Corso di Ingegneria del Software Deliverable di progetto	**2024-2025**

“Ingegneria del Software”
2024-2025

Docente: Prof. Angelo Furfaro

Book Manager

Data	15/07/2025
Documento	Documento Finale – D3

Team Members		
Nome e Cognome	**Matricola**	**E-mail address**
Alessandro Bruno	**240036**	**brnlsn04b20d086e@studenti.unical.it**

List of Challenging/Risky Requirements or Tasks

Challenging Task	Date the task is identified	Date the challenge is resolved	Explanation on how the challenge has been managed
Rappresentazione in memoria della libreria e relativa implementazione	**8/07/2025**	**10/07/2025**	Ho deciso di supportare più formati di persistenza (JSON e SQLite) e permetterne nuove implementazioni
Implementazione del sistema di ricerca e filtro dei libri sfruttando le ottimizzazioni di ricerca dello specifico formato di persistenza	**10/07/2025**	**13/07/2025**	Introdotta l'interfaccia OptimizedSearch per distinguere i DAO con ricerca ottimizzata e utilizzarla in caso di ricerca con filtri.
Implementazione funzionalità Undo/Redo	**12/07/2025**	**13/07/2025**	Implementato CommandHistory con stack per gestire cronologia comandi. Ogni operazione CRUD incapsulata in command separato.
Implementazione GUI	**11/07/2025**	**13/07/2025**	Utilizzo della libreria JavaFX

A. Stato dell'Arte

Il progetto BookManager si inserisce nel dominio delle applicazioni di gestione bibliotecario personale. L'analisi dello stato dell'arte ha rivelato diverse soluzioni esistenti:

Soluzioni Commerciali Analizzate

- Calibre: Software desktop open-source per gestione e-book con funzionalità avanzate di conversione
- TV-Time: Software mobile per gestione di tracking di film e serie tv viste e da vedere con sistema di rating

Ispirazione Architettuale

Il design di BookManager si ispira ai principi di Calibre per la gestione locale dei dati, combinandoli con l'usabilità di TV-Time e le capacità di filtering avanzato di applicazioni moderne.

B. Raffinamento dei Requisiti

A.1 Servizi (con prioritizzazione)

- **S01 – Visualizzazione libri (Importanza:** Alta **| Complessità:** Bassa**)**
  L'utente è in grado di visualizzare tutti i libri presenti nella libreria catalogati a tabella. Per ogni libro sarà possibile visualizzare titolo, autore, stato lettura e valutazione.

- **S02 – Dettagli libro (Importanza:** Alta **| Complessità:** Bassa**)**
  L'utente è in grado di selezionare un libro e di esso visualizzarne i dettagli, includendo dunque genere e codice ISBN.

- **S03 – Gestione CRUD Libri (Importanza:** Alta **| Complessità:** Media**)**
  L'utente è in grado di: aggiungere nuovi libri fornendone gli attributi obbligatori (codice ISBN, titolo, stato lettura) ed eventualmente gli attributi opzionali (autore, genere); modificare libri presenti nella libreria aggiornando tutti gli attributi tranne il codice ISBN (identificativo del libro); rimuovere libri dalla libreria selezionandoli e premendo il tasto 'rimuovi'.

- **S04 – Ricerca e Filtri (Importanza:** Alta **| Complessità:** Alta**)**
  L'utente è in grado di cercare libri per titolo, autore o codiceISBN, fornendo di questi attributi anche solo una parte. Inoltre è possibile filtrare i libri visualizzati nella libreria in base alla valutazione, allo stato di lettura, al genere o ad una combinazione di questi.

- **S05 – Sistema Undo/Redo (Importanza:** Media **| Complessità:** Media**)**
  L'utente può annullare le ultime operazioni (CRUD) eseguite tramite il tasto 'undo' e in caso rieseguirle tramite il tasto 'redo'. L'utente può aspettarsi il comportamento dei due tasti visualizzando l'operazione che verrà annullata o rieseguita portando il cursore del mouse sopra ai due pulsanti.

- **S06 – Persistenza multi-formato (Importanza:** Media **| Complessità:** Alta**)**
  All'avvio dell'applicazione, l'utente potrà scegliere se creare una nuova libreria o caricarne una esistente, in entrambi i casi l'utente può scegliere se lavorare con file di tipo JSN o SQLite. È inoltre possibile cambiare la libreria corrente chiudendo la sua finestra di visualizzazione e caricandone una nuova.

- **S07 – Sistema di cache (Importanza:** Bassa **| Complessità:** Media**)**
  Sistema di cache in memoria per migliorare performance con dataset grandi, consentendo l'applicazione di operazioni di ricerca, filtro e ordinamento senza ricaricare i dati dalla memoria.

- **S08 – Ordinamento dinamico (Importanza:** Media **| Complessità:** Bassa**)**
  L'utente può selezionare un attributo per il quale ordinare i libri nella visualizzazione, e scegliere se applicare l'ordinamento in senso crescente o decrescente.

- **S09 – Pulizia filtri e ordinamento (Importanza:** Bassa **| Complessità:** Bassa**)**
  L'utente può facilmente pulire i valori di ricerca, filtro e ordinamento precedentemente imposti.

- **S10 – Validazione (Importanza:** Alta **| Complessità:** Bassa**)**
  Prima di essere salvati in memoria, i libri aggiunti o modificati dall'utente vengono validati per assicurarsi che il codice ISBN sia valido (10 o 13 cifre) e unico nella libreria, che gli attributi obbligatori siano non vuoti e non troppo lunghi.

A.2 Requisiti non Funzionali

Usabilità:
L'interfaccia è comprensibile a tutti gli utenti, attraverso un UI responsive e con feedback visuale. Qualsiasi funzionalità primaria è raggiungibile con non più di 3 click.

Affidabilità:
l'applicazione garantisce che tutte le operazioni eseguite dall'utente siano coerenti rispetto alla logica applicativa definita, assicurando l'integrità e la precisione dei dati gestiti.

Evolvibilità:
Il sistema è strutturato in modo modulare e flessibile in modo da supportare futuri aggiornamenti e l'introduzione di nuove funzionalità.

Verificabilità:
Per l'applicazione è prevista una fase di testing per verificare il corretto funzionamento delle principali funzionalità dell'applicazione e per assicurare la qualità e la robustezza del prodotto finale.

A.3 Scenari d'uso dettagliati

Scenario 1: Aggiunta Nuovo Libro
Precondizioni: Applicazione avviata con libreria caricata o creata
Flusso Principale:

1. Utente clicca pulsante '+' nella toolbar
2. Si apre pannello di inserimento con form vuoto
3. Utente compila titolo, autore, ISBN, seleziona genere e stato
4. Sistema valida dati in tempo reale mostrando errori
5. Utente clicca "Salva" e libro viene aggiunto alla collezione
6. Vista principale si aggiorna mostrando il nuovo libro
7. Viene visualizzata finestra dettagli del libro appena creato

Flusso Alternativo 4a: Dati non validi

4a1. Sistema mostra errore sotto il pulsante Salva
4a2. Pulsante 'Salva' rimane disabilitato
4a3. Ritorna al passo 3

Scenario 2: Recupero da errore con Undo
Precondizioni: Utente ha appena eliminato o modificato un libro per sbaglio
Flusso Principale:

1. Utente si accorge dell'errore immediatamente
2. Tooltip su pulsante 'Undo' mostra "Annulla:Rimozione(o modifica) libro X"
3. Utente clicca pulsante 'Undo' nella toolbar in alto
4. Sistema ripristina il libro a prima della rimozione o modifica
5. Vista si aggiorna mostrando il libro ripristinato
6. Pulsante 'Redo' diventa abilitato
7. Tooltip su pulsante 'Redo' mostra "Ripeti:Rimozione(o modifica) libro X"

Scenario 3: Modifica Stato di Lettura e Valutazione
Precondizioni: Libro presente nella libreria con stato "In Lettura"
Flusso Principale:

1. Utente cerca il libro tramite titolo nella barra di ricerca
2. Clicca sulla card del libro per aprire i dettagli
3. Clicca pulsante "Modifica" (icona matita)
4. Cambia stato da "In Lettura" a "Letto"
5. Aggiorna valutazione usando lo slider (es. 4 stelle)
6. Clicca "Salva" per confermare modifiche
7. Sistema salva le modifiche
8. Dettagli e vista della libreria si aggiornano mostrando nuovo stato e valutazione

Flusso Alternativo 1a: Libro non trovato nella ricerca

1a1. Sistema non mostra risultati per la ricerca
1a2. Utente può provare termine o attributo di ricerca diverso
1a3. Ritorna al passo 1

Scenario 4: Caricamento Libreria Esistente
Precondizioni: File libreria esistente (JSON o SQLite) sul sistema
Flusso Principale:
1. Utente avvia applicazione e vede schermata di benvenuto
2. Clicca "Carica Libreria Esistente"
3. Si apre dialog di selezione file
4. Naviga e seleziona file libreria (es. "mia_collezione.db")
5. Sistema carica e valida il file
6. Sistema carica automaticamente tutti i libri
7. Vista principale mostra collezione con tutti i libri catalogati
8. Utente può iniziare a navigare, cercare e modificare

Flusso Alternativo 5a: File corrotto o non valido
5a1. Sistema rileva errore
5a2. Mostra dialog error
5a3. Permette selezione di file diverso
5a4. Ritorna al passo 2

Scenario 5: Ordinamento e filtro libreria grande
Precondizioni: Collezione grande caricata in memoria
Flusso Principale:
1. Utente vuole trovare i libri con valutazione più alta
2. Seleziona "Valutazione" nel menu ordinamento nella toolbar
3. Attiva Checkbox "Decrescente" per ordine inverso
4. Sistema riordina immediatamente libri per valutazione (5→1)
5. Utente scorre le prime posizioni vedendo tutti i libri a 5 stelle
6. Decide di filtrare solo genere "Fantascienza" per affinare ricerca
7. Selezione menu filtri
8. Sistema mostra un menu a tendina con i possibili filtri da selezionare
9. Utente seleziona checkbox ‘Fantascienza’
10. Sistema mostra solo fantascienza ordinata per valutazione

