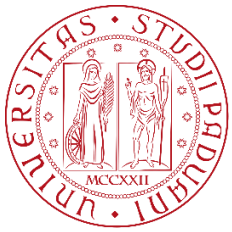
Relazione progetto

Alessandro Poloni *matricola 1224444*

Luca Polese *matricola 1225425*

Corso di Programmazione ad Oggetti

Anno accademico 2020-2021



1. SCOPO DEL PROGETTO

Negli ultimi anni sono comparsi sul mercato molti servizi di cloud storage, la maggior parte dei quali offre spazi di archiviazione gratuiti dalle dimensioni non indifferenti. Da ciò nasce l'idea dell'applicazione QtDrive, un programma dall'interfaccia semplice ed intuitiva che permetta ai suoi utenti di gestire i propri account di servizi cloud ed i relativi file personali, attraverso un unico software.

L'applicazione, dunque, consente all'utente di memorizzare le credenziali di accesso a tali servizi, visualizzare informazioni importanti, come lo spazio ancora disponibile e la distribuzione dei file all'interno dei singoli account, inserire diversi tipi di file all'interno di essi, visualizzare e modificare le informazioni dei file già presenti.

Molto spesso risulta difficile tenere traccia dei propri file importanti, soprattutto se suddivisi all'interno di piattaforme diverse; pertanto, QtDrive offre anche funzionalità di ricerca che coinvolgono l'insieme di tutti i file distribuiti nei diversi account.

L'applicazione permette inoltre il salvataggio di dati e la loro importazione tramite opportuni file in formato XML generabili dalla stessa.

1.1 Funzionalità

Funzionalità più rilevanti dell'applicazione sono dunque le seguenti:

- Apertura, chiusura e salvataggio di un profilo utente, tramite file in formato XML
- Visualizzazione e modifica delle informazioni relative ad un account di servizio di cloud storage
- Visualizzazione e modifica delle informazioni relative ai file presenti all'interno dei suddetti account
- Ricerca di file all'interno degli account, tramite nome, descrizione o informazioni dipendenti dal tipo di file

1.2 Premesse

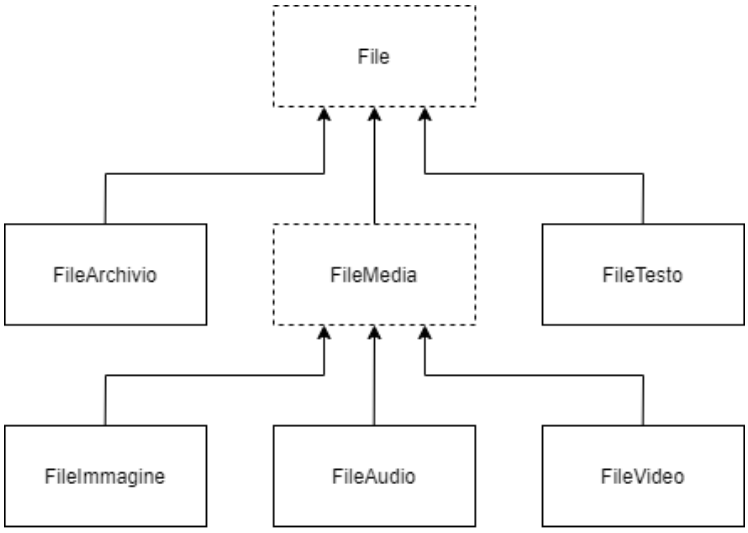
Idea alla base dell'applicazione è che in futuro risulti possibile implementare opportune API che consentano un recupero delle informazioni degli account e dei file in modo automatico dai server dei diversi host.

Data la natura del progetto, tuttavia, il controllo di tali elementi risulta essere completamente manuale e gestito tramite file salvati localmente. Si suppone inoltre che la gestione dei dati trattati sia la medesima per ogni servizio di cloud.

2. PROGETTAZIONE

Per lo sviluppo del programma, si è deciso di adottare il pattern Model-View-Controller, al fine di garantire una separazione tra modello logico e interfaccia grafica.

2.1 Gerarchia File



Un bordo tratteggiato indica che la classe non è istanziabile.

Base della gerarchia è la classe astratta File, che implementa alcuni metodi virtuali, in particolare:

- Il metodo distruttore ed il metodo `clone() const`, per una gestione profonda della memoria
- Il metodo `getInformazioniFile() const`, che restituisce le informazioni specifiche al tipo di file
- I metodi `serializza(QXmlStreamWriter) const`, `deserializza(QXmlStreamReader)` e `getTipoFile()` const per la lettura e scrittura di file XML
- Il metodo `ricercaAvanzata(QString, Qt::CaseSensitivity)`, per effettuare operazioni di ricerca che tengano conto anche del tipo di file specifico.

