QtLibrary

Alberto Suar — Matricola 2101051

11 giugno 2025

Indice

1 Descrizione del progetto

QtLibrary è un'applicazione sviluppata in **C++**, utilizzando il framework grafico **Qt**, con lo scopo di fornire un software interattivo e user-friendly che consenta agli utenti di *ricercare*, *aggiungere*, *modificare* e *rimuovere contenuti* da una biblioteca virtuale multimediale.
Le tipologie di contenuti attualmente supportate sono **Libri**, **Film**, **Manga** e **Anime**, ciascuna dotata di attributi specifici. Il progetto è stato concepito con una forte *impronta scalabile*, rendendo semplice l'aggiunta di nuovi tipi di contenuto in futuro, sia dal punto di vista logico che grafico.
La progettazione ha seguito rigorosamente i principi fondamentali della **Programmazione Orientata agli Oggetti**, adottando con coerenza i concetti di *Incapsulamento*, *Astrazione*, *Ereditarietà* e *Polimorfismo*. Sono stati utilizzati pattern come `Factory` per l'istanziazione dinamica dei contenuti e `Visitor` per l'elaborazione di comportamenti specifici senza ricorrere al `type checking`.
Particolare attenzione è stata dedicata allo sviluppo della **Graphical User Interface (GUI)**, progettata per prevenire errori comuni e garantire una navigazione fluida e intuitiva. Sono stati analizzati e gestiti scenari potenzialmente problematici, come:

- L'interruzione della creazione di un contenuto passando alla sezione di ricerca.
- La ricerca di un contenuto non presente in memoria.
- Il tentativo di modificare più contenuti simultaneamente.

Per affrontare queste situazioni, sono state integrate **finestre di dialogo contestuali**, che informano l'utente delle operazioni possibili e delle limitazioni momentanee. La GUI reagisce in modo dinamico alle azioni dell'utente: componenti non compatibili vengono disabilitati automaticamente, e il passaggio tra le varie modalità (ricerca, inserimento, modifica) avviene in modo coerente, evitando conflitti di stato.
La persistenza dei dati è stata implementata mediante il formato `JSON`, garantendo compatibilità e facilità di estensione. Tutti i contenuti vengono serializzati e deserializzati automaticamente, mantenendo la coerenza con la struttura delle classi.
Infine, il codice è stato strutturato con un'attenzione particolare alla **manutenibilità**, separando la logica dell'applicazione dalla presentazione e favorendo il riuso dei componenti. L'interfaccia è progettata in modo **user-centric**, adattandosi ai comportamenti dell'utente e semplificando operazioni complesse attraverso meccanismi visivi e interattivi.
Il risultato è un'applicazione solida, estendibile e pronta per futuri sviluppi, con una base architetturale robusta e un'interfaccia curata nei dettagli.
Lo sviluppo di **QtLibrary** ha richiesto una significativa fase di **studio e approfondimento**, sia sul piano teorico che pratico. In particolare, è stato necessario acquisire familiarità con il framework **Qt**, affrontando temi come la gestione dei segnali e slot, la progettazione di interfacce reattive, la manipolazione dei layout e la gestione dei widget dinamici.
Parallelamente, il progetto ha previsto l'impiego di concetti avanzati della programmazione orientata agli oggetti, come l'utilizzo del pattern **Visitor** per la gestione polimorfica dei contenuti, evitando l'uso di meccanismi espliciti di `type checking`. Questo ha comportato uno studio approfondito dei *design pattern*, con particolare attenzione a quelli orientati alla scalabilità e alla separazione delle responsabilità, come `Visitor`, e altre soluzioni architetturali.
L'intero processo di sviluppo ha rappresentato un'opportunità formativa concreta, permettendomi di conoscere concetti nuovi della programmazione orientata agli oggetti e anche di analizzare il processo di sviluppo per poter costruire da zero una applicazione.