Scenario 6: Pulizia filtri
Precondizioni: Multipli filtri attivi, termine di ricerca non vuoto e attributo di ordinamento diverso da ‘Titolo’
Flusso Principale:
1. Utente clicca pulsante ‘Pulisci filtri’ nella toolbar
2. Sistema rimuove tutti i filtri
3. Pulisce ricerca testuale
4. Imposta ordinamento di default per titolo crescente
5. Vista torna a mostrare tutta la collezione ordinata per titolo

6. Utente può ricominciare a navigare

Scenario 7: Cambio libreria
Precondizioni: libreria1 caricata e visualizzata, file libreria2 esistente nel sistema
Flusso Principale:
1. Utente decide di spostarsi sulla libreria2
2. Chiude la finestra di visualizzazione di libreria1
3. Sistema apre automaticamente schermata di benveuto
4. Segue Scenario 4 – Caricamento Libreria Esistente

Scenario 8: Visualizzazione libri
Trigger: Libreria appena caricata o libro appena aggiunto, modificato o rimosso
Flusso Principale:
1. Utente ha appena caricato la libreria o aggiornato la collezione
2. Vista dei libri si aggiorna
3. Sistema visualizza tutti i libri
4. Ordina libri visualizzati secondo criterio di ordinamento impostato

Flusso Alternativo 3a: Sono impostati dei filtri e/o termini di ricerca
3a1. Sistema visualizza libri conformi ai filtri e al termine di ricerca (rispetto all'attributo di ricerca selezionato)
3a2. Ritorna al passo 4

A.4 Excluded Requirements

Multi-utente:
Applicazione pensata per uso singolo

Paginazione risultati:
Le collezioni personali raramente superano dimensioni che richiedono paginazione. Aggiungere paginazione complicherebbe inutilmente l'interfaccia utente.

Lazy loading/ Virtual scrolling
Per lo stesso motivo della paginazione. JavaFX gestisce efficacemente il rendering di migliaia di elementi nella FlowPane. L'overhead implementativo non è giustificato per l'uso target.

Conversione formato libreria:
Non è supportata la conversione diretta da JSON a SQLite o viceversa. L'utente deve scegliere il formato alla creazione e mantenerlo. Motivazioni: complessità implementativa alta, use case raro (l'utente tipicamente sceglie un formato e lo mantiene).

Supporto database esterni:

Non è stato implementato il supporto per MySQL, PostgreSQL o altri RDBMS. Nonostante ciò, la modularizzazione del progetto ne rende abbastanza semplice la realizzazione.

A.5 Assunzioni

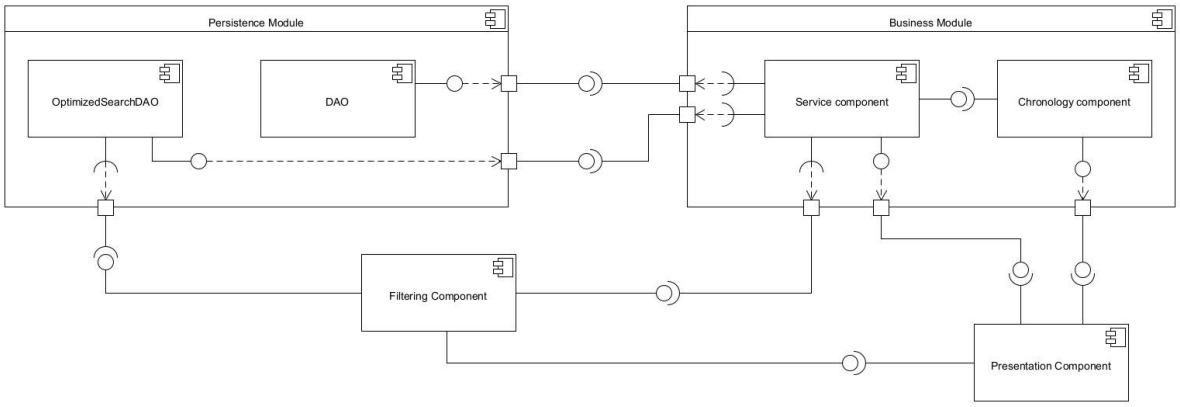
AS-01: Collezioni tipiche non superano 5000 libri per uso personale
AS-02: Dati inseriti manualmente dall'utente (no import automatico)
AS-03: ISBN utilizzati sono unici per libro e seguono gli standard internazionali (ISBN-10 o ISBN-13)
AS-04: Un libro può avere al più un genere (il più rilevante)

A.6 Use Case Diagrams

C. Architettura Software

C.1 The static view of the system: Component Diagram

Il seguente Component Diagram mostra l'architettura modulare del sistema BookManager organizzata in quattro moduli principali. Le dipendenze seguono una struttura layered unidirezionale: Presentation → Business → Persistence, con il Filtering Component che agisce come utility condivisa accessibile da tutti i layer senza creare dipendenze circolari.



Presentation Comonent:
Contiene i controller JavaFX responsabili dell'interfaccia utente (WelcomeController, LibreriaController, UnifiedBookPanelController). Richiede servizi dal Business Module per le operazioni sui dati e dal Filtering Component per la costruzione e gestione dei filtri nell'interfaccia utente. Corrisponde totalmente al package *view.*

Business Module:
Nucleo della logica applicativa, composto da Service Component (coordinamento operazioni CRUD e ricerca) e Chronology Component (gestione sistema undo/redo tramite pattern Command). Il Service Component fornisce interfacce di alto livello al Presentation Layer e coordina l'accesso ai dati tramite il Persistence Module. Corrisponde al package *service* e *command*.

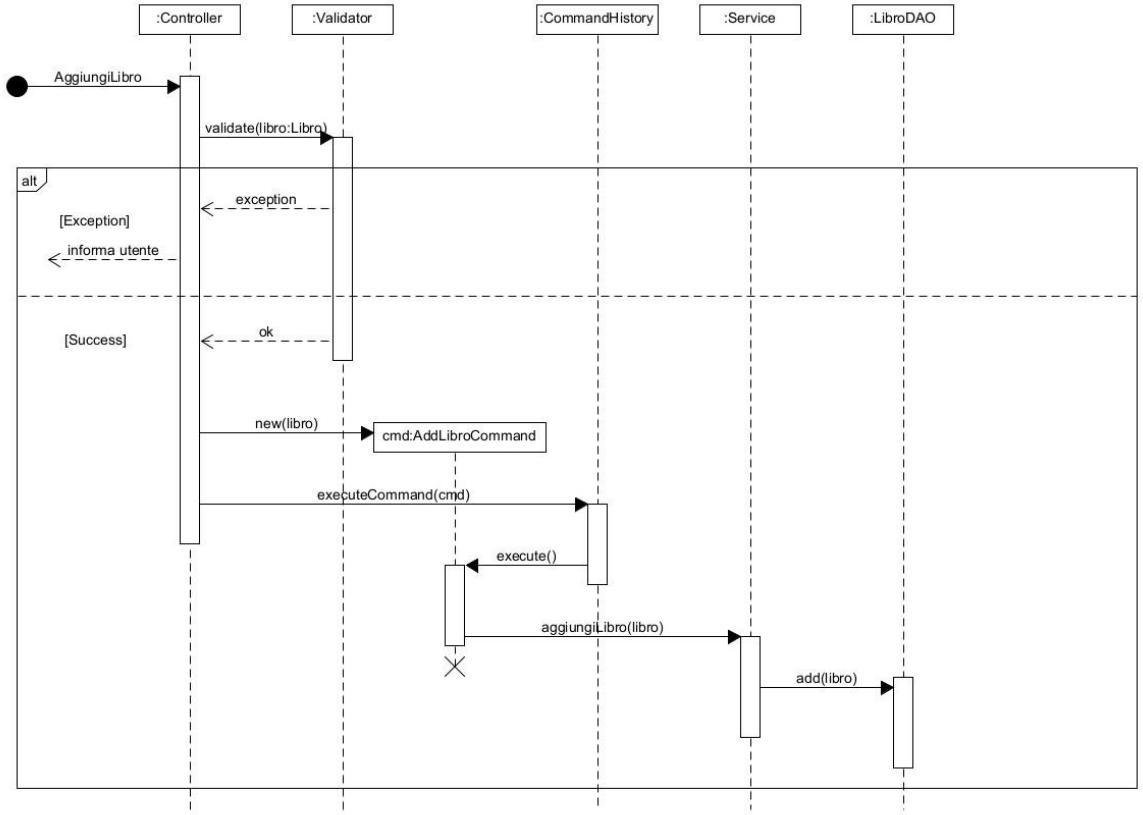
Persistence Module:
Gestisce l'accesso e la persistenza dei dati attraverso il componente DAO che implementa l'interfaccia LibroDAO. Il componente OptimizedSearchDAO fornisce capacità di ricerca ottimizzate (principalmente per implementazioni database) attraverso l'interfaccia OptimizedSearch opzionale. Corrisponde al package *dao*.

Filtering Component:
Modulo cross-cutting che fornisce il sistema di filtri componibili utilizzato da tutti gli altri layer. Contiene l'interfaccia Filter<T> con le sue implementazioni concrete (TitoloFilter, GenereFilter, etc.) e la classe SearchCriteria che coordina filtri e criteri di ordinamento per query complesse. Corrisponde al package *filter*.

