Progetto Biblioteca

Riccardo Graziani, Matricola 2101054

Giugno 2025

Indice

1 Introduzione

Il progetto esposto in questa relazione consiste in un programma con interfaccia grafica, realizzato in C++ mediante il framework *Qt*, che permette all'utente di gestire una biblioteca virtuale. La biblioteca supporta diversi tipi di *media*, come ad esempio **libri**, **film** e **brani musicali**. Per ogni tipo di media, l'utente può svolgere le principali operazioni *CRUD*: creare nuovi oggetti, modificare oggetti esistenti, eliminare oggetti e ricercare oggetti specifici. Ogni tipo di media possiede attributi caratteristici e una propria visualizzazione grafica unica.
La biblioteca virtuale permette all'utente di salvare i media creati in file **JSON** e di caricare file già esistenti, siano essi file vuoti o popolati di media.
I media creati dall'utente o caricati da file sono visualizzabili attraverso la lista dei media, che li presenta sotto forma di piccole card che contengono le informazioni principali. Oltre alla lista dei media è presente la schermata di visualizzazione dei dettagli, che permette di visualizzare ogni attributo del media scelto.

2 Descrizione del modello

Il progetto è stato sviluppato seguendo il pattern software architetturale **Model-View**, il quale ha permesso una netta separazione tra la logica di accesso e rappresentazione dei media (il *Model*) e la logica di visualizzazione e interazione con i media (la *View*).

2.1 Model

Il diagramma di *Figura 1* rappresenta il *Model* attraverso un linguaggio di modellazione standard, ossia lo **Unified Model Language** (UML). La gerarchia del *Model* parte dalla classe *AbstractItem*, che rappresenta in maniera concettuale un media. Tale classe raccoglie gli attributi comuni a tutti i media, come ad esempio: identificatore, autore, titolo, etc.
AbstractItem implementa, oltre ai metodi classici come vari *getter* e *setter*, i metodi virtuali **accept** con argomento **ItemVisitor** e **ConstItemVisitor**, utilizzati per implementare il pattern **Visitor**. Dalla classe *AbstractItem* discendono le tre classi concrete:

1. *Book*: rappresenta un oggetto libro e possiede attributi caratteristici quali **publisher**, **language** e **genre**.

2. *Movie*: rappresenta un oggetto film e possiede attributi caratteristici quali **language**, **duration**, **genre** e **ageRating**.

3. *Music*: rappresenta un oggetto brano musicale e possiede attributi caratteristici quali **album**, **duration**, **explicitLanguage** e **genre**.

Per rappresentare i valori dei campi **genre** (presente in *Book*, *Movie* e *Music*) e **age rating** (presente in *Movie*) sono stati definiti tre enum corrispondenti:

- *BookMovieGenre*: rappresenta i generi letterari per i libri.

- *MusicGenre*: rappresenta i generi musicali.

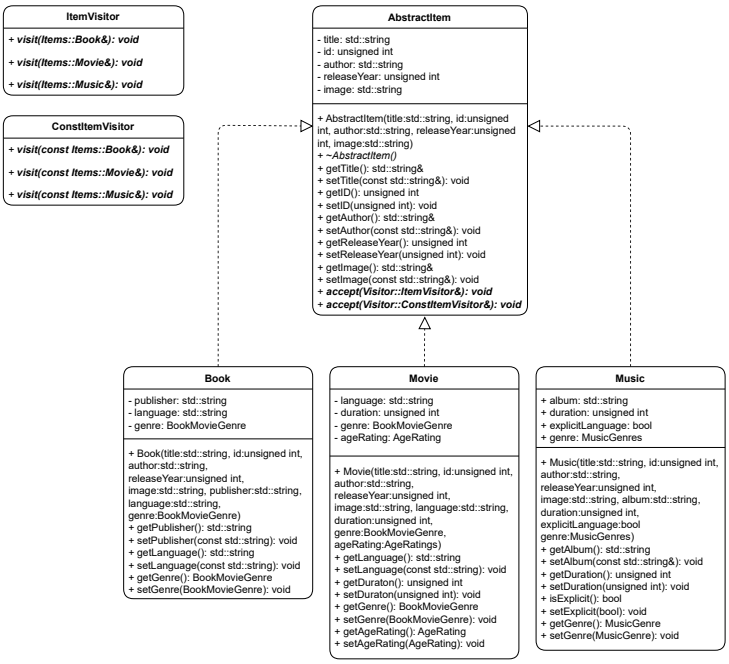

Figura 1: Diagramma UML

- *AgeRatings*: rappresenta i limiti di età attribuibili ai film.

