Università degli Studi di Padova

Relazione Progetto di Porgrammazione ad Oggetti

A.A. 2024-2025

Carmelo Russello

Indice

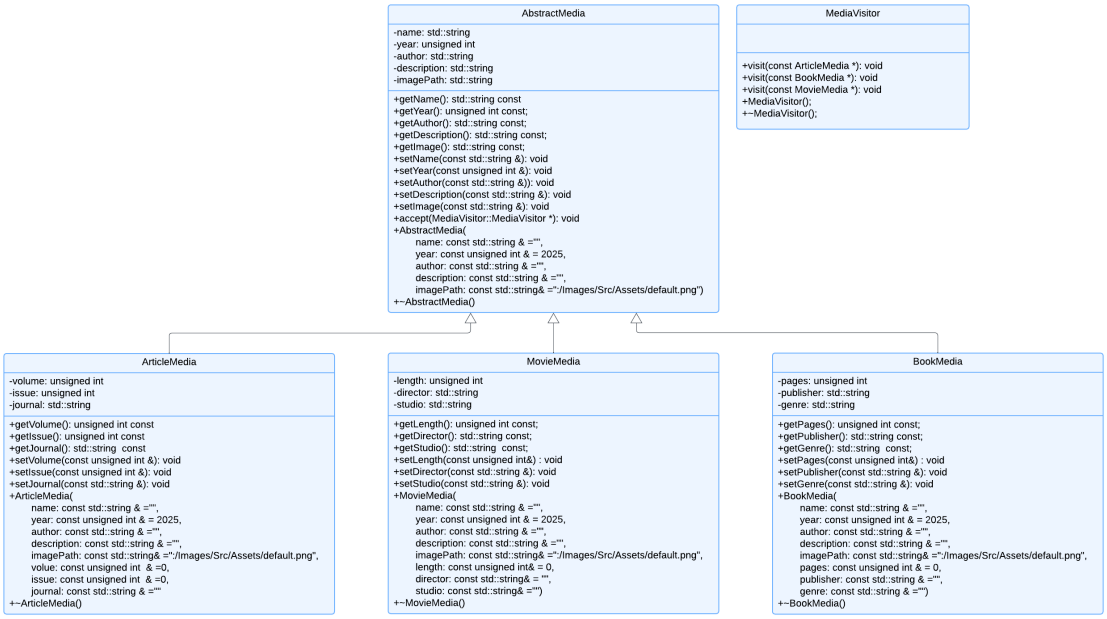
1 Introduzione

Il mio progetto consiste in un'interfaccia di gestione e visualizzazione di una libreria multimediale, composta da film, articoli e libri. Questa libreria fa parte di un software pensato per organizzare, catalogare e analizzare contenuti multimediali, con l'obiettivo di fornire agli utenti un'esperienza completa per la gestione e la consultazione delle proprie risorse.
Ogni categoria di media ha caratteristiche specifiche che possono essere personalizzate e visualizzate. Ad esempio, per i film sono disponibili informazioni come la duarata, il regista e lo studio che lo ha prodotto; per gli articoli, il volume, la rivista di pubblicazione e la edizione; per i libri, il numero di pagine, l'editore e il genere. I media possono essere aggiunti manualmente o importati da un file JSON, consentendo agli utenti di popolare rapidamente la libreria.
L'interfaccia principale presenta un pannello laterale che elenca i media disponibili in ordine verticale, con la possibilità di selezionarli per visualizzare i dettagli associati. Per ogni elemento selezionato, viene visualizzato l'insieme delle loro informazioni e l'immagine del contenuto multimediale.
Un elemento fondamentale dell'interfaccia è la search bar, che permette agli utenti di trovare rapidamente i media desiderati tramite il nome. Questo strumento semplifica la navigazione nella libreria, specialmente

quando il numero di contenuti aumenta, consentendo di localizzare in modo immediato film, articoli o libri specifici senza dover scorrere l'intero elenco. La search bar rende l'esperienza utente più fluida ed efficiente. Infine la status bar in basso che guida l'utente mostrando le operazioni eseguite.

