Progetto in C++, framework Qt

MyDigitalLibrary

Relazione tecnica

Università degli Studi di Padova
L.T. in Informatica
A.A. 2024/25

Autore Thomas Morettin
Matr. 2111001

Data 07.09.2025
Corso Programmazione ad Oggetti
Cod. SC02123180

Indice

1 Introduzione

MyDigitalLibrary è un prodotto software che opera come *libreria digitale*. Esso permette la memorizzazione e la gestione di diverse tipologie di **Media**, i quali si dividono in: **Album, Film, Libri e Riviste**.
Le funzionalità principali che offre riguardano la memorizzazione dei Media, i quali vengono visualizzati all'interno di un catalogo che offre una vista su tutti o alcuni elementi filtrati grazie a criteri di ricerca, e la loro gestione; nello specifico si fa riferimento ad operazioni che permettono:

- inserimento di nuovi Media;
- visualizzazione in dettaglio del Media;
- eliminazione del Media;
- modifica di tutti o alcuni dati del Media;
- import/export da file JSON.

Per concludere, l'applicazione è progettata con un'interfaccia grafica comprensibile e minimale, per offrire all'utente un'esperienza di utilizzo semplice e curata.

2 Descrizione del modello

Di seguito viene riportato un estratto del **diagramma UML** delle classi dell'applicazione, reperibile nella sua forma integrale all'interno della cartella contenente la presente relazione tecnica, al nome di `DiagrammaUML.svg`.