Assieme alla definizione di tali enum, sono state definite funzioni di utility per facilitare la conversione da enum a std::string e viceversa.
La sotto-cartella **Memory** contiene la definizione della classe responsabile per la gestione dei media creati. La classe *ItemRepository* sfrutta il pattern **Singleton**, che restringe il numero di istanze globali della classe a una sola e ne permette l'accesso alle altre classi. In questo modo si facilita l'accesso ai media da parte delle classi che ne fanno richiesta, e si centralizza la gestione degli oggetti creati. Al suo interno *ItemRepository* sfrutta uno std::vector come *container* principale, ed espone una serie di metodi per poter interagire con tale vector.
Infine la sotto-cartella **JSON** gestisce la lettura/scrittura dei media da/verso file in formato JSON. In particolare:

- *JsonVisitor* eredita da *ConstItemVisitor* ed è responsabile per la scrittura del valore degli attributi dei media in formato JSON: tramite il pattern Visitor questa classe implementa i metodi visit per poter correttamente ricavare ogni campo del media che si sta salvando.

- *JsonReader* è la classe responsabile per la lettura da file: in base al campo **type** capisce che oggetto bisogna creare e richiama la funzione corrispondente.

- *JsonRepository* rappresenta una memoria intermedia tra il programma e i file JSON. Mantiene gli oggetti letti/creati in una std::map, e si aggiorna in parallelo alla *ItemRepository* in caso di operazioni di creazione, modifica

o eliminazione. Anche *JsonRepository* espone metodi per poter interagire con la std::map, tra cui in particolare i metodi *readAll()*, che viene usato quando si carica un file JSON per popolare la *ItemRepository*, e *store()*, che viene usato quando si decide di salvare il contenuto di *ItemRepository* nel file JSON aperto in quel momento.

2.2 View

La finestra principale è il widget *MainWindow*: in questo widget si trovano tutti i componenti della GUI, ed è qui che avviene la comunicazione tra il *Model* e la *View*. La *MainWindow* è infatti responsabile per l'inserimento e l'eliminazione dei media dalla memoria centrale e dalla repository JSON: tali azioni sono gestite da funzioni apposite che richiamano i metodi delle classi del *Model*. *MainWindow* tiene inoltre traccia del file JSON attualmente aperto, gestisce il caricamento/salvataggio dei media da file e informa l'utente in caso abbia delle modifiche non salvate. La struttura della *MainWindow* si compone di **tre** elementi principali:

- In alto si trova una **ToolBar** che racchiude le principali operazioni sull'interazione con i file JSON (nel sottomenu *File* è possibile **creare** un nuovo file JSON su cui scrivere, **aprire** un file JSON esistente e **salvare** i media presenti nel file JSON scelto), la creazione di nuovi media (gestita dal widget *EditWidget*) e informazioni sulle *scorciatoie* da tastiera implementate. La Toolbar è resa fissa e ogni azione presente viene identificata sia in modo testuale sia tramite icone.

- A sinistra si trova il widget *ItemListWidget*, che racchiude i media creati/caricati e li presenta come card, modellate dal widget *ItemCardWiget*, il quale crea una nuova card assegnando dinamicamente un icona appropriata al tipo e mostrando le informazioni principali quali *titolo* e *autore*. All'interno di *ItemListWidget* si trova, in alto, il widget *SearchWidget*, usato per implementare la ricerca in **tempo reale** dei media, basandosi sull'attributo comune *titolo*. In base ai caratteri digitati, la lista dei media viene filtrata per mostrare solo gli elementi che hanno un match con la stringa ricercata: viene applicato un delay interno di *200ms* al refresh della lista filtrata, che si avvia ad ogni carattere digitato. Inoltre viene reso disponibile una funzionalitá di **filtro** in base al tipo dell'oggetto: tramite una *Combobox* si sceglie il tipo per cuio si desidera filtrare, mentre per rimuovere i filtri si sceglie il valore *None*.

- A destra è presente il widget *DetailsWidget*, il quale ha il compito di presentare in maniera visiva gli tutti gli attributi del media selezionato dall'utente dalla lista di *ItemListWidget*. *DetailsWidget* crea dinamicamente per ogni tipo di media una struttura grafica differente, e mette a disposizione due pulsanti usati per **modificare** il media attualmente selezionato o **eliminare** tale media. Scegliendo di modificare un media esistente, i campi di inserimento presenti in *EditWidget* verranno precompilati con le informazioni attuali del media che si sta modificando.

Il widget *EditWidget* permette all'utente di creare o modificare un media: si compone di una serie di campi di input che riflettono gli attributi comuni a

tutti i media, e una *Combobox* che permette di selezionare il tipo di media che si desidera creare. In base alla scelta verranno mostrati i campi specifici per quel tipo di media. Nel caso si scelga di modificare un media esistente, *EditWidget* si appoggia a due visitor per recuperare tutte le informazioni di tale media: *TypeSelector* rileva il tipo del media che si sta modificando e imposta la *Combobox* sull'editor corretto, mentre *ItemEditInjector* imposta i valori specifici del media nei campi di input dell'editor.