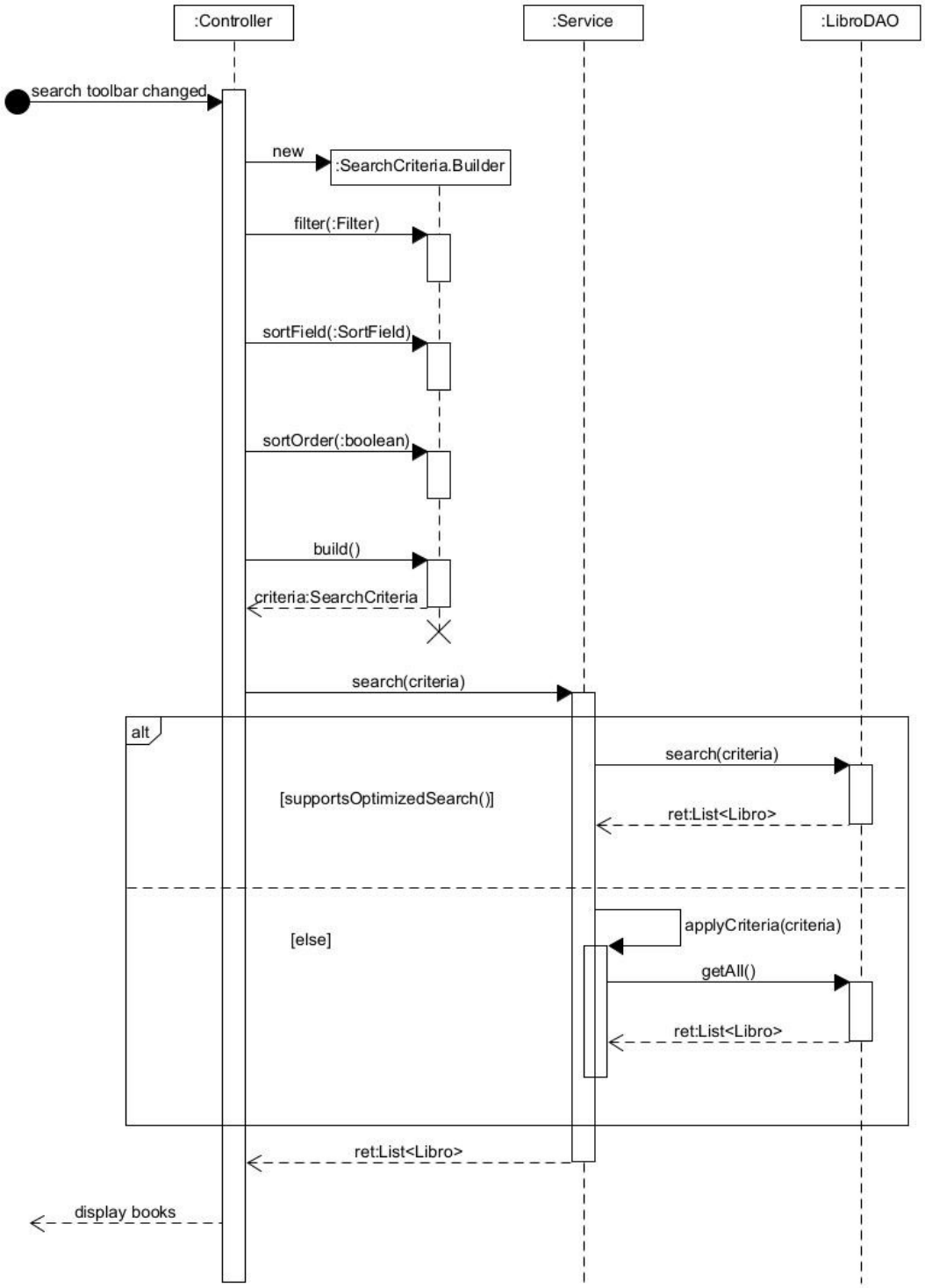
Model Dependencies:
Tutti i componenti dipendono dalle entità del dominio (Libro, Genere, StatoLettura) che rappresentano il modello dati condiviso. Queste dipendenze non sono rappresentate esplicitamente nel diagram per mantenere la leggibilità, ma costituiscono il vocabolario comune utilizzato da tutti i layer per la rappresentazione e manipolazione dei dati.

C.2 The dynamic view of the software architecture: Sequence Diagram

Scenario: Aggiunta libro



Scenario: Ricerca con filtri e ordinamento



D. Dati e loro modellazione (se il sistema si interfaccia con un DBMS)

Dati Principali:

- **Collezione Libri:**
  Metadati bibliografici inseriti manualmente dall'utente

- **Enumerazioni**:
  Valori predefiniti per Genere e StatoLettura (embedded nel codice)

- **Nessun dato esterno:**
  Il sistema non si integra con database bibliografici o servizi cloud esterni.

Schema Database SQLite

```
CREATE TABLE libri (
   isbn TEXT PRIMARY KEY,
   titolo TEXT NOT NULL,
   autore TEXT,
   genere TEXT NOT NULL,
   valutazione INTEGER CHECK(valutazione >= 0 AND valutazione <= 5),
   stato TEXT NOT NULL
);
```

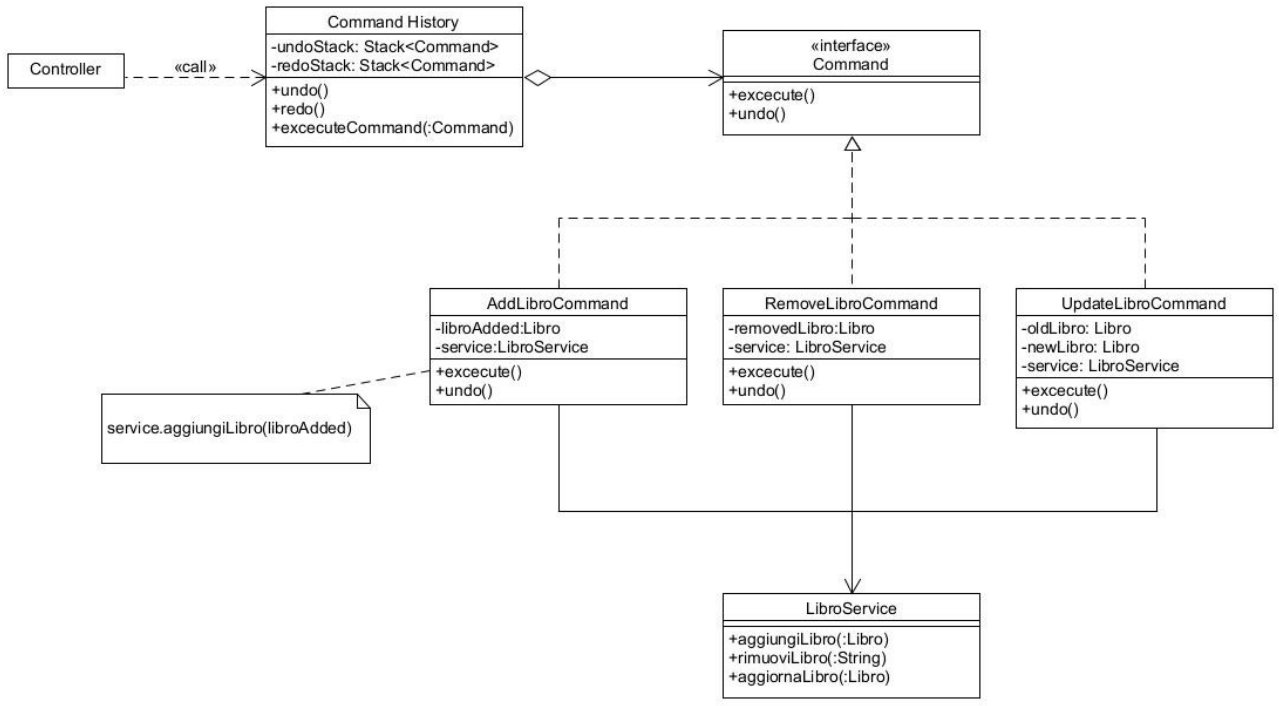
E. Scelte Progettuali (Design Decisions)

Le scelte progettuali da affrontare per la realizzazione di questo progetto sono state molteplici, si è cercato di dare importanza ad aspetti che permettessero, fin dalla fase di progettazione, di garantire il soddisfacimento dei principali requisiti definiti inizialmente. Di seguito le 5 decisioni progettuali principali:

Pattern Command per Undo/Redo

Problema:
Necessità di supportare annullamento e ripetizione di tutte le operazioni di modifica.

Soluzione: Implementazione pattern Command con CommandHistory centralizzata per incapsulare ogni operazione di modifica e gestirne la cronologia tramite un history stack-based.



Elementi chiave del Diagram:

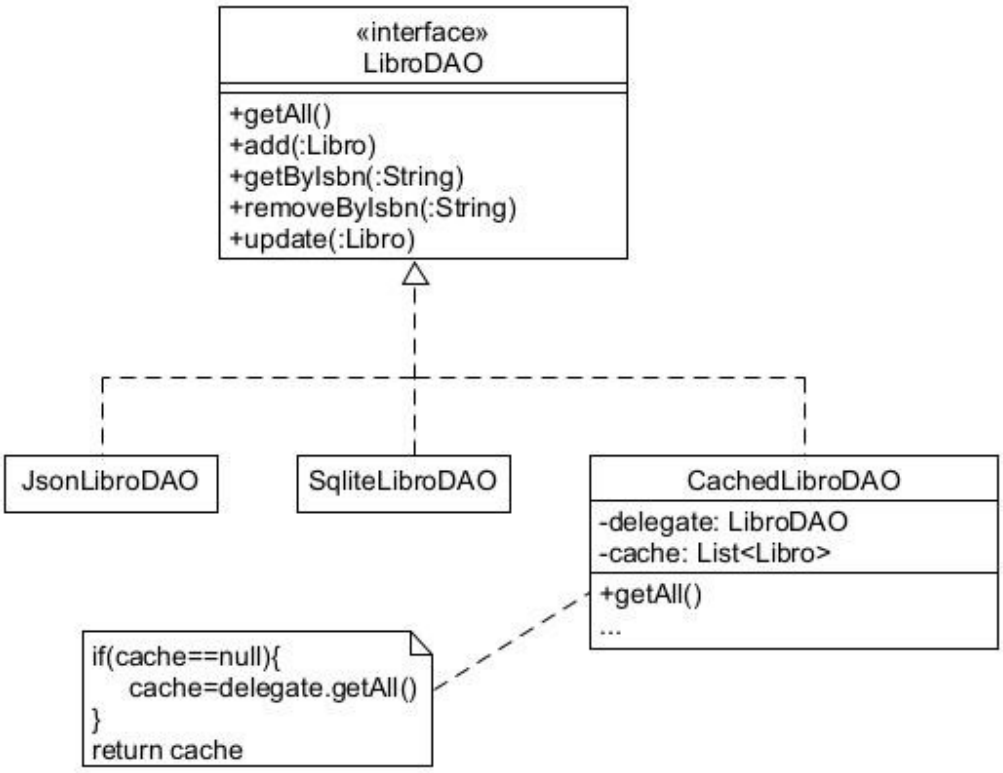
- **Interfaccia Command:**
  Definisce il contratto per incapsulare operazioni reversibili

