Programmazione ad oggetti
Progetto "Semilibertá" – FileShare

Relazione di **Cailotto Mirco**
Matricola **1123521**
Universitá di Padova

Anno 2016/2017

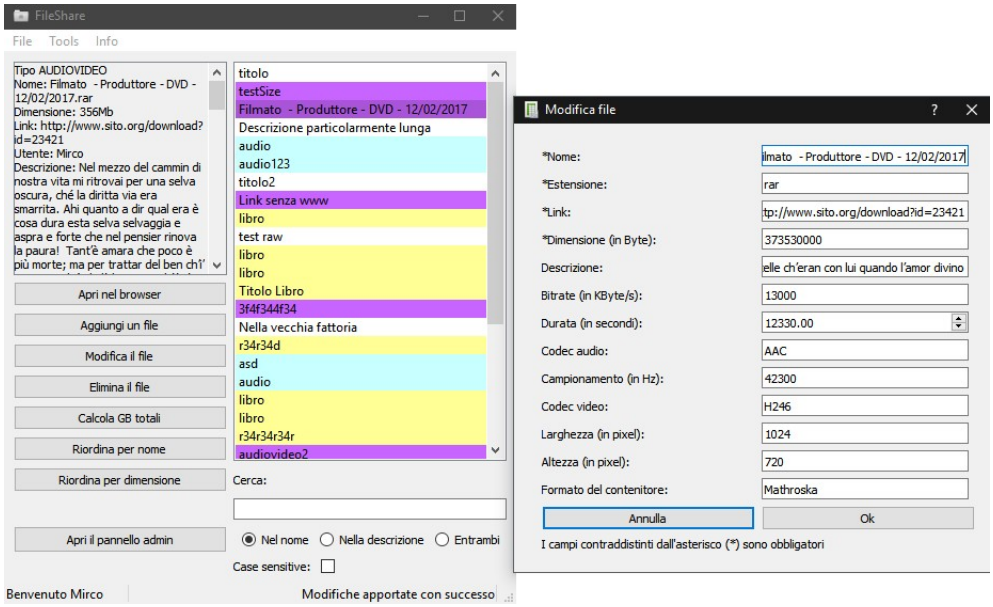

Figure 1: Anteprima del programma

Contents

1 Scopo del progetto

Il progetto si prefigge lo scopo di creare una applicazione per condividere dei link riguardanti file presenti sulla rete internet, concedendo vari livelli di permessi agli utenti e la possibilità di gestire i collegamenti in relazione ai permessi concessi dagli amministratori.
I file possono essere di vario tipo, ad esempio generici oppure multimediali o testuali, con tutti i relativi sottotipi, che sono audio, video, audiovideo, epub o testi semplici.
Con file video si intende un filmato senza la presenza di audio, poco usato se non in particolari ambiti, ad esempio i file .webm sui social network.
La possibilità della creazione degli utenti è riservata agli amministratori, in quanto si desidera impedire agli utenti non registrati la possibilità di vedere i dati.
Si vuole anche fornire la possibilità di effettuare ricerche in tempo reale, filtrando nel nome, nella descrizione o in entrambe, con o senza case sensitive.
I file possono anche essere riordinati per nome o per dimensione, in modo ascendente o discendente.
Ci sono in totale quattro classi di utenza presenti: gli amministratori che hanno il controllo completo, i moderatori che possono compiere qualsiasi azione riguardante i file, gli editori che hanno il compito di aggiungere nuovi file e gli utenti standard, che possono solamente consultare l'elenco dei file.

2 Descrizione delle gerarchie di tipi usate

Il progetto presenta tre gerarchie, due nella parte logica che rappresentano i file e gli utenti e una nella parte grafica, che permettono di costruire la finestra per l'inserimento e la modifica dei file.
Per quanto riguardano le gerarchie nella parte logica si è voluto utilizzare il più possibile la STL, in modo da non dover dipendere dal framework QT.

