Relazione Progetto Programmazione ad Oggetti: MediaManager

Giuseppe De Fina, matricola 2113187

Luglio 2025

Il codice è disponibile su **GitHub** all'indirizzo: `https://github.com/blavkice/MediaManager`.

1 Introduzione

MediaManager è un gestore di media che permette di creare, modificare, cancellare, cercare, visualizzare, importare ed esportare quest'ultimi. Nella versione attuale, i prodotti che è possibile gestire sono ***Poesie*** e ***Libri*** (appartenenti alla categoria "Letteratura") ma anche ***Articoli di Giornale*** e ***Articoli Accademici*** (appartenenti alla categoria "Articoli").

Due punti di forza del gestore di media sono: **l'interfaccia UI** minimalista e molto semplice che permette anche di visualizzare gli elementi con due viste differenti (una classica a "lista" e una *fullscreen* a "grid") e la **gestione delle immagini integrata**: una volta assegnata l'immagine ad un *media*, non servirà più conservare l'immagine di partenza poiché la sua gestione è demandata al programma. Inoltre, il programma utilizza estensivamente il ***visitor*** *design pattern* che garantisce una piena estensibilità (è semplicissimo aggiungere in futuro altri tipi di *media*) e anche gli **smart pointers** che permettono di minimizzare al massimo gli errori anche dopo prolungate sessioni di utilizzo.

2 Descrizione del Modello

Il modello progettuale di **MediaManager** si basa su una gerarchia di classi che rappresentano le diverse tipologie di media gestite dal programma. Alla base di tutto c'è la classe astratta ***Media***, che contiene gli attributi e i metodi comuni a tutte le tipologie di media, come ad esempio l'*ID*, il *titolo*, la *descrizione breve*, l'*anno di pubblicazione* e il *percorso* dell'immagine associata. Si noti, si è scelto di utilizzare un **UUID** come identificatore e questo garantisce l'unicità assoluta di ogni elemento, eliminando le possibilità di conflitto anche nel caso di importazione di dati esterni. Un ulteriore vantaggio di questa scelta si riflette nella gestione delle immagini: ogni file immagine associato a un *media* viene salvato con il nome corrispondente all'**UUID** del *media* stesso.

Media, Articles, Literature Da ***Media*** derivano due categorie: ***Literature*** e ***Articles***. La prima racchiude tutte le *opere letterarie* come i ***libri*** e le ***poesie***, mentre la seconda comprende gli *articoli*, che possono essere sia di ***giornale*** sia ***accademici***. Ciascuna di queste sottoclassi introduce proprietà specifiche: ad esempio, ***Literature*** potrebbe includere il *genere letterario*, il *numero di pagine* o anche una *descrizione lunga* (che può essere usata come riassunto del ***libro*** o per mostrare l'intera ***poesia***), mentre ***Articles*** gestisce campi come l'*url* o il *numero di parole* contenute in un ***articolo***. Il tutto è mostrato dalle figure 1, 2 e 3.

Classi Specifiche Queste categorie vengono ulteriormente specializzate: dalla classe ***Literature*** derivano ***Book*** (dove si aggiungono ad esempio il *numero di pagine* e la *casa editrice*) e ***Poem***, mentre da ***Articles*** derivano ***Newspaper-Article*** (dove si aggiungono il *titolo d'apertura* dell'articolo e se l'articolo tratta

di *politica*, con un `bool`) e ***AcademicArticle*** (dove si gestisce l'*università* di appartenenza e se l'articolo è stato *revisionato*). Tramite questa gerarchia si riesce sia a gestire in modo uniforme tutte le operazioni comuni, sia ad aggiungere facilmente nuove tipologie di media in futuro, semplicemente estendendo la gerarchia nei suoi vari livelli. Si noti, i diagrammi *UML* delle specifiche classi sono presenti all'interno del diagramma in allegato e non vengono mostrati in questa relazione per questioni di spazio.

