Relazione sul progetto di Programmazione ad oggetti

Macrì Antonino, Matricola: 1045245
7 settembre 2015

Abstract

Il software `e stato sviluppato in linea con il design pattern MVC (Model-View-Controller), mantenendo una separazione delle parti tramite sottocartelle che raggruppassero i concetti diversi rappresentati. In particolare nella parte del model si è posta cura nel dividere le cartelle in modo da sottolineare i 3 concetti fondamentali da esprimere:

1. Le caratteristiche della persona, lavoratore con esperienze(cartella Persona)
2. Le caratteristiche dell'utente, persona all'interno della struttura LinQedIn, con un tipo di account ed una rete di contatti LinQedIn (cartella Utente)
3. Il database e le due figure che vi hanno accesso con rispettivi privilegi, client ed admin (cartella Database)

L'attenzione all'estensibilità è stata resa nella cura della suddivisione del codice e mediante derivazione gerarchica. Cercando di incapsulare in ogni classe un concetto specifico di cui poi sarebbe stata l'unica ad occuparsi.
Il software è stato sviluppato usando solo classi Qt.

GERARCHIA DI CLASSI: Utente

PERSONA

La persona è la classe che si preoccupa dei metodi setter e getter per le caratteristiche del lavoratore linQedIn. É la classe base della gerarchia di classi che porta hai 3 tipi di utente presenti.
Ha al suo interno 6 campi dati: QString cognome, QString nome. competenze competente, Lingue poliglotta, Residenza abitazione, Nato origine.
In particolare:

- Per la classe Lingue ho scelto un campo dati di tipo QSet<QString> perché il contenitore associativo è il più adatto per ricerche pur avendo basse prestazioni per operazioni di rimozione che comunque saranno assolutamente minori e quindi meno significative per le prestazioni.
- Per la classe competenze ho scelto un campo dati di tipo Qmap<QString,mansione>, dove mansione è una classe che rappresenta il concetto generale di abilità acquisita svolgendo qualcosa (studio, corso specifico, esperienza lavorativa). La chiave di ricerca è un titolo dato alla mansione. Anche in Questo caso la scelta del contenitore è stato di tipo associativo per l'importanza delle prestazioni della ricerca a discapito di operazioni di eliminazione e modifica che sono come intuibile meno presenti.

UTENTE

L'utente è una classe astratta che rappresenta una persona all'interno del database. Quindi eredita tutti i metodi setter e getter di persona con il contratto di di aggiungere una rete di contatti linQedin e una username univoca di tipo Qstring.
Le funzioni obbligatorie per la gerarchia di Classi derivata da Utente sono:

- Virtual QString typeUtente() const =0: restituisce in modo polimorfo il nome del tipo di utente. Utilizzata solo ottenere nella view la stampa del tipo di utente. La ricerca e' effettuata in modod dinamico tramite ricerca.
- Virtual bool ricerca(const SmartUtente& sm, const QString& s) const =0: restituisce un booleano true/false se l'utente sm contiene in uno dei suoi dati personali la stringa s.
  Il controllo e' effettuato in base al proprio tipo di utente.
  Ad esempio: se chiamo u->ricerca(sm,"user") con l'utente u di tipo basic, controllera' se sm ha la stringa "user" nel nome,cognome o username. Cosi' facendo quando viene richiamata la ricerca non importa sapere di che tipo e' l'utente che chiama la ricerca, ma sara' il polimorfismo a effettuare dinamicamente la scelta della funzione ricerca tramite l'overriding.
- Virtual Utente* clone() const =0: Utilizzata per il costruttore del puntatore Smart con cui si tiene in memoria un utente nel database.

L'utente è costruito solo mediante cognome,nome e username. Le altre caratteristiche saranno settate tramite i metodi setter derivati da persona.
Per la classe Rete ho scelto un campo dati di tipo Qset<QString> contenente la username per le stesse ragioni già esposte per la classe lingue.

UTENTEGRATIS ED UTENTEPAY

La Struttura della gerarchia prosegue dividendosi in due utenti a loro volta astratti: UtenteGratis ed UtentePay.
UtenteGratis di fatto non aggiunge nulla ad utente ma rappresenta il concetto di utente che non paga.
UtentePay rappresenta invece l'utente pagante, che quindi fornisce una carta di credito per il pagamento annuale e 2 metodi statici virtuali per modificare il valore fisso dell'abbonamento.

UTENTEBASIC

Classe concreta derivata da UtenteGratis. Soddisfa il contratto di essere un utente non pagante che è iscritto a LinQedIn.
Il suo significato è quello di fornire i propri dati per la ricerca degli altri utenti ed una rete di contatti basata sulla possibilità di inserire qualcuno cercandolo tramite username, nome e cognome.

