LinQedIn

Davide Polonio (1070162)

August 25, 2015

Contents

1 Introduzione

1.1 Ambiente di sviluppo

La realizzazione di LinQedIn è avvenuta nel seguente ambiente:

- Sistema operativo di test e sviluppo: ArchLinux x86_64 (rolling release)
- Compilatore: GCC versione 5.2.0
- Qt: versione 5.5.0
- QMake: versione 3.0
- Qt Creator: versione 3.4.2

1.2 Descrizione del progetto

Come da consegna, il progetto si propone di ricreare le principali funzionalità di LinkedIn. Sono presenti 3 tipi di iscrizione al servizio: iscrizione come membro Basic, Business o Executive. L'amministrazione del sistema è gestita da un Admin. Gli iscritti al servizio possono compiere le seguenti azioni:

- Modifica del proprio profilo, con possibilità di aggiungere i propri Hobby, i propri Interessi e le proprie Esperienze (lavorative)
- Possibilità di ricercare altri iscritti al servizio in base alla loro sottoscrizione:
  - Basic: possibilità di ricerca solamente per Nome e Cognome
  - Business: tutte le funzionalità di basic con l'aggiunta della ricerca per una specifica data di nascita o per un particolare Hobby
  - Executive: possiede tutte le funzionalità di ricerca del servizio. Rispetto agli iscritti Business, i membri Executive possono ricercare anche tramite Interessi
- Visualizzare i profili degli altri membri dalla sezione di ricerca. La visualizzazione dei profili cambia per tipologia di iscritto:
  - Basic: visualizzazione delle informazioni generali (nickName, nome, cognome, data di nascita) e delle amicizie
  - Business: stessa vista dei membri Basic con l'aggiunta degli Hobby e degli Interessi
  - Executive: stessa vista dei membri Business con l'aggiunta delle Esperienze lavorative
- Gestione delle proprie amicizie con possibilità di aggiunta e rimozione

L'amministratore di LinQedIn ha a disposizione le seguenti funzionalità:

- Creazione di un nuovo membro con la possibilità di scelta di tipo del nuovo utente e con una procedura guidata per l'aggiunta delle informazioni principali.

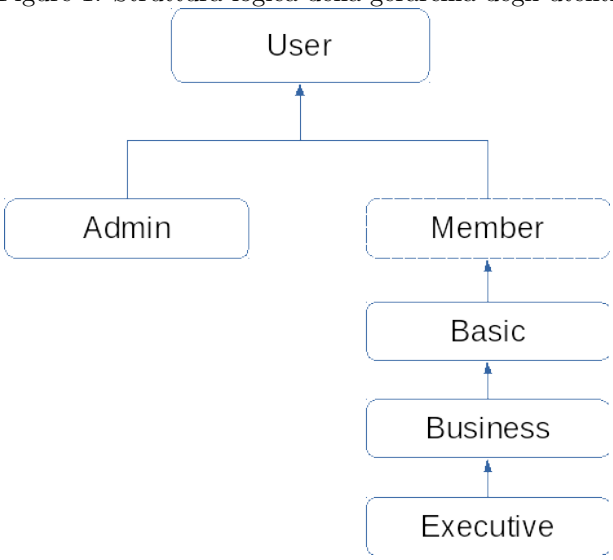
- Ricerca degli iscritti al servizio con possibilità di visualizzare il loro profilo
- Rimozione di un membro iscritto
- Cambiamento di tipologia di un utente Iscritto
- Salvataggio del Database

LinQedIn si appoggia su un database scritto nel formato xml, dove vengono salvate le informazioni relative agli Iscritti e le loro amicizie.

2 Descrizione delle classi

2.1 Gerarchia degli utenti

Figure 1: Struttura logica della gerarchia degli utenti



Alla base di tutta la gerarchia degli utenti è presente la classe *Users*. Questa è una classe virtuale, che contiene informazione di base, come il tipo di account in uso, il Database a cui appartiene e la sua validità. Dato che logicamente non ha senso che esista un User, esso non è istanziabile.
Derivato a User c'è *Admin*, che è una classe istanziabile e concreta. Essa implementa le funzionalità dell'amministratore, come il metodo di ricerca, il cambiamento di tipo di un Membro, la sua aggiunta e la sua cancellazione.

Derivata da User c'è anche la classe *Member*, virtuale pura. Essa pone le basi per tutte le tipologie di iscritti a LinQedIn. Qui è presente il metodo search da implementare per ogni classe derivata. Member implementa già la scrittura e la lettura delle informazioni dal database xml per ogni tipologia di iscritto. Basic, Business, Executive sono classi derivate che implementano la ricerca sul Database.

2.1.1 Classi contenute in Member

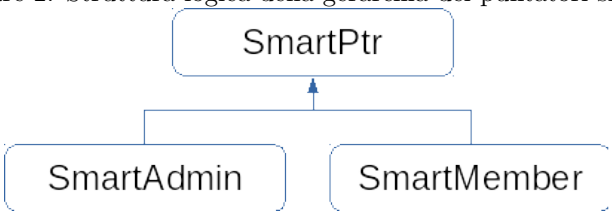
Member contiene diverse classi, *Profile* e *Friendships* a loro volta formate da più classi o derivate da altre.

