MediaLibrary

oggetto	Relazione progetto Programmazione a Oggetti
gruppo	Dennis Parolin, mat. 2113203 Tommaso Ceron, mat. 2101045

Introduzione

MediaLibrary è un sistema che simula una biblioteca digitale personale organizzando i file presenti nel computer dell'utente come se fossero raccolti in un sito web, strutturato e facilmente consultabile. L'obiettivo principale del progetto è quello di offrire una gestione intuitiva di diversi contenuti multimediali.
All'interno della libreria virtuale è possibile gestire cinque diverse tipologie di oggetti: album, libri, fumetti, videogiochi e film. Ogni tipologia ha caratteristiche proprie che ne definiscono le modalità di visualizzazione nel dettaglio. Ma è possibile anche visualizzare tutti gli elementi nel loro insieme ponendoli su una griglia. L'utente può aggiungere nuovi elementi alla libreria (permettendo di salvarli per non perderli), eliminarli o modificarne i dettagli in qualsiasi momento. Inoltre, è disponibile un sistema di ricerca filtrata che consente di trovare rapidamente contenuti in base alla loro categoria o al titolo.
Queste funzionalità disponibili sono gestite dalla classe Library che raccoglie gli oggetti in una lista apposita.
Ho scelto di sviluppare questo progetto per esplorare l'uso del polimorfismo applicato a oggetti multimediali differenti e simulare una struttura ordinata e interattiva per la gestione dei file personali.

Descrizione del modello

Il modello logico si divide in 3 parti principali: oggetti, libreria e la gestione dei file XML/JSON
Alla base della gerarchia si trova la classe astratta AbstractItem. Questa fornisce una rappresentazione comune per tutti i media. Definisce infatti campi comuni a tutti come: il titolo dell'elemento, una descrizione e il percorso all'immagine di copertina, l'anno, il paese e il genere. Questi attributi costituiscono il nucleo informativo di ogni oggetto e sono gestiti tramite metodi getter e setter. Poiché le classi del modello si comportano come dei Data Transfer Object (DTO) e non espongono funzionalità autonome, si è adottato il design pattern Visitor per arricchirne dinamicamente il comportamento.
Sulla base di AbstractItem si sviluppano più sottoclassi concrete, una per ciascuna tipologia di media: Book, Album, Comic, Movie e Videogame. Ogni sottoclasse estende la rappresentazione base con attributi specifici. Per supportare operazioni polimorfiche, ogni sottoclasse implementa il metodo accept, necessario per l'applicazione del pattern Visitor. Questo consente, ad esempio, al SaveEditsVisitor di aggiornare i dati specifici in