2 Modello Logico

Per comprendere la struttura delle classi, si consiglia di fare riferimento all'immagine `UML_PaO.jpeg` fornita in allegato. Di seguito, viene descritto l'approccio adottato nella progettazione del modello logico dell'applicazione, con particolare enfasi sul ruolo del **ContentManager**.
Dal punto di vista implementativo, una scelta ricorrente è stata l'utilizzo di **enumerazioni** e **mappe associative** per rappresentare attributi discreti e controllati, come le **Lingue**, i **Generi**, le **Risoluzioni**, le **Copertine** e le **Cadenze**. Ogni classe concreta (Libro, Film, Manga, Anime) dispone di una propria definizione specifica di genere, mantenuta all'interno di una mappa costante che associa l'enumerazione ad una stringa rappresentativa. Questo approccio ha permesso di vincolare l'input dell'utente a valori validi e coerenti, evitando errori di inserimento che avrebbero potuto compromettere la qualità dei dati e l'efficacia della funzionalità di ricerca. Ad esempio, senza questo controllo, l'utente avrebbe potuto inserire un genere arbitrario per un film, rendendo tale contenuto non più ricercabile tramite i filtri prestabiliti.
Elemento centrale della gestione logica dei dati è il **ContentManager**, ovvero il modulo responsabile della gestione unificata dell'intero insieme di contenuti multimediali. L'idea chiave alla base del suo funzionamento è l'utilizzo di un sistema di **indicizzazione basato su vettori separati per ogni tipologia di contenuto**, in cui l'ordine fisso delle categorie (Libri $\rightarrow$ 0, Manga $\rightarrow$ 1, Film $\rightarrow$ 2, Anime $\rightarrow$ 3) viene utilizzato come riferimento costante. In questo modo, ciascuna operazione – che si tratti di creazione, ricerca, modifica o eliminazione – è parametrizzata da un indice intero che rappresenta la categoria di appartenenza.
Questo design ha permesso di evitare l'introduzione di un attributo esplicito del tipo `tipoContenuto` all'interno della classe base, delegando invece tale responsabilità ad un **Visitor specializzato**, denominato **IndexVisitor**. Questo visitor implementa una logica di dispatch polimorfa, che associa dinamicamente ciascun oggetto derivato al suo indice corrispondente, mantenendo così separata la logica di classificazione dal modello dei dati stesso. Questo approccio rispetta il principio di apertura/chiusura e rende il sistema altamente scalabile: l'aggiunta di nuove tipologie richiede modifiche minime, localizzate unicamente all'interno del visitor e del ContentManager.
Un ulteriore aspetto chiave del ContentManager è la gestione della **unicità dei contenuti in memoria**. È stato progettato un sistema di verifica dei duplicati che impedisce l'inserimento o la modifica di contenuti già presenti. La verifica avviene confrontando tutti gli attributi semanticamente rilevanti (eccetto l'immagine, che non viene utilizzata come discriminante), garantendo che ogni contenuto presente nella biblioteca virtuale sia univoco. Questo ha consentito di mantenere l'integrità dell'archivio e di evitare ambiguità durante la ricerca o la modifica dei dati.
Infine, si segnala che tutte le operazioni del ContentManager, ad eccezione di quelle dedicate alla persistenza su file `.json` (che saranno discusse nella sezione **Persistenza dei dati**), sono basate su questa logica di gestione indicizzata, la quale ha costituito il fondamento dell'intera architettura del progetto.

3 Polimorfismo

L'adozione del polimorfismo rappresenta uno degli aspetti cardine e più significativi del progetto, incidendo profondamente sia sulla logica applicativa sia sulla progettazione della GUI.

CreationVisitor

Il **CreationVisitor** è uno dei visitor principali all'interno della logica applicativa, responsabile della specializzazione degli oggetti durante la loro creazione. Esso opera su un contenuto inizialmente impostato con valori di default, al quale viene passata una mappa di attributi definiti dall'utente. Invocando i metodi *setter* specifici per ogni tipo di contenuto, il visitor assegna correttamente i valori ai relativi attributi, personalizzando così l'istanza. Questo visitor è utilizzato nella fase di creazione, dopo la quale il contenuto viene inserito nel vettore corrispondente, seguendo la medesima logica di indicizzazione illustrata nella sezione **Modello Logico**.

SearchVisitor

Il **SearchVisitor** funziona in modo analogo al *CreationVisitor*, utilizzando anch'esso una mappa di attributi fornita dall'utente, ma con finalità di ricerca. In particolare, viene impiegato per la ricerca avanzata, sfruttando i filtri basati sui contenuti inseriti dall'utente. La procedura inizia con l'identificazione dei contenuti che corrispondono esattamente al nome fornito nella barra di ricerca, limitando l'indagine al vettore associato al tipo selezionato. Successivamente, vengono confrontati gli attributi della mappa con quelli degli oggetti candidati, restituendo un valore booleano `true` per i contenuti che soddisfano tutti i criteri di ricerca.