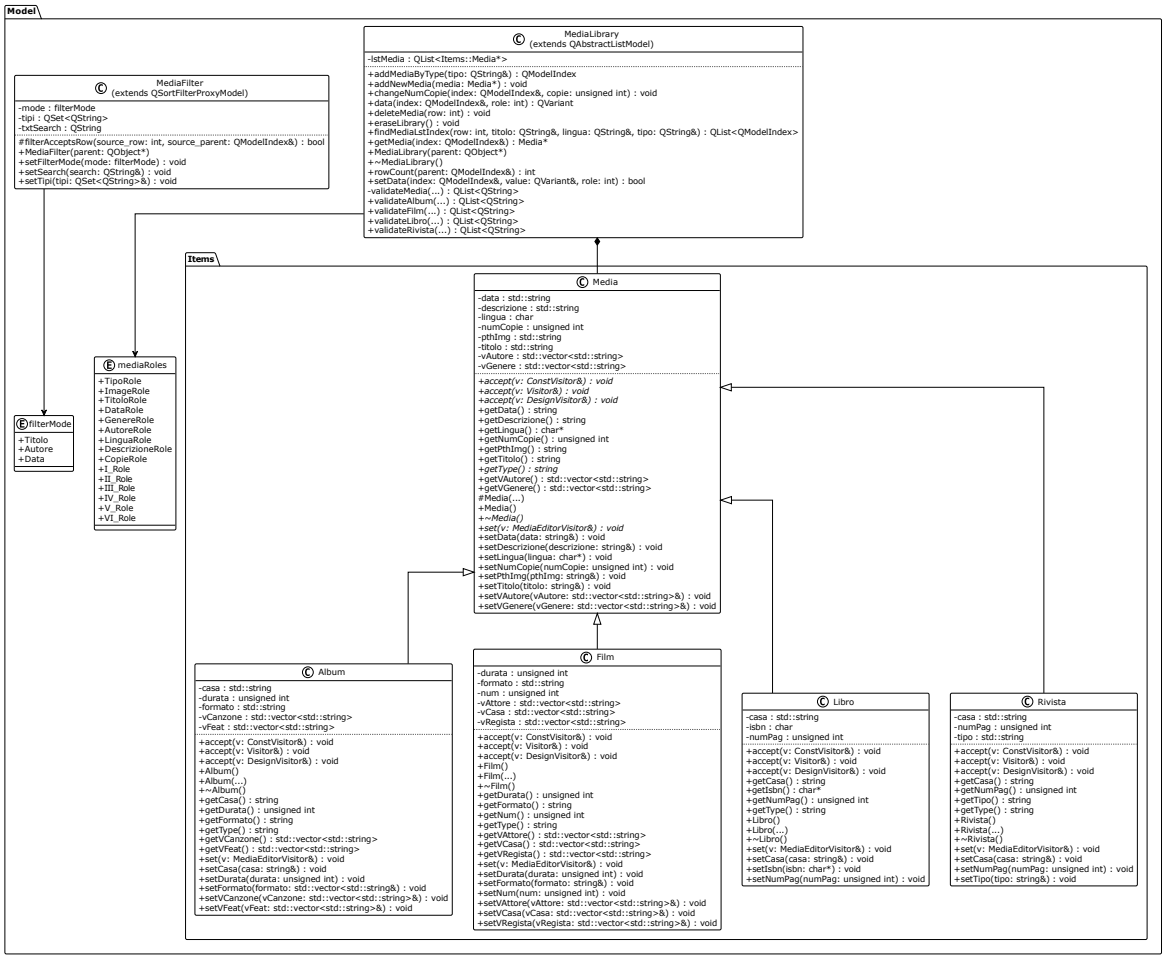


Figura 1: Diagramma UML delle classi fondamentali

Per la realizzazione del prodotto si è fatto riferimento al **design pattern MVC**, ovvero lo scheletro principale dell'intera applicazione.

> **MVC**, acronimo di Model View Controller, è un design pattern architetturale che impone la separazione tra la logica di presentazione dei dati e la loro logica di business (manipolazione).
> La struttura è cosi composta:
>
> - **Model** ha il compito di memorizzare i dati dell'applicazione;
> - **View** rappresenta la visualizzazione grafica degli elementi;
> - **Controller** è il gestore del *Model*, pertanto ha il compito di notificare la *View* dei cambiamenti del Model e manipolare quest'ultimo.

Il progetto, seguendo le specifiche del design pattern MVC, è così organizzato:

- `MediaLibrary` è la classe con il ruolo di *Controller*, pertanto al suo interno sono stati implementati tutti i metodi che permettono la manipolazione del *Model*, il quale è rappresentato da una `QList<Items::Media*>` (privata) per la memorizzazione degli oggetti Media.
  La suddetta classe estende la classe `QAbstractListModel`, ovvero un'interfaccia standard, offerta dal framework Qt, per l'implementazione del design pattern MVC. Inoltre, presenta un campo `enum` adibito al riconoscimento dei ruoli di ciascun Media, ovvero i suoi dati sui quali possono essere implementate le operazioni CRUD.

- `Media` rappresenta la classe base astratta (con campi generici comuni a tutti gli oggetti concreti), da cui ereditano tutte le classi concrete che rappresentano gli oggetti manipolabili all'interno dell'applicazione con i propri campi specifici.

- `MainWindow, FnsIniziale, FnsCatalogo, FnsDettagli e FrmEditor` sono le classi che hanno il ruolo di *View*, ovvero permettono la visualizzazione dei dati. Per semplicità le classi non sono state riportate nel Figura 1 a pag. 2, ma di seguito viene riportata la loro struttura:
  - `MainWindow` rappresenta la finestra principale, la quale grazie ad un oggetto `QStackedWidget` permette di gestire dinamicamente le finestre per le operazioni all'utente;
  - `FnsIniziale` rappresenta la finestra iniziale dalla quale è possibile istanziare la libreria. Successivamente all'instanza della libreria, prima di essere distrutta, viene inviato un segnale a tutte le finestre per la memorizzazione della libreria creata;
  - `FnsCatalogo` rappresenta la finestra principale dove è possibile svolgere la maggior parte delle operazioni, inclusa la visualizzazione della `QListView` per la lista di Media. Il precedente oggetto è collegato alla classe `MediaFilter`, la quale estendendo `QSortFilterProxyModel` ha il compito di visualizzare gli elementi filtrati tramite i criteri di ricerca, prelevando i dati direttamente dal *Model* dell'applicazione;
  - `FnsDettagli` rappresenta la finestra di visualizzazione di tutti i dettagli di un oggetto Media, facendo uso del Visitor MediaDettagliVisitor (Sezione 3.3 a pag. 4, il quale si occupa della creazione dei campi generici e specifici (con il loro popolamento);
  - `FrmEditor` rappresenta la finestra di modifica dei dati di un oggetto Media, facendo uso del Visitor `MediaEditorVisitor` (Sezione 3.3 a pag. 4, il quale si occupa della creazione dei campi di input generici e specifici (con il loro popolamento).