UTENTEBUSINESS

Classe concreta derivata da UtentePay. Soddisfa il contratto di essere un utente pagante che è iscritto a LinQedIn con un proprio abbonamento fisso da pagare annualmente.
Il suo significato è quello di fornire i propri dati per la ricerca degli altri utenti ed una rete di contatti basata sulla possibilità di inserire qualcuno cercandolo tramite username, cognome, nome, competenza, lingua, residenza e luogo di nascita.

UTENTEEXECUTIVE

Classe concreta derivata da UtentePay. Soddisfa il contratto di essere un utente pagante che è iscritto a LinQedIn con un proprio abbonamento fisso da pagare annualmente.
Il suo significato è quello di fornire i propri dati per la ricerca degli altri utenti ed una rete di contatti basata sulla possibilità di inserire qualcuno cercandolo tramite username, cognome, nome, competenza, lingua, residenza, luogo di nascita e contenuto nella rete di utenti.

GERARCHIA DI CLASSI: SmartUtente

SMARTUTENTE

Classe base della gerarchia. Il suo contratto è quello di smart pointer per la classe Utente e ne ridefinisce la distruzione del puntatore ad utente.

SMARTCHANGETYPE

Classe derivata da SmartUtente. Risulta quindi essere uno smart pointer che ridefinisce la distruzione del puntaore, ma in aggiunta il suo contratto prevede di occuparsi solamente della gestione del cambio di tipo di utente.
Attualmente ridefinisce solo l'operazione di cambio tipo ma è stata pensata con l'idea di staccare il concetto di smart pointer inteso come puntatore ad utente dal concetto di cambio di tipo dove magari in un futuro sarà necessario aggiungere ulteriori proprietà tra i vari tipi di utenti. Queste possibilità di aggiungere proprietà mediante i puntatori sono quindi incapsulate in questa classe.
La ragione princitpale di Questa scelta è stata l'importanza che riveste il cambio di tipo all'interno del progetto, che anche se per ora non esegue molte operazioni concettualmente era uno dei punti principali che caratterizzano LinQedIn e verso cui è naturale ipotezzare dei cambiamenti anche importanti in futuro
Questa è la classe effettivamente usata anche all'interno del database (non si usa SmartUtente).

PARTE DATABASE

DATABASE

Il Database è la classe che si occupa di creare il database, gestirlo inserendo, eliminando e modificando gli utenti e la rete di utenti e occupandosi della sua memorizzazione tramite salvataggio su file e caricamento in memoria RAM.
Possiede un unico campo dati Qmap<QString,SmartChangeType*> le ragioni di questa scelta ricadono sempre sulla proprietà di ricerca data dal Qmap, che come key possiede la username.
La username viene cercata sempre in modo case sensitive mentre la ricerca effettuata per gli altri campi è di tipo case insensitive.

- SALVATAGGIO SEMPLICE: Avviene usando le classi fornite da Qt per il salvataggio dati mediante linguaggio: XML.
  Il campo dati statico filename contiene il percorso della posizione di memorizzazione del database.xml. Attualmente è impostata in modo che cerchi il file .xml nella cartella dove crea i file il compilatore e che in caso di fallimento di lettura ritornerà la risposta al

terminale ma eseguirà lo stesso il programma che ha chiusura salverà il nuovo database nella posizione indicata.

- SALVATAGGIO SICURO: È possibile salvare dal client usando la funzione:
  void save_const() const; è stata definita costante poiché questa funzione salva su un secondo file (database2.xml). In questo modo l'admin ed il client salvano su 2 file diversi ma in lettura il database attraverso QfileInfo ottiene l'informazione sul file modificato più di recente che è salvato in memoria e quindi carica quest'ultimo.
  Visto che il programma comunque NON offre la possibilità di lavorare sia come client che come admin, questa operazione è per una eventuale possibilità futura di implementare questa possibilità. In secondo luogo è stata inserita anche per garantire maggiore sicurezza durante questa delicata operazione per i dati.
- Non c'è un comando previsto per cambiare il tipo di salvataggio visto che il sistema non offre la possibilità di lavorare sia come client che come admin, Il salvataggio è quindi fatto in modo semplice. Il caricamento è effettuato dalla funzione load( ) con il costruttore del database e la scrittura dalla funzione save() con il distruttore del database.
  Per vederne il funzionamento del salvataggio sicuro basta richiamare le funzioni “save()” di admin e “save() const” di client togliendo la save() dal distruttore di database.

ADMIN

L' Admin è la classe che esegue le operazioni di gestione in modalità admin del database cioè permettendone le operazioni previste di inserimento,eliminazione, ricerca (tramite nome,cognome e username), cambio tipologia dell'account.
Il campo dati è di tipo DataBase* potendo effettuare tutte le operazioni.