CheckVisitor

Il **CheckVisitor** condivide molte somiglianze con il *SearchVisitor*, ma, data la diversa finalità e il contesto di utilizzo, è stato definito come classe distinta. La sua funzione principale è garantire l'unicità dei contenuti all'interno della memoria. Per ogni contenuto da verificare, il visitor confronta sistematicamente gli attributi con quelli di tutti gli elementi presenti nel vettore corrispondente. Se viene identificato un contenuto con tutti gli attributi coincidenti (esclusa l'immagine, non rilevante ai fini della verifica), restituisce `true`, impedendo così la creazione o la modifica che porterebbe a duplicati indesiderati.

IndexVisitor

L'**IndexVisitor** ha un ruolo semplice ma fondamentale: data l'istanza di un contenuto, restituisce l'indice numerico associato alla sua categoria, seguendo l'ordinamento definito nel modello logico (Libri = 0, Manga = 1, Film = 2, Anime = 3). Questa informazione viene utilizzata per individuare rapidamente il vettore in cui operare.

FilterVisitor

Il **FilterVisitor** si occupa della costruzione dinamica dei widget utilizzati per la visualizzazione dei risultati all'interno dell'applicazione. Riceve come input un contenuto e genera il filtro corrispondente, utilizzando il polimorfismo per identificare con certezza il tipo di contenuto trattato. In questo processo, sfrutta i metodi polimorfici `fromObjectToMap()` e `setAttributes()`. Il primo costruisce una mappa degli attributi a partire dall'oggetto contenuto, mentre il secondo imposta i valori dei widget filtro basandosi su tale mappa, permettendo così all'utente di visualizzare e interagire con i risultati della ricerca in modo coerente e tipizzato.

Oltre ai visitor, altri metodi polimorfici importanti nell'applicazione includono:

- `reset()`: ripristina i filtri alla loro condizione iniziale, operando sia nella fase di creazione che in quella di ricerca, facilitando il passaggio fluido tra diverse sezioni dell'interfaccia.
- `mostraFiltro()`: differenzia il comportamento di visualizzazione tra modalità *Creation* e *Search*, mostrando in quest'ultima un messaggio di avvertimento all'utente qualora si tenti un'azione non consentita.

L'intera architettura polimorfica è stata progettata in modo da evitare la necessità di utilizzare `dynamic_cast`, garantendo al contempo la massima sicurezza e correttezza nell'assegnazione dei comportamenti specifici a ciascun tipo di contenuto. Questo ha permesso di mantenere il codice pulito, estensibile e conforme ai principi di programmazione orientata agli oggetti.

4 Persistenza dei Dati

La persistenza dei dati nel progetto è gestita tramite la classe `JsonHandler`, che si occupa di salvare e caricare le informazioni in formato JSON utilizzando le classi del framework Qt.

Salvataggio e caricamento degli enum

La classe fornisce un metodo template `saveEnumToJson` che permette di serializzare in un file JSON una mappa di enumerazioni associate a stringhe. Questa funzione:

- Apre (o crea) un file JSON specificato da un percorso;
- Carica il contenuto esistente per evitare di sovrascrivere altri dati presenti;
- Scrive un array JSON contenente gli oggetti con i campi `"nome"` (la rappresentazione testuale) e `"enum"` (il valore numerico dell'enumerazione);
- Salva il file aggiornato con indentazione per una migliore leggibilità.

Il caricamento avviene tramite il metodo `loadEnumFromJson`, che prende in input il percorso del file, la chiave JSON da cui leggere e un puntatore a una `QComboBox`. Il metodo:

- Apre e legge il file JSON;
- Estrae l'array corrispondente alla chiave specificata;
- Per ogni elemento dell'array aggiunge una voce nella `QComboBox` con il nome visualizzato e il valore enum associato come dato utente.