2 Modello

Il modello logico di questa applicazione è progettato per gestire e organizzare una libreria multimediale contenente diversi tipi di contenuti, come articoli, libri e film. L'obiettivo principale è quello di creare una struttura flessibile e scalabile che permetta di rappresentare, catalogare e manipolare in modo efficace queste diverse tipologie di media, mantenendo al contempo un alto grado di estensibilità e riusabilità del codice.
La struttura delle classi è organizzata secondo un approccio orientato agli oggetti, dove una classe base astratta (AbstractMedia) fornisce la base comune per tutti i tipi di media, mentre le classi concrete (come ArticleMedia, BookMedia e MovieMedia) estendono questa base, aggiungendo caratteristiche specifiche e metodi dedicati per gestire i dati propri di ciascun tipo di media.
Un aspetto fondamentale di questo modello è l'utilizzo del pattern Visitor, che consente di separare l'implementazione delle operazioni da quella dei dati. In questo modo, è possibile applicare operazioni personalizzate sui diversi tipi di media senza dover modificare direttamente le classi concrete. Questo approccio rende il sistema altamente estensibile, poiché l'aggiunta di nuove operazioni non richiede modifiche alle classi che rappresentano i media.
L'interazione tra il modello logico e l'interfaccia grafica è facilitata dalla presenza di due varianti del metodo accept. Ciò permette al modello logico di interagire con componenti grafici o operazioni esterne, mantenendo il codice pulito e modulare.



Classe `AbstractMedia`

La classe astratta `AbstractMedia` rappresenta la base comune per tutte le tipologie di media.

- **Attributi principali**:
  - `name`: il nome del media,
  - `year`: l'anno di pubblicazione,
  - `author`: l'autore del media,
  - `description`: una breve descrizione,
  - `imagePath`: il percorso dell'immagine associata al media (valore predefinito:"/Images/Src/Assets/default.png")
- **Metodi principali**:

  - Getter e setter per ciascun attributo, consentendo l'accesso e la modifica dei dati.
  - Il metodo `accept(MediaVisitor *visitor)` consente di utilizzare il pattern Visitor per applicare operazioni specifiche a seconda della tipologia di media.

Classi concrete

Le classi derivate da `AbstractMedia` rappresentano i tipi specifici di media gestiti dal sistema.

Classe `ArticleMedia`

Questa classe rappresenta gli articoli.

- **Attributi aggiuntivi**:
  - `volume`: il volume del giornale o rivista,
  - `issue`: il numero dell'edizione,
  - `journal`: il nome del giornale o della rivista.
- **Metodi principali**:
  - Getter e setter per i nuovi attributi (`getVolume()`, `getIssue()`, `getJournal()`), consentendo l'accesso e la modifica dei dati specifici di un articolo.

Classe `BookMedia`

Questa classe rappresenta i libri.

- **Attributi aggiuntivi**:
  - `pages`: il numero di pagine del libro,
  - `publisher`: l'editore,
  - `genre`: il genere del libro.
- **Metodi principali**:
  - Getter e setter per i nuovi attributi (`getPages()`, `getPublisher()`, `getGenre()`), consentendo la gestione dei dati specifici relativi ai libri.

Classe `MovieMedia`

Questa classe rappresenta i film.

- **Attributi aggiuntivi**:
  - `length`: la durata del film in minuti,
  - `director`: il regista,
  - `studio`: lo studio di produzione.
- **Metodi principali**:
  - Getter e setter per i nuovi attributi (`getLength()`, `getDirector()`, `getStudio()`), permettendo di gestire i dati specifici di un film.

Classe `MediaVisitor`

La classe `MediaVisitor` utilizza il pattern Visitor per consentire operazioni personalizzate su ciascun tipo di media.

- **Metodi principali**:
  - `visit(const ArticleMedia *)`, `visit(const BookMedia *)` e `visit(const MovieMedia *)` permettono di definire comportamenti specifici per ogni tipo di media senza modificare le classi concrete.

3 Polimorfismo

Nel mio progetto ho utilizzato ampiamente il design pattern del **visitor**, che si è rivelato fondamentale per la gestione delle operazioni specifiche sui vari tipi di media. In particolare, il pattern è stato sfruttato per applicare operazioni sui media senza modificare direttamente le classi concrete. La struttura prevede l'utilizzo di un **MediaVisitor** e un **ConcreteVisitor** per gestire operazioni come l'impostazione dell'icona corretta per ogni tipo di media, oltre a un **JSONVisitor** che consente di esportare i dati in un file JSON.

