Relazione progetto Programmazione ad Oggetti

Alessandro Bertolazzi

1227274

Sommario

Relazione del progetto "Biblioteca Virtuale" per il corso "Programmazione ad Oggetti". Un'applicazione scritta in C++ con framework Qt che permette la gestione di una biblioteca personale, implementando design pattern avanzati e un'architettura modulare scalabile.

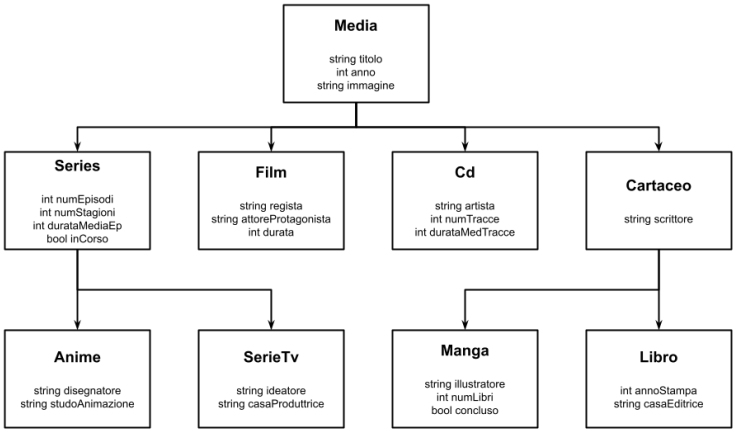
1 Introduzione

Bibliotheca procurator dal latino "direttore della biblioteca", il nome che ho deciso di dare all'applicazione. Il progetto proponeva di creare un'applicazione che permettesse la gestione di una biblioteca. L'applicazione che ho sviluppato non riguarda l'organizzazione di una biblioteca pubblica, bensì l'organizzazione di una biblioteca privata in modo da poter strutturare e catalogare libri, film, CD, manga, anime e serie TV per un privato con collezioni ampie. Mi sono dedicato a questo obiettivo poiché, oltre a poter avere un'utilità per me stesso, ho notato che non si trovano prodotti simili online e, se si trovano, si tratta di applicazioni datate, a pagamento o non più mantenute.

2 Struttura del Progetto

2.1 Logic

Nella cartella `Logic` si trova tutta la parte logica dell'applicazione. Ho implementato 6 classi concrete: `Anime`, `SerieTv`, `Film`, `Cd`, `Manga` e `Libro`. Come classi astratte ho utilizzato `Media` come superclasse e due classi intermedie `Series` e `Cartaceo` per raggruppare funzionalità comuni. Ho inoltre implementato il pattern Visitor tramite `MediaVisitor` per gestire operazioni polimorfiche sui diversi tipi di media.



Struttura logica dell'applicazione

2.2 UI

Per l'interfaccia grafica ho adottato un'architettura modulare dove `mainWindow` coordina diversi widget specializzati:

- `topMenuWidget`: gestisce la barra superiore con pulsanti iconografici per le operazioni principali (carica, salva, crea, toggle tema)
- `rightLayoutWidget`: si occupa della visualizzazione dinamica dei media utilizzando il pattern Visitor per creare widget specifici per ogni tipo
- `createItemWidget`: finestra modulare per creazione e modifica che utilizza form generati dinamicamente tramite `FormWidgetVisitor`
- `mediaWidgetVisitor`: implementa il pattern Visitor per creare rappresentazioni UI specifiche per ogni tipo di media
- `formWidgetVisitor`: genera automaticamente form di inserimento/modifica basati sul tipo di media selezionato

2.3 Services

Ho implementato un'architettura a servizi:

- `JsonService`: gestisce la persistenza dei dati utilizzando un pattern Factory per la creazione dinamica di oggetti Media dal JSON
- `MediaService`: service layer principale che incapsula tutta la business logic per la gestione dei media, inclusi validazione, operazioni CRUD e filtering
- `UIService`: si occupa della formattazione dei dati per la visualizzazione e della gestione delle immagini con placeholder automatici
- `StyleUtils`: sistema centralizzato di gestione degli stili con supporto per temi chiaro/scuro e palette di colori coerenti
- `JsonTypeVisitor`: visitor specializzato per determinare il tipo di media durante la serializzazione

