

Progetto PAO-2014/2015

The logo for LinQedIn, featuring the word "Lin" in blue, a red "Q" with a small tail, and "edIn" in blue. The logo is centered within a light gray rectangular box with a subtle gradient and a thin black border.

LinQedIn


Progetto PAO-2014/2015

1 Scopo

LinkedIn è il principale servizio web di rete sociale per contatti professionali, gratuito ma con servizi opzionali a pagamento. Lo scopo del progetto è lo sviluppo in C++/Qt di un sistema minimale per l'amministrazione ed utilizzo tramite interfaccia utente grafica di un (piccolo) database di contatti professionali ispirato a LinkedIn.

2.1 Tipologie di account

LinkedIn prevede (attualmente) quattro tipologie di account:

Confronta i piani	Gratis <i>Il tuo piano attuale</i>	Business	Business Plus	Executive
Prezzi: Annuale Mensile  Risparmia fino al 25%		EUR 17,99/MESE EUR 22,13/MESE incluso IVA Fatturazione annuale	EUR 34,99/MESE EUR 43,04/MESE incluso IVA Fatturazione annuale	EUR 59,99/MESE EUR 73,79/MESE incluso IVA Fatturazione annuale
		Comincia subito	Comincia subito	Comincia subito
Visibilità				
Chi ha visitato il tuo profilo? Ottieni l'elenco completo e scopri come ti hanno trovato.	Limitato	✓	✓	✓
Profili completi Vedi i profili completi di tutti i collegamenti di 1°, 2° e 3° grado all'interno della tua rete.	Limite fino al 2° grado	✓	✓	✓
Visibilità completa del nome Visualizza i nomi completi dei collegamenti di 3° grado e dei membri del tuo gruppo.				✓
Comunicazione				
Messaggi InMail Invia messaggi diretti a qualsiasi utente su LinkedIn. Risposta garantita. ¹		3 al mese	10 al mese	25 al mese
Presentazioni Chiedi ai tuoi collegamenti di presentarti ai professionisti delle aziende che ti interessano.	5	15	25	35
Open Profile Consenti a chiunque su LinkedIn di visualizzare il tuo profilo completo e contattarti gratuitamente. ²		✓	✓	✓
Ricerca				
Ricerca Premium Utilizza fino a 8 filtri di ricerca avanzata. ³		4	4	8
Profili per ricerca Visualizza un maggior numero di profili quando effettui una ricerca.	100	300	500	700
Avvisi di ricerche salvate Salva le ricerche e ricevi avvisi quando nuovi profili corrispondono ai tuoi criteri di ricerca.	3 settimanalmente	5 settimanalmente	7 settimanalmente	10 quotidianamente
Ricerca referenze Ottieni un elenco di persone nella tua rete che può fornire una referenza per qualcuno che ti interessa.		✓	✓	✓

Per semplicità nel progetto consideriamo due tipologie di account a pagamento, Business ed Executive, oltre a quella gratuita Basic.

2.2 Utente LinkedIn

Un utente LinkedIn è caratterizzato almeno da:

1. Profilo (ad esempio: dati personali, esperienze professionali, competenze lavorative, lingue conosciute, titoli di studio, etc.).
2. Tipologia dell'account di iscrizione: Basic, Business o Executive.
3. Rete di contatti professionali con altri utenti LinkedIn.

Le diverse tipologie di account determinano quindi diversi tipi di utenti che devono essere modellati mediante una opportuna gerarchia di classi.

2.3 DB LinkedIn

Sarà quindi necessario modellare opportunamente il database (DB) degli utenti iscritti a LinkedIn. Si valutino le diverse possibilità di modellazione del DB e di memorizzazione degli utenti in tale DB, ad esempio mediante puntatori (smart) polimorfi ad una classe base di una gerarchia di classi di utenti.