    > **NB** La suddetta classe viene utilizzata sia per l'inserimento di un nuovo Media che per la sua successiva modifica, questo per rendere lo sviluppo più efficiente e mantenibile nel tempo.
    > Il segnale di annullamento dell'inserimento/modifiche aziona due slot differenti all'interno della `MainWindow`, per il corretto funzionamento del programma.

Per l'applicazione, oltre al design pattern MVC, è stato fatto uso del **design pattern Visitor** per la corretta gestione degli oggetti concreti derivanti da Media (spiegato esaustivamente nella sezione successiva) e del **design pattern Factory**, il quale viene implementato nella classe `MediaFactory` per la creazione di un oggetto `QJsonObject` per ciascun Media, da poter inserire all'interno del file portabile JSON.

3 Polimorfismo

Nella presente applicazione si è sfruttato il concetto di polimorfismo sia nella variante "banale" imposta dal linguaggio di programmazione C++, che "non banale" tramite l'utilizzo di **design pattern Visitor**.

Visitor è un design pattern comportamentale, il quale ha il compito di separare un algoritmo dalla struttura dati alla quale è applicato. Ciò viene utilizzato per aggiungere nuove funzionalità senza dover modificare la struttura base degli oggetti sui quali è applicato.
Il suo utilizzo risulta fondamentale, nel contesto di quest'applicazione, nella gestione di algoritmi differenti per tipo concreto dell'oggetto visitato.

> **NB** Tutte le classi che fanno uso di design pattern Visitor sono state create con lo scopo sia di gestire i campi generici di ciascun Media (**es** lingua, titolo, ...) che quelli specifici per tipologia (**es** Libro → ISBN, Album → durata, ...) per maggiore manutenibilità del codice.

3.1 Visitor::Items::MediaGetterVisitor/MediaSetterVisitor

Il primo utilizzo di polimorfismo "non banale" riguarda le classi che permettono di estrarre e settare i parametri, di ciascun Media. Le classi, rispettivamente figlie delle classi `ConstVisitor e Visitor`, permettono di manipolare i ruoli definiti all'interno del *Model* che gestisce la `QList<Items::Media*>`.

3.2 Visitor::JSON::ToJsonVisitor

Il secondo utilizzo avviene nella classe adibita alla creazione di un oggetto `QJsonObject` a partire dal Media, per tipologia, presente nel catalogo, per il popolamento del file JSON.

3.3 Visitor::View::MediaDettagliVisitor/MediaEditorVisitor

L'utilizzo più significativo di polimorfismo all'interno dell'applicazione avviene nelle suddette classi, le quali ereditano dalla classe base astratta `DesignVisitor`.
La classe `MediaDettagliVisitor` fa uso di design pattern Visitor per la creazione e il popolamento dei widget `QLabel` adibiti alla visualizzazione dei dettagli generici e specifici (per tipologia) di ciascun Media. La classe `MediaEditorVisitor` si occupa della creazione, del popolamento e del controllo dei widget `QLineEdit, QTextEdit e QComboBox`, adibiti all'inserimento/modifica delle informazioni generiche e specifiche (per tipologia) di ciascun Media. Ulteriormente, la classe `MediaEditorVisitor` implementa metodi `set()`, non presenti nella classe base astratta, per effettuare il popolamento/modifica (dagli elementi widget) delle informazioni di ciascun Media tramite i ruoli del *Model* mantenendo separate le responsabilità dettate dal design pattern MVC (Sezione 2 a pag. 2).
Quest'ultima classe, inoltre, implementa metodi, propri per ciascuna tipologia di Media, per la chiamata a funzioni di validazione dei dati (interne alla classe `MediaLibrary`) forniti dall'utente (**es** il Libro presenta codice ISBN nel formato corretto), con conseguente avviso all'utente nel caso in cui tenti di salvare dati non coerenti con la struttura degli oggetti del prodotto software.