3 Polimorfismo

Il polimorfismo è stato realizzato sfruttando ampiamente il design pattern **Visitor** nella *View*:

- *AbstractItemEditWidget*: rappresenta un'astrazione degli editor di ogni media e implementa un metodo virtuale **create** che viene poi specializzato nelle classi figlie. Inoltre la classe *EditWidget* sfrutta due classi **visitor** (ossia *TypeSelector* e *ItemEditInjector*) per decidere, quanto si sta modificando un media esistente, il tipo del media: in tal modo si può mostrare fin da subito l'editor corretto e riempire i campi dati.
- *ItemCardWidget* è la classe responsabile della creazione delle **card** che sono inserite all'interno della lista dei media. Tale classe deriva dalla classe *Widget* di *Qt* e dalla classe *ConstItemVisitor*: in tal modo *ItemCardWidget* implementa i metodi **visit** in modo da assegnare a ciascun tipo di media un icona appropriata.
- *DetailsWidget* è la classe responsabile di mostrare a schermo i dettagli del media selezionato. Anche questa classe deriva dalla classe *Widget* di *Qt* e dalla classe *ConstItemVisitor*: in tal modo *DetailsWidget* implementa i metodi **visit** in modo da presentare le informazioni di ciascun tipo di media in maniera differente.

4 Persistenza dei dati

La persistenza dei dati è stata implementata attraverso lo standard **JSON**: per poter iniziare a creare media è **necessario** che sia aperto un file JSON. Ciò avviene quando viene creato un file vuoto o quando viene aperto un file esistente: solo dopo aver svolto una di queste operazioni il programma permetterà di creare nuovi media. Tutte le informazioni riguardanti i media vengono salvate nel file JSON in uso: in aggiunta viene inserito un campo *type* che denota il tipo di media salvato. Tale campo *type* viene utilizzato durante il caricamento di media da file per poter assegnare il tipo corretto ai nuovi oggetti.
Nella cartella del progetto viene messo a disposizione un file di test JSON precompilato, contenente alcuni esempi di media per ogni tipo (file *test_file.json*).

5 Funzionalità implementate

Le funzionalità implementate nel progetto sono le seguenti:

- Gestione di **tre** tipi di media e relative operazioni *CRUD*: creazione, ricerca, modifica, eliminazione.
- Gestione della **persistenza** dei dati in formato JSON: lettura e salvataggio dei media in file JSON.
- **Toolbar** delle operazioni disponibili.
- **Scorciatoie** da tastiera per rendere la navigazione più semplice.
- Utilizzo di **icone** per rendere la navigazione semplice e comprensibile.
- Visualizzazione dei media sotto forma di **lista** scorrevole e interattiva.
- Barra di ricerca dei media in **tempo reale**.
- Visualizzazione dei **dettagli** di un media, diversificato in base al tipo di media selezionato.
- Finestre e widget si adattano al ridimensionamento.
- Navigazione delle diverse schermate gestita nella **finestra principale**, limitando le finestre pop-up al minimo.
- Controllo di modifiche **non salvate** in chiusura del programma.
- Controllo di **unicità** degli ID.
- Supporto delle immagini per i media.
- Funzionalitá di filtro dei media per tipo.

6 Rendicontazione ore

Attività	**Ore previste**	**Ore effettive**
Studio e progettazione	5	7
Sviluppo del *Model*	10	8
Studio del framework *Qt*	5	8
Sviluppo della *View*	10	15
Test e debug	5	5
Stesura della relazione	5	3
Totale	40	46

Tabella 1: Rendicontazione ore utilizzate

La stima iniziale delle ore necessarie è stata superata: in particolare, lo studio del framework *Qt* e la progettazione concettuale dell'applicazione hanno richiesto più tempo del previsto. Parte del tempo è stato speso anche per comprendere a pieno il funzionamento del pattern Visitor, che si è rivelato cruciale

nello svolgimento del progetto.
La scrittura del codice del *Model* si è conclusa prima di quanto previsto, anche grazie alla suite di strumenti messi a disposizione da *Qt* per interagire con i file JSON. Al contrario, lo sviluppo della View ha richiesto un monte ore maggiore: si sono verificati problemi nella gestione della visualizzazione dei dettagli dei media e nella lista dei media.

7 Conclusioni

Lo svolgimento di questo progetto, assieme alle lezioni del corso di Programmazione ad Oggetti, mi ha fatto molto apprezzare il linguaggio C++ per la sua versatilità e anche per la sua difficoltà iniziale. Sono stato molto felice di aver potuto mettere in pratica ogni aspetto del linguaggio imparato a lezione, e di aver avuto la possibilità di apprendere nuovi aspetti del C++ attraverso la sua estensiva documentazione.
Durante lo svolgimento del progetto ho anche preso confidenza con tool di analisi del codice come **Valgrind**, che mi ha molto aiutato nell'individuare le cause di errori legati alla memoria e nella loro correzione. Ritengo dunque che lo svolgimento di questo progetto sia stato di grande beneficio sia didattico che di arricchimento personale.