2.4 LinkedIn lato amministratore

È richiesto lo sviluppo di una GUI per la gestione basilare del database (DB) degli utenti LinkedIn da parte dell'amministratore. Le funzionalità disponibili devono includere:

1. Inserimento nel DB di un nuovo utente LinkedIn; i dati per l'inserimento saranno nome, cognome, codice univoco dell'utente (ad esempio, l'email), tipologia di account (Basic, Business, Executive).
2. Ricerca nel DB di un utente mediante nome/cognome o codice utente.
3. Rimozione dal DB di un utente LinkedIn.
4. Cambio di tipologia di account (Basic, Business, Executive) per un utente LinkedIn.
5. Salvataggio/lettura su file del DB degli utenti LinkedIn.

2.5 LinkedIn lato utente

È richiesto lo sviluppo di una GUI per l'utilizzo dei servizi LinkedIn da parte di un utente già iscritto a LinkedIn. Le funzionalità disponibili devono includere:

1. Aggiornamento del proprio profilo.
2. Inserimento di un nuovo contatto nella propria rete.
3. Rimozione di un contatto dalla propria rete.
4. Funzionalità di ricerca sul DB LinkedIn, come autorizzate dalla propria tipologia di account. Le tre tipologie di account dovranno permettere delle funzionalità di ricerca crescenti: come semplice esempio non vincolante, Basic può permettere una ricerca che controlla solamente se nel DB LinkedIn esiste un utente con un dato nome e/o cognome; Business permette di ottenere da una ricerca il profilo completo di un utente; Executive permette inoltre di ottenere anche la rete dei contatti di un utente.

2.6 Interfaccia Grafica

Si richiede di sviluppare le GUI usando la libreria Qt. Naturalmente sarà possibile lo sviluppo di una unica GUI che permetta l'accesso al DB LinkedIn in due diverse modalità: amministratore ed utente. Si valuti l'opportunità di aderire al design pattern Model-View-Controller per la progettazione architetturale della/e GUI. Qt include un insieme di classi di “view” che usano una architettura “model/view” per gestire la relazione tra i dati logici della GUI ed il modo in cui essi sono presentati all'utente della GUI (si veda <http://qt-project.org/doc/qt-5/model-view-programming.html>).

Come noto, la libreria Qt è dotata di una documentazione completa e precisa che sarà la principale guida di riferimento nello sviluppo della GUI, oltre ad offrire l'IDE QtCreator ed il tool QtDesigner. La libreria Qt offre una moltitudine di classi e metodi per lo sviluppo di GUI curate, dettagliate e user-friendly.

Model–view–controller

From Wikipedia, the free encyclopedia

Model–view–controller (MVC) is a [software pattern](#) for implementing [user interfaces](#). It divides a given software application into three interconnected parts, so as to separate internal representations of information from the ways that information is presented to or accepted from the user.^{[1][2]} The central component, the *model*, consists of application data, business rules, logic, and functions. A *view* can be any output representation of information, such as a chart or a diagram. Multiple views of the same information are possible, such as a bar chart for management and a tabular view for accountants. The third part, the *controller*, accepts input and converts it to commands for the model or view.^[3]

Contents [\[hide\]](#)

- 1 Component interactions
- 2 Use in web applications
- 3 History
- 4 See also
- 5 References
- 6 External links

Model-View-Controller (MVC) [\[edit\]](#)

A pattern often used by applications that need the ability to maintain multiple views of the same data. The model-view-controller pattern was until recently a very common pattern especially for graphic user interlace programming, it splits the code in 3 pieces. The model, the view, and the controller.

The Model is the actual data representation (for example, Array vs Linked List) or other objects representing a database. The View is an interface to reading the model or a fat client GUI. The Controller provides the interface of changing or modifying the data, and then selecting the "Next Best View" (NBV).

Newcomers will probably see this "MVC" model as wasteful, mainly because you are working with many extra objects at runtime, when it seems like one giant object will do. But the secret to the MVC pattern is not writing the code, but in maintaining it, and allowing people to modify the code without changing much else. Also, keep in mind, that different developers have different strengths and weaknesses, so team building around MVC is easier. Imagine a View Team that is responsible for great views, a Model Team that knows a lot about data, and a Controller Team that see the big picture of application flow, handing requests, working with the model, and selecting the most appropriate next view for that client.