3 Design Pattern Implementati

3.1 Visitor Pattern

Il Visitor pattern è stato implementato in diversi contesti per separare gli algoritmi dalle strutture dati:

```
class MediaVisitor {
public:
    virtual ~MediaVisitor() = default;
    virtual void visit(Film* film) = 0;
    virtual void visit(SerieTv* serieTv) = 0;
    virtual void visit(Anime* anime) = 0;
    virtual void visit(Libro* libro) = 0;
    virtual void visit(Manga* manga) = 0;
    virtual void visit(Cd* cd) = 0;
};

// Utilizzo
media->accept(widgetVisitor);
QWidget* specificWidget = widgetVisitor->getResultWidget();
```

Ho utilizzato questo pattern per:

- Creazione di widget UI specifici (`MediaWidgetVisitor`)
- Generazione di form dinamici (`FormWidgetVisitor`)
- Determinazione del tipo per serializzazione (`JsonTypeVisitor`)

3.2 Factory Method Pattern

Implementato per la creazione dinamica di oggetti Media dal JSON senza conoscere il tipo specifico a compile-time:

```
void JsonService::initializeFactories() {
    mediaFactories["Film"] = [](const QJsonObject& json) -> std::unique_ptr<Media> {
        std::string titolo = json["titolo"].toString().toStdString();
        int anno = json["anno"].toInt();
        std::string immagine = json["immagine"].toString().toStdString();

        auto film = std::make_unique<Film>(titolo, anno, immagine, "", "", 0);
        film->fromJsonSpecific(json);
        return film;
    };

    mediaFactories["Serie Tv"] = [](const QJsonObject& json) -> std::unique_ptr<Media> {
        // Factory per Serie TV...
    };
}
```

3.3 Template Method Pattern

Utilizzato per la validazione unificata dei form con algoritmi personalizzabili:

```
template<typename Validator>
bool MediaService::validateFieldAtIndex(const QList<QLineEdit*>& fields, int index,
                                        const QString& fieldName, QWidget* parent,
                                        Validator validator) const {
    if (index >= fields.size()) return false;
    return validator(fields[index], fieldName, parent);
}

auto validateFields = [this, &fields, parent](
    const QVector<QPair<int, QString>>& fieldConfigs,
    auto validator) -> bool {
    for (const auto& field : fieldConfigs) {
        if (!validateFieldAtIndex(fields, field.first, field.second, parent, validator)) {
            return false;
        }
    }
    return true;
};
```

4 Persistenza dei Dati

Ho implementato un sistema di persistenza basato su JSON. Implementando **Factory Pattern** per la creazione dinamica di oggetti dal JSON, **Template Method** per la serializzazione/deserializzazione specifica per tipo, **Visitor Pattern** per determinare il tipo durante la serializzazione. Ogni classe concreta implementa metodi virtuali per la gestione specifica dei propri dati:

```
// Metodi virtuali per serializzazione
virtual QJsonObject toJsonSpecific() const = 0;
virtual void fromJsonSpecific(const QJsonObject& json) = 0;

QJsonObject SerieTv::toJsonSpecific() const {               // Implementazione in SerieTv
    auto json = getSeriesBaseJson(); // Eredita da Series
    json["type"] = "Serie Tv";
    json["ideatore"] = QString::fromStdString(ideatore);
    json["casaProduttrice"] = QString::fromStdString(casaProduttrice);
    return json;
}
```

Il file JSON é cosi strutturato:

```
"media": [    {
        "anno": 1986,
        "disegnatore": "Akira Toriyama",
        "durataMediaEp": 24,
        "immagine": "../resources/img/dragonball.jpg",
        "inCorso": false,
        "numEpisodi": 153,
        "numStagioni": 5,
        "studioAnimazione": "Toei Animation",
        "titolo": "Dragon Ball",
        "type": "Anime"
    }]
```

5 Polimorfismo e Architettura Modulare

5.1 Serializzazione Polimorfica

Ho eliminato l'uso di dynamic_cast sostituendolo con metodi virtuali per una gestione più pulita e sicura:

```
virtual QJsonObject toJsonSpecific() const = 0;
virtual void fromJsonSpecific(const QJsonObject& json) = 0;
virtual Media* clone() const = 0;
virtual bool matchesCategory(const string& category) const = 0;
```

5.2 Gestione Memory-Safe

Utilizzo di smart pointer e RAII (Resource Acquisition Is Initialization) per una gestione automatica della memoria:

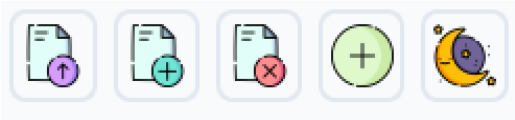
```
std::function<std::unique_ptr<Media>(const QJsonObject&)> mediaFactories;

Media* Anime::clone() const {       // Clone pattern per duplicazione
    return new Anime(getTitolo(), getAnno(), getImmagine(),
                     getNumEpisodi(), getNumStagioni(), getDurataMediaEp(),
                     getInCorso(), disegnatore, studioAnimazione);
}
```

6 Form Dinamici e UI Generata

Il sistema di creazione/modifica utilizza form generati automaticamente tramite il pattern Visitor:

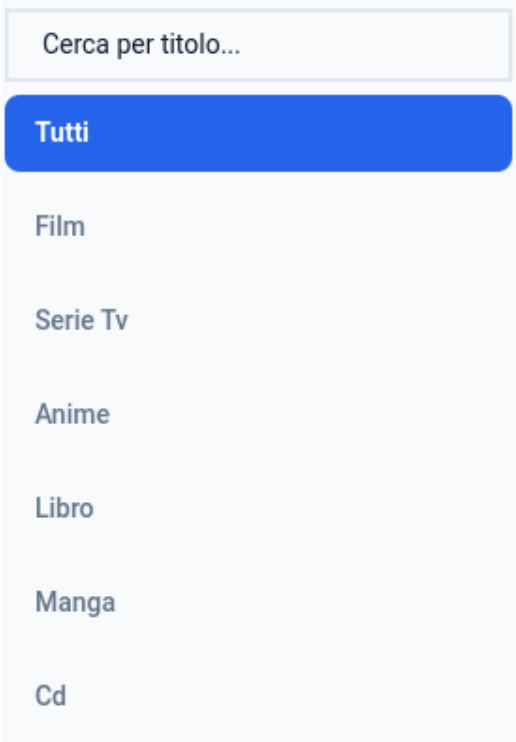
```
const QVector<FieldConfig> FILM_FIELDS = {       // Configurazioni statiche per ogni tipo di media
    {"Titolo", "Inserisci titolo"},
    {"Immagine", "Inserisci percorso immagine", FieldType::IMAGE},
    {"Anno", "Inserisci anno", FieldType::INTEGER},
    {"Regista", "Inserisci regista"},
    {"Attore Protagonista", "Inserisci attore protagonista"},
    {"Durata (min)", "Inserisci durata in minuti", FieldType::POSITIVE_INTEGER}
};

template<typename MediaType>    // Generazione automatica del form
void FormWidgetVisitor::visitGeneric(MediaType* media, const QVector<FieldConfig>& fieldConfigs) {
    QStringList values;
    if (media) {
        values = extractValues(media);
    }
    resultWidget = createStandardForm(fieldConfigs, values);
}
```

7 Interfaccia Grafica

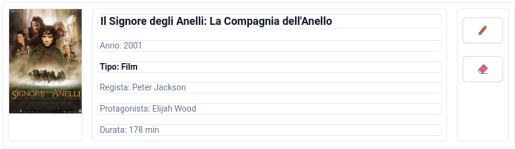


Barra superiore con sistema iconografico:

- Carica biblioteca esistente
- Crea nuova biblioteca con auto-save della cache
- Chiudi biblioteca corrente
- Crea nuovo elemento media
- Toggle tema chiaro/scuro dinamico



Nel leftMenu viene creata la barra di ricerca che, ad ogni cambiamento (ogni volta che viene digitato un carattere), cerca l'elemento nella biblioteca. La ricerca è "case insensitive" e funziona anche con ricerca parziale. Sotto sono presenti i media categorizzati per tipologia.