- Il **ConcreteVisitor** è il visitor principale che applica operazioni specifiche a ciascun tipo di media. Tra le operazioni più importanti ci sono:
  - **'setIcon'**: permette di assegnare l'icona corretta a ciascun tipo di media. A seconda del tipo di media (Articolo, Libro, Film), il **ConcreteVisitor** seleziona e applica l'icona appropriata.
  - **'setAttributes'**: permette di raccogliere e settare gli attributi specifici di ogni media, come il numero di pagine di un libro o la durata di un film, e di aggiornarli dove necessario, ad esempio nell'interfaccia utente o in un database.
- Il **JSONVisitor** è un visitor che permette di esportare i dati specifici di ciascun tipo di media in un formato JSON. Esso visita gli oggetti di tipo **ArticleMedia**, **BookMedia** e **MovieMedia**, raccoglie i relativi attributi (nome, anno di pubblicazione, autore, ecc.) e li serializza in un file JSON. Questo approccio consente di salvare i dati in un formato strutturato, che può essere facilmente letto, modificato o importato in altre sessioni dell'applicazione. I metodi di visita del **JSONVisitor** sono:
  - **'visit(const ArticleMedia *)'**: esporta i dati relativi agli articoli, come volume, numero di edizione, e nome della rivista.
  - **'visit(const BookMedia *)'**: esporta i dati relativi ai libri, come il numero di pagine, l'editore, e il genere.
  - **'visit(const MovieMedia *)'**: esporta i dati relativi ai film, come la durata, il regista e lo studio di produzione.
- Il **MediaVisitor** è la classe astratta che definisce le operazioni comuni per ogni tipo di media. I metodi **visit** sono dichiarati come puri in questa classe, mentre la loro implementazione viene fornita dal **ConcreteVisitor** e dal **JSONVisitor**, che gestiscono rispettivamente le operazioni di visualizzazione (come l'icona e gli attributi) e di esportazione dei dati (in formato JSON).

L'utilizzo del design pattern **visitor** consente di mantenere il codice flessibile ed estensibile. Ogni volta che viene introdotto un nuovo tipo di media, è sufficiente aggiungere un nuovo metodo di visita, senza la necessità di modificare il codice delle classi concrete. Questo approccio separa anche le preoccupazioni: la logica per gestire l'aspetto visivo dei media (icone, attributi) è separata dalla logica di gestione dei dati (esportazione in JSON).

4 Persistenza dei dati

Nel mio progetto, la persistenza dei dati è gestita tramite la classe **JsonStorage**, che si occupa di serializzare e deserializzare i media (articoli, libri, film) in formato JSON. Questo approccio permette di salvare e caricare i dati in modo strutturato, consentendo al sistema di mantenere i dati anche dopo che l'applicazione è stata chiusa. La classe **JsonStorage** interagisce con il pattern **visitor**, utilizzando un **JSONVisitor** per la conversione degli oggetti **AbstractMedia** in oggetti JSON e viceversa.
Ecco come ho implementato la persistenza dei dati:

- **Classe JsonStorage**: La classe **JsonStorage** è il cuore della persistenza dei dati e si occupa della serializzazione e deserializzazione degli oggetti. Ha due metodi principali:
  - **'turnToObject'**: Questo metodo prende un oggetto di tipo **AbstractMedia** (che può essere un articolo, un libro o un film) e lo converte in un oggetto **QJsonObject**. Per fare ciò, utilizza un **JSONVisitor**, che visita l'oggetto media e raccoglie i suoi attributi per costruire l'oggetto JSON. Il visitor è passato all'oggetto tramite il metodo **accept**, e i dati vengono estratti in modo che possano essere serializzati.

  - **'turnToMedia'**: Questo metodo esegue l'operazione inversa, ovvero converte un oggetto **QJsonObject** in un oggetto concreto di tipo **AbstractMedia**. La funzione verifica prima il tipo di media (Libro, Articolo, o Film) controllando la presenza di determinati campi nel JSON. A seconda dei campi rilevati, vengono creati gli oggetti corretti (**BookMedia**, **ArticleMedia**, o **MovieMedia**) con i dati estratti dal JSON.

- **JSONVisitor**: Il visitor utilizza il pattern **visitor** per raccogliere i dati specifici di ogni tipo di media (Libro, Articolo, Film) e convertirli in un formato JSON. Ogni tipo di media ha un proprio metodo di visita, che è responsabile per raccogliere i dati dell'oggetto e costruire un **QJsonObject** che rappresenta tale media.

