Università degli Studi di Padova

Relazione Progetto

Programmazione ad Oggetti
Titolo: **Biblioteca virtuale**

Draghici Ana Maria
Matricola: 2101044

Anno Accademico 2024/2025

1 Introduzione

Questo progetto consiste nello sviluppo di una **biblioteca virtuale** che consente la gestione e l'interazione con oggetti digitali come libri, riviste e film. L'obiettivo principale è permettere agli utenti di *visualizzare*, *prenotare* e *restituire* contenuti, offrendo allo stesso tempo agli amministratori strumenti avanzati di gestione.

L'interfaccia è suddivisa in due sezioni principali:

- **Sezione Admin**, dedicata alla gestione della biblioteca. Qui è possibile aggiungere, modificare o eliminare oggetti. L'inserimento è differenziato in base alla tipologia (libro, rivista o film) e le funzionalità disponibili permettono una gestione approfondita delle risorse.
- **Sezione Utente**, pensata per la consultazione e la prenotazione. Gli utenti possono navigare tra gli oggetti disponibili, filtrare per tipo o lingua, e accedere alla propria lista prenotazioni tramite un'apposita sezione nella barra superiore.

Il sistema include inoltre funzionalità di filtraggio e conteggio dei prestiti attivi, con informazioni sempre accessibili nella parte superiore dell'interfaccia. Ogni oggetto è rappresentato con specifiche dettagliate e un'icona distintiva, per favorire una rapida identificazione. Il progetto è stato pensato per sfruttare i concetti di *polimorfismo* e *incapsulamento*, offrendo una struttura flessibile ed estendibile per futuri sviluppi.

2 Descrizione del modello e uso del polimorfismo

Il modello logico del progetto si basa su una gerarchia di classi orientata a rappresentare gli oggetti gestiti nella biblioteca digitale. Alla radice della gerarchia vi è la classe astratta `BibliotecaItem`, che incapsula attributi comuni a tutte le risorse bibliotecarie, quali titolo, anno, genere, lingua, disponibilità, numero di copie e prestiti, immagine rappresentativa e descrizione.

Da `BibliotecaItem` derivano tre classi concrete specializzate:

- `Libro`, con attributi specifici come autore, numero di pagine, codice ISBN, target di età e indicazione della presenza di ebook.
- `Film`, che aggiunge informazioni come regista, protagonista, durata, premi ricevuti, paese di origine, sottotitoli e valutazione a stelle.
- `Rivista`, con dati specifici quali periodicità, editore, direttore, ISSN, elenco di articoli e volume.

Questa struttura consente di utilizzare il **polimorfismo** per trattare in modo uniforme collezioni eterogenee di oggetti `BibliotecaItem*`. Ad esempio, una lista di puntatori a `BibliotecaItem` può contenere oggetti di tipo `Libro`, `Film` o `Rivista`, e chiamando metodi virtuali come `prenota()` o `accept(Visitor*)` si attiveranno le versioni specifiche di ciascuna sottoclasse, senza bisogno di effettuare controlli espliciti sul tipo.

Questa dinamica è fondamentale in scenari quali:

- Filtraggio e ordinamento dinamico degli oggetti in base a categoria, lingua o altri attributi specifici, utilizzando metodi virtuali `getCategoria()`, `getLingua()` e simili.

Figura 1: Diagramma UML

- Supporto alla serializzazione e deserializzazione automatica in formato JSON tramite Visitor, che visita ogni oggetto concreto per estrarne i dati specifici.

- Personalizzazione dell'interfaccia utente con icone, colori e widget distintivi, generati dinamicamente in base al tipo dell'oggetto.

L'implementazione C++ delle classi riflette queste scelte architetturali: il costruttore di `BibliotecaItem` gestisce gli attributi comuni, mentre ciascuna sottoclasse estende la funzionalità con i propri campi e ridefinisce il metodo virtuale `accept` per supportare il pattern Visitor. I metodi `prenota` e `restituisci` sono implementati nella classe base, garantendo una logica coerente e riutilizzabile, con possibilità di override se necessario.

2.1 Pattern Visitor per estensione dinamica del comportamento

Per supportare operazioni diverse sugli oggetti derivati da `BibliotecaItem` senza modificare le classi stesse, è stato adottato il **pattern Visitor**. Questo pattern permette di definire nuove funzionalità esterne alla gerarchia, mantenendo separata la logica specifica per ogni tipo concreto.

Nel progetto è definita un'interfaccia astratta `Visitor` con metodi virtuali puri `visit` sovraccaricati per ogni classe derivata: `Libro`, `Film` e `Rivista`. Ogni sottoclasse implementa il metodo `accept` che accetta un puntatore a `Visitor` e invoca il metodo `visit` corrispondente al proprio tipo.

Sono stati implementati diversi Visitor concreti:FilterVisitor, e StyleVisitor

L'uso del pattern Visitor consente quindi di aggiungere facilmente nuove operazioni sugli oggetti senza alterare la struttura esistente delle classi modello, rispettando il principio di responsabilità singola e favorendo l'estensibilità del sistema.

3 Polimorfismo

Nel progetto è stato utilizzato un polimorfismo non banale, partendo da una classe base comune `BibliotecaItem` da cui derivano tre categorie distinte di oggetti della biblioteca: `Libro`, `Film` e `Rivista`. Ciascuna categoria possiede attributi specifici e unici che influenzano sia la visualizzazione che le operazioni di creazione e modifica.

Ad esempio, per i `Film` sono presenti attributi come la valutazione a stelle (da 1 a 5), premi vinti, nome del regista, protagonista e sottotitoli; per le `Riviste` sono definiti articoli, ISSN, volume, direttore e periodicità; mentre per i `Libri` sono inclusi ebook, target di età, numero di pagine e altri dettagli specifici.

Il polimorfismo viene sfruttato non solo per restituire oggetti specifici, ma anche in operazioni quali il filtraggio dinamico, la conversione automatica in formato JSON, la costruzione di widget personalizzati e l'assegnazione automatica di icone e colori distintivi per ciascuna categoria (libro, film, rivista), facilitando così la distinzione visiva all'interno dell'interfaccia utente.

Grazie al polimorfismo, chiamando metodi virtuali come `prenota()`, `getCategoria()` o `getLingua()` su ciascun puntatore, il programma invoca automaticamente la versione specifica implementata nella sottoclasse concreta (`Libro`, `Film` o `Rivista`). Questo permette di trattare uniformemente oggetti eterogenei, semplificando operazioni come il filtraggio e la visualizzazione dinamica.

In questo modo, l'interfaccia riesce a gestire dinamicamente la varietà di oggetti senza bisogno di controlli espliciti sul tipo o cast, mantenendo il codice pulito, modulare e facilmente estendibile.

4 Design Pattern: Visitor

All'interno del progetto è stato implementato il **design pattern Visitor** per gestire in maniera modulare e scalabile la logica associata ai diversi tipi di oggetti della gerarchia `BibliotecaItem` (`Libro`, `Film`, `Rivista`). Questo approccio ha permesso di separare la logica di visualizzazione, filtraggio e modifica dei dati dalla logica interna degli oggetti, rendendo il sistema più estendibile.

- **PageVisitor**: responsabile della costruzione dei widget personalizzati per ciascun tipo di oggetto, sia nella visualizzazione che nella modifica o creazione. Implementa i metodi `visit` per ogni sottoclasse di `BibliotecaItem`, richiamando funzioni specifiche (es. `setupFilmFields(film)`) per impostare i campi dedicati a ciascun tipo (es. "premi" per i film, "articoli" per le riviste).

- **FilterVisitor**: utilizzato nelle sezioni `AdminPage` e `UserPage` per implementare il filtraggio dinamico della collezione. Restituisce il tipo dell'oggetto, utile per applicare filtri per categoria (libro, film, rivista) e lingua. Grazie a questo approccio, l'interfaccia utente può adattarsi dinamicamente ai criteri scelti.