Model/View Programming

Introduction to Model/View Programming

Qt contains a set of item view classes that use a model/view architecture to manage the relationship between data and the way it is presented to the user. The separation of functionality introduced by this architecture gives developers greater flexibility to customize the presentation of items, and provides a standard model interface to allow a wide range of data sources to be used with existing item views. In this document, we give a brief introduction to the model/view paradigm, outline the concepts involved, and describe the architecture of the item view system. Each of the components in the architecture is explained, and examples are given that show how to use the classes provided.

The model/view architecture

Model-View-Controller (MVC) is a design pattern originating from Smalltalk that is often used when building user interfaces. In **Design Patterns**, Gamma et al. write:

MVC consists of three kinds of objects. The Model is the application object, the View is its screen presentation, and the Controller defines the way the user interface reacts to user input. Before MVC, user interface designs tended to lump these objects together. MVC decouples them to increase flexibility and reuse.

If the view and the controller objects are combined, the result is the model/view architecture. This still separates the way that data is stored from the way that it is presented to the user, but provides a simpler framework based on the same principles. This separation makes it possible to display the same data in several different views, and to implement new types of views, without changing the underlying data structures. To allow flexible handling of user input, we introduce the concept of the *delegate*. The advantage of having a delegate in this framework is that it allows the way items of data are rendered and edited to be customized.

Esempio concettuale di MVC

```
class StudentModel {
private:
    string matricola;
    string nome;
public:
    string getMatricola() const { return matricola; }
    void setMatricola(string m) { matricola = m; }

    string getNome() const { return nome; }
    void setNome(string n) { nome = n; }
};
```

```
class StudentView {
private:
    ostream os;
public:
    StudentView(): os(std::cout) {}
    // metodo di istanza con una view di invocazione
    void printStudentDetails(StudentModel s) const {
        os << "STUDENTE:" << std::endl;
        os << "Nome: " << s.getNome() << std::endl;
        os << "Matricola: " << s.getMatricola() << std::endl;
    }
};
```

```
class StudentController {
private:
    StudentModel model;
    StudentView view;
public:
    StudentController(StudentModel m, StudentView v): model(m), view(v) {}

    void setNomeStudente(string n){ model.setNome(n); }
    string getNomeStudente() const { return model.getNome(); }

    void setMatricolaStudente(string m){ model.setMatricola(m); }
    string getMatricolaStudente() const { return model.getMatricola(); }

    void updateView() const { view.printStudentDetails(model); }
};
```

Esempio concettuale di MVC

```
class MVCPatternDemo {
private:
    static StudentModel retrieveStudentFromDB(string m) {
        StudentModel student;
        student.setMatricola(m);
        student.setNome("Roberto");
        return student;
    }

public:
    MVCPatternDemo() {
        // Crea un model recuperando uno studente dal DB
        StudentModel model = retrieveStudentFromDB("999");
        // Crea una view per l'output su cout dei dettagli dello studente
        StudentView view;
        // Crea un controller su model e view
        StudentController controller(model, view);

        controller.updateView();

        controller.setNomeStudente("Paolo");
        controller.updateView();
    }
};

int main() {
    MVCPatternDemo app;
}
```

3 Valutazione del Progetto

Un buon progetto dovrà essere sviluppato seguendo i principi fondamentali della programmazione orientata agli oggetti, anche per quanto concerne lo sviluppo dell'interfaccia grafica. La valutazione del progetto prenderà in considerazione i seguenti criteri:

1. **Correttezza:** il progetto deve compilare e funzionare correttamente, e raggiungere correttamente e pienamente gli scopi previsti.
2. **Orientazione agli oggetti:** (A) progettazione ad oggetti, (B) modularità (in particolare, massima separazione tra il codice logico del progetto ed il codice della GUI del progetto), (C) estensibilità e (D) qualità del codice sviluppato.
3. **Quantità e qualità:** quante e quali funzionalità il progetto rende disponibili, e la loro qualità.
4. **GUI:** utilizzo corretto della libreria Qt, qualità ed usabilità della GUI.