Visitor Al centro dell'architettura di **MediaManager** è stata progettata una gerarchia di **visitor** che prende ispirazione dal ***visitor*** *design pattern*, ma viene adattata e arricchita per rispondere in modo flessibile sia alle esigenze di gestione dei dati sia a quelle dell'interfaccia utente. Questa struttura di ***visitor*** rappresenta uno dei punti di forza principali del sistema, poiché consente di trattare in maniera uniforme tutte le diverse tipologie di media e di implementare in modo modulare funzioni di ricerca, filtro e creazione dinamica dei widget grafici.

Se nel modello classico il ***visitor*** *design pattern* viene adottato per separare le logiche operative dalla struttura degli oggetti, in questo progetto tale principio viene applicato in modo ancora più flessibile: ogni **visitor** (come ad esempio ***ViewVisitor***, ***AddVisitor***, ***EditVisitor***, ***JSONVisitor***, ecc.) racchiude una specifica responsabilità, che va dalla visualizzazione alla modifica degli oggetti, fino alla gestione dell'import/export in formato *JSON*.

Questa scelta architetturale permette, tra le altre cose, di riutilizzare buona parte del codice tra *visitor* apparentemente diversi, come ***AddVisitor*** ed ***EditVisitor***, che condividono la logica per la gestione dei campi e dei widget dell'interfaccia, differenziandosi solo nei dettagli necessari alla creazione o alla modifica di un elemento. Allo stesso modo, la classe ***ViewVisitor*** è impiegata per generare dinamicamente la visualizzazione dei vari tipi di media, evitando duplicazioni di codice e rendendo superfluo l'utilizzo di strutture condizionali complesse all'interno della **UI**.

Un ulteriore aspetto positivo di questa soluzione riguarda l'estensibilità del software: l'introduzione di nuove categorie di *media* o di nuovi comportamenti è resa estremamente agevole, grazie alla possibilità di aggiungere semplicemente un nuovo *visitor* o un overload del metodo `visit`, senza dover modificare la struttura delle classi già esistenti.

Nota sulla gerachia *Visitor* Si noti, come è possibile notare dalla figura 4, le linee che collegano i vari *visitor* alla base ***Visitor*** sono continue: questo perché essa non è propriamente un'*interfaccia pura*. La classe ***Visitor*** infatti, eredita da ***QObject*** per poter utilizzare i meccanismi di *signals/slots* di **Qt**, necessari ad esempio per notificare la **UI** di modifiche ai dati. In questo modo, i *visitor* diventano oggetti "*Qt-compliant*" e possono integrarsi senza problemi con l'infrastruttura dell'applicazione, pur rimanendo (per il resto) una classe base astratta con soli metodi virtuali puri come da specifica del *design pattern*.

3 Polimorfismo

Nel progetto, ciascun *visitor* concreto implementa i metodi necessari per "visitare" oggetti di tipo diverso, permettendo così un comportamento *polimorfo*: ogni oggetto *visitor*, quando passato a un oggetto ***Media*** tramite il metodo `accept`, esegue l'operazione corretta a seconda del tipo concreto del media. Esempio: Per mostrare i dati di un media all'utente, viene creato un oggetto ***ViewVisitor*** che, in base al tipo del media, genera dinamicamente tutti i widget necessari (ad esempio, campi di testo, etichette, immagini e anche tutti i campi specializzati come *scelta della data*, `si/no` per il campo `is_politics` di tipo `bool`, ecc...) e li aggiunge all'interfaccia. Ad esempio, se il *media* è un **libro**, il visitor genera anche i campi per *autore*, *titolo* e *casa editrice*; se invece è un **articolo di giornale**, i campi saranno diversi. Tutto questo avviene senza dover ricorrere a catene di `if` o `switch`, ma semplicemente sfruttando il polimorfismo delle chiamate `visit`.

Visitor per la logica di aggiunta e modifica La struttura dei *visitor* è stata ulteriormente raffinata per evitare duplicazioni e aumentare la riusabilità del codice. In particolare, le classi ***AddVisitor*** ed ***EditVisitor*** condividono gran parte della logica, generando i widget dell'interfaccia in modo simile. La differenza principale è che ***EditVisitor*** precompila i campi coi dati esistenti, mentre ***AddVisitor*** lascia i campi vuoti per la creazione di un nuovo oggetto. Questa soluzione ha permesso di centralizzare e standardizzare tutta la logica di creazione e modifica dei *media*, semplificando moltissimo sia la manutenzione sia le future estensioni.

Visitor anche per la serializzazione Il pattern è stato applicato anche alla serializzazione/deserializzazione dei dati: il ***JSONVisitor*** si occupa della conversione in formato *JSON*, sempre attraverso i metodi `visit` specifici per ogni tipo di *media*. In questo modo, l'aggiunta di un nuovo formato di esportazione richiederebbe solo la creazione di un nuovo *visitor*, senza modificare il modello dati né la logica dell'interfaccia.

Un impatto diretto sulla UI (Qt e *signals/slots*) Un aspetto particolarmente efficace di questa implementazione è l'integrazione con la logica di **Qt**: tutti i *visitor* ereditano da ***QObject*** e possono quindi utilizzare `signals` e `slots` per notificare in tempo reale la **UI** su eventuali cambiamenti. Ad esempio, quando un *media* viene modificato, il *visitor* può emettere un segnale (`mediaEdited`) che aggiorna immediatamente la visualizzazione, garantendo così un'interfaccia sempre coerente e responsiva.

Punti di forza della soluzione adottata

- **Modularità ed Estensibilità**: la gerarchia dei *visitor* consente di aggiungere facilmente nuovi tipi di media o nuove operazioni senza toccare il codice esistente delle altre classi.

- **Separazione delle responsabilità**: ogni *visitor* incapsula un comportamento specifico (visualizzazione, editing, serializzazione, ecc.), evitando il classico "*code bloat*" delle classi monolitiche.
- **UI completamente dinamica**: tutti i widget dell'interfaccia sono generati a *runtime* in modo polimorfo; ciò permette di modificare o arricchire l'applicazione senza dover riscrivere la logica di base.
- **Riuso del codice**: logiche comuni a più contesti (ad esempio, creazione e modifica) sono centralizzate e riusate senza duplicazione.

4 Persistenza dei Dati

Il sistema di persistenza dei dati in **MediaManager** si basa su un meccanismo che sfrutta la serializzazione e deserializzazione tramite file *JSON*, delegando queste operazioni alla classe `JSONVisitor`. Grazie al pattern *visitor*, la logica di esportazione e importazione dei dati è separata dalle classi principali del modello.

Le principali operazioni di salvataggio, esportazione e importazione dei dati sono accessibili direttamente dal menu di sistema dell'applicazione (`MenuBar`), tramite le azioni `Save`, `Import` ed `Export`. Quando l'utente seleziona l'azione di salvataggio, l'applicazione invoca il metodo `exportToFile()` del ***JSONVisitor***, che si occupa di serializzare l'intero elenco dei media in un file *JSON* (`saves/last.json`) posto nella stessa directory dell'eseguibile. Questo permette la persistenza dei dati anche dopo la chiusura e riapertura dell'applicazione.

Gestione delle immagini Un aspetto distintivo della persistenza è la gestione automatica delle immagini associate ai media: ogni immagine viene salvata nella cartella `media/` con un nome corrispondente all'**UUID** del media stesso, garantendo così un'associazione univoca e la possibilità di riconoscere ed eliminare facilmente le immagini non più utilizzate in fase di esportazione (il ***JSONVisitor*** ripulisce automaticamente la cartella da immagini "orfane").