- **Concrete Commands:**
  Implementano le diverse operazioni fornendone un metodo per eseguirle ed uno per annullarle
- **Command History (Invoker):**
  Gestisce i command, salvandoli in due stack per consentire le operazioni di undo e redo.
- **Libro Service (Reciever):**
  Esegue operazioni effettive sui dati, è agnostico del pattern Command
- **Controllers (Client):**
  Creano command specifici e li invocano tramite CommandHistory

Pattern Decorator per supporto cache

Problema: Miglioramento performance per dataset grandi senza modificare DAO esistenti

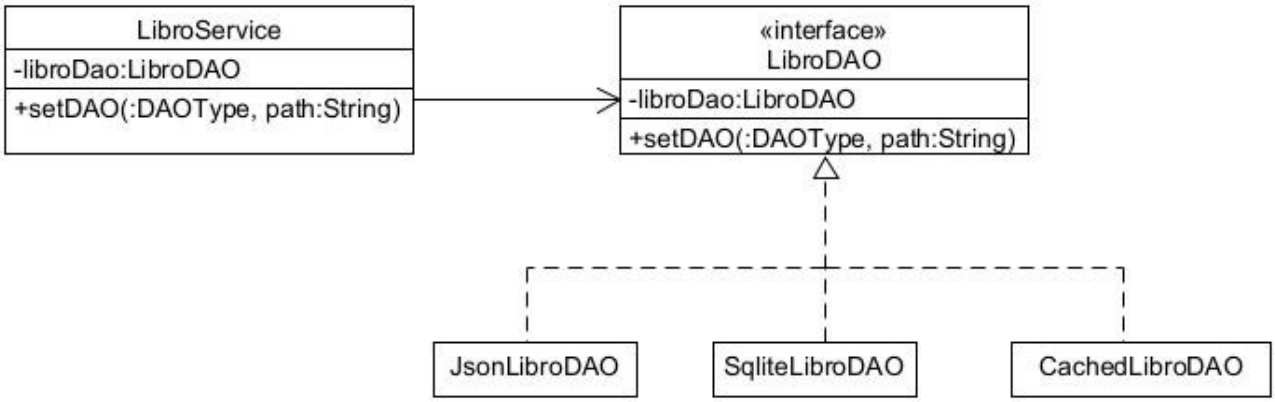
Soluzione: Pattern Decorator per aggiungere cache trasparente



Pattern Strategy per multiple tecniche di persistenza

Problema: Supportare multiple strategie di persistenza (JSON, SQLite, Cached) e permettere switch a runtime

Soluzione: Rendere l'interfaccia LibroDAO Strategy



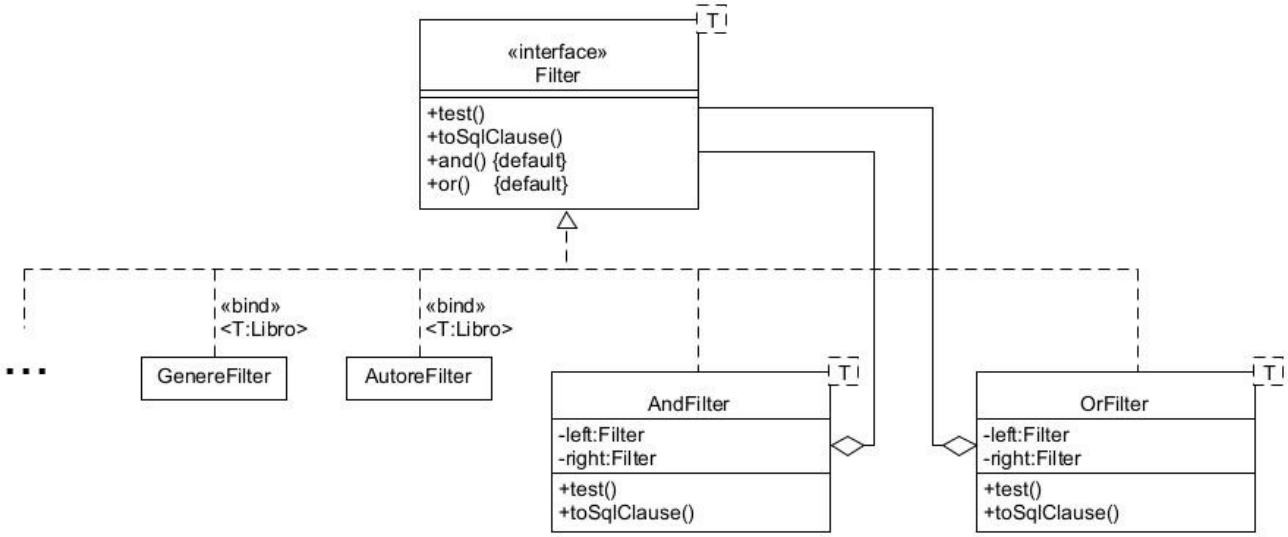
Elementi chiave del Diagram:

- **LibroDAO (Strategy Interface):**
  Contratto comune per tutte le strategie, comportamento intercambiabile a runtime.
- **Concrete Strategies:**
  Implementazioni delle varie strategie di persistenza (JSON, SQlite, Cached)
- **Libro Service (Context):**
  Mantiene riferimento alla strategia corrente a cui delega le operazioni ai dati.
  Permette switching di strategia tramite setDAO.

Pattern Composite per i filtri

Problema: combinare filtri semplici (titolo, autore, genere) in espressioni complesse usando operatori logici (AND, OR) creando alberi di filtri arbitrariamente profondi e complessi.

Soluzione: Composite pattern



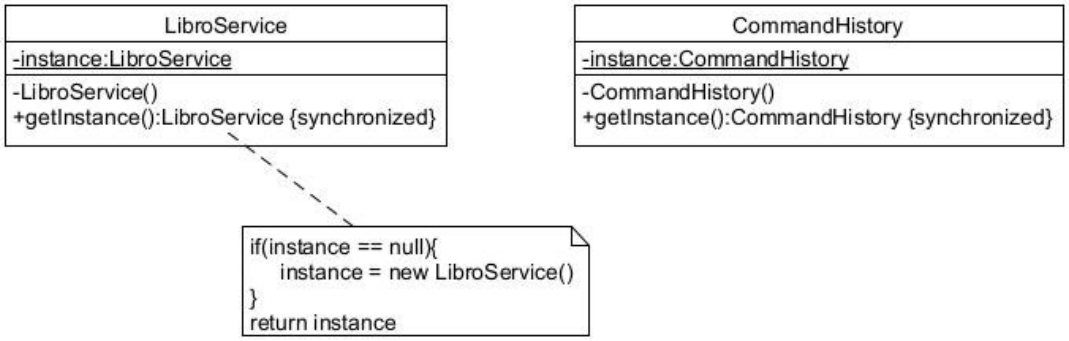
Elementi chiave del Diagram:

- **Interface Filter<T> (Comonent):**
  Interfaccia comune per tutti i filtri
- **Leaf Components (GenereFilter, AutoreFilter, ...)**
  Filtri atomici su singoli campi
- **Composite Components (AndFilter, OrFilter):**
  Contengono due child filters che compongono le due clausole logiche da porre in and o or. Possono avere come figli altri composite components, creando alberi di filtri arbitrariamente profondi ed attraversandoli ricorsivamente tramite le chiamate test() e toSqlClause()

Pattern Singleton per LibroService e CommandHistory

Problema: LibroService e CommandHistory devono essere condivisi tra tutti i controller dell'applicazione mantenendo stato globale consistente e coordinando operazioni su un'unica istanza di configurazione e cronologia.

Soluzione: Rendere le due classi singleton, garantendo l'esistenza di un unica istanza per l'intera esecuzione dell'applicazione.



F. Progettazione di Basso Livello

Questa sezione presenta i dettagli implementativi delle scelte architetturali più significative, mostrando come i design pattern identificati nelle scelte progettuali sono stati concretamente implementati nel codice per garantire flessibilità, manutenibilità ed estensibilità del sistema.

