Elaborato di Ingegneria del Software

Docente: Enrico Vicario

Studenti: Luca Lascialfari, Marco Siani, Tommaso Puzzo

Contenuti

1 Introduzione

1.1 Finalità del progetto

L'applicativo ha come obiettivo quello di creare un sistema di prenotazione e restituzione di libri, riviste e media digitali, dinamico, intuibile e disponibile da qualsiasi sistema con il software installato. Lo scopo di questo applicativo è quindi quello di facilitare, ad utenti e lavoratori, la visione e la presa in prestito degli elementi del catalogo presente all'interno della biblioteca.

1.2 Vantaggi

Utilizzando questo sistema ci sarà la possibilità di diminuire gli spostamenti all'interno di strutture che potrebbero risultare piccole, facilitando i lavoratori interni alla biblioteca, i clienti e anche persone con problemi di mobilità, che potrebbero avere bisogno di più spazio e tempo per cercare ciò di cui hanno bisogno. Inoltre tramite il sistema la presa in prestito e la restituzione di un elemento saranno più veloci e sempre sotto controllo. Tutto questo lo otterremo grazie ad una visione dinamica degli elementi disponibili e del loro stato, facilitando e ottimizzando la gestione della biblioteca, dei suoi elementi e dei suoi spazi.

1.3 Tecnologie e standard utilizzati

L'applicativo è sviluppato in Java. Per garantire una permanenza in memoria è stato creato un database locale gestito tramite PostgreSQL. Il progetto resta comunque aperto all'estensione ad un database connesso alla rete. La connessione al suddetto database sfrutta invece le librerie di JDBC.//L'interfaccia grafica è realizzata tramite le librerie open source di JavaFX. Durante la progettazione è stato utilizzato il tool SceneBuilder, fornito da JavaFX per permettere un design pratico e celere dell'interfaccia.//Per i diagrammi di classi viene utilizzato lo standard UML.//Il testing è realizzato sfruttando le librerie di JUnit.

2 Progettazione

2.1 Attori coinvolti e relativi casi d'uso

L'applicativo individua 3 diversi attori:

- Amministratore,
- Utente,
- Sistema.

Ognuno degli attori è caratterizzato da diverse funzionalità, delle quali alcune, come il login, sono comuni, in quanto tutti possono effettuarlo da un'interfaccia identica, che tu sia un amministratore o un cliente della biblioteca. L'applicativo prevede l'autenticazione e l'accesso al sistema tramite l'utilizzo di una e-mail e di una password, inseriti in precedenza tramite un'operazione di signup. L'esistenza di almeno un amministratore è prevista con il rilascio del software. L'amministratore potrà gestire il database e i suoi elementi e, se lo riterrà necessario, potrà anche creare nuovi utenti amministratori, che siano o meno già presenti nel database. Ogni nuovo utente dovrà quindi specificare la e-mail con la quale accedere e la sua password, inserendoli opportunatamente nell'interfaccia di login.
Alcuni **Use Case** coinvolgono più di un attore, come ad esempio **Descrizione elemento**, che mostrerà, sia ai clienti, sia agli amministratori, informazioni sull'elemento selezionato, con opportune differenze (l'utente potrà solo prendere in prestito l'elemento, l'admin potrà rimuoverlo e modificarlo).

2.1.1 Use Case

Gli **Use Case** sono una tecnica utilizzata nell'ingegneria del software per descrivere l'interazione tra un attore e un sistema, con l'obbiettivo di soddisfare un requisito o di raggiungere uno scopo definito.