CLIENT

La classe client è la classe che esegue le operazioni di gestione in modalità client del database cioè permettendone le operazioni previste di aggiornamento profilo, modifica della rete di contatti e ricerca in base al proprio tipo di utente.
Il campo dati è di tipo const DataBase*; e grazie al salvataggio definito const poiché effettuato su un altro file.xml diverso da quello dell'admin, al client può essere facilmente inserita la possibiltà di richiamare il salvataggio prima che lo faccia il distruttore del database.
Punta poi all'utente della sessione mediante un campo dati SmartChangeType*;

TYPEDEF PER GLI ITERATORI

Le Classi Rete, database, lingue e competenze sono state dotare tramire typedef di iteratori con il proprio nome questo permette di effettuare modifiche anche pesanti all'interno delle classi garantendo la possibilità di avere degli iteratori che se chiamati esternamente non saranno da ridefinire in ogni codice che usi quella classe.
Questa possibilità probabilmente non servirà mai, ma dal mio punto di vista mi permetteva di leggere meglio il codice grazie al nome dell'iteratore ridefinito a doc per ogni classe perciò ho voluto comunque inserirla.

PARTE CONTROLLER-VIEW

CONTROLLER

I due controller sono AdminControl e ClientControl.
Loro compito è quello appunto di far comunicare la parte logica con la parte view del progetto richiamando levarie costruzioni delle tabelle per tenere i dati aggiornati con le modifiche avvenute in ternamente al db ad opera delle richieste inviate dalla view.
Hanno entrambi un valore booleano che indica se ho meno devono creare la view in base al comando all'input corretto dell'utente.

- Nel ClientControl è presente una funzione chiamata: `void viewReadMode() const;` che chiama la funzione readMode(); presente in ClientView che apre il client in modalità sola lettura. La funzione è stata definita in vista di una possibilità futura di aggiungere al progetto una modalità di sola lettura.
  Per vedere la schermata di una user in ReadMode, basta eliminare nel costruttore di ClientControl il commento della funzione (//viewReadMode()) presente in fondo al costruttore.

VIEW

Le due classi che si occupano dell'interfaccia grafica sono admincontrol e clientview. Entrambe cono create dal rispettivo controller che a loro volta è chiamato a costruire o meno le interfacce in base ad un input dato da una classe Qdialog.

La view è stata curata per essere resa chiara ed immediata forzando a eseguire le operazioni in modo corretto mediante controlli dell'interfacccia; ad esempio non è possibile inviare la form di iscrizione di un nuovo utente senza averne compitati tutti i dati minimi necessari

ClientView è organizzata: dati personali: i dati anagrafici previsti, dati lavorativi: le competenze con i dettagli delle mansioni e le lingue conosciute, la modifica della rete di contatti (inserimento e rimozione di un contatto) e la ricerca di un contatto.
La modifica di una competenza si effettua semplicemente inserendo nuovamente la competenza nel form “inserisci competenza”, infatti le mansioni con uguale titolo effettuano l'operazione di modifica.

AdminView è organizzata dividendola in duè metà, a sinistra tutte le operazioni possibili da admin e a destra la tabella con il database che contiene le informazioni di user,cognome,nome e tipo di utente.
L'inserimento di un nuovo utente non mette la carta di credito che è possibile inserire tramite la cella del form modifica a condizione che si sia inserito il tipo corretto nella casella del cambio tipo.

In entrambi la selezione di un utente e fatta tramite clic con il mouse sulla riga della tabella che lo caratterizza questo attiverà i tasti legati a quella tabella permettendo le operazioni sulla username di quell'utente.
Le uniche operazioni che non lavorano trami questa logica click-pulsante attivato, sono:

1. il doppio click in una cella della tabella competenze nel client che restituisce la descrizione della competenza.
2. La ricerca effettuata dinamicamente ad ogni lettera inserita dall'utente.

LinQedIn

Prima di far partire il programma per usare il piccolo database già presente bisogna inserire i file database.xml e database2.xml nella cartella dove genera i file il compilatore(di default quindi carichera' il database all'esecuzione del programma).

1. Per far partire la modalità admin inserire la parola admin (controllata in modo case Insensitive) nella casella del dialog ed inviare.
   Le parole che cominciano con “admin” non possono essere il nome di nessun utente. Questo controllo è effettuato mediante l'uso di espressione regolare in modo case Insensitive.
   Il controllo è nella view, perché si vuole lasciare tutte le parole che iniziano per “Admin” come possibili parole speciali relative all'accesso al lato admin.
2. Per far partire la modalità client inserire l'utente:
   EduBic(Basic)
   Tullio(Business)
   Giuly(Executive)
   ed inviare. la username è controllata in modo case Sensitive.
3. Per chiudere la finestra di dialogo senza effettuare l'accesso digitare “close” ed inviare. Il controllo effettuato è case Insensitive. Anche in questo caso tutte le parole che iniziano per close non possono essere inseriti come nomi utente ed il controllo è fatto tramite espressine regolari.