> **NB** Entrambe le classi restituiscono un oggetto `QWidget`, contenente le informazioni generiche e specifiche per tipologia di Media.
> Inoltre, entrambe le classi `MediaDettagliVisitor` e `MediaEditorVisitor` sovrascrivono un metodo virtuale pure, definito nella classe base `DesignVisitor`, dal nome `clear()` che permette di distruggere il `QWidget` contenente i campi specifici per ciascun Media. In questo modo, nel caso in cui il Media selezionato dall'utente dovesse essere differente rispetto al precedente, la finestra viene aggiornata inserendo i campi di input corretti; se questo dovesse essere identico, invece, il visitor non procederà alla distruzione del `QWidget` per gli elementi specifici ma aggiornerà semplicemente i campi nel caso in cui siano stati aggiornati.

3.4 Visitor::Items::CheckFormatoVisitor

L'ultimo utilizzo di polimorfismo avviene nella classe che, ereditando da `ConstVisitor`, ha il compito di controllare l'uguaglianza tra un parametro fornito dalla ricerca ne `QList<Items::Media*>` e i seguenti campi per ciascun Media: **Album → formato, Film → formato, Libro → ISBN e Rivista → tipo**. Questa classe viene utilizzata per la gestione dei duplicati, insieme alla corrispondenza lingua/titolo (Sezione 5.1 a pag. 6).

4 Persistenza dei dati

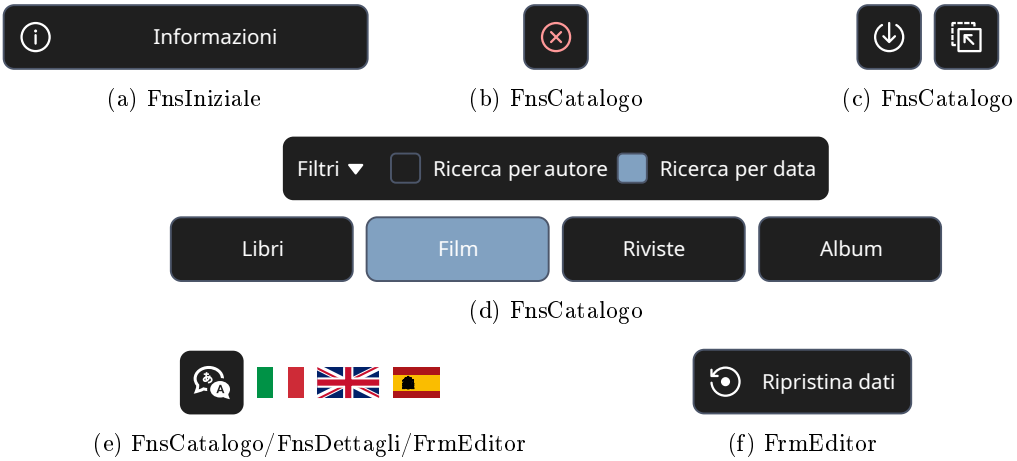
Come da commessa accademica è stato progettato un sistema di persistenza dei dati su file system tramite file in formato **JSON**. Nello specifico, il suddetto file contiene la memorizzazione dell'intero catalogo di oggetti Media, gestendone la specificità tramite un attributo *media*.
Viene consegnato, all'interno della cartella di progetto, un file `Savings/Salvataggio.json` (presente, identico, nella cartella `EsempioPersistenza` della **cartella ZIP** contenente l'intero lavoro), contenente diversi oggetti per tipologia, in modo da poter avviare il prodotto con Media pre registrati.

> **NB** Il file JSON, oltre a contenere oggetti Media correttamente memorizzati, ne contiene alcuni che presentano dati mancanti, richiesti dalla struttura del codice per l'importazione. Ciò a dimostrazione di come alcuni Media potranno non essere importati correttamente, a causa delle **politiche di controllo** del file .json.