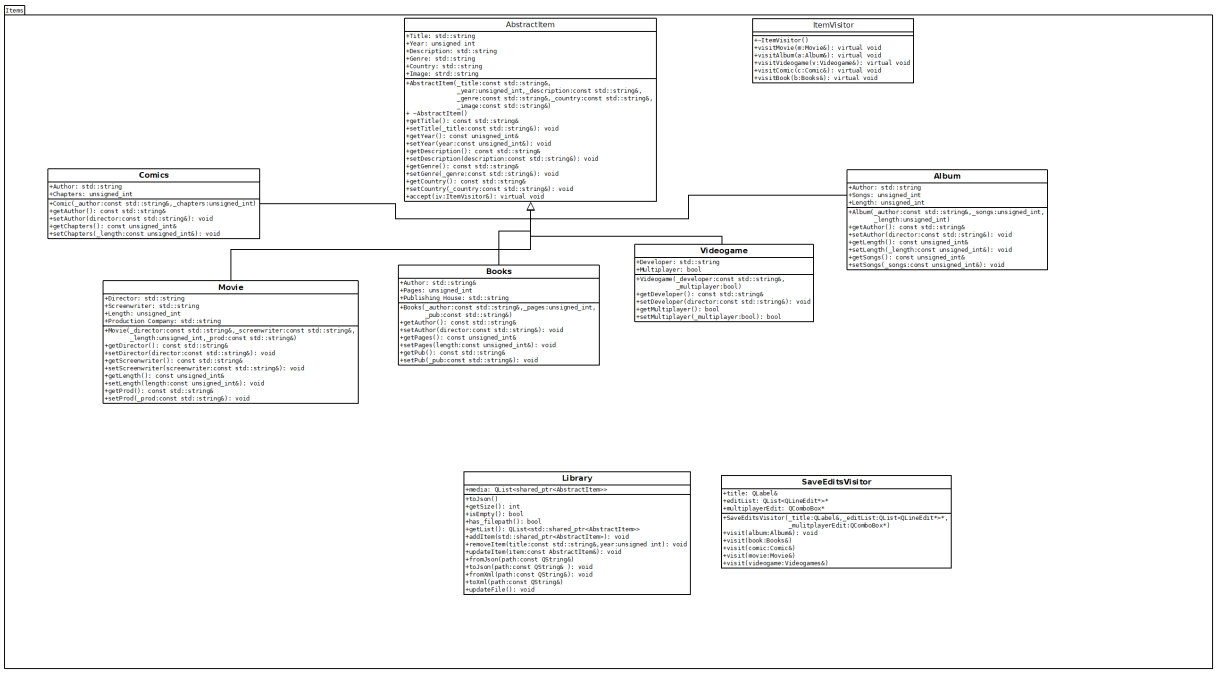

Figura 1: Didascalia dell'immagine

base al tipo effettivo dell'oggetto.
Il modello è stato concepito fin dall'inizio per essere estensibile: aggiungere un nuovo tipo di media richiede solo la definizione di una nuova sottoclasse di AbstractItem, con i relativi campi, senza compromettere la struttura esistente. Questa progettazione si presta bene a evoluzioni del software, permettendo la crescita della base dati e delle funzionalità senza modifiche invasive.
Dal punto di vista della gestione degli oggetti in memoria, l'intera collezione è affidata alla classe Library, che funge da contenitore centrale. La sua implementazione segue il pattern Singleton, assicurando un'unica istanza globale dell'archivio. Library gestisce operazioni di base come l'aggiunta, la rimozione e la modifica di oggetti, ed è responsabile della persistenza dei dati attraverso file XML o JSON. Ogni modifica alla collezione comporta un aggiornamento automatico del file associato.
La serializzazione e la deserializzazione degli oggetti sono gestite da moduli separati, incaricati rispettivamente della lettura (XmlParser, JsonReader) e della scrittura (XmlWriter, JsonWriter) nei formati XML e JSON. In particolare, l'analisi del file XML è effettuata mediante QXmlStreamReader, e il file JSON tramite QJsonObject.
Infine, la classe ItemController, realizzata secondo il pattern Controller, gestisce la creazione degli oggetti e il loro inserimento nella libreria, fornendo un'interfaccia tipizzata per l'interazione con i moduli dell'interfaccia grafica.

Polimorfismo

Nel progetto, il polimorfismo è impiegato in modo significativo tramite l'adozione del design pattern Visitor, utilizzato per gestire comportamenti complessi che variano dinamicamente in base al tipo concreto di oggetto visitato. Questo approccio consente una netta separazione tra la struttura dei dati e la logica delle operazioni, migliorando l'esten-

dibilità e la manutenibilità del codice.
Un esempio centrale è l'utilizzo del SaveEditsVisitor, che applica modifiche persistenti su istanze derivate da AbstractItem. Ogni sottoclasse (ad esempio Album, Book, Movie, ecc.) implementa un metodo accept che inoltra il controllo al visitor, il quale gestisce internamente il comportamento specifico per ciascun tipo. Allo stesso modo, l'ItemDetailVisitor viene utilizzato per costruire dinamicamente interfacce grafiche specifiche per ciascun tipo di elemento selezionato. In base alla classe concreta dell'oggetto (Album, Book, Comic, Movie), il visitor genera widget, layout e campi appropriati come:
per il tipo Videogame, viene utilizzata una QComboBox per rappresentare l'attributo booleano Multiplayer, mentre per gli altri media si impiegano QLineEdit standard. Inoltre, la gestione del campo Length (che assume significati diversi: minuti, pagine, capitoli) è stata incapsulata in una classe personalizzata LengthEdit, la quale estende QLineEdit per mostrare l'unità di misura corretta e imporre vincoli sull'input numerico. Questo approccio evita l'uso di blocchi condizionali come if o switch, sostituendoli con un meccanismo polimorfico più pulito e scalabile.
Anche la serializzazione degli oggetti sfrutta il polimorfismo grazie all'implementazione di due visitor distinti per l'esportazione in formato JSON e XML. Questi visitor attraversano la struttura dell'oggetto e producono una rappresentazione testuale adatta al salvataggio, al backup o allo scambio dati. Ogni nodo della gerarchia richiama il metodo accept, che consente al visitor di eseguire la logica specifica per quel tipo.
Infine, il polimorfismo è presente nella gerarchia che separa l'interfaccia AbstractItem dalle sue sottoclassi. Questa struttura consente una gestione uniforme degli oggetti, ad esempio in collezioni o funzioni generiche, ma trova il suo massimo valore nell'integrazione con i visitor, che ne sfruttano il comportamento virtuale per operare su oggetti di tipo eterogeneo senza conoscere a priori il tipo esatto.
Un ulteriore componente è la classe ItemController, responsabile della costruzione e dell'inserimento degli oggetti nella libreria. È stata realizzata seguendo il pattern Controller.
Nel complesso, l'uso del polimorfismo nel progetto è funzionale a obiettivi concreti: favorisce l'estensibilità del sistema, la separazione delle responsabilità e la riduzione della complessità condizionale, migliorando la leggibilità e la manutenibilità del codice.

