1. Introduzione

Questo progetto implementa un sistema di gestione per una biblioteca multimediale che supporta diversi tipi di media, come libri, film, riviste ed e-book utilizzando il linguaggio di programmazione C++ e il framework Qt per la creazione di un'interfaccia grafica. Il programma consente agli utenti di gestire la propria collezione multimediale, aggiungendo, modificando e rimuovendo media, nonché di visualizzare le informazioni specifiche di ciascun tipo di media in un'interfaccia chiara e intuitiva.

Il progetto è stato sviluppato seguendo i principi della programmazione orientata agli oggetti, implementando pattern di design comuni come Visitor, Strategy e Factory per garantire un'architettura flessibile ed estensibile. La separazione netta tra modello logico e interfaccia grafica consente il riutilizzo del codice e facilita la manutenzione.

2. Descrizione delle classi principali del modello logico

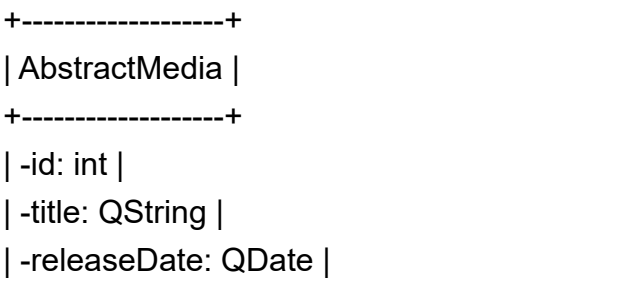
Gerarchia di media

La gerarchia di classi è stata progettata per rappresentare diversi tipi di media attraverso una classe base astratta e le sue derivate:

- **AbstractMedia**: Classe base astratta che definisce l'interfaccia comune per tutti i tipi di media. Contiene attributi generali come id, titolo, data di pubblicazione, descrizione e percorso dell'immagine di copertina.
- **Book**: Rappresenta un libro, con attributi specifici come autore, ISBN, numero di pagine e genere.
- **Movie**: Rappresenta un film, con attributi specifici come regista, durata e cast.
- **Magazine**: Rappresenta una rivista, con attributi specifici come numero, volume, numero di uscita e argomento.
- **EBook**: Rappresenta un libro elettronico, con attributi specifici come autore, ISBN, numero di pagine, formato e dimensione del file.

Questo approccio permette di gestire in modo uniforme diversi tipi di media, consentendo al contempo l'estensione con nuovi tipi in futuro.

Diagramma UML



| ... |
+------------------+
| +getMediaType(): QString |
| +accept(visitor): void |
| +toJson(): QJsonObject |
+------------------+
/_
|
+-------------------+-----------------+-----------------+
| | | |
+---------------+ +-----------------+ +---------------+ +---------------+
| Book | | Movie | | Magazine | | EBook |
+---------------+ +-----------------+ +---------------+ +---------------+
-author: QString		-director: QString		-issue: QString		-author: QString
-isbn: QString		-duration: int		-volume: int		-isbn: QString
-pageCount: int		-cast: QString		-issueNumber: int		-pageCount: int
-genre: QString				-topic: QString		-format: QString
+---------------+ +-----------------+ +---------------+	-fileSizeMB: float
+accept()		+accept()		+accept()	+---------------+
+toJson()		+toJson()		+toJson()		+accept()
+---------------+ +-----------------+ +---------------+ | +toJson() |
+---------------+``

3. Utilizzo del polimorfismo non banale

Il progetto implementa il polimorfismo in modo non banale attraverso l'uso di diversi pattern di design:

Pattern Visitor

Il pattern Visitor è stato implementato per separare gli algoritmi dalle strutture di dati. Questa implementazione consente l'aggiunta di nuove operazioni senza modificare le classi base.

```
class MediaVisitor {
public:
    virtual ~MediaVisitor() = default;

    virtual void visit(Book &book) = 0;
    virtual void visit(Movie &movie) = 0;
    virtual void visit(Magazine &magazine) = 0;
    virtual void visit(EBook &ebook) = 0;
};
```

Ogni classe concreta nella gerarchia dei media implementa il metodo `accept()` che accetta un visitor:

```cpp
// Nel file Book.cpp
void Book::accept(MediaVisitor &visitor) {
    visitor.visit(*this);
}
```

Il progetto implementa diverse classi visitor concrete:

1. **MediaRendererVisitor**: Genera widget di visualizzazione specifici per ciascun tipo di media.
2. **MediaExportVisitor**: Esporta i media in diversi formati (JSON, CSV, XML).
3. **MediaSearchVisitor**: Implementa algoritmi di ricerca specializzati per ciascun tipo di media.