5 Funzionalità implementate

In aggiunta alle funzionalità richieste all'interno della commessa, vengono riportate di seguito le migliorie funzionali/grafiche implementate all'interno del prodotto software.

5.1 Implementazioni funzionali



(a) FnsIniziale (b) FnsCatalogo (c) FnsCatalogo

(d) FnsCatalogo

(e) FnsCatalogo/FnsDettagli/FrmEditor (f) FrmEditor

Figura 2: Elementi grafici in evidenza

Si considerino le seguenti:

- gestione di **quattro tipologie** di oggetti Media;
- importazione ed esportazione dei dati in formato JSON:
  - controllo sulla **presenza e correttezza** dei campi dati memorizzati; nel caso di dati mancanti/incorretti espressamente richiesti a livello funzionale il Media non viene importato nella libreria, con avviso all'utente. Questo per mantenere il sistema robusto e permettere all'utente di poter inserire manualmente, nel file JSON, solamente i dati minimali, per completare il Media all'interno dell'applicazione.

    > **ES** Per tutti gli oggetti Libro è richiesto obbligatoriamente il campo ISBN in formato 13 char, altrimenti il Media non viene caricato.
    > Per tutti gli oggetti Rivista viene controllata la presenza del campo casa editoriale, se questo dovesse essere errato o inesistente il Media viene importato correttamente nella libreria, ma alla prossima modifica ne è obbligatorio l'inserimento.
- pulsante di apertura della documentazione tecnica dalla finestra di inizializzazione della libreria (path individuato automaticamente tramite variabile globale *DIR_RELAZIONE*) (Figura 2a a pag. 5);

- pulsante di **cancellazione** completa della libreria (Figura 2b a pag. 5);
- casella di testo `QLineEdit` per la ricerca in **tempo reale**;
- ricerca dell'oggetto Media desiderato tramite i seguenti criteri (Figura 2d a pag. 5):
  - filtri di ricerca (applicati alla `QLineEdit`) per titolo (default), autore o data di pubblicazione;
  - filtri di ricerca per tipologia di Media.
- azioni mouse-click sugli elementi grafici:
  - *click-dx* sull'elemento grafico dell'oggetto Media nel catalogo → apertura della scheda dettagli;
  - *click-sx* sull'elemento grafico dell'oggetto Media nel catalogo → apertura di un menu a tendina per modificare/eliminare;
- **controllo automatico della presenza/correttezza** dei dati in tutti i campi di input (esclusa l'immagine di copertina) sia nell'inserimento che nella modifica di un Media, con avviso all'utente;

  NB I campi *featuring* e *attori* rispettivamente degli oggetti concreti Album e Film **non sono obbligatori**;
- gestione degli oggetti **Media duplicati**, i quali, se presenti, vengono accorpati all'interno del Media esistente (somma tra le copie presenti e le nuove copie) nei seguenti scenari:
  - importazione da file JSON;
  - inserimento dell'oggetto Media da applicazione.

  NB Un oggetto Media viene considerato *duplicato* di uno già esistente se presenta i seguenti campi identici:
  - per tutti i Media: lingua e titolo;
  - per tutti i Libri: ISBN 13 char;
  - per tutti gli Album/Film: formato;
  - per tutte le Riviste: tipo.

  ES Per due oggetti Album che presentano stessa lingua e titolo, ma **diverso formato** (CD e musicassetta) vengono creati due Media separati.
  Per due oggetti Rivista che presentano stessa lingua, titolo e tipo (entrambe mensili) viene aggiunto al numero di copie del Media già esistente il numero di copie del Media che si è intenzionati ad aggiungere.
- gestione grafica per l'inserimento della **lingua**, il quale parametro viene rappresentato dalla *bandiera* corrispondente (Figura 2e a pag. 5);
- pulsante per il **ripristino dei dati** precedenti alle modifiche apportate dall'utente (Figura 2f a pag. 5);