2.1 Gerarchia dei file

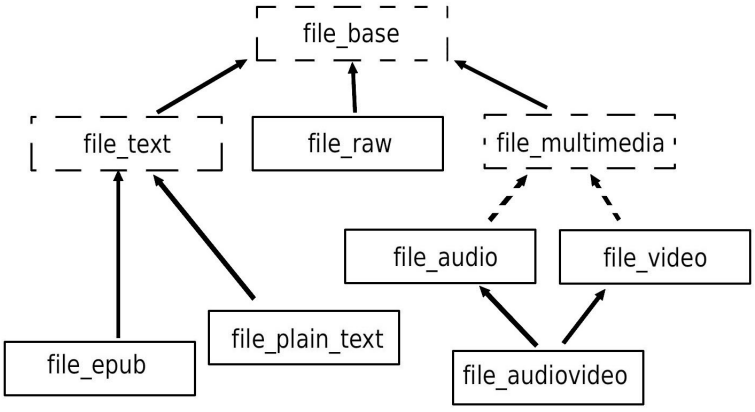

Figure 2: Gerarchia dei file

La prima gerarchia costituisce l'insieme dei file raccolti nel contenitore C richiesto nei vincoli obbligatori del progetto e rappresenta dei file presenti su determinati siti web.
Si parte da un generico file di base astratto, con campi nome, estensione, link, descrizione e il nome dell'utente che ha inserito il file.
Il file di base viene concretizzato nel file raw, "grezzo", che non aggiunge campi rispetto alla classe base e indica un file qualsiasi, senza nessuna informazione aggiuntiva.
Il file di testo invece rappresenta un file testuale, che quindi possiede anche i campi autore e titolo dell'opera, concretizzato senza ulteriori informazioni in file di testo semplice (file_plain_text).
Dal file di testo deriva anche il file epub, che rappresenta un libro in formato digitale, molto diffuso sigli e-reader, e opzionalmente può tenere traccia del numero di immagini contenute.
file multimedia invece riguarda i file multimediali, caratterizzati da un bitrate e da una durata, che vengono concretizzati in file audio e file video, con ereditarietà virtuale.
Nel caso del file audio vengono anche mantenute informazioni riguardanti il codec audio utilizzato e il campionamento, mentre i file video presentano il codec video utilizzato e la risoluzione.
La gerarchia viene chiusa dal file audiovideo, che rappresentano i filmati compresi di audio, e aggiunge un nuovo campo per mantenere traccia del codec utilizzato per contenere i due flussi multimediali.

Ogni classe presenta metodo virtuale `getInfo()` che restituisce una stringa contenente tutte le informazioni riguardanti il file.
La gerarchia presenta anche una funzione virtuale `getColor()` pubblica che restituisce una struct indicante un colore, che serve a dare a ogni tipo di file un colore simbolico che può essere usato in più aspetti dell'interfaccia.
Il colore viene rappresentato mediante una struct e non una classe in quanto si ha la necessità di information hiding o di ereditarietà e si é scelto di non utilizzare QColor per mantenere la classe il più possibile indipendente dal framework QT.
Ogni classe, per facilitare l'esportazione in XML, presenta la funzione virtuale exportMyXmlData, che scrive sul QxmlStreamWriter indicato tutti i campi dati che contiene.

2.2 Gerarchia degli utenti

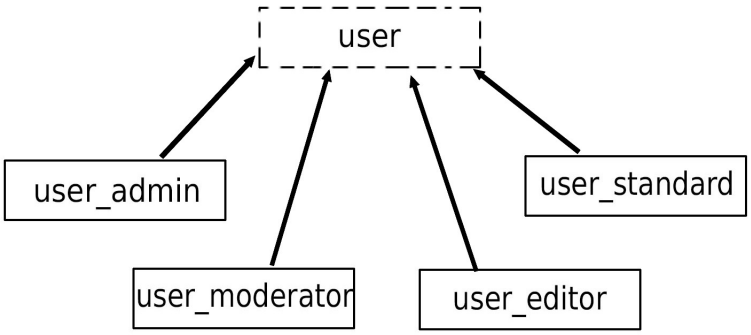

Figure 3: Gerarchia degli utenti

La gerarchia utenti consiste in una classe astratta user, resa astratta dai metodi di

clonazione e di controllo dei permessi, dalla quali derivano quattro classi concrete.
I metodi che consentono l'esportazione delle informazioni su file sono exportXml e getLabel, la prima salva le informazioni su file mentre la seconda, virtuale, fornisce per ogni casse l'informazione sul tipo.
Per quanto riguarda i permessi si è scelto di non mantenerli all'interno di variabili, in quanto avere una copia dei permessi per ogni utente e non per ogni tipologia di utenti sarebbe stato uno spreco di memoria in quanto comuni.
Una possibile alternativa era utilizzare dei campi statici, ma così facendo le funzioni sarebbero state identiche in tutte le classi di utenti ed inutilmente virtuali, mentre nella base dovevano rimanere virtuali pure.
Ulteriormente, grazie a questa gestione dei permessi, in caso di estensione della gerarchia si potrebbero creare nuove tipologie di utenti con permessi dinamici, modificabili nel tempo.