- **Deserializzazione**: Quando si caricano i dati da un file JSON, la funzione **turnToMedia** esamina l'oggetto JSON e, in base ai campi presenti, crea l'istanza del tipo di media appropriato. Se l'oggetto JSON contiene il campo **Publisher**, viene creato un oggetto di tipo **BookMedia**; se contiene il campo **Journal**, viene creato un oggetto **ArticleMedia**; se contiene il campo **Director**, viene creato un oggetto **MovieMedia**. Se i dati non corrispondono a nessuna di queste tipologie, viene generata un'eccezione per indicare che il tipo di media non è valido o che mancano i campi richiesti.

- **Eccezioni**: La funzione **turnToMedia** solleva un'eccezione di tipo `std::invalid_argument` se il JSON non contiene i campi necessari o se il tipo di media è sconosciuto. Questo meccanismo di gestione delle eccezioni garantisce che i dati vengano trattati in modo sicuro e che il sistema non continui l'esecuzione con dati incompleti o errati.

5 Funzionalità implementate

Le principali funzioni implementate nella mia applicazione sono le seguenti:

- **Shortcuts da tastiera**: Implementazione di scorciatoie da tastiera per facilitare l'interazione con l'interfaccia dell'applicazione, inclusi tasti per la navigazione e l'esecuzione di azioni rapide.

- **Searchbar per cercare i vari media per nome**: Una barra di ricerca che consente all'utente di cercare facilmente media specifici per nome, permettendo una gestione rapida dei contenuti.

- **Visualizzazione di media**: La possibilità di visualizzare i dettagli dei media selezionati, mostrando informazioni come nome, anno, autore e descrizione, insieme a un'immagine.

- **Capacità di modificare o cancellare i media**: Funzionalità che permettono all'utente di modificare i dettagli di un media esistente o di eliminarlo dalla libreria.

- **Possibilità di caricare immagini per i media**: Supporto per caricare immagini personalizzate per ogni tipo di media, migliorando la visualizzazione e l'organizzazione della libreria.

- **Gestione di tre tipi di media (Movie, Book, Article)**: L'applicazione è progettata per gestire tre tipologie di media: film, libri e articoli, ciascuna con attributi specifici.

- **Introduzione della Status Bar**: Implementazione di una barra di stato che fornisce informazioni sull'operazione in corso e lo stato dell'applicazione.

- **Persistenza dei dati in formato JSON con apertura e salvataggio file**: Funzione di persistenza dei dati che permette di salvare e caricare i media in formato JSON, supportando l'apertura e il salvataggio dei file JSON.

- **Unicitá del nome del media**: Nessun media puo avere nome uguale ad un altro.

6 Rendicontazione ore

A causa della mia inesperienza con il framework Qt e del fatto che questo è il mio primo progetto effettivo, ho sforato leggermente le ore previste. Nonostante gli sforzi per rispettare la pianificazione, la necessità di imparare le basi di Qt e di affrontare le sfide del progetto ha comportato un investimento di tempo maggiore rispetto a quanto inizialmente stimato. Questo mi ha permesso, però, di acquisire una comprensione più approfondita del framework e di affrontare con maggiore consapevolezza le fasi successive del progetto.

Attività	**Ore previste**	**Ore effettive**
Studio e progettazione	5	5
Codice del modello	5	10
Studio del framework Qt	10	10
Sviluppo interfaccia grafica	10	15
Debugging e test	5	10
Stesura della Relazione	5	5
Totale	40	55

Tabella 1: Rendicontazione ore

7 Conclusione

In conclusione, questo progetto mi ha offerto un'importante opportunità di crescita, permettendomi di prendere familiarità con il framework Qt e di apprendere le basi dello sviluppo di applicazioni grafiche. L'esperienza con Qt mi ha consentito di comprendere meglio la struttura e le funzionalità del framework, mentre l'uso di Qt Creator come ambiente di sviluppo mi ha facilitato nella gestione del progetto, nella scrittura del codice e nel controllo degli errori. Inoltre, ho acquisito competenze nel debugging utilizzando il debugger integrato di Qt Creator e GDB, degli strumenti che si sono rivelati fondamentali per analizzare e risolvere i bug in modo più efficace e preciso (in particolare i segmentation fault). Oggi, dopo aver completato questo progetto, mi sento decisamente più capace e sicuro nelle mie abilità. Sento di aver imparato concetti e tecniche che sono fondamentali non solo per il framework Qt, ma anche per il debugging in generale, e sono convinto che queste esperienze saranno determinanti per i miei futuri progetti di sviluppo software.