Questa classe è caratterizzata da alcuni campi dati comuni a tutte le classi che da essa derivano:

- `nome` ed `estensione` del file, che contraddistinguono univocamente un file all'interno di un account
- `dimensione` del file, gestita in megabyte
- `dataCreazione` e `dataInserimento` del file all'interno del cloud
- una `descrizione` facoltativa, inseribile dall'utente

La classe **FileArchivio** viene utilizzata per gestire le informazioni di file di tipo archivio (.zip, .rar, …), pertanto differisce da File per tre campi dati: `dimensioneOriginale`, che indica la dimensione non compressa del file stesso; `numeroDiFile`, ovvero il numero di file compressi al suo interno e `protetto`, variabile booleana che indica se l'archivio è protetto da password

La classe **FileTesto** gestisce le informazioni di file testuali (.txt, …) e perciò introduce i campi dati `numeroCaratteri` e `numeroParole`.

La classe **FileMedia** è un'altra classe astratta che introduce il concetto di file multimediale. Unico campo dati di questa classe è `tipoCompressione` che modella appunto il tipo di compressione del file, mediante una enumerazione. Tale concetto accomuna le tre classi che derivano da essa.

La classe **FileImmagine** gestisce le informazioni di file di tipo immagine (.png, .jpeg, …) attraverso i campi dati `larghezza` e `altezza`, che ne definiscono la risoluzione, e l'enumerazione `tipo` che gestisce la natura dell'immagine (raster o vettoriale).

La classe **FileAudio** introduce i campi dati `bitrate` e `durata`, che contraddistinguono file di tipo audio (.mp3, .flac).

Infine, **FileVideo** modella il concetto di filmato, tramite i campi `larghezza` e `altezza` che, come per i file immagine, ne definiscono la risoluzione, la `durata` dello stesso, il `codec` e gli `FPS`, ovvero i fotogrammi al secondo.

Tutte queste classi, con un'ovvia eccezione per FileMedia, forniscono un'implementazione dei metodi dichiarati virtuali nella classe base File.

La gerarchia è stata pensata per risultare eventualmente estensibile mediante polimorfismo, al fine di garantire il supporto ad altri tipi di file. Esempio di possibile evoluzione può essere una ipotetica classe Canzone, che deriva da FileAudio ed introduce nuovi campi dati, quali autore e genere. Altro esempio di evoluzione coinvolge invece la classe FileTesto, dalla quale potrebbero ereditare classi per la modellazione di file di testo più specifici, come documenti PDF od ePub.

2.2 Account

Classe fondamentale all'interno del programma., che tuttavia non fa parte di una gerachia, è **Account**, che modella le informazioni di un account di un servizio di cloud. Tali informazioni sono costituite dai campi dati `email`, `password`, `spazioFornito` e l'enumerazione `host`, che specifica il provider del servizio (Amazon, GDrive, …). Ogni oggetto di classe Account ha inoltre tra i suoi campi dati `listaFile`, container di deep-pointers ad oggetti di tipo File, che costituisce l'insieme dei file associati a quell'account.

2.3 Container e Deepptr

Come struttura dati per il progetto, si è ritenuto opportuno sviluppare un template Container<T> come lista doppiamente linkata, che prevede tutti i metodi e gli operatori solitamente forniti da una struttura di questo tipo, come push_back e push_front, erase e clear, e operatori di dereferenziazione e subscripting, oltre agli iteratori, anche costanti, con i loro metodi di accesso.

All'utilizzo di questo Container si affianca quello dei Deepptr, che garantiscono una gestione della memoria in modalità profonda.

2.4 View

Tutte le classi utilizzate per implementare la GUI del programma ereditano pubblicamente dalla classe `QWidget`, al fine di poter utilizzare pratici metodi per la disposizione su schermo degli elementi.

3. POLIMORFISMO

3.1 Metodo distruttore e metodo clone() const

Il metodo distruttore della gerarchia File ed il metodo `clone() const` sono resi virtuali al fine di evitare memory leak e garantire una corretta gestione della memoria.

3.2 Metodo getInformazioniFile() const

Il metodo getInformazioniFile() const della gerarchia File restituisce una stringa contenente tutte le informazioni specifiche al tipo di file, pertanto è stato reso virtuale.

3.3 Metodi serializza(QXmlStreamWriter) const e deserializza(QXmlStreamReader)

Questi due metodi della gerarchia File consentono rispettivamente di salvare su un file e di leggere da un file le informazioni che compongono un oggetto di un qualunque sottotipo della gerarchia. Il polimorfismo risulta quindi fondamentale al fine di leggere e scrivere tutti i campi che lo compongono.

Questi metodi vengono utilizzati all'interno dei metodi dallo stesso nome della classe Account: per ogni file contenuto nell'account, viene quindi effettuata una chiamata polimorfa.

3.4 Metodo ricercaAvanzata(QString, Qt::CaseSensitivity)

Questo metodo viene utilizzato per effettuare la ricerca di una stringa indicata dall'utente all'interno delle informazioni specifiche di un file. Il metodo pertanto richiama la funzione `getInformazioniFile() const`, anch'esso polimorfo e all'interno della stringa restituita effettua la ricerca.