2.3 Gerarchia delle view per l'aggiunta e modifica

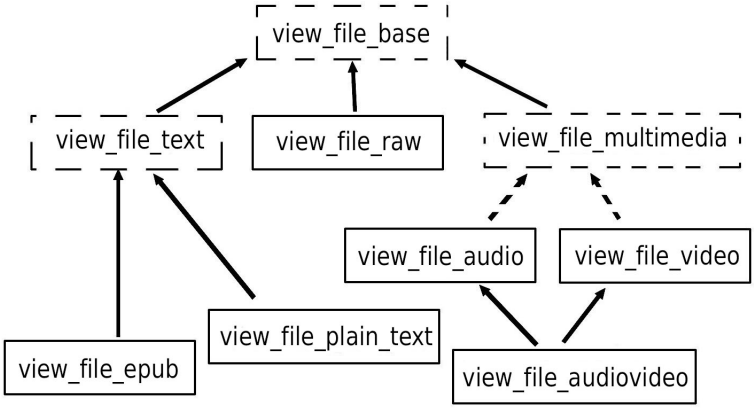

Figure 4: Gerarchia delle view

Parallelamente alla gerarchia di file si sviluppa anche una gerarchia di view, derivate da QWidget, che hanno il compito di fornire una GUI finalizzata ad aggiungere o modificare i file.
Oltre a creare l'interfaccia la gerarchia è anche responsabile di controllare che i campi inseriti siano validi e, nel caso l'utente accetti le modifiche o l'inserimento, modificare o creare il file in esame.

2.4 Classi contenitori

Le classi contenitori costruiscono una piccola gerarchia, anche se non forniscono un vero polimorfismo.
La classe base è mylist, classe templetizzata a due parametri, il primo indica il tipo di contenitore utilizzato, specificando anche il contenuto, ed il secondo il contenuto.
Vengono forniti iteratori sia costanti che con possibile side-effect, i quali utilizzano gli

iteratori presenti nel contenitore che incapsulano.

Gli utenti sono raccolti utilizzando la classe listUser, che istanza il template con una std::list di user*, mentre i file sono mantenuti in listFile, che invece utilizza un std::vector di file_base*
ListFile, chiamata semplicemente list intendendola come lista nel linguaggio colloquiale, cioè una serie di informazioni, ignorandone l'implementazione, presenta un std::vector in quanto l'utilizzo principale sarà la consultazione, quindi il tempo di accesso sarà costante e minore rispetto ad una std::list.
Ulteriormente listFile presenta anche funzioni di riordino della lista e di accesso casuale, mentre listUser presenta funzioni di ricerca degli utenti e di gestione della password.

3 Descrizione dell'uso di codice polimorfo

3.1 Utilizzo polimorfico della gerarchia degli utenti

Innanzitutto l'uso polimorfico della classe user, per non creare memory leak il distruttore è stato reso virtuale, in modo che quando si distrugge un puntatore a user viene richiamato il distruttore corretto.
Oltre a ciò user presenta anche la funzione super polimorfa `user* clone()`, che crea una copia dell'oggetto e ne restituisce un puntatore, utilizzato nel file `mylist.h` alla riga 124.
Altre funzioni che fanno uso del polimorfismo nella gerarchia delle classi user (in questo caso super polimorfiche) sono `bool isAdmin()`, `bool canItemAdd()`, `bool canItemEdit()`, `bool canItemDelete()`, che ritornano le informazioni riguardo ai permessi di una classe di utenza, questi metodi sono utilizzati nella parte grafica per disabilitare o abilitare funzionalitá, ad esempio in `main_widget.cpp` righe 13, 42, 143 e 151.