- **UpdateVisitor**: definito all'interno della classe `EditCreatePage`, viene utilizzato per aggiornare un oggetto esistente con i valori inseriti nei campi della GUI. Poiché è fortemente legato alla struttura dei widget, è implementato come classe interna per garantire coesione e semplicità di accesso agli elementi grafici.

- **StyleVisitor**: utilizzato per determinare dinamicamente l'icona e il colore associati a ciascun tipo di oggetto, migliorando la coerenza grafica dell'interfaccia. Evita l'uso di `dynamic_cast`, facendo uso del dispatch dinamico del pattern Visitor. È impiegato in contesti come la visualizzazione degli elementi grafici delle card.

L'approccio adottato consente di estendere facilmente il sistema: ad esempio, aggiungendo un nuovo tipo (`Vinile`), è sufficiente definire un nuovo metodo `visit` in ciascun visitor e una nuova sottoclasse di `BibliotecaItem`, senza dover modificare codice esistente.

5 Funzionalità implementate

Il progetto implementa una biblioteca virtuale con un'interfaccia utente interattiva, suddivisa in due sezioni principali: area **Admin** e area **Utente**, ognuna con funzionalità specifiche. Le funzionalità sono raggruppate in due categorie: **funzionali** ed **estetiche**.

5.1 Funzionalità Funzionali

- Gestione di tre tipologie di oggetti: **libro**, **rivista** e **film**.
- Conversione e **salvataggio automatico in formato JSON** al momento di aggiunta, modifica o eliminazione.
- Sistema di **ricerca testuale** e **filtri multipli** per categoria (libro, rivista, film) e lingua (italiano, spagnolo, inglese), con possibilità di selezionare o deselezionare singoli filtri oppure applicarli tutti.
- Conteggio in tempo reale dei **prestiti attivi**, aggiornato dinamicamente con ogni operazione di prenotazione o restituzione.
- Sistema di prenotazione che **impedisce duplicati** per lo stesso libro finché non viene restituito (area utente).
- Controllo sulle disponibilità: messaggi di avviso se l'oggetto è esaurito o se rimane una sola copia.
- **Paginazione iniziale** per distinguere l'accesso utente da quello amministrativo.
- Accesso Admin protetto da **password** (admin/admin) con opzione di *mostra/nascondi password con cambiemnto icona relativa.*
- In area Utente: funzionalità di **visualizzazione** e **prenotazione**, con possibilità di visualizzare e restituire i propri ordini da un menu a lista puntata.

5.2 Funzionalità Estetiche e Interfaccia Grafica

Barra dei menù superiore con funzionalità comuni e specifiche:

- Le funzionalità comuni comprendono: *home*, *ricerca per testo*, *filtro per categoria e lingua*, *conteggio prestiti attivi* (aggiornato dinamicamente) e *uscita dall'applicazione*, che richiede conferma da parte dell'utente.

- In modalità **Admin** è presente un pulsante *"Aggiungi"*, che apre un menu per la creazione di un nuovo oggetto (libro, rivista o film).
- In modalità **Utente**, al posto del pulsante "Aggiungi", è presente un pulsante *"Ordini"*, che mostra la lista dei contenuti prenotati, da cui è possibile effettuare la restituzione.

Icone grafiche nei pulsanti e nel menù per una migliore usabilità.

Ridimensionamento gestito, incluso scroll per descrizioni lunghe e campi aggiuntivi.

Navigazione intuitiva tra schermate (es. ritorno a menu scelta o admin dopo creazione/modifica).

- **Pulsante "Salva"** attivo solo in modalità modifica/creazione, disattivato nel menù scelta.
- Interfacce **diverse per Admin e Utente**, con stili e funzionalità dedicate.
- **Colori e stili grafici coerenti**, inclusi effetti *hover*, barra laterale, e testo selezionabile per copia/incolla.
- **Feedback grafico tramite messaggi e icone** per:
  - Password errata.
  - Prenotazione, modifica, creazione o restituzione riuscita.
  - Nessuna copia disponibile o tentativo di prenotare un oggetto già prenotato.
  - Ricerca senza risultati, con ritorno facilitato alla schermata precedente.
  - **Scorciatoie da tastiera implementate nella pagina iniziale principale**:
    - - CTRL+Q: chiude l'applicazione
    - - CTRL+A: apre la schermata di identificazione per l'accesso alla pagina Admin
    - - CTRL+U: apre direttamente l'area Utenti