Profile è una classe che contiene le informazioni personali e la carriera dell'iscritto, ingloba tramite una relazione *has-a* le classi *Personal* ed *Experiences*.
Personal contiene a sua volta *Bio, Hobby, Interests*. È da notare che *Hobby, Interests* e *Experiences* derivano da classi contenitori. Questa scelta di creare dei wrapper è dovuta alla maggiore estendibilità del codice: se un domani si volesse cambiare contenitore, le modifiche andrebbero eseguite solo in quelle classi.

Friendships deriva da un contanier *vector* e si occupa di salvare le amicizie di ogni Membro. L'amicizia viene salvata tramite il nickName, che è univoco per tutto il Database.

2.2 Gerarchia degli SmartPointer

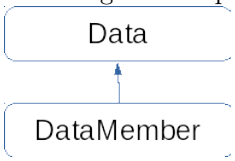
Figure 2: Struttura logica della gerarchia dei puntatori smart



Nonostante la gestione della memoria condivisa non sia stato un problema data la natura implementativa del progetto, è stato deciso comunque di utilizzare puntatori smart. SmartPtr è una classe base con il costruttore dichiarato protected, questo per impedirne una sua istanziazione tranne che per le classi derivate da essa: avere una classe base comune a tutti i smartPtr permette di aggiungere funzionalità a tutte le classi derivate, e di introdurre metodi virtuali o puri. *SmartAdmin* e *SmartMember* gestiscono la condivisione in memoria rispettivamente di Admin e Member, utilizzando il reference counting.

2.3 Gerarchia della gestione degli User

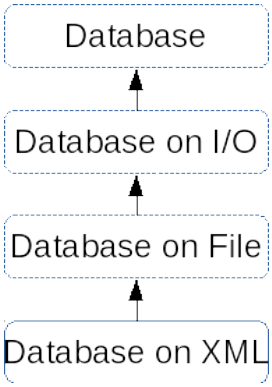
Figure 3: Struttura logica della gerarchia per la gestione degli User



Come scelta progettuale, si è deciso di non tenere la lista degli utenti direttamente sul Database, ma di creare una gerarchia di classi per la gestione di essi che poi verranno gestite a loro volta dal Database. Questo permette una più facile gestione degli utenti (attualmente solo Membri, ma potrebbero esserci altre tipologie). La gerarchia ricalca la stessa struttura data per gli smart pointer e per gli user.

2.4 Database

Figure 4: Gerachia del Database



Il Database è stato strutturato in maniera tale da rendere possibile la scrittura della base di dati in diversi formati o in diversi device, questo grazie alla derivazione da Database, classe virtuale pura in cui vengono implementati solamente i metodi di ricerca (*select*) sull'oggetto *DataMember*, che si occupa come descritto prima di mantenere in memoria gli iscritti al servizio. Per questo progetto è stato deciso di salvare il database nel formato xml, ma grazie a questa gerarchia è possibile implementare un diverso metodo di salvataggio. Si noti come in DatabaseOnFile (nel codice viene chiamato *DBonFile*) si ha una derivazione multipla: una dalla classe *DBonIO*, l'altra da *QFile*; questo in quanto le classi derivate necessiteranno delle funzioni di *QFile* per riuscire a implementare le funzioni pure di *Database*.

2.5 Interfaccia grafica

2.5.1 Pattern MVC

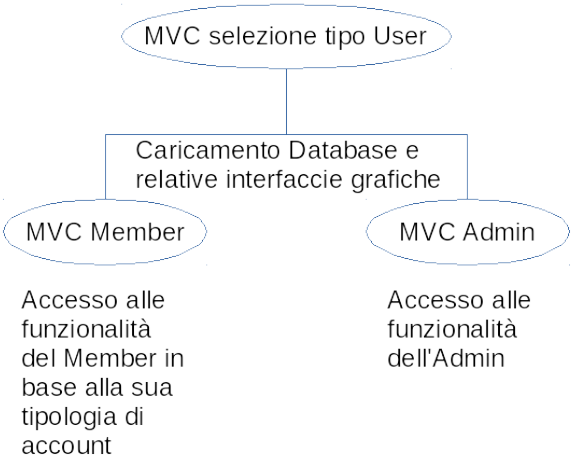
Nella scrittura della GUI si è optato per l'uso del pattern MVC: si ha quindi il *Controller* che fa da ponte tra la *View* e il *Model*, e si occupa di effettuare eventuali controlli di consistenza con i dati e di elaborare gli imput provenienti dalla *View* stessa.

2.5.2 Idea e schema di funzionamento

La gestione dell'interfaccia grafica, basata sul pattern MVC, si prepone di caricare in RAM solo la parte effettivamente utilizzata dal programma. Dato che non è possibile cambiare utente dopo la selezione del tipo, caricare in RAM anche l'altra parte di interfaccia grafica che non verrà certamente usata sarebbe un inutile spreco di risorse. Quindi, il *MainWindowController* si occupa di far istanziare al *MainWindowModel* solamente la parte effettivamente utilizzata.

Per ogni funzionalità è stato deciso di creare un proprio pattern MVC: questo permette di isolare ogni parte in una propria finestra e di rendere le modifiche a quella specifica sezione più semplici.

Figure 5: Schema di funzionamento della GUI



Gestione del Database In base alla scelta del tipo di User, viene creato un oggetto *DBonXml*:

```
Database* db = new DBonXml(‘‘database’’);
```

questo porta a una riduzione dell'estendibilità del codice (se per esempio si decidesse di apportare una modifica comune all'istanziazione del database ciò dovrebbe essere ripetuto per tutti i controller che istanziano il database), ma permette di apportare modifiche singole per ogni controller che istanzia un Database.