Questo approccio permette di separare la logica di visualizzazione, esportazione e ricerca dal modello dati, rispettando il principio di responsabilità singola.

Pattern Strategy

Il pattern Strategy è stato implementato per gestire diverse strategie di visualizzazione e modifica dei media:

```cpp
class MediaViewStrategy {
public:
    virtual ~MediaViewStrategy() = default;
    virtual QWidget* createDetailView(AbstractMedia *media) = 0;
    virtual QWidget* createSummaryView(AbstractMedia *media) = 0;
    virtual QWidget* createEditView(AbstractMedia *media) = 0;
};
```

Ogni tipo di media ha la sua strategia concreta (BookViewStrategy, MovieViewStrategy, ecc.) che implementa metodi per creare widget specifici per quel tipo di media.

Pattern Factory Method

Il pattern Factory Method è stato implementato per facilitare la creazione di nuovi oggetti media e i relativi widget di editing:

```cpp
class MediaFactory {
public:
    virtual ~MediaFactory() = default;
    virtual AbstractMedia* createMedia() = 0;
    virtual QWidget* createEditDialog() = 0;
    virtual QString getMediaType() const = 0;
};
```

Ogni tipo di media ha la sua factory concreta (BookFactory, MovieFactory, ecc.) che implementa metodi per creare nuove istanze di quel tipo di media e i relativi widget di editing.

La combinazione di questi pattern crea un sistema flessibile ed estensibile, che permette di aggiungere facilmente nuovi tipi di media e nuove funzionalità senza modificare il codice esistente.

4. Persistenza dei dati

Il progetto implementa un sistema di persistenza dei dati in formato JSON attraverso la classe `JsonPersistenceManager`. Questa classe fornisce metodi per salvare e caricare la biblioteca di media in/da un file JSON.

Formato di serializzazione

Ogni media viene serializzato in un oggetto JSON che include sia i campi comuni (definiti in AbstractMedia) che quelli specifici del tipo. Per esempio, un libro viene serializzato come:

```
{
  "id": 1,
  "type": "Book",
  "title": "The Great Gatsby",
  "releaseDate": "1925-04-10",
  "publisher": "Charles Scribner's Sons",
  "description": "Set in the Jazz Age on Long Island...",
  "coverPath": "covers/gatsby.jpg",
  "author": "F. Scott Fitzgerald",
  "isbn": "978-0743273565",
  "pageCount": 180,
  "genre": "Novel"
}
```

Implementazione della persistenza

La classe `JsonPersistenceManager` implementa due metodi principali:

```
bool saveLibrary(const QString &filePath, const QVector<AbstractMedia*>
&mediaList);
QVector<AbstractMedia*> loadLibrary(const QString &filePath);
```

Il salvataggio avviene convertendo ogni media in un oggetto JSON e poi salvando l'array risultante in un file. Il caricamento avviene leggendo il file JSON, estraendo l'array di media e convertendo ogni oggetto JSON nel corrispondente oggetto C++.

Il campo "type" in ogni oggetto JSON viene utilizzato per determinare quale tipo concreto di media creare durante il caricamento:

```
AbstractMedia* AbstractMedia::fromJson(const QJsonObject &json) {
    QString type = json["type"].toString();

    if (type == "Book") {
        return Book::fromJson(json);
    } else if (type == "Movie") {
        return Movie::fromJson(json);
    } else if (type == "Magazine") {
        return Magazine::fromJson(json);
    } else if (type == "EBook") {
        return EBook::fromJson(json);
    }

    return nullptr;
}
```

5. Funzionalità aggiuntive implementate

1. Sistema di ricerca avanzata

Il sistema di ricerca implementa una funzionalità avanzata che consente di cercare in tutti i campi di tutti i tipi di media. Il pattern Visitor viene utilizzato per implementare algoritmi di ricerca specifici per ciascun tipo di media.

2. Esportazione di dati in diversi formati

L'applicazione consente di esportare i dati della biblioteca in diversi formati:

- CSV per l'importazione in fogli di calcolo
- XML per l'interoperabilità con altri sistemi
- Testo semplice per la leggibilità umana

3. Visualizzazione multipla dei media

L'interfaccia utente supporta diverse modalità di visualizzazione dei media:

- Vista a elenco per la visualizzazione compatta
- Vista a dettaglio per informazioni complete
- Vista a miniature per una visione d'insieme

4. Gestione delle immagini di copertina

L'applicazione supporta la gestione delle immagini di copertina per ciascun media, consentendo agli utenti di selezionare immagini dal proprio sistema

5. Interfaccia responsive con layout adattivi

L'interfaccia utente è stata progettata per adattarsi automaticamente alle dimensioni della finestra, rendendo l'applicazione utilizzabile su schermi di diverse dimensioni. Questo è stato realizzato utilizzando layout annidati di Qt (QVBoxLayout, QHBoxLayout, QFormLayout) che si adattano automaticamente.

6. Supporto per scorciatoie da tastiera

Sono state implementate numerose scorciatoie da tastiera per migliorare l'usabilità dell'applicazione:

- Ctrl+N: Nuovo file
- Ctrl+O: Apri file
- Ctrl+S: Salva file
- Ctrl+Shift+S: Salva come
- Ctrl+F: Cerca
- Ctrl+N (nell'interfaccia principale): Aggiungi nuovo media

7. Sistema di filtraggio per tipo di media

È stata implementata una funzionalità di filtraggio che consente agli utenti di visualizzare solo i media di un determinato tipo. Questo è particolarmente utile quando la biblioteca contiene molti elementi.

8. Statistiche sulla collezione

L'applicazione può generare statistiche sulla collezione di media, come:

- Numero totale di media per tipo
- Media più recente/più vecchio
- Distribuzione per genere/argomento

6. Rendicontazione delle ore

Attività	Ore previste	Ore effettive
Analisi e progettazione	8	10
Implementazione del modello dati	6	7
Implementazione dei pattern di design	8	9
Implementazione della persistenza dati	4	5

Attività	Ore previste	Ore effettive
Sviluppo dell'interfaccia grafica	10	12
Testing e debugging	4	6
Totale	**40**	**49**

La realizzazione del progetto ha richiesto circa 49 ore di lavoro, superando leggermente le 40 ore previste. I motivi principali sono stati:

1. Maggior complessità nell'implementazione dei pattern di design, in particolare nell'integrazione tra Visitor e Strategy
2. Tempo aggiuntivo richiesto per lo sviluppo dell'interfaccia grafica con una buona esperienza utente
3. Testing più approfondito per garantire robustezza nell'uso dei pattern implementati

7. Suddivisione delle attività (per progetti di coppia)

Nota: Questa sezione è applicabile solo per progetti svolti in coppia. Per un progetto individuale, può essere omessa.

Attività	Componente del gruppo
Progettazione dell'architettura	Entrambi
Implementazione del modello dati	Studente 1
Implementazione dei pattern Visitor e Strategy	Studente 1
Implementazione del pattern Factory	Studente 2
Sviluppo della persistenza dei dati	Studente 2
Sviluppo dell'interfaccia grafica principale	Studente 2
Sviluppo delle schermate di modifica	Studente 1
Testing e debugging	Entrambi
Redazione della relazione	Entrambi

8. Conclusioni

Il progetto Library Management System dimostra i vantaggi dell'approccio orientato agli oggetti nella creazione di software complessi ed estensibili. L'uso di pattern di design come Visitor, Strategy e Factory ha permesso di creare un'architettura flessibile che separa chiaramente responsabilità diverse e facilita l'estensione del sistema.

La chiara separazione tra modello logico e interfaccia grafica ha reso possibile lo sviluppo indipendente di queste componenti, migliorando la manutenibilità del codice. La persistenza dei dati in formato JSON offre un approccio moderno e flessibile alla gestione dei dati.

Tra le sfide affrontate, la più significativa è stata l'integrazione dei diversi pattern di design in un'architettura coerente. L'applicazione dei principi SOLID, in particolare il Principio di Singola Responsabilità e il Principio di Sostituzione di Liskov, ha guidato lo sviluppo e contribuito alla creazione di un codice ben strutturato.

Il sistema risultante è estensibile sia in termini di tipi di media supportati (è possibile aggiungere facilmente nuovi tipi come Audiobook, Game, ecc.) sia in termini di funzionalità (nuovi visitor possono implementare nuove operazioni senza modificare le classi esistenti).

In futuro, il sistema potrebbe essere esteso con funzionalità come:

- Supporto per database relazionali o NoSQL invece dei file JSON
- Integrazione con API di servizi online per recuperare informazioni sui media
- Funzionalità di prestito e tracciamento per una vera biblioteca
- Interfaccia utente web o mobile utilizzando lo stesso modello dati