore al database. Sono stati definiti ulteriori metodi per poter accedere ai dati degli utenti e poter iterare sul database.

1.7 Cambio iscrizione e la classe "SmartUtente"

Una delle funzionalità offerte da LinkedIn è quella di permettere il passaggio di un utente di tipo basic ad un utente di tipo business o executive, pagando. Il pagamento deve avvenire mensilmente e nel caso un utente decidesse di non pagare più, il suo account verrebbe retrocesso ad un account di tipo basic. Uno dei compiti della classe amministratore è quello di permettere il cambio di tipo utente.

La classe "SmartUtente" è nata per semplificare questo tipo di passaggio. Essa contiene nella sua parte privata un puntatore ad un "Utente*". È stato poi ridefinito l'operatore di assegnazione la cui definizione è la seguente: "SmartUtente& operator=(Utente*)". Quando viene invocato l'operatore di assegnazione tra uno "SmartUtente" ed un "Utente*", il puntatore contenuto dentro "SmartUtente" viene sostituito con il nuovo puntatore ad "Utente" mentre sul vecchio puntatore viene invocata la delete standard di "Utente". È necessario invocare la delete standard poiché tra il puntatore ad il nuovo utente ed il puntatore ad il vecchio utente c'è condivisione di memoria.

Il cambio di iscrizione funziona così:

1) Dalla classe "Amministratore" viene invocato il metodo "cambiaTipoUtente" che prende come parametri due stringhe, una contenete lo username dell'utente da modificare e l'altra che contiene il nome del nuovo tipo di utente. Questo metodo invoca attraverso il puntatore al "Database" il metodo "cambiaTipoIscrizione" definito in esso.

2) Supponendo di voler cambiare il tipo dell'utente in "UtenteBase", il metodo "cambiaTipoIscrizione" dentro la classe "Database", una volta trovato lo SmartUtente che punta all'utente da modificare, esegue la seguente istruzione: " it->second= new UtenteBase(&*(it->second))". Il campo it->second è di tipo "SmartUtente" e punta allo "SmartUtente" contente il puntatore all'utente da modificare. Con l'istruzione "new UtenteBase(&*(it->second))" viene invocato il costruttore di "UtenteBase" ad un parametro che si aspetta un "Utente*". Questo costruttore costruisce un nuovo utente assegnando ai puntatori alle classi "Profilo", "Login" e "Rete" i puntatori dell'utente passato, che sarà l'utente di cui dobbiamo modificare il tipo. In questo modo otteniamo un utente del nuovo tipo senza dover fare copie profonde. Il nuovo utente ritornato viene assegnato allo "SmartUtente".

3)Viene allora invocato l'operatore di assegnazione di "SmartUtente" con il puntatore all'oggetto "Utente" del nuovo tipo. A questo punto il comportamento di questo operatore è lo stesso descritto in precedenza ed avremmo così ottenuto l'utente con il nuovo tipo desiderato.

2. View

La view comprende la realizzazione dell'interfaccia grafica. Nella schermata iniziale sono presenti due bottoni che consento di decidere se effettuare il login come amministratore o come utente. Entrando come amministratore verranno caricati dal database tutti gli utenti presenti e compariranno in una tabella sulla sinistra della finestra. Sulla destra invece è possibile scegliere quali operazioni effettuare. Per realizzare questa finestra è stato creata la classe "AmministratoreView" che si occupa del caricamento di tutti i widget necessari.

Effettuando il login come utente, verranno richiesti username e password. Una volta inseriti, si aprirà una finestra che a sinistra presenta tutte le informazioni di profilo mentre sulla destra tutte le possibili operazioni che un utente può effettuare. La classe "UtenteView" è quella che si occupa della gestione e del caricamento

di tutti i widget di questa finestra.
Sulla destra è possibile aprire la finestra della ricerca che consente di poter cercare nuovi utenti. A seconda del tipo di utente, sulla destra comparirà una piccola descrizione dei campi in cui può cercare. Questa descrizione è del tutto polimorfa e per realizzarla, è stato aggiunto il metodo virtuale puro "getDescrizioneRicerca()" dentro la classe "Utente" che restituisce un vettore contente le stringhe rappresentanti i campi in cui un utente può cercare.
Nelle view troviamo quindi diverse classi che caricano finestre differenti. In ogni view è sempre presente un puntatore al model, che può essere un puntatore al "Database" oppure un puntatore ad "Utente" o ad "Amministratore", a seconda della finestra in cui ci troviamo. Questi puntatori servo esclusivamente per la lettura dei dati dal model

3. Controller

Il controller è l'intermediario tra le view ed il model. In ogni classe del controller è sempre presente un puntatore alla view che sta gestendo ed un puntatore al modello a cui fa riferimento. La classe "AmministratoreGeneralController" serve per gestire il cambio di finestre che ci può essere all'interno della classe "AmministratoreView". La classe "AmministratoreController" invece raccoglie i segnali che gli arrivano dalla classe "AmministratoreView" e comunica con la classe "Amministratore" andando a modificare i dati del model in base al segnale ricevuto. Una volta che i dati sono stati rielaborati, la classe invia un segnale alla view, andando ad aggiornala.
Per la parte utente è stata fatta la stessa cosa. È stata creata la classe "UtenteGeneralController" e la classe "UtenteController" il cui comportamento è speculare alle classi sopra descritte andando a cambiare solamente il model a cui fanno riferimento.
È stata creata un'ulteriore classe "Controller" il cui compito è quello di fare da avvio dell'applicazione e di gestire la schermata iniziale dell'applicazione. Essa infatti possiede un puntatore alla schermata iniziale, un puntatore al controllore generale dell'amministratore ed un puntatore al controllore generale dell'utente. Grazie a questa classe è quindi possibile fare il logout da amministratore, loggarsi come utente e viceversa.

4. Note

Per quanto riguarda la compilazione del progetto, una volta lanciato il comando "make", verrà creata una nuova cartella "Build" all'interno del progetto stesso nella quale sono presenti i file compilati e l'eseguibile. Se l'eseguibile dovesse essere spostato da quella cartella, non riuscirebbe più a trovare i file del database e cui non verrebbe caricato.