3.5 Metodo getTipoFile() const

Questo metodo virtuale della gerarchia File viene utilizzato all'interno dei metodi per la lettura e la scrittura dei file XML supportati dal programma e restituisce il tipo dell'oggetto di invocazione sottoforma di stringa, evitando quindi il type checking dinamico.

3.6 Template di funzione ContaFile() const

Questo metodo della classe Account viene utilizzato per calcolare il numero di file di una determinata classe, che costituisce il parametro di tipo, presenti all'interno di un account. Per fare ciò si scorre il container di deep-pointers a File dell'oggetto Account di invocazione e, per ogni puntatore, si effettua un controllo sul tipo dinamico tramite **dynamic_cast**.

3.7 Strutture dati utilizzate

La struttura dati principale del programma è il campo dati `Container<Deepptr<Account>> listaAccount` della classe Controller.

Ogni Account a sua volta possiede il campo dati `Container<Deepptr<File>> listaFile`, che consente una gestione profonda della memoria, utilizzando i metodi virtuali descritti in precedenza.

4. FUNZIONALITÀ DI INPUT/OUTPUT

L'applicazione consente il salvataggio di dati e la loro visualizzazione tramite file in formato XML. Al loro interno, questi file presentano diversi elementi:

- `file`, contiene le informazioni di un singolo file inserito dall'utente
- `listaFile`, contiene più elementi file, rappresentando quindi la lista dei file presenti all'interno di un account
- `account`, contiene le informazioni di accesso ad un servizio e un elemento listaFile
- `listaAccount`, elemento radice del file, contiene più elementi account

Un opportuno file in questo formato può essere aperto nel programma selezionando dal menù le voci **Profilo > Apri**.

Insieme al codice del programma, è incluso un file di esempio già contenente al suo interno alcune informazioni.

5. MANUALE UTENTE

Un manuale utente completo è visionabile all'interno della applicazione, selezionando dal menù le voci **Aiuto > Guida** oppure premendo la combinazione di tasti **Ctrl+H**.

6. ISTRUZIONI DI COMPILAZIONE ED ESECUZIONE

Per compilare correttamente il programma è necessario il file `progetto.pro`, incluso nella consegna.

I comandi da eseguire sono i seguenti:

```
qmake progetto.pro
```

```
make progetto
```

A questo punto, il programma risulta eseguibile tramite il comando:

```
./progetto
```

7. TEMPISTICHE

- Analisi preliminare del problema: 4 ore
- Progettazione modello: 3 ore
- Progettazione GUI: 5 ore
- Apprendimento libreria Qt: 4 ore
- Codifica modello: 12 ore

- Codifica GUI: 16 ore
- Debugging: 5 ore
- Testing: 6 ore

In totale il progetto ha richiesto circa 55 ore di lavoro individuale, 5 ore in più rispetto a quelle previste. Motivo di tale sforamento è dovuto principalmente allo sviluppo della parte grafica dell'applicazione, più complesso di quanto previsto, ed al collaudo della stessa.

Infine, per quanto riguarda l'apprendimento della libreria, non si è tenuto conto delle ore di tutorato frequentate e della consultazione della documentazione di Qt in corso d'opera.

8. SUDDIVISIONE DEL LAVORO PROGETTUALE

Classi a cui ha lavorato individualmente lo studente Alessandro Poloni:

- File
- AccountWidget
- GuidaWidget
- InfoFileWidget
- ModificaAccountWidget
- NuovoFileWidget

Classi a cui ha lavorato individualmente lo studente Luca Polese:

- Container
- DeepPtr
- Xmlify
- Intro

Segue una lista delle classi a cui hanno lavorato entrambi i componenti del gruppo, con una spiegazione di quanto implementato dal sottoscritto Alessandro Poloni:

- Controller, funzionalità che riguardano la gestione degli account
- MainWindow, elementi grafici delle schede della finestra principale, implementazione funzionalità di inserimento e modifica di account e file e funzioni di ricerca
- Account, ad eccezione dei metodi `serializza(QXmlStreamWriter) const`, `deserializza(QXmlStreamReader)`, `riempiTipiDiFile()` e del template di funzione `contaFile() const`
- Classi relative ai tipi di file, ad eccezione dei metodi `serializza(QXmlStreamWriter) const` e `deserializza(QXmlStreamReader)`

9. AMBIENTE DI SVILUPPO

Sistema operativo: Windows 10 Home

Compilatore: MinGW 5.3.0 32bit

Libreria Qt: versione 5.15.2

Sono stati inoltre effettuati test di compatibilità con la macchina virtuale fornita dal docente, dotata di sistema operativo Ubuntu 18.04.3 LTS, compilatore GCC 64-bit e libreria Qt alla versione 5.9.5.