Estrazione di array di mappe da JSON

Il metodo `estraiArrayDiMappe` consente di caricare da un file JSON un array di oggetti, convertendoli in un `std::vector` di `std::unordered_map<std::string, std::string>`. Ogni oggetto JSON viene trasformato in una mappa chiave-valore, dove sia le chiavi che i valori sono stringhe standard C++. Questa funzione è utile per ricostruire i dati in una forma generica indipendente dal tipo concreto di contenuto.

Salvataggio di contenuti

La funzione `salvaArrayDiMappe` accetta un vettore di puntatori a oggetti `Contenuto`, ed effettua il salvataggio dei dati in un file JSON come array di oggetti. Per ogni contenuto:

- Viene invocato il metodo polimorfico `fromObjectToMap()`, che converte l'oggetto in una mappa di stringhe rappresentanti gli attributi e i rispettivi valori;
- Questa mappa viene trasformata in un `QJsonObject`;
- Ogni `QJsonObject` viene aggiunto all'array JSON complessivo.

Il file viene quindi scritto in modo sicuro, con sovrascrittura e formattazione indentata.

Selettore interattivo dei file JSON

Per migliorare l'usabilità, è stata implementata la classe `JsonSelector`, un widget Qt che permette all'utente di selezionare i file JSON da cui caricare i dati relativi ai diversi tipi di contenuto (libri, manga, film, anime).
Il widget presenta:

- Quattro pulsanti per il caricamento di ciascun tipo di contenuto, ciascuno associato all'apertura di una finestra di dialogo per la selezione di un file JSON;
- Un pulsante `Avvia Applicazione` che controlla che tutti i file siano stati selezionati correttamente, mostrando un messaggio di errore in caso contrario, ed emette un segnale per avviare il caricamento dei contenuti.

Ogni selezione di file salva un percorso relativo al file JSON scelto, facilitando così la gestione dei dati nell'ambito dell'applicazione e mantenendo un riferimento compatto e portabile.

Considerazioni generali

L'uso di `unordered_map<string, string>` per la serializzazione degli attributi consente una gestione generica e flessibile degli oggetti, senza dipendere strettamente da classi specifiche o strutture rigide. L'integrazione con i widget Qt, sia per la gestione degli enum (tramite `QComboBox`) sia per la selezione interattiva dei file JSON, permette una perfetta sinergia tra la persistenza dei dati e l'interfaccia utente. Questo sistema di persistenza garantisce:

- Estensibilità verso nuovi tipi di contenuto, grazie all'approccio polimorfico e generico;
- Facilità di manutenzione e aggiornamento dei dati in formato leggibile e strutturato;
- Una UX chiara e controllata nella fase di caricamento dati.

5 Funzionalità Aggiuntive

5.1 Utilizzo di Finestre di Dialogo per fornire interattività

Per migliorare l'esperienza utente e la gestione degli errori nell'applicazione *QtLibrary*, sono stati implementati diversi widget di dialogo personalizzati per la visualizzazione di messaggi di errore e conferma. Tali widget estendono la classe `QDialog` di Qt e offrono un'interfaccia grafica coerente e stilizzata per notificare all'utente situazioni specifiche durante l'uso dell'applicazione.

5.1.1 Architettura e Implementazione

Tutti i messaggi di errore e le notifiche di successo derivano da una classe base chiamata `ErrorStructure`, che incapsula la struttura comune a tutti i dialoghi, come la label del testo e il layout verticale principale. La finestra è configurata per essere modale e priva di bordo, in modo da focalizzare l'attenzione dell'utente. Ogni tipo di errore o messaggio è rappresentato da una sottoclasse specifica che personalizza il testo del messaggio e i pulsanti di azione disponibili. Ad esempio:

- **ErrorChanging**: Notifica un avviso all'utente quando l'azione in corso comporta la perdita di tutti i filtri inseriti. Include i pulsanti *Conferma* e *Annulla* per proseguire o interrompere l'azione.
- **ErrorClosing**: Chiede conferma prima di uscire dall'applicazione.
- **ErrorDuplicate**: Avvisa che un contenuto con lo stesso titolo è già presente, bloccando l'operazione di inserimento o modifica.
- **ErrorMissing**: Indica che uno o più filtri obbligatori non sono stati dichiarati, pertanto l'operazione non è stata effettuata.
- **ErrorNoResult**: Informa l'utente che la ricerca non ha prodotto alcun risultato.
- **ErrorNoTitle**: Segnala che non è stato inserito un titolo, impedendo l'operazione.
- **ErrorNoFile**: Indica che uno o più file JSON necessari per il caricamento non sono stati specificati.
- **MessageSuccess**: Conferma all'utente il successo di un'operazione, come l'inserimento o la modifica di un contenuto.