Importazione ed esportazione Durante l'esportazione, oltre al file *JSON*, è consigliato copiare anche il contenuto della cartella `media/`, così da conservare anche le immagini associate ai media. Al momento dell'importazione di un salvataggio infatti, il programma importa esclusivamente il file *JSON* (che contiene tutti i dati testuali), mentre le immagini saranno correttamente visualizzate solo se la cartella `media/` viene copiata e incollata nella stessa posizione dell'eseguibile. Questo meccanismo offre flessibilità all'utente e rende il programma facilmente portabile su altri dispositivi o directory.

Prima esecuzione Per facilitare il primo avvio, nella **directory principale del progetto** sono fornite le cartelle `media/` e `saves/`, che vanno manualmente copiate nella stessa directory in cui viene generato l'eseguibile, assicurando così

il corretto funzionamento della persistenza di dati e immagini già dalla prima esecuzione (nella *virtual machine*, basta inserirle nella cartella `build` generata).

5 Funzionalità Aggiuntive

Di seguito vengono descritte le funzionalità aggiuntive implementate:

- **Gestione immagini integrata**: È possibile associare un'immagine personalizzata a ciascun media. Le immagini vengono gestite e rinominate automaticamente dal programma in base all'**UUID**, non serve quindi mantenere l'immagine "sorgente" poichè viene copiata all'interno del programma
- **Interfaccia utente moderna**: L'**UI** utilizza un design a *card* e una *palette* minimalista, rendendo l'interfaccia molto semplice ma funzionale. Inoltre sono presenti effetti di selezione delle cards/elementi, colori, bottoni con icone per l'aggiunta e la rimozione e anche animazioni.
- **Visualizzazione personalizzata**: I vari *media* vengono mostrati in base alle loro caratteristiche. Per esempio, è possibile scegliere e visualizzare date tramite l'apposito widget di *Qt*.
- **Doppia modalità di visualizzazione**: Ogni "collezione" può essere visualizzata sia nella classica modalità *lista* (con informazioni sintetiche), sia in modalità *griglia fullscreen*, che permette di sfruttare al massimo lo spazio disponibile e rende particolarmente immediata la consultazione di grandi raccolte.
- **Visualizzazione, editing e creazione in fullscreen**: L'interazione con i singoli media (visualizzazione dettagliata, modifica, inserimento) avviene tramite widget a schermo intero, migliorando l'usabilità anche su schermi piccoli o con dimensioni ridotte (questo in modalità *griglia fullscreen*).
- **Filtraggio**: È possibile filtrare la visualizzazione dei media per tipologia (*Letteratura*, *Articoli*, etc.), rendendo semplice la consultazione di raccolte anche molto ampie.
- **Ricerca in tempo reale**: Una barra di ricerca sempre attiva consente di individuare rapidamente un media, con risultati che si aggiornano dinamicamente in base sia al testo inserito, sia ai filtri attivi (quindi ricerca anche in base al tipo!)
- **Shortcut da tastiera**: Durante l'inserimento di nuovi media è possibile confermare l'operazione semplicemente premendo `Invio` (*Enter*), velocizzando così la compilazione di grandi quantità di dati.
- **Gestione intelligente delle immagini**: Il sistema si occupa autonomamente di eliminare immagini non più collegate ad alcun media (quando si salva), mantenendo ordinata e pulita la cartella delle immagini.

Nota Si noti inoltre che il codice è stato formattato tramite `clang-format` per garantire leggibilità e standardizzazione. In esso inoltre si fa ampio uso di `smart pointers` e funzioni `lambda`.

6 Rendicontazione ore

Attività	**Ore Previste**	**Ore Effettive**
Studio e progettazione	8	8
Sviluppo del codice del modello	8	14
Studio del framework Qt	8	8
Sviluppo del codice della GUI	8	14
Test e debug	4	7
Stesura della relazione	4	4
Totale	**40**	**55**

Il tempo aggiuntivo impiegato è dovuto principalmente alla cura grafica dell'interfaccia e all'utilizzo ibrido dei vari *visitor*.