Command Pattern

Interfaccia base del command:

```
/**
 * Interfaccia base per il pattern Command.
 * Permette di incapsulare operazioni come oggetti, abilitando undo/redo per il controller.
 */
public interface Command {  14 usages  3 implementations  isederiso351 *
    /**
     * Esegue il comando.
     *
     * @throws BookManagerException se l'operazione fallisce
     */
    void execute() throws BookManagerException;  12 usages  3 implementations  isederiso351

    /**
     * Annulla l'operazione eseguita dal comando.
     *
     * @throws BookManagerException se l'annullamento fallisce
     */
    void undo() throws BookManagerException;  6 usages  3 implementations  isederiso351

    /**
     * Indica se questo comando può essere annullato.
     * Alcuni comandi potrebbero non essere reversibili.
     *
     * @return true se il comando può essere annullato
     */
    default boolean canUndo() { return true; }

    /**
     * Restituisce una descrizione dell'operazione per logging e UI.
     *
     * @return descrizione dell'operazione
     */
    String getDescription();  18 usages  3 implementations  isederiso351
}
```

Esempio di implementazione concreta:

```java
public class AddLibroCommand implements Command {  9 usages  isederiso351 *

    private static Logger logger = LoggerFactory.getLogger(AddLibroCommand.class);  2 usages

    private final LibroService service;  3 usages
    private final Libro libro;  5 usages

    public AddLibroCommand(LibroService service, Libro libro) {  4 usages  isederiso351
        this.libro = libro;
        this.service = service;
    }

    @Override  12 usages  isederiso351
    public void execute() throws BookManagerException {
        service.aggiungiLibro(libro);
        logger.info("Eseguito: {}", getDescription());
    }

    @Override  6 usages  isederiso351
    public void undo() throws BookManagerException {
        service.rimuoviLibro(libro.getIsbn());
        logger.info("Annullato: {}", getDescription());
    }

    @Override  18 usages  isederiso351 *
    public String getDescription() {
        return "Aggiunta libro: " + libro.getTitolo() + " (ISBN: " + libro.getIsbn() + ")";
    }
}
```

CommandHistory, l'Invoker che gestisce le cronologia, ogni volta che verrà richiesta l'invocazione di un comando con executeCommand(), oltre ad eseguirlo, lo salverà nello stack undoStack, pronto ad essere prelevato e annullato in caso di undo(). In caso di undoStack pieno, l'invoker "dimenticherà" il comando più vecchio, consentendo quindi sempre all'utente di tornare indietro di maxHistorySize(50) passi.

```java
/**
 * Esegue un comando e lo aggiunge alla cronologia.
 *
 * @param command comando da eseguire
 * @throws BookManagerException se l'esecuzione fallisce
 */
public void executeCommand(Command command) throws BookManagerException {  10 usages  isederiso351 *
    command.execute();

    undoStack.push(command);
    redoStack.clear(); // Cancella la cronologia redo dopo una nuova operazione

    while (undoStack.size() > maxHistorySize) {
        undoStack.removeFirst();
    }

    logger.debug("Comando eseguito e aggiunto alla cronologia: {}", command.getDescription());
}
```

Di seguito il metodo undo() che preleva tramite pop() l'ultimo command eseguito e chiama su esso il metodo undo(). Il metodo redo() è analogo.

```java
/**
 * Annulla l'ultimo comando eseguito.
 *
 * @throws BookManagerException se l'annullamento fallisce
 */
public void undo() throws BookManagerException {  5 usages  isederiso351
    if (undoStack.isEmpty()) {
        throw new BookManagerException("Nessun comando da annullare");
    }

    Command command = undoStack.pop();

    if (!command.canUndo()) {
        throw new BookManagerException("Il comando non può essere annullato: " + command.getDescription());
    }

    command.undo();
    redoStack.push(command);

    logger.info("Comando annullato: {}", command.getDescription());
}
```

Decorator Pattern

Data la seguente interfaccia del servizio LibroDAO:

```java
/**
 * Interfaccia per l'accesso e la gestione dei dati relativi ai libri.
 * Rappresenta un contratto che può essere implementato con varie tecnologie di persistenza
 * (es. file JSON, database SQLite, cache in memoria, ecc.).
 */
public interface LibroDAO {  25 usages  3 implementations  isederiso351

    /** Restituisce la lista completa dei libri memorizzati. ...*/
    List<Libro> getAll() throws DAOException;  20 usages  3 implementations  isederiso351

    /** Sovrascrive l'intera collezione di libri persistendo i dati forniti. ...*/
    void saveAll(List<Libro> libri) throws DAOException;  13 usages  3 implementations  isederiso351

    /** Cerca un libro tramite il suo ISBN ...*/
    Optional<Libro> getByIsbn(String isbn) throws DAOException;  5 usages  3 implementations  ised

    /** Aggiunge un nuovo libro alla collezione ...*/
    void add(Libro libro) throws LibroAlreadyExistsException, DAOException;  3 implementations

    /** Rimuove un libro tramite il suo ISBN ...*/
    void removeByIsbn(String isbn) throws LibroNotFoundException, DAOException;  8 usages  3 im

    /** Aggiorna le informazioni di un libro già presente, identificato da ISBN ...*/
    void update(Libro libro) throws LibroNotFoundException, DAOException;  6 usages  3 implement

    /** Indica se questa implementazione preferisce operazioni batch (saveAll) ...*/
    default boolean prefersBatchOperations() { return false; }
}
```

Il Decorator CachedLibroDAO, aggiunge a qualsiasi LibroDAO datogli nel costruttore, la funzionalità di caching, attingendo ad essa ogni volta che è possibile e invalidandola ogni volta che si presenta un errore. Si veda la seguente sua implementazione del metodo add():

```
@Override  isederiso351
public void add(Libro libro) throws LibroAlreadyExistsException, DAOException {
    try {
        if (getCache().contains(libro)) {
            logger.warn("Tentativo di aggiunta libro già presente in cache con ISBN {}", libro.getIsbn());
            throw new LibroAlreadyExistsException(libro.getIsbn());
        }

        cache.add(libro);

        if (delegate.prefersBatchOperations()) {
            delegate.saveAll(cache);
            logger.debug("Usata strategia batch per delegate {}", delegate.getClass().getSimpleName());
        } else {
            delegate.add(libro);
            logger.debug("Usata strategia singola per delegate {}", delegate.getClass().getSimpleName());
        }

        logger.info("Libro aggiunto a cache e persistenze: {} (ISBN: {})", libro.getTitolo(), libro.getIsbn());

    } catch (DAOException | LibroAlreadyExistsException e) {
        logger.error("Errore durante add, invalidazione cache", e);
        invalidateCache();
        throw e;
    }
}
```

Si osserva la presenza nell'interfaccia LibroDAO del metodo prefersBatchOperations() per ottimizzare l'aggiunta di libri del CachedLibroDAO. Se infatti si volesse aggiungere un libro appena inserito in cache, ma il libroDAO da delegare per aggiungere un libro è costretto a scaricare tutta la collezione, aggiungere il libro, e poi ricaricarla (esempio JSON) allora converrebbe chiamare direttamente delegate.saveAll(cache) che carica la cache in memoria. Se invece il libroDAO da delegare supporta nativamente l'inserimento di un singolo libro (esempio SQL) allora converrà chiamare delegate.add(libro).

Strategy Pattern

L'interfaccia Libro DAO appena visto, funge pure da Strategy nei confronti di LibroService, è infatti possibile nel service, modificare a runtime la strategia di persistenza scelta tramite il metodo setDAO():

```
public final class LibroService {  32 usages  isederiso351 *
    public void setDAO(DAOType type, String path) {  6 usages  isederiso351
        this.libroDAO = DAOFactory.createDAO(type, path);
        logger.info("Strategia DAO cambiata a: {}", type.name());
    }
```

Per creare il nuovo libroDAO, il service utilizza una classe DAOFactory che, dato il DAOType e il percorso del file di persistenza, restituisce una nuova strategia:

```
/**
 * Factory per la creazione di istanze LibroDAO.
 * Nasconde i dettagli di implementazione e le dipendenze specifiche dei DAO
 * dal resto dell'applicazione.
 */
public class DAOFactory {  3 usages  isederiso351 *
    private static final Logger logger = LoggerFactory.getLogger(DAOFactory.class);  1 usage

    /**
     * Crea un'istanza di LibroDAO del tipo specificato.
     *
     * @param type tipo di DAO da creare
     * @param path percorso del file o database (per JSON sarà il file .json,
     *             per SQLite sarà il nome del file .db)
     * @return istanza del DAO richiesto
     * @throws IllegalArgumentException se il tipo non è supportato
     */
    public static LibroDAO createDAO(DAOType type, String path) {  1 usage  isederiso351

        if (path == null || path.trim().isEmpty()) {
            throw new IllegalArgumentException("Il percorso non può essere null o vuoto");
        }
        LibroDAO dao = switch (type) {
            case SQLITE -> new SqliteLibroDAO( url: "jdbc:sqlite:" + path);
            case JSON -> new JsonLibroDAO(path);
            case CACHED_JSON -> new CachedLibroDAO(new JsonLibroDAO(path));
        };
        logger.debug("Creato DAO di tipo {} con successo", type);
        return dao;
    }
}
```

Composite Pattern

Interfaccia Filter<T> che rappresenta il componente generico della struttura ad albero che il composite ci permetterà di costruire:

```java
public interface Filter<T> {  8 implementations
    boolean test(T item);  8 implementations  ise
    String toSqlClause();  8 implementations  ise

    default Filter<T> and(Filter<T> f) {  5 us
        return new AndFilter<>( left: this, f);
    }

    default Filter<T> or(Filter<T> f) {  5 usa
        return new OrFilter<>( left: this,f);
    }
}
```

Esempio di un'implementazione foglia (filtro atomico su singolo attributo):

```java
public class AutoreFilter implements Filter<Libro>{  3 usages  isederiso351
    private final String autore;  3 usages

    public AutoreFilter(String autore) {  2 usages  isederiso351
        this.autore = autore != null ? autore.trim() : "";
    }

    @Override  isederiso351
    public boolean test(Libro libro) {
        return libro.getAutore().toLowerCase().contains(autore.toLowerCase());
    }

    @Override  isederiso351
    public String toSqlClause() {
        return "autore LIKE '%" + autore + "%'";
    }
}
```

Una delle due implementazione di Composite:

```java
public class AndFilter<T> implements Filter<T>{  1 usage  isederiso351

    private final Filter<T> left, right;  3 usages

    public AndFilter(Filter<T> left, Filter<T> right) {  1 usage  isederiso351
        this.left = left;
        this.right = right;
    }

    @Override  isederiso351
    public boolean test(T item) {
        return left.test(item) && right.test(item);
    }

    @Override  isederiso351
    public String toSqlClause() {
        return "(" + left.toSqlClause() + " AND " + right.toSqlClause() + ")";
    }
}
```

Singleton pattern:

Singleton LibroService

```java
public final class LibroService {  32 usages  isederiso351 *

    private static final Logger logger = LoggerFactory.getLogger(LibroService.class);

    // Singleton instance
    private static volatile LibroService instance;  3 usages

    private LibroDAO libroDAO;  12 usages

    private LibroService() {  1 usage  isederiso351
        logger.info("LibroService inizializzato");
    }

    public synchronized static LibroService getInstance() {  8 usages  isederiso351
        if (instance == null) {
            instance = new LibroService();
        }
        return instance;
    }
```

Singleton CommandHistory

```java
public final class CommandHistory {  12 usages  isederiso351 *

    private static volatile CommandHistory instance;  3 usages

    private static final Logger logger = LoggerFactory.getLogger(CommandHistory.class);

    private final Stack<Command> undoStack = new Stack<>();  11 usages
    private final Stack<Command> redoStack = new Stack<>();  8 usages
    private final int maxHistorySize = 50;  1 usage


    private CommandHistory() { }  1 usage  isederiso351

    public synchronized static CommandHistory getInstance() {  3 usages  isederiso351 *
        if (instance == null) {
            instance = new CommandHistory();
        }
        return instance;
    }
```

G. Spiegare come il progetto soddisfa i requisiti funzionali (FRs) e quelli non funzionali (NFRs)

Questa sezione dimostra come l'implementazione del sistema BookManager soddisfi i requisiti funzionali e non funzionali specificati, attraverso scelte architetturali mirate, pattern di design appropriati e una struttura modulare che garantisce robustezza, usabilità ed estensibilità del prodotto finale.

Requisiti Funzionali (FRs)
S01 - Visualizzazione libri
Implementazione:

- LibreriaController gestisce vista principale con FlowPane per layout a griglia
- createBookCard() genera card visuali per ogni libro mostrando titolo, autore, stato lettura (emoji) e valutazione (stelle)
- displayBooks() popola la vista con collezione completa
- Layout responsive con scroll automatico per grandi collezioni

S02 - Dettagli libro
Implementazione:

- Click su card libro → showBookDetails() apre pannello laterale dettagliato
- UnifiedBookPanelController.showDetailsView() visualizza tutti i metadati inclusi genere e ISBN
- Layout organizzato con TextFlow per presentazione chiara dei dati aggiuntivi

S03 - Gestione CRUD Libri
Implementazione:

- **Aggiunta**: handleAddBook() → form con campi obbligatori (ISBN, titolo, stato) e opzionali (autore, genere)
- **Modifica**: Click "Modifica" → showEditView() con ISBN disabilitato (non modificabile)
- **Rimozione**: Pulsante "Elimina" con dialog di conferma in pannello dettagli

S04 - Ricerca e Filtri
Implementazione:

- searchField con searchTypeComboBox per ricerca su titolo/autore/ISBN
- Ricerca parziale case-insensitive con aggiornamento real-time
- AdvancedFilterComponent per filtri su valutazione, stato lettura, genere

- Pattern Composite per combinazioni multiple di filtri
- SearchCriteria per coordinare ricerca + filtri + ordinamento

S05 - Sistema Undo/Redo
Implementazione:
- Pattern Command con CommandHistory singleton per cronologia operazioni
- Pulsanti undo/redo nella toolbar con stato enabled/disabled appropriato
- Tooltip dinamici che mostrano descrizione operazione da annullare/ripetere
- Stack management con limite 50 operazioni per controllo memoria

S06 - Persistenza multi-formato
Implementazione:
- WelcomeController per scelta formato all'avvio (JSON/SQLite)
- Pattern Strategy con DAOFactory per creazione runtime del DAO appropriato
- JsonLibroDAO e SqliteLibroDAO come strategie concrete
- Chiusura finestra libreria → ritorno welcome screen per cambio libreria
- File picker per selezione/creazione file in entrambi i formati

S07 - Sistema di cache
Implementazione:
- CachedLibroDAO decorator pattern per cache trasparente in memoria
- Cache lazy-loaded al primo accesso con invalidation su errori
- Ottimizzazione operazioni ricerca/filtro/ordinamento senza ricaricamento dati
- Batch operations detection per performance

S08 - Ordinamento dinamico
Implementazione:
- sortComboBox per selezione attributo ordinamento (titolo, autore, valutazione, genere, stato)
- sortOrderCheckBox per direzione crescente/decrescente
- SearchCriteria.Builder.sortBy() per coordinamento con ricerca e filtri
- Aggiornamento real-time della vista all'cambio ordinamento

S09 - Pulizia filtri e ordinamento
Implementazione:
- Pulsante "Pulisci Filtri" nella toolbar
- handleClearFilters() resetta searchField, filtri avanzati, ordinamento ai valori default
- AdvancedFilterComponent.clearAll() per reset filtri complessi
- Un click per tornare alla vista completa non filtrata

S10 - Validazione
Implementazione:

- Validator.validateLibro() e validateIsbn() per controlli pre-salvataggio
- Regex validation per ISBN-10 e ISBN-13: ^(\\d{9}[\\dX]|\\d{13})$
- Controllo unicità ISBN tramite LibroAlreadyExistsException
- Validazione lunghezza campi (titolo 1-255 char, autore 0-255 char)
- Feedback real-time nel form con disabilitazione pulsante "Salva" se errori

Requisiti Non Funzionali (NFRs)
Usabilità
Implementazione:

- **UI responsive**: Design CSS con hover effects, focus states, feedback visuale
- **3-click rule**: Qualsiasi funzione raggiungibile in max 3 click (es: Home → Add Book → Save)
- **Feedback visuale**: Tooltips, messaggi di errore real-time, pulsanti disabled appropriatamente
- **Comprensibilità**: Icone intuitive, layout familiare, terminologia standard

Affidabilità
Implementazione:

- **Validazione robusta**: Controlli multi-livello (applicativo + service + database per SQLite)
- **Exception handling**: Gerarchia eccezioni custom per gestione errori specifici
- **Transazioni**: rollback su errori per consistency
- **Cache invalidation**: Cache invalidation su errori per prevenire stato inconsistente

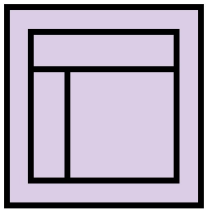
Evolvibilità
Implementazione:

- **Architettura modulare**: Separation of concerns con layer ben definiti
- **Design patterns**: Strategy, Command, Decorator, Composite per flessibilità
- **Interfacce**: LibroDAO, Filter<T>, Command per estendibilità

Verificabilità
Implementazione:

- **Unit testing completo**: 85% line coverage con JUnit e Mockito
- **Parametric testing**: Test parametrici su tutti i DAO types
- **Mock framework**: Isolamento dipendenze per testing robusto

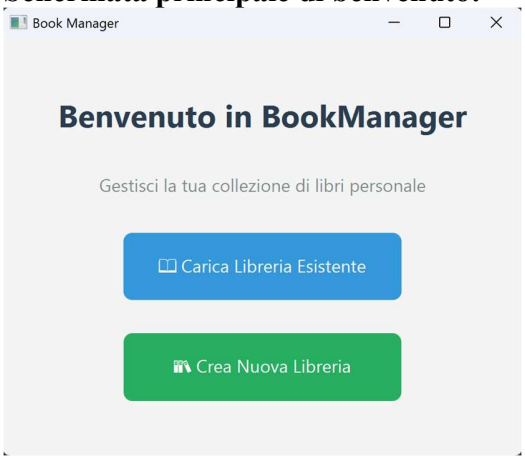
Appendix. Prototype



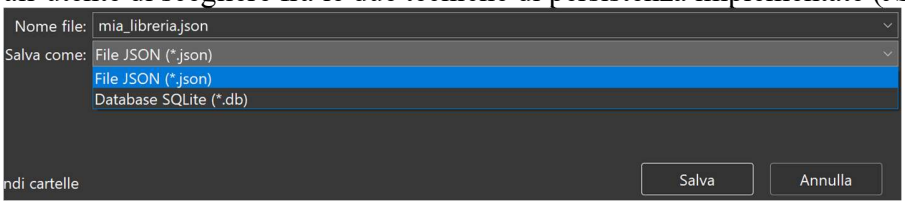
Il prototipo sviluppato rappresenta un'implementazione completa e funzionante del sistema BookManager con tutte le funzionalità core specificate nei requisiti. Il sistema è stato sviluppato in Java 21 con JavaFX per l'interfaccia grafica e include supporto per persistenza multi-formato (JSON/SQLite) con sistema di cache avanzato.

Verrà di seguito esposta brevemente l'applicazione con degli screenshots mostrando come rispetta i vari requisiti funzionali e non.