2.1.2 Use Case scenario

Prima di vedere nel dettaglio i vari **casi d'uso** vediamo come essi interagiscono con gli attori:

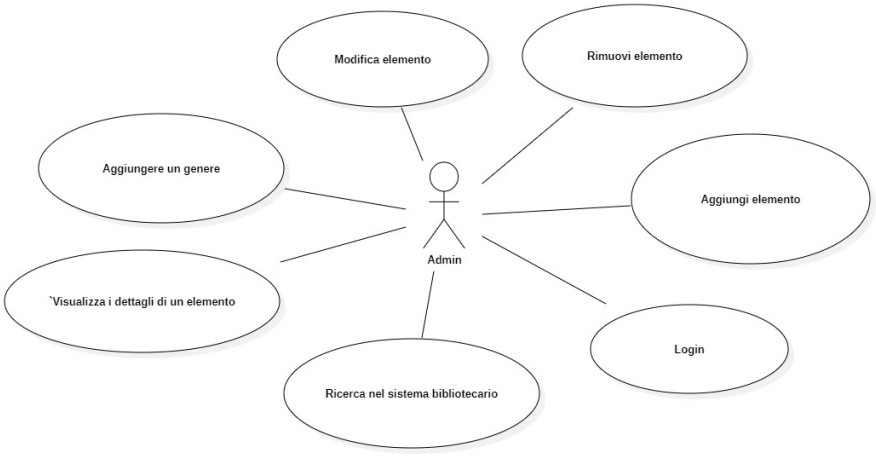


Figure 1: Use Case Diagram per l'Admin

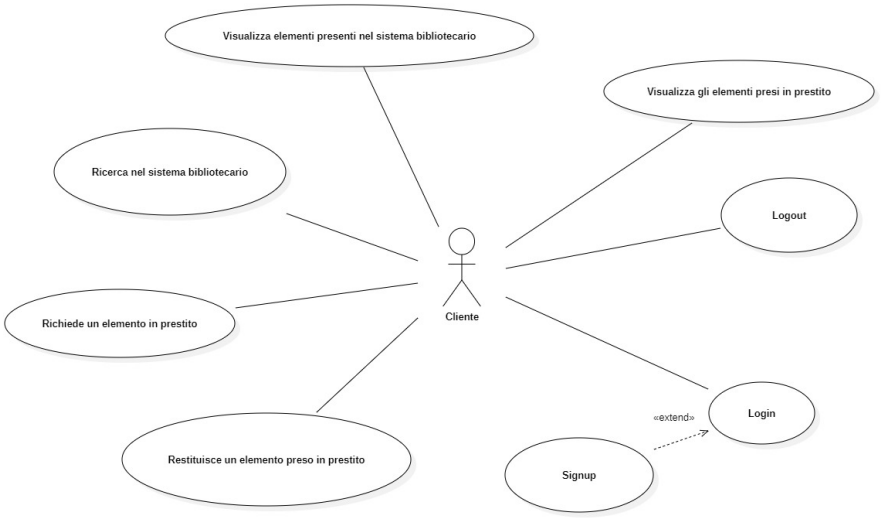


Figure 2: Use Case Diagram per l'utente

2.1.3 Shared Use Case

In questa sezione saranno mostrati tutti gli **Use Case** che comprendono due o più attori e verranno spiegate in breve sotto i casi d'uso corrispondenti.

Use Case	**Login**
Level*	Function
Description*	Utente usa e-mail e password per fare il login
Actors	Cliente, Manager
Pre-conditions*	Disporre di un account Essere connessi al database
Post-conditions*	Utente esegue l'accesso correttamente o viene segnalato errore al login
Normal flow*	1. Utente inserisce identificativo e password 2. Utente preme il tasto Login 3. Login avviene con successo
Variations	1. Utente può inserire username o e-mail come identificativo
Alternative flows	3A. Se identificativo non nel database, compare scritta "Identificativo non presente" 3B. Se identificativo esiste ma password sbagliata, compare scritta "Password errata" 3C. Se non c'è connessione al database, compare scritta "Impossibile connettersi con il database"

Table 1: Tabella Use Case Login

Come specificato prima l'interfaccia di 1 è identica sia per i clienti e sia per gli amministratori.

La tabella 2 sia per i clienti e sia per gli amministratori mostra informazioni sull'elemento selezionato. A differenza del caso **login** in questo caso l'interfaccia avrà delle piccole differenze:

1. Il cliente potrà solamente selezionare il tasto per prendere in prestito il libro, mentre l'amministratore vedrà anche i tasti per eliminare e modificare l'elemento

Use Case	**Dettagli elemento**
Level*	Function
Description*	Utente visualizza informazioni dettagliate, a schermo, