

Libreria Multimediale

Oggetto: Relazione progetto Programmazione a Oggetti

Studente: Difino Andrea, mat. 2101072

Titolo: Libreria Multimediale

1 Introduzione

Libreria Multimediale è un gestionale che consente di aggiungere, modificare, cancellare e cercare i media presenti al suo interno. I contenuti gestibili sono di diversa natura: libri, film e musica, ciascuno caratterizzato da attributi specifici e da una propria modalità di visualizzazione.

L'applicazione è dotata di un'interfaccia grafica moderna, progettata per essere **user-friendly**. Sono stati utilizzati contrasti visivi elevati per mettere in risalto sia i media che le principali funzionalità, come la cancellazione, la modifica e la visualizzazione dei contenuti.

Il motore di ricerca si basa su un algoritmo semplice, che utilizza il nome del media come chiave di ricerca. Per migliorarne l'efficacia, è stato integrato un sistema basilare di filtraggio che consente di selezionare gli elementi in base alla tipologia desiderata (libro, film o musica).

L'impiego di diverse tipologie di media ha rappresentato inoltre un'ottima occasione per applicare il polimorfismo in maniera concreta e non banale, come richiesto dalle specifiche, consentendo di gestire ciascun tipo in modo modulare ma coerente.

2 Descrizione del modello

Il modello logico si articola in due parti: la gestione dei media e il loro salvataggio in un file JSON.

La prima parte comprende le classi che descrivono i media e la libreria, come illustrato nel diagramma in Figura 1. La seconda include le classi di supporto necessarie per il salvataggio, la modifica e la gestione del file JSON. A tal fine vengono utilizzati gli strumenti offerti da Qt, in particolare *QJsonObject* e *QJsonArray*, per rappresentare i media in un formato compatibile con JSON.



Figura 1: Diagramma UML delle classi del modello

Il modello parte da una classe base astratta, *Item*, che rappresenta le informazioni comuni a tutti i media gestibili: titolo, descrizione, anno, id e il campo booleano *isSaved*.

Il campo statico *idCount* è una semplice variabile intera di appoggio che tiene traccia del numero totale di *Item*, evitando soluzioni più complesse a livello computazionale. L'id viene assegnato a ciascun *Item* come valore intero incrementale, utile soprattutto per identificare e gestire l'eliminazione dei media.

Infine, il campo *isSaved* si è rivelato fondamentale per la gestione del visitor, poiché permette di applicarlo solo ai media effettivamente "nuovi" (non ancora salvati) e ha risolto il problema della duplicazione dei media all'interno del *QJsonArray*.

Per ognuno di questi campi sono implementati i metodi *getter* e *setter*, fatta eccezione per *idCount* che non ha un metodo setter in quanto variabile statica che viene inizializzata a 0 al momento della inizializzazione della libreria.

Il metodo *getExtendedLength* utilizzato in entrambe le classi *Film* e *Music* restituisce la loro durata scritta nel formato "hh:mm:ss" utile per la visualizzazione estesa nella GUI.

Poiché le classi del modello si comportano come dei *Data Transfer Object* (DTO) e non espongono alcuna funzionalità rilevante, si è scelto di utilizzare il *design pattern Visitor* per consentirne la memorizzazione dinamica nel file JSON. A tal fine è stata realizzata la classe base astratta *VisitorInterface*. Di conseguenza, in *Item* è stato inserito il metodo virtuale puro *accept* per accettare il *Visitor*.

Vorrei marcare la presenza di una funzione *isValidJson* nel file *JSONFileHandler* che ha il compito di verificare che i dati contenuti in un file JSON siano strutturati correttamente e compatibili con l'applicazione. Essa riceve in input un array di oggetti JSON (*QJsonArray*) e controlla che ogni elemento sia un oggetto valido contenente tutti i campi richiesti, sia comuni (come id, title, description, year) sia specifici in base al tipo di media (*Music*, *Film*, *Book*).

Per ciascun tipo, la funzione verifica inoltre che i campi abbiano il formato corretto (ad esempio, che durata sia un numero per i film e i brani musicali, o che author sia una stringa per i libri).

In caso di errore, la funzione restituisce false e stampa un messaggio diagnostico utile al debugging tramite qDebug(). Questo processo consente di prevenire la lettura e l'elaborazione di file JSON non conformi, garantendo la robustezza dell'applicazione.

3 Polimorfismo

L'utilizzo principale del polimorfismo si riscontra nell'adozione del design pattern *Visitor* all'interno della gerarchia *Item*. Le classi *Book*, *Film* e *Music* estendono la classe base *Item* e implementano il metodo *accept*, permettendo così di accettare un oggetto *Visitor*.

Il *Visitor* è rappresentato dalla classe *JSONVisitor*, che implementa l'interfaccia *VisitorInterface* definendo il metodo *visit* per ciascun tipo di media. Questo consente di serializzare ogni tipo di contenuto in un oggetto *QJsonObject* specifico. La serializzazione e la deserializzazione sono delegate a una classe ausiliaria, *JSONSerializer*, che si occupa della conversione bidirezionale tra oggetti *Item* e dati in formato JSON.

Il pattern Visitor è stato utilizzato anche per la gestione della componente grafica dell'applicazione. In particolare, sono state definite due classi che implementano l'interfaccia *VisitorInterface*:

ModifyWidgetVisitor: si occupa della creazione dei widget per la modifica dei media, specifici per ciascun tipo di media;

OverviewWidgetVisitor: gestisce la visualizzazione delle informazioni dettagliate relative a ciascun media.

In particolare, la classe **ModifyWidgetVisitor** implementa il metodo *areInputsValid* che verifica che i campi di testo non siano vuoti.

Oltre al pattern Visitor, è stato impiegato anche il design pattern *Observer* per gestire dinamicamente le modifiche alla home page dell'applicazione. La classe **MainWidget**, oltre a *QWidget*, implementa l'interfaccia *ObserverInterface* e definisce i metodi *addW_notification* e *removeW_notification*. Questi vengono invocati dalla classe *JSONVisitor* quando si fa il loading di un file JSON e serve quindi l'aggiornamento dei widget nella GUI.

4 Persistenza dei dati

Per la persistenza dei dati viene utilizzato il formato JSON, un unico file per tutti i media, contenente un vettore di oggetti. Gli oggetti sono perlopiù semplici associazioni chiave-valore, e la serializzazione delle sottoclassi viene gestita aggiungendo un attributo "type". Un esempio della struttura dei file è dato dai JSON forniti assieme al codice, in particolare "defaultDB.json" contiene due prodotti per ciascuna tipologia, in modo da illustrare brevemente le diverse strutture.

5 Funzionalità implementate

Le funzionalità implementate sono, per semplicità, suddivise in due categorie:

Funzionali:

- Gestione di 3 tipi di media
- Conversione e salvataggio in JSON
- Implementazione di una funzione per il controllo della compatibilità dei file JSON importati
- Funzionalità di ricerca basata sul nome

- Sistema di filtraggio semplice basato sul tipo dell'oggetto
- Il tasto salva funziona sia per il salvataggio del file sia per la creazione di un nuovo database nel caso in cui non sia stato ancora creato

Grafiche:

- Sezione superiore con pulsanti per il salvataggio, upload, creazione database e creazione dei media
- Utilizzo di icone per quasi tutti i pulsanti in modo da rendere la GUI più semplice e visivamente più appagante
- Controllo dinamico del testo del pulsante per l'aggiunta dei media, basato sulla pagina corrente
- Controllo della presenza di modifiche non salvate prima di uscire
- Utilizzo di immagini chiare per l'identificazione del tipo di media
- Utilizzo di colori e stili grafici
- Effetti grafici come cambio del colore al passaggio del mouse
- Utilizzato il *QFlowLayout* di Qt per la visualizzazione dinamica dei media, che consente di adattare la visualizzazione in base alla dimensione della finestra

6 Rendicontazione delle ore

Attività	Ore Previste	Ore Effettive
Studio e progettazione	9	10
Sviluppo del codice del modello	8	10
Studio del framework Qt	5	5
Sviluppo del codice della GUI	10	14
Test e debug	6	6
Stesura della relazione	2	2
totale	40	47

Il superamento del monte ore previsto è stato contenuto e dovuto principalmente a difficoltà incontrate nella creazione della GUI e nella gestione del salvataggio dei media per la base di dati.

7 Differenze rispetto alla consegna precedente

Rispetto alla consegna precedente ho implementato alcune delle funzionalità mancanti e ho migliorato la GUI con l'utilizzo di immagini personalizzate.

Andando in ordine ho apportato le seguenti modifiche affiancate dal punto negativo della consegna precedente a cui si riferiscono:

- **(Migliorabile con scorciatoie da tastiera):** Ho aggiunto delle scorciatoie da tastiera nella home dell'applicazione. In particolare, sono state implementate le seguenti combinazioni:
 1. **Ctrl + S:** salvataggio del file e creazione di un nuovo file nel caso non esistesse.
 2. **Ctrl + N:** creazione di un nuovo file JSON.
 3. **Ctrl + O:** caricamento di un file JSON esistente.

- **(Migliorabile con utilizzo pertinente di immagini o altri elementi multimediali):** Ho introdotto la possibilità di associare immagini personalizzate agli *Item*. Di conseguenza, la classe **Item** è stata aggiornata per includere il campo `image_path`. Per la gestione delle immagini è stata aggiunta la classe **ImageHandler**.

Per creare un *Item* con immagine personalizzata, basta cliccare sul pulsante "Aggiungi Immagine", il cui testo cambia in "Immagine Selezionata!" una volta completata la selezione. Le immagini personalizzate sono visibili solo in due contesti:

- Quando si accede alla pagina di visualizzazione completa dell'*Item*, tramite l'icona "occhio" presente in basso a sinistra di ogni *ItemCard*.
- Quando si apre la pagina di modifica dell'*Item*, dove è disponibile anche una preview dell'immagine prima del salvataggio.

Ho scelto di non mostrare le immagini direttamente nelle Card nella home per mantenere un'interfaccia più pulita.

Le immagini vengono copiate e salvate nella cartella `itemsCover/`, e il relativo path è memorizzato nel file JSON come percorso relativo (es. `itemsCover/nomeImmagine`). Per ogni **Item** eliminato, anche l'immagine associata viene rimossa, evitando così inutili sprechi di memoria. In caso di immagini con lo stesso nome, viene automaticamente aggiunto un suffisso numerico incrementale per evitare conflitti.

- **(Migliorabile con altre funzionalità pertinenti):** Ho aggiunto la possibilità di mettere "mi piace" agli *Item*. Facendo doppio clic su una **ItemCard** nella home, compare un'icona animata che rappresenta lo stato di "like" (non il cuore standard ma una stellina). La presenza dell'icona indica se l'*Item* è stato apprezzato o meno.

Per supportare questa funzionalità, è stata aggiunta una nuova proprietà `liked` nel file JSON, e la classe **Item** è stata aggiornata con un nuovo campo pubblico booleano `is_liked`, inizializzato a `false` di default.

- **(Altre funzionalità):** Ho migliorato sia l'aspetto estetico che l'efficienza dei filtri nella home. Ora è inoltre possibile filtrare e visualizzare esclusivamente gli *Item* marcati come preferiti.
- **(Piccole modifiche):** Ogni *Item* viene ora salvato immediatamente non appena viene modificato. Di conseguenza, se l'applicazione viene chiusa senza salvare, l'unico caso possibile è quello della creazione di un nuovo *Item* non ancora salvato. Questa modifica semplifica notevolmente la gestione dell'eliminazione delle immagini in caso di cancellazione di un *Item* non salvato.