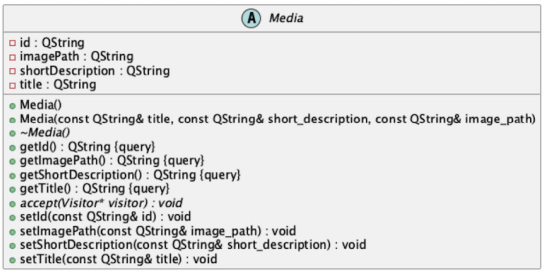

Figure 1: UML della classe ***Media***

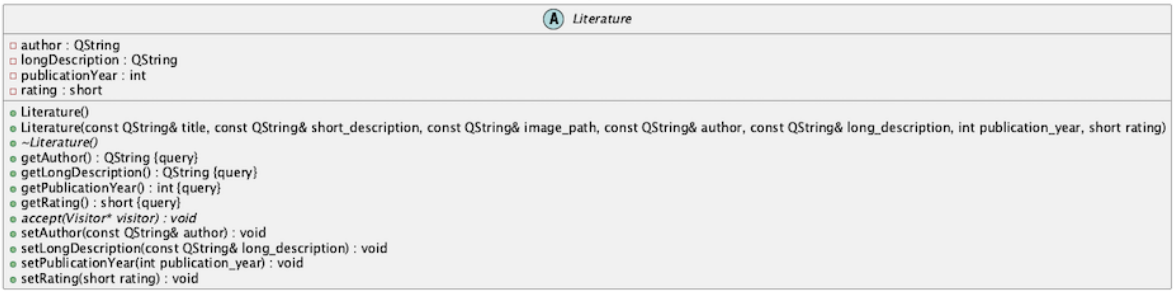

Figure 2: UML della classe ***Literature***

Figure 3: UML della classe ***Articles***

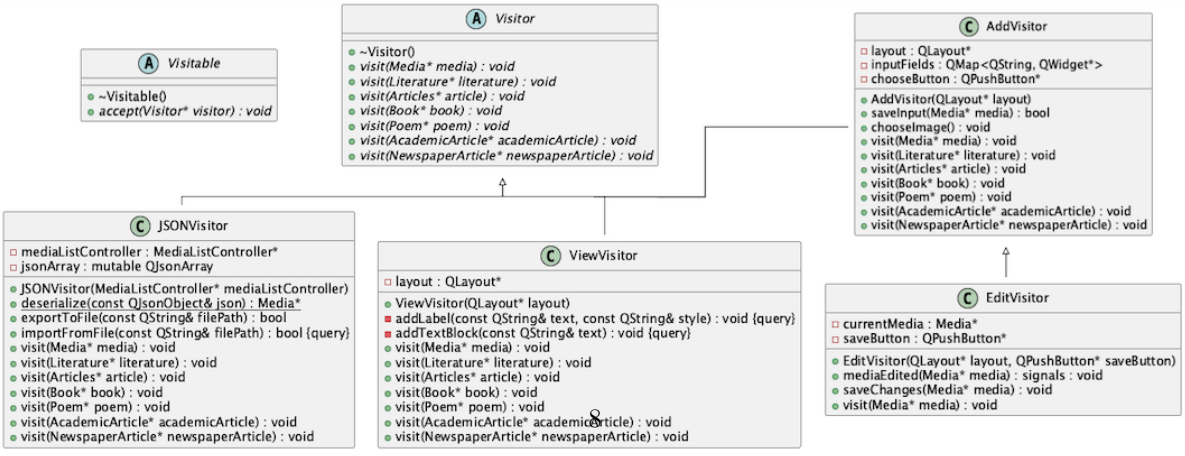

Figure 4: UML della gerarchia *visitor*