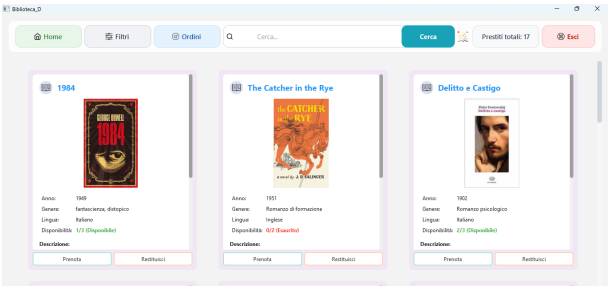

Figura 2: Interfaccia Utente

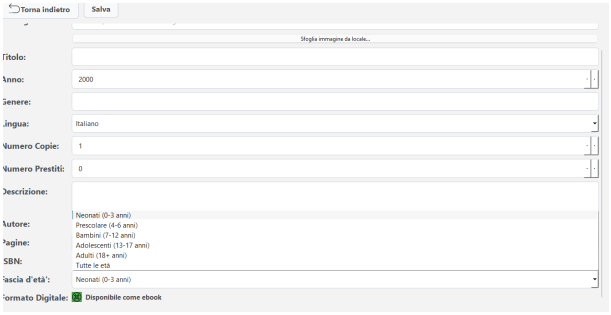

Figura 3: Per aggiungere

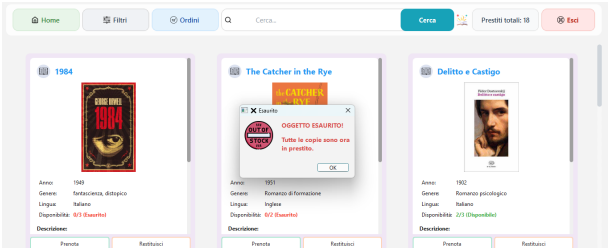

Figura 4: Errore: non ci sono più copie da prenotare

Figura 5: Una sola copia disponibile

6 Persistenza dei dati

Per la persistenza delle informazioni nella biblioteca virtuale ho scelto di utilizzare un file in formato `JSON`. Questo file contiene un vettore di oggetti, ognuno rappresentante un elemento della biblioteca come libri, riviste o film. Ogni oggetto è strutturato con coppie chiave-valore, in cui il tipo di classe è esplicitamente indicato tramite l'attributo `"classe"`.

Il file JSON viene aggiornato automaticamente ogni volta che, tramite l'applicazione, si inserisce o modifica un oggetto. Questa scelta garantisce una gestione semplice e diretta dei dati senza necessità di un database complesso.

Ogni oggetto include inoltre un percorso (path o URL) relativo all'immagine rappresentativa. Sebbene tale percorso sia specifico per ogni macchina su cui gira il programma, ho deciso di adottare questa soluzione per semplificare la fase di inserimento di nuovi elementi, accettando come compromesso una minor portabilità del progetto.

7 Rendicontazione ore

Attività	**Ore Previste**	**Ore Effettive**
Studio e progettazione	10	11
Sviluppo del codice del modello	10	12
Studio del framework Qt	10	12
Sviluppo del codice della GUI	10	18
Test e debug	5	7
Stesura della relazione	5	4
Totale	**50**	**64**

Tabella 1: Confronto tra ore previste ed effettive per le attività svolte

Commento finale: Ho impiegato più tempo nello sviluppo del codice GUI per capire come collegare fra di loro le varie interfacce e le scorciatoie per muovermi fra loro. Inoltre, ho impiegato più tempo nello studio di possibili stili grafici per aumentare l'esperienza visiva; in particolare mi sono soffermata anche su stili come CSS per implementare colori, effetti hover, stili grafici, ecc.