Schermata principale di benvenuto:

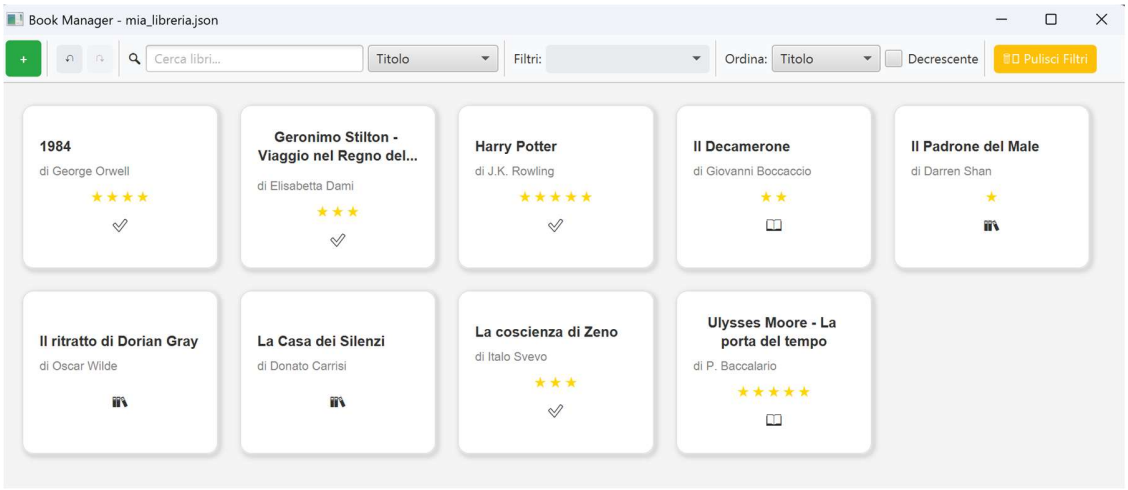


Da questa schermata è possibile scegliere se caricare una libreria esistente o crearne una nuova.
Nel caso di creazione di una nuova libreria, il FileSelector che si aprirà permetterà all'utente di scegliere fra le due tecniche di persistenza implementate (JSON e SQLite):



A questo punto, che si abbia creato o caricato la libreria, verrà aperta la schermata di visualizzazione principale

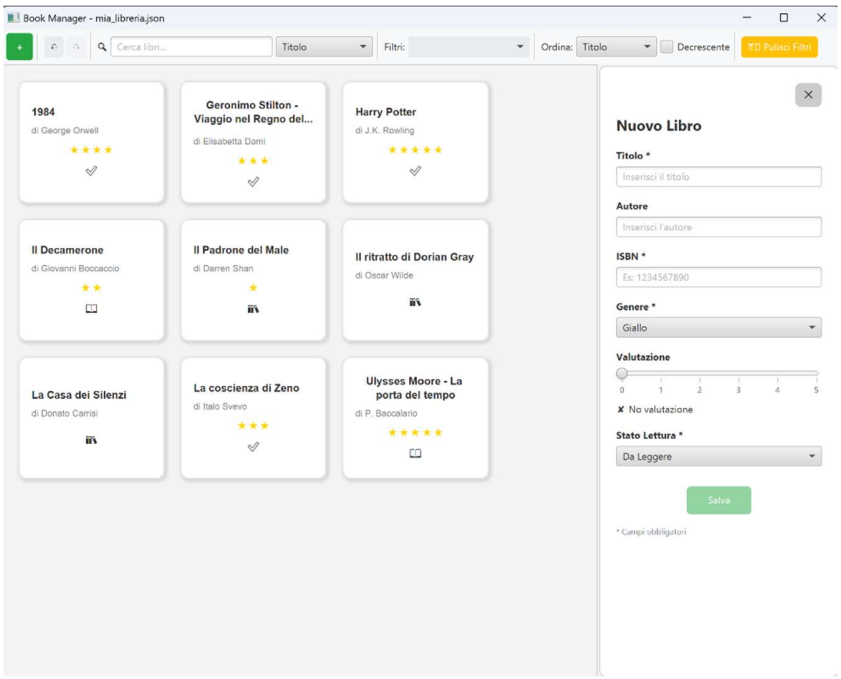
Schermata principale



È qui possibile visualizzare tutti i libri salvati nella libreria, nel caso in cui i libri siano troppi per essere visualizzati in un unica vista, comparirà lo slider per poter scrollare la griglia.

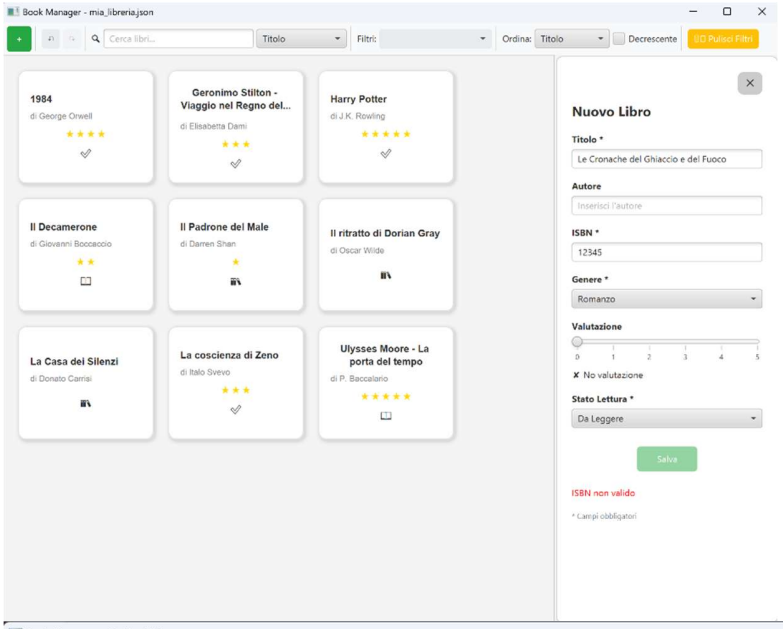
Aggiunta libro

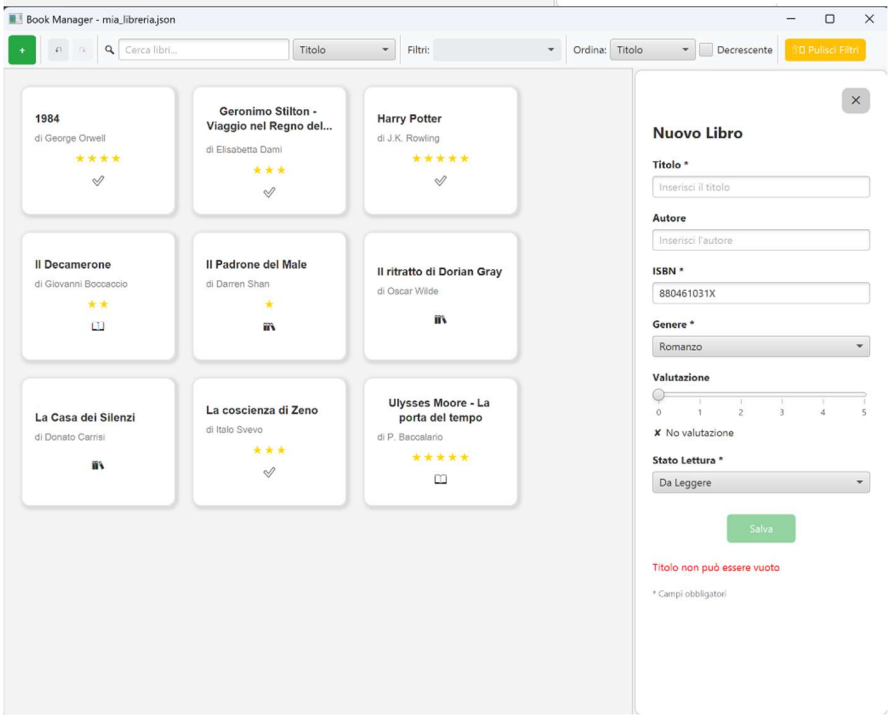
Premendo il tasto verde '+' in alto a sinistra comparirà una finestra laterale nel quale potremo aggiungere gli attributi necessari per creare il nuovo libro:



Attributi invalidi

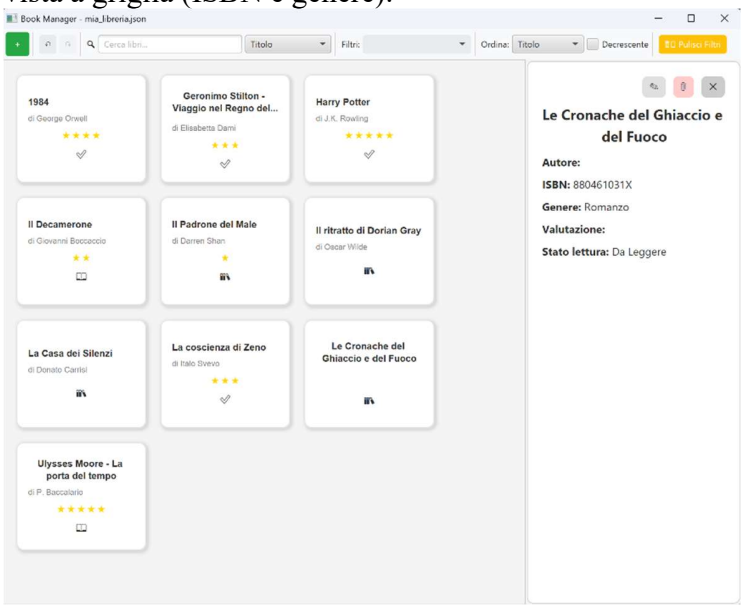
Nel caso in cui si inserisca un ISBN non valido (non conforme agli standard ISBN-10 e ISBN-13), un attributo con numero di caratteri superiore a 255, o nel caso in cui non si inserisca un titolo, verrà visualizzato un messaggio di errore sotto al pulsante Salva, che sarà disabilitato (colore sbiadito)





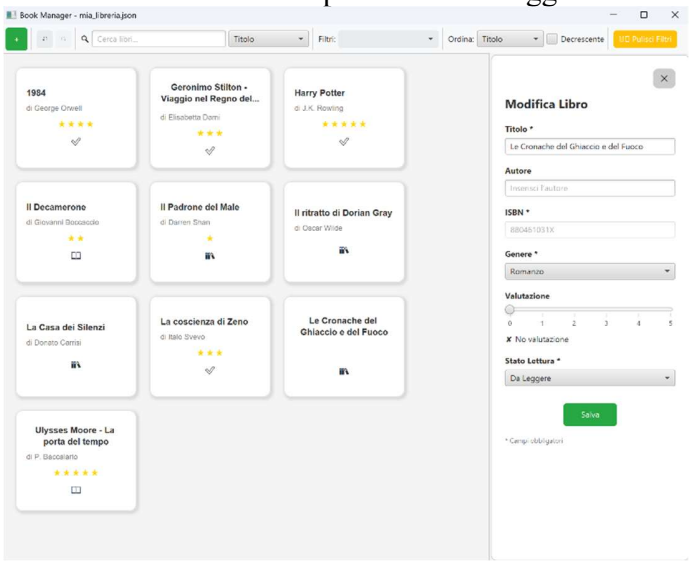
Visualizzazione Dettagli

Ogni qualvolta si clicchi su un libro nella vista o si aggiunga un nuovo libro, si aprirà la scheda dei dettagli del libro, in cui si potranno visualizzare attributi non visibili dalla vista a griglia (ISBN e genere):

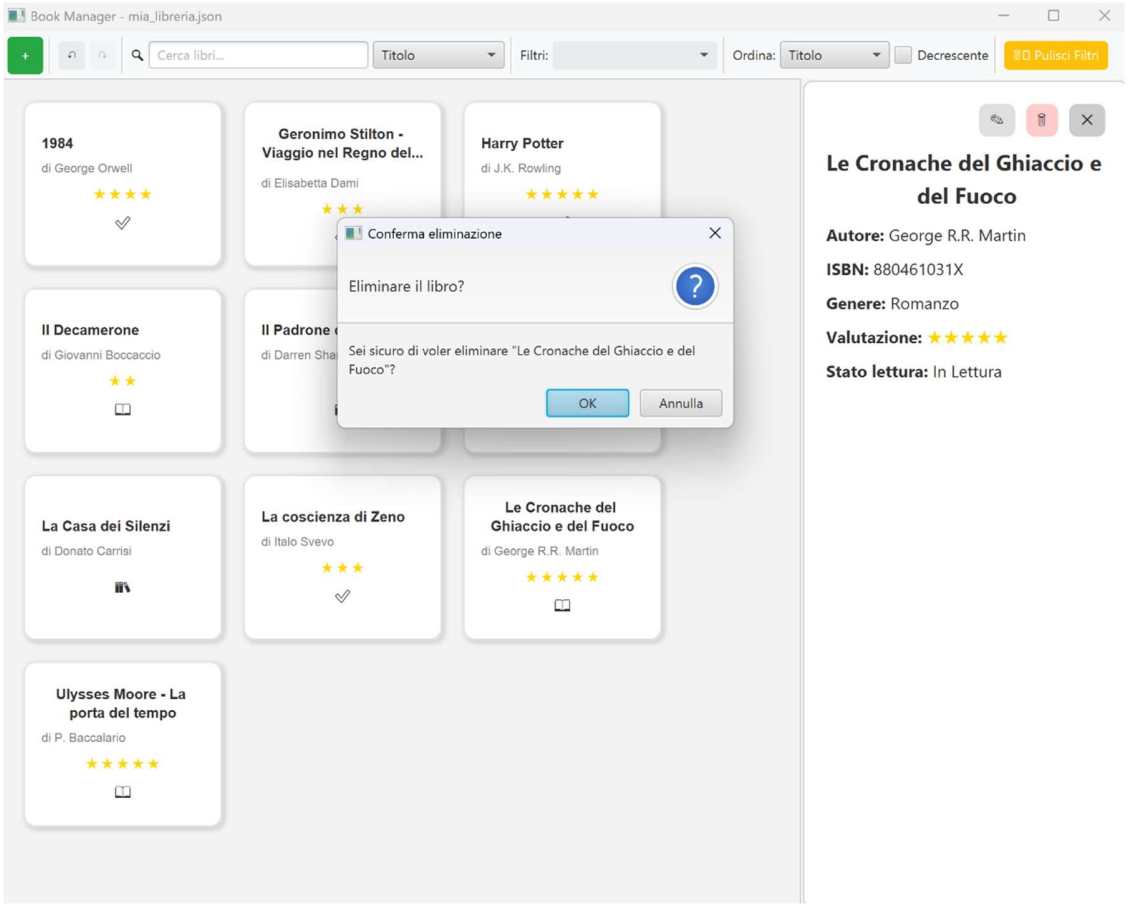


Modifica Libro

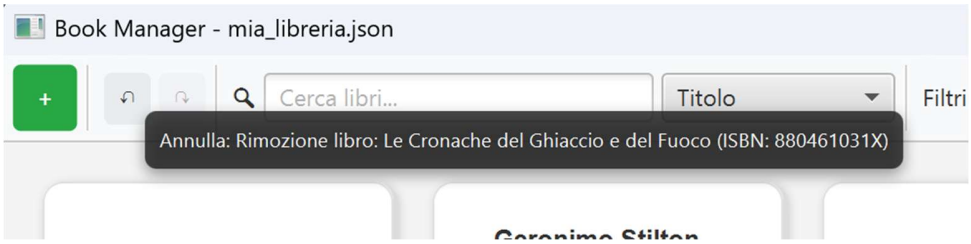
Cliccando sul pulsante a forma di matita in alto nella scheda dei dettagli del libro, si entrerà in modalità modifica, in cui l'utente potrà modificare tutti gli attributi del libro eccetto l'ISBN. Fatto ciò potrà salvare e aggiornare il libro nella visualizzazione.



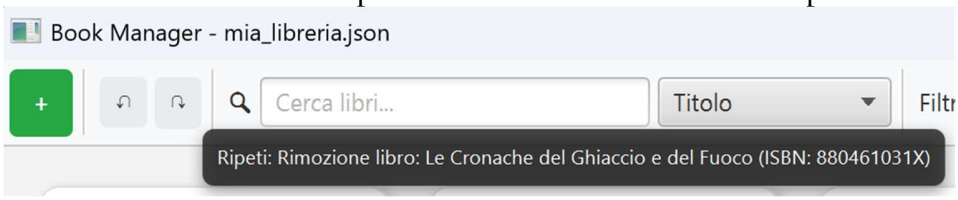
Rimozione Libro
Similmente alla modifica, per rimuovere il libro basterà cliccare il tasto rosso a forma di cestino in alto alla scheda di dettagli libro. Un messaggio di conferma apparirà al centro dello schermo.



Undo e Redo
È possibile porre il cursore del mouse sul tasto undo nella toolbar in alto per far comparire un Tooltip che informa l'utente sull'ultima azione annullabile

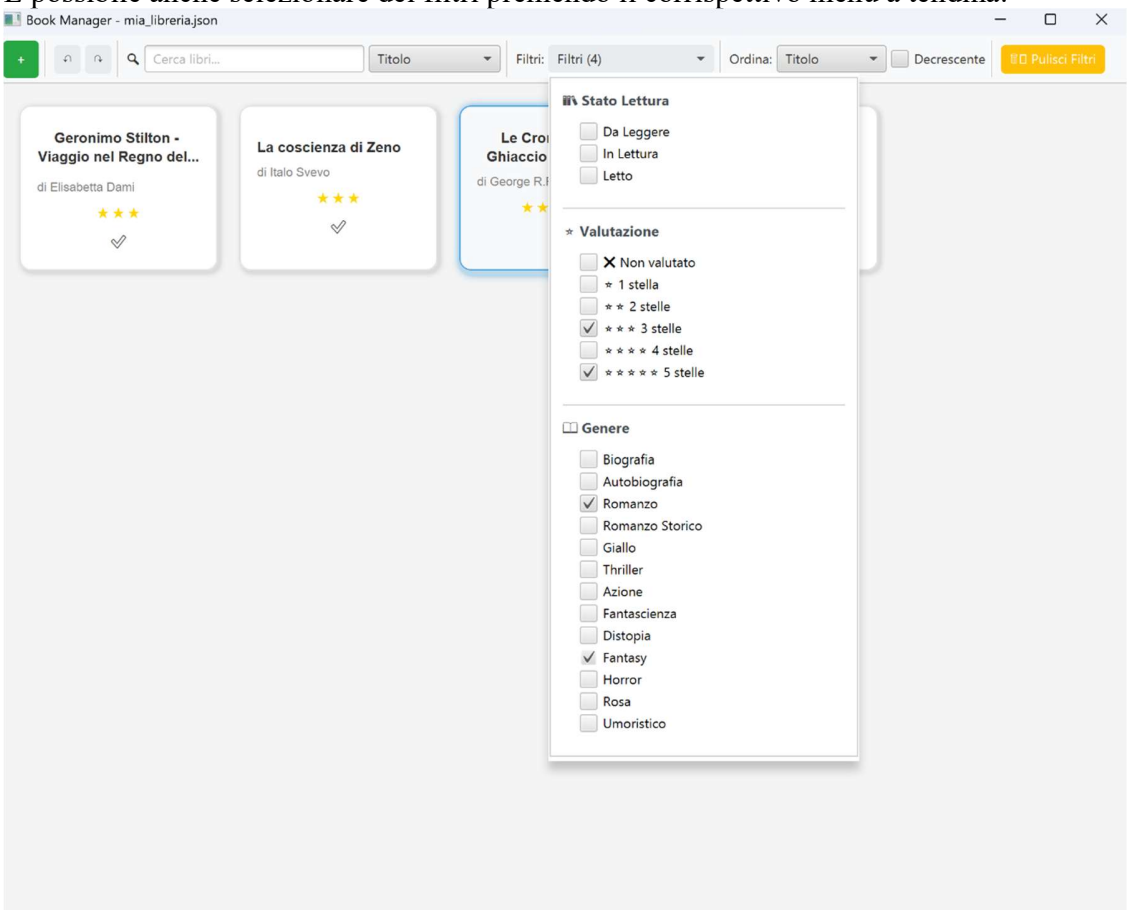


Analogamente, se si ha appena annullato almeno un'azione, il tasto redò diventerà abilitato e mostrerà un tooltip con l'ultima azione annullata da ripetere.



Filtri e ricerca

Scrivendo sulla barra di ricerca, dopo aver selezionato l'attributo di ricerca desiderato, la vista si aggiornerà dinamicamente mostrando i soli libri conformi alla ricerca.
È possibile anche selezionare dei filtri premendo il corrispettivo menù a tendina:



È inoltre possibile selezionare un attributo di ordinamento e, selezionando la checkbox con la freccia in basso, il senso di ordinamento.
Infine, premendo il tasto giallo in alto a sinistra 'Pulisci Filtri', verranno puliti tutti i filtri, la barra di ricerca, e l'ordinamento tornerà all'impostazione di default (per Titolo crescente).