4 Esame Orale e Registrazione Voto

La partecipazione all'esame orale è possibile solo dopo:

1. avere superato con successo (cioè, con voto $\geq 18/30$) l'esame scritto
2. avere consegnato il progetto entro la scadenza stabilita
3. essersi iscritti alla lista Uniweb dell'esame orale

Il giorno dell'esame orale (nel luogo ed all'orario stabiliti) verrà comunicato l'esito della valutazione del progetto (non vi saranno altre modalità di comunicazione della valutazione del progetto). Tre esiti saranno possibili:

- (A) Valutazione positiva del progetto con registrazione del voto complessivo proposto **con esenzione dell'esame orale**. Nel caso in cui il voto proposto non sia ritenuto soddisfacente dallo studente, sarà possibile richiedere l'esame orale, che potrà portare a variazioni in positivo o negativo del voto proposto.
- (B) Valutazione del progetto da completarsi con un **esame orale obbligatorio**. Al termine dell'esame orale, o verrà proposto un voto complessivo sufficiente oppure si dovrà riconsegnare il progetto per un successivo esame orale.
- (C) Valutazione negativa del progetto che comporta quindi la **riconsegna del progetto** per un successivo esame orale (il voto dell'esame scritto rimane valido).

Si ricorda inoltre che all'eventuale esame orale lo studente dovrà saper motivare **ogni** scelta progettuale e dovrà dimostrare la **piena conoscenza** di ogni parte del progetto.

5.5 Scadenze di consegna

Il progetto dovrà essere consegnato rispettando **tassativamente** le scadenze **ufficiali** (data e ora) previste che verranno rese note tramite le liste Uniweb di iscrizione agli esami scritti ed orali e tramite il gruppo Facebook del corso <https://www.facebook.com/groups/pa014.15>. Approssimativamente la scadenza sarà circa 8-10 giorni prima dell'esame orale.

Per i progetti ritenuti insufficienti, lo studente dovrà consegnare una nuova versione del progetto per un successivo appello orale.

Prima sessione regolare di esami orali: Le date degli esami orali della sessione regolare con relative scadenze tassative di consegna del progetto sono le seguenti:

Primo orale: lunedì 9 febbraio 2015, scadenza di consegna: domenica 1 febbraio 2015 ore 23:59

Secondo orale: venerdì 27 febbraio 2015, scadenza di consegna: mercoledì 18 febbraio 2015 ore 23:59

Scheletro logico

```
class Info {};  
  
class Profilo {  
public:  
    void modificaProfilo() {}  
    Info* visualizzaProfilo() const {return 0;}  
};  
  
class Rete;  
  
class Username {  
public:  
    string login;  
    Username(string s): login(s) {}  
};  
  
// base polimorfa  
class Utente {  
public:  
    Profilo pf;  
    Rete* rete;  
    Username un;  
    Utente(Username u): un(u) {}  
    virtual ~Utente() {}  
};
```

```
// possibile modellazione
class UtenteBasic: public Utente {};
class UtenteBusiness: public Utente {};
class UtenteExecutive: public Utente {};
// meglio una diversa relazione gerarchica?

class SmartUtente {
public:
    Utente* u; // puntatore polimorfo
};

class Rete {
public:
    vector<SmartUtente> net; // vector ok?
    void add(Username u) {}
    void remove(Username u) {}
};
```

```

// il database di utenti LinQedIn
class DB {
public:
    vector<SmartUtente> db; // vector ok?
    void load() {}          // carica da file
    void save() const {}    // salva su file
    // cerca l'utente (con il suo tipo)
    // avente Username u nel DB,
    Utente* find(Username u) {
        return new Utente(u);
    }
};

// lato amministratore
class LinQedInAdmin {
public:
    DB* db; // inizializzato da file,
            // gestito in memoria,
            // salvabile su file

    LinQedInAdmin() {
        // carica da file
        db->load();
    }
    void insert() {}
    void find(Username u) const {}
    void remove(Username u) {}
    // cambia tra Basic, Business, Executive
    void changeSubscriptionType(Username u) {}
    void save() const { db->save(); }
};

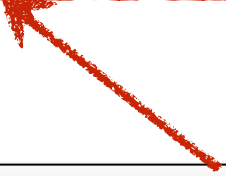
```

```
// lato client
class LinkedInClient {
public:
    Utente* u;
    DB* db; // inizializzato da file,
           // usato in sola lettura
    LinkedInClient(Uusername s) {
        // carica il DB
        db->load();
        // cerca l'utente client con Username s
        u = db->find(s);
    }
    void aggiornaProfilo() {
        (u->pf).modificaProfilo();
    }
    void showProfilo() {
        (u->pf).visualizzaProfilo();
    }
    void insertRete(Uusername un) {
        u->rete->add(un);
    }
    void removeRete(Uusername un) {
        u->rete->remove(un);
    }
};
```

2.5 LinkedIn lato utente

È richiesto lo sviluppo di una GUI per l'utilizzo dei servizi LinkedIn da parte di un utente già iscritto a LinkedIn. Le funzionalità disponibili devono includere:

1. Aggiornamento del proprio profilo.
2. Inserimento di un nuovo contatto nella propria rete.
3. Rimozione di un contatto dalla propria rete.
4. Funzionalità di ricerca sul DB LinkedIn, come autorizzate dalla propria tipologia di account. Le tre tipologie di account dovranno permettere delle funzionalità di ricerca crescenti: come semplice esempio non vincolante, Basic può permettere una ricerca che controlla solamente se nel DB LinkedIn esiste un utente con un dato nome e/o cognome; Business permette di ottenere da una ricerca il profilo completo di un utente; Executive permette inoltre di ottenere anche la rete dei contatti di un utente.



Come implementare le funzionalità di ricerca autorizzate dalla propria tipologia di account?

Funtori

Un funtore e' un oggetto di una classe che puo' essere trattato come fosse una funzione (o un puntatore a funzione):

```
FunctorClass fun;  
fun(1,4,5);
```

E' possibile fare cio' mediante l'overloading di operator(), l'operatore di "chiamata di funzione": puo' avere un qualsiasi numero di parametri di qualsiasi tipo e ritornare qualsiasi tipo. Quando si invoca operator() su un oggetto, si puo' quindi pensare di "invocare" quel funtore.

```
class FunctorClass {  
private:  
    int x;  
public:  
    FunctorClass(int n): x(n) {}  
    int operator() (int y) const {return x+y;}  
};  
  
int main() {  
    FunctorClass sommaCinque(5);  
    cout << sommaCinque(6); // stampa 11  
}
```

Funtori

```
class MoltiplicaPer {
private:
    int factor;
public:
    MoltiplicaPer(int x): factor(x) {}
    int operator() (int y) const {return factor*y;}
};

int main() {
    vector<int> v;
    v.push_back(1); v.push_back(2); v.push_back(3);
    std::transform(v.begin(), v.end(), v.begin(), MoltiplicaPer(2));
    cout << v[0] << " " << v[1] << " " << v[2]; // stampa 2 4 6
}
```

```
template <class InputIterator, class OutputIterator, class UnaryOperation>
OutputIterator transform (InputIterator first, InputIterator last,
                        OutputIterator result, UnaryOperation op);

/*
Applica op ad ogni elemento in [first,last) and memorizza il valore ritornato da ogni
applicazione di op nel segmento di contenitore che inizia da result.
Equivalente a:
*/

template <class InputIterator, class OutputIterator, class UnaryOperator>
OutputIterator transform (InputIterator first, InputIterator last,
                        OutputIterator result, UnaryOperator op) {
    while (first != last) {
        *result = op(*first);
        ++result; ++first;
    }
    return result;
}
```


Funtori

```
class UgualA {
private:
    int number;
public:
    UgualA(int n): number(n) {}
    bool operator() (int x) const {return x==number;}
};

// template di funzione, con parametro di tipo "funtore"
template<class Functor>
vector<int> find_matching(const vector<int>& v, Functor pred) {
    vector<int> ret;
    for(vector<int>::const_iterator it = v.begin(); it<v.end(); ++it)
        if( pred(*it) ) ret.push_back(*it); // deve essere disponibile bool operator() (int)
    return ret;
}

int main() {
    vector<int> w;
    w.push_back(1); v.push_back(5); v.push_back(1); v.push_back(3);
    vector<int> r = find_matching(w, UgualA(1));
    for(int i=0; i<r.size(); ++i) cout << r[i] << " ";
    // stampa 1 1
};
```