Persistenza dei dati

La persistenza dei dati nel progetto è gestita attraverso l'uso del pattern Visitor, con supporto ai formati JSON e XML.
Per la scrittura vengono utilizzati due visitor: JsonVisitor e XmlVisitor. Questi visitano ciascun oggetto (Album, Book, Movie, Comic, ecc.) e ne generano una rappresentazione testuale rispettivamente in formato JSON o XML. La scrittura si basa su un'unica chiamata accept per ciascun oggetto, che consente ai visitor di accedere in modo polimorfico alle proprietà specifiche.
La lettura dei dati è gestita dalle classi JsonReader e XmlParser. Entrambe si occupano di leggere un file strutturato contenente un array (in JSON) o una sequenza di tag (in XML), estrarre il tipo tramite un campo "type" (in JSON) o il nome del tag (in XML), e creare dinamicamente l'istanza della sottoclasse corrispondente. Ogni oggetto viene poi inserito nella lista interna che rappresenta la libreria.
All'avvio dell'applicazione, viene automaticamente caricato un file predefinito contenente almeno un oggetto per ogni tipologia. Questo file è fornito insieme al progetto e può

essere usato come riferimento per comprendere la struttura di serializzazione. In JSON, ogni oggetto è rappresentato come un dizionario con chiavi che corrispondono ai campi dell'oggetto e un campo "type" che ne indica la classe concreta.

Funzionalità implementate

Le funzionalità sviluppate si possono suddividere in due grandi categorie: funzionali e grafiche.

Funzionalità funzionali

- Capacità di leggere e scrivere dati nei formati JSON e XML.
- Ricerca e filtro per categoria, che facilitano la navigazione nella libreria.
- Supporto per la gestione di cinque diverse categorie di oggetti multimediali.

Funzionalità grafiche

- Uso di un ItemDelegate per mostrare la locandina e il titolo di ogni elemento nella vista a griglia.
- Diverse modalità di visualizzazione: a griglia e in dettaglio.
- Pannello laterale dotato di barra di ricerca, filtri per categoria e icone intuitive.
- Barra dei menù principale con opzioni per aprire file, salvare e uscire dall'applicazione.
- Implementazione di scorciatoie da tastiera per velocizzare l'uso delle funzioni principali.
- Pulsanti dedicati alle operazioni CRUD (creazione, lettura, aggiornamento e cancellazione).

L'interfaccia si basa sul modello architetturale Model-View, che separa nettamente il modello dei dati dalla loro rappresentazione grafica. In questo schema, il modello si occupa di fornire i dati e di eseguire eventuali operazioni su di essi. Un esempio è l'uso della classe QItemDelegate per la visualizzazione della locandina con titolo in sovraimpressione. La vista si limita invece alla loro visualizzazione, senza conoscere lo stato interno degli oggetti o della libreria.
Questa separazione semplifica gli aggiornamenti dell'interfaccia: ogni modifica dei dati genera un segnale che il modello riceve, aggiornando automaticamente la vista. Grazie a questo approccio, si garantisce una gestione efficiente e modulare delle informazioni, con un'interfaccia responsiva e coerente.

Rendicontazione Ore

Attività	**Ore Previste**	**Ore Effettive**
Studio e progettazione	5	10
Sviluppo del codice del modello	15	18
Studio del framework Qt	5	7
Sviluppo del codice della GUI	10	13
Test e debug	5	6
Stesura della relazione	2	6
Totale	42	**60**

Abbiamo superato il totale di ore previste a causa dello sviluppo dell'interfaccia grafica. Non ci erano totalmente chiari alcuni particolari del framework Qt e questa ci ha richiesto uno studio più approfondito. Soprattutto per implementare alcuni pattern.

Suddivisione del Lavoro

Attività	**Componente del gruppo**
Progettazione	Entrambi
Sviluppo del modello logico (oggetti)	Parolin
Sviluppo del modello logico (Library/gestione JSON)	Ceron
Sviluppo del modello logico (Gestione XML)	Parolin
Sviluppo della interfaccia grafica a griglia	Ceron
Sviluppo di ItemController e SaveEditsVisitor	Ceron
Sviluppo della interfaccia grafica dettagliata, con relativo visitor	Parolin
Test e debugging	Entrambi
Stesura relazione	Entrambi