3.2 Utilizzo polimorfico della gerarchia dei file

Per quanto riguarda la gerarchia dei file, esattamente come per la classe user, il distruttore è virtuale e presenta il metodo `file_proj* clone()` utilizzato sempre in `mylist.h`.
Un altro metodo polimorfico presente è `string getInfo()`, che restituisce una stringa contenente tutte le informazioni relative alla classe, utilizzato nello stesso metodo in modo non polimorfico, mentre viene usato in modo polimorfico nel file `main_widget.cpp` riga 121.
Altri metodi che usano il polimorfismo sono `string getLabel()` e `color_file getColor()`, il primo restituisce un identificatore dell'oggetto e il secondo un colore caratteristico, utilizzati rispettivamente in `user.cpp` riga 21 all'interno di `exportXml()` e nel file `main_widget.cpp` riga 83.
L'ultimo metodo che sfrutta il polimorfismo è `void exportMyXmlData(QXmlStreamWriter&)`, questo metodo è utilizzato in modo polimorfo in `file_base.cpp`, riga 41.
Il metodo `exportXml()` non necessita di essere virtuale, in quanto il suo comportamento non cambia, in quanto sfrutta due chiamate polimorfiche, `getLabel()` e `exportMyXmlData()`.

3.3 Utilizzo polimorfico della gerarchia delle viste

La gerarchia delle viste presenta innanzitutto il distruttore virtuale in modo da evitare memory leak, esattamente come le altre gerarchie precedentemente analizzate.
Ulteriormente presenta tre funzioni virtuali: `bool check()`, `void edit()` e `void build_field()`.
`Bool check()` serve a controllare la validità dei dati inseriti, non viene mai effettuato un override nel progetto ma per permettere l'estensibilità deve essere virtuale, in quanto potrebbero essere necessari dei controlli su nuovi campi in caso di aggiunta di nuove classi alle gerarchie, viene utilizzato nel file `view_base_file.cpp` riga 95.
`Void edit()` modifica l'oggetto file corrente, riportando le modifiche effettuate nei campi di inserimento da parte dell'utente nell'oggetto, viene utilizzato nel file `view_base_file.cpp` riga 97.
`Void build_field()` aggiunge alla view i campi necessari per effettuare modifiche all'oggetto, viene utilizzato nel file `view_base_file.cpp` riga 2.

4 Manuale utente

L'applicazione fa uso di due file presenti nella directory del file eseguibile per salvare le informazioni, userdata.xml per gli utenti e filedata.xml per i link.
Non è presente la necessità che questi file siano presenti, in tal caso verrà creato un utente admin di default con nome **admin** e password **admin**.
L'interfaccia grafica è ridimensionabile per consentire una agevolazione nella lettura e nell'inserimento dei campi di testo, per aprire un link si può anche effettuare un doppio click sul file interessato.
L'interfaccia di amministrazione è accessibile attraverso il pulsante "Apri il pannello admin", che appare a sinistra della tabella di ricerca nel caso l'utente abbia i permessi necessari.
Nel caso si desideri provare l'applicazione con dei dati già inseriti basta spostare i due file presenti nella cartella esempi nella locazione del file eseguibile.
Per impedire ad utilizzatori sprovvisti di account la consultazione dei file condivisi si è scelto di precludere la possibilità ad un utente non registrato di creare un account standard, lasciando agli amministratori l'onere di gestire gli utenti, compreso il loro inserimento.
Il riordinamento viene effettuato in modo ascendente per quanto riguarda il nome dei file e discendente per la dimensione, per capovolgere l'ordine basta effettuare un secondo click sullo stesso pulsante.
Nella seguente tabella sono riportati i permessi delle tipologie di utenti:

Tipo di utente	gestione utenti	eliminazione	modifica	aggiunta	consultazione
amministratore	✓	✓	✓	✓	✓
moderatore	✗	✓	✓	✓	✓
editore	✗	solo se di sua proprietá	solo se di sua proprietá	✓	✓
utente standard	✗	✗	✗	✗	✓

5 Ore utilizzate

progettazione: 5h
progettazione grafica (inclusa la scelta dei layout): 4h
scrittura codice: 12h
scrittura gui: 36h
test: 3h
debug (compresi test per il memory leak): 2h

6 Ambiente di sviluppo

Sistema operativo: Microsoft Windows 10 Education 64-bit 10.0.10586
Compilatore: g++ (tdm64-1) 4.9.2
QT: 5.7.0
Qmake: 3.0