  NB Nello scenario di una modifica ad un Media preesistente, vengono ripristinati i dati salvati all'interno del catalogo.
  Nello scenario di un nuovo inserimento, vengono ripuliti i campi di input, poiché il Media non presentava dati salvati precedentemente.
- **salvataggio automatico dell'immagine di copertina**, selezionata al momento dell'inserimento/modifica, all'interno della cartella dedicata (path individuato automaticamente tramite variabile globale *DIR_COPERTINE*);
- **esportazione automatica** dei dati presenti nella libreria su file JSON alla chiusura dell'applicazione (path individuato automaticamente tramite variabile globale *DIR_SALVATAGGIO*);

  NB I dati vengono salvati automaticamente solamente se si ha già istanziato una libreria o quest'ultima non risulta vuota.
- utilizzo di **best practices** nella stesura del codice, come per esempio la creazione di costruttori `protected` all'interno delle classi base astratte o passaggio di parametri per riferimento all'interno dei metodi per evitare la creazione di copie dei valori in memoria.

5.2 Implementazioni grafiche

Si considerino le seguenti:

- studio della **palette grafica** *Nord Color Palette* (Arctic Ice Studio), del **font** *Inter* (Google Fonts), delle **icone** (iconmonstr) e degli **elementi** della pagina per la coerenza visiva del prodotto;
- gestione del **ridimensionamento della finestra**, con conseguente adattamento degli oggetti grafici;
- ridefinizione di tutti gli **elementi grafici** dell'applicazione tramite file .qss;
- creazione dell'oggetto grafico **MediaDelegate**, utilizzato all'interno della struttura `QListView` nel catalogo principale, per la visualizzazione dei dettagli principe di un Media;
- gestione grafica della **mancanza dell'immagine di copertina** con immagini studiate ad-hoc per ciascuna pagina (catalogo/dettagli/editor);

6 Rendicontazione delle ore

Attività	Ore previste	Ore effettive
Studio del framework Qt e ripasso dei concetti C++	10	10
Progettazione grafica/funzionale	5	15
Sviluppo del codice (Model/Controller)	40	50
Sviluppo del codice GUI (View)	30	25
Testing, debugging e popolamento	20	20
Stesura della relazione	5	8
Totale	**110**	**128**

In seguito all'analisi delle ore previste/effettive per la realizzazione di questo prodotto software è importante precisare il seguente aspetto: il prodotto vuole essere il più simile possibile ad un'applicazione di uso comune, mettendo in primo piano la gestione di tutte le possibili azioni da parte dell'utente (controlli inclusi) e ponendo attenzione alla resa grafica finale.
Successivamente, si evidenzia un superamento del 16% delle ore previste, dovuto principalmente alle seguenti ragioni:

- **studio** di un framework nuovo e complesso, con interesse particolare alle classi che hanno reso possibile l'applicazione del design pattern MVC e Visitor;
- **progettazione** di un'interfaccia complessa, sia dal punto di vista grafico che funzionale per la gestione del sistema *signals/slots*;
- sviluppo del codice (*Model/Controller*) con funzionalità aggiuntive (**es** gestione dei duplicati) e controlli su tutte le possibili azioni dell'utente;
- **stesura della relazione** con creazione del grafico con tecnica di reverse engineering grazie al software Enterprise Architect e rappresentazione grafica tramite PlantUML.

7 Conclusioni e sviluppi futuri

Il prodotto software, al termine dello sviluppo e della fase di testing, risulta correttamente funzionante, rispettando tutte le caratteristiche sopracitate.
Si vuole evidenziare la possibilità di aggiungere le seguenti funzioni in un possibile aggiornamento dell'applicazione:

- funzionalità di import/export da database programmato in **PostgreSQL**, situato su server remoto all'interno della piattaforma Supabase, in quanto la maggior parte del codice di funzionamento è stata sviluppata per il progetto del corso di **Basi di Dati** (cod. *SCP4065533*);
- implementazione del codice per eseguire registrazione/login all'applicazione con account personale (e-mail e password) o guest, con salvataggio dei dati su database remoto.