5.1.2 Aspetti grafici e interattivi

Tutti i dialoghi utilizzano uno stile uniforme, con testo bianco su sfondo grigio scuro e bordi blu, mantenendo un carattere monospaziato per migliorare la leggibilità del testo. I pulsanti di conferma sono verdi o blu scuro, mentre quelli di annullamento sono rossi, per favorire una comprensione immediata da parte dell'utente.
L'interazione è gestita tramite `QButtonGroup` che consente di connettere il clic su un pulsante a specifici slot per l'emissione di segnali personalizzati, così da permettere alla finestra principale o ad altri componenti di reagire adeguatamente alle scelte dell'utente.

5.1.3 Utilizzo nel flusso applicativo

Questi widget di errore e messaggi vengono richiamati in varie fasi dell'applicazione, ad esempio:

- Prima di procedere con operazioni che comportano la perdita di dati temporanei, come la cancellazione dei filtri.
- In caso di tentativi di inserimento di contenuti duplicati.
- Se mancano dati obbligatori per completare un'operazione.
- Quando si avviano processi che richiedono file JSON e tali file non sono stati specificati.
- Per confermare l'uscita dall'applicazione.

In questo modo si migliora la robustezza dell'interfaccia, prevenendo errori e offrendo all'utente un feedback chiaro e immediato.

5.2 Ricerca Parziale e Case-Insensitive

La funzionalità di ricerca avanzata è stata progettata con particolare attenzione all'usabilità e alla flessibilità, implementando due caratteristiche fondamentali: la ricerca parziale e la ricerca case-insensitive. La ricerca *parziale* consente di ottenere risultati anche quando il testo inserito dall'utente corrisponde solo a una parte del valore presente negli attributi degli elementi. Ad esempio, nel caso di un manga il cui campo `mangaka` fosse valorizzato con *"Kentaro Miura"*, una ricerca contenente semplicemente *"Miura"* avrebbe comunque prodotto un risultato positivo. Ciò garantisce un'esperienza di ricerca più tollerante e naturale, evitando la necessità di inserire esattamente l'intera stringa.
Parallelamente, la ricerca è stata resa *case-insensitive*, ovvero non sensibile alle maiuscole o minuscole. Questo significa che, indipendentemente dal modo in cui l'utente digita la query (*"miura"*, *"Miura"*, *"MIURA"*), i risultati saranno comunque trovati correttamente. Questa scelta aumenta ulteriormente la comodità d'uso, riducendo gli errori dovuti a differenze di maiuscole.
Inoltre, il sistema di ricerca è stato concepito per supportare ricerche basate su filtri multipli, ma con un comportamento molto flessibile anche in presenza di filtri parziali o incompleti. Ad esempio, qualora un utente desiderasse effettuare una ricerca specificando solamente l'anno di pubblicazione di un libro, lasciando tutti gli altri filtri indefiniti, il motore di ricerca implementato dal `SearchVisitor` restituirebbe tutti gli elementi del tipo selezionato che corrispondono a quell'anno specifico, indipendentemente dal valore degli altri attributi. In altre parole, la ricerca filtra gli elementi in base ai criteri esplicitamente indicati dall'utente, ignorando quelli non specificati, senza imporre una corrispondenza totale su tutti i campi.
Questa modalità consente una ricerca molto granulare e personalizzabile, in cui l'utente può combinare diversi filtri a piacimento, ottenendo sempre un insieme coerente e pertinente di risultati. L'approccio adottato rende quindi il sistema di ricerca robusto, intuitivo e adatto a scenari d'uso reali, in cui spesso non si dispone di tutte le informazioni o non si desidera limitarne eccessivamente il campo di ricerca.

6 Rendicontazione delle ore di sviluppo del progetto

Attività	**Ore previste**	**Ore effettive**
Studio e progettazione	8	10
Sviluppo del codice del modello	4	6
Studio del framework Qt	5	13
Sviluppo del codice della GUI	15	30
Test e debug	5	15
Stesura della relazione	3	2
Totale	**40**	**76**