Ogni elemento media utilizza il pattern Visitor per generare layout specifici con:

- Caricamento delle immagini
- Bottoni di azione per la modifica e la cancellazione
- Layout responsive che si adatta al contenuto
- Dettagli specifici per ogni tipo di media

8 Ricerca e Filtering

Ho implementato un sistema di ricerca case-insensitive con filtering per categorie:

```
bool Media::matchesSearch(const string& searchText) const {
    if (searchText.empty()) return true;

    string lowerTitle = titolo;
    string lowerSearch = searchText;

    // Conversione case-insensitive
    std::transform(lowerTitle.begin(), lowerTitle.end(), lowerTitle.begin(), ::tolower);
    std::transform(lowerSearch.begin(), lowerSearch.end(), lowerSearch.begin(), ::tolower);

    return lowerTitle.find(lowerSearch) != string::npos;
```

```
}

// Filtering combinato categoria + ricerca
QVector<Media*> MediaService::filterMedia(const QString& category, const QString& searchText) const {
    QVector<Media*> filtered;
    std::string categoryStd = category.toStdString();
    std::string searchStd = searchText.toStdString();

    for (Media* media : mediaCollection) {
        if (media->matchesCategory(categoryStd) && media->matchesSearch(searchStd)) {
            filtered.append(media);
        }
    }
    return filtered;
}
```

9 Rendiconto delle Tempistiche

Lo sviluppo del progetto ha richiesto circa 80 ore di lavoro, distribuite come segue:

Jobs	**Scheduled hours**	**Actual hours**	**Notes**
Modellazione diagramma UML	1	2	Tempo aggiuntivo per progettazione pattern
Implementazione modello Logico	5	6	Aggiunta pattern Visitor e metodi virtuali
Implementazione Design Pattern	8	12	Visitor, Factory, Template Method
Implementazione Interfaccia grafica	15	20	Sistema di temi e widget modulari
Implementazione parte Service	15	18	Architettura a servizi e validazione
Sistema di Validazione	5	8	Validazione robusta con template
Affinamento Progetto	5	8	UI/UX improvements e ottimizzazioni
Debug e Testing	5	6	Testing approfondito e correzioni

10 Possibili Miglioramenti

- Implementare una ricerca online automatica delle copertine dei media
- Per Serie TV, Anime e Manga, permettere all'utente di specificare quali volumi/episodi possiede effettivamente
- Migliorare il sistema di scaling e ridimensionamento delle immagini
- Aggiungere supporto per database relazionali oltre al JSON
- Implementare sistema di backup automatico e cronologia delle modifiche
- Aggiungere statistiche avanzate sulla collezione con grafici
- Supportare collezioni multiple e condivisione tra utenti
- Implementare sistema di importazione da cataloghi online (IMDb, MyAnimeList, etc.)

11 Conclusioni

Il progetto Bibliotheca Procurator rappresenta un'applicazione per la gestione di biblioteche personali che implementa diversi design pattern avanzati e un'architettura modulare scalabile. L'uso del pattern Visitor per la generazione dinamica dell'UI, del Factory pattern per la persistenza dei dati, e del Template Method per la validazione, combinati con un sistema di servizi, rendono l'applicazione facilmente estensibile e manutenibile.

12 Terminologie Utilizzate

- **Factory Method Pattern**: Pattern creazionale implementato tramite lambda functions in `JsonService` per creare istanze specifiche di Media dal JSON senza conoscere il tipo a compile-time.
- **Visitor Pattern**: Pattern comportamentale utilizzato per separare algoritmi dalle strutture dati, implementato per generazione UI (`MediaWidgetVisitor`), form dinamici (`FormWidgetVisitor`) e serializzazione (`JsonTypeVisitor`).
- **Template Method Pattern**: Pattern utilizzato per definire algoritmi di validazione con parti personalizzabili, implementato tramite template functions in `MediaService`.
- **Service Layer Architecture**: Architettura che separa la business logic in servizi specializzati (`MediaService`, `JsonService`, `UIService`, `StyleUtils`).
- **SOLID Principles**: Principi di design object-oriented applicati per garantire codice mantenibile e estensibile.
- **Media**: Termine generico per indicare qualsiasi elemento della collezione (Film, Libro, CD, Serie TV, Anime, Manga).
- **Smart Pointers**: Utilizzo di `std::unique_ptr` per gestione automatica della memoria e ownership chiara.
- **RAII (Resource Acquisition Is Initialization)**: Tecnica di gestione delle risorse che lega il ciclo di vita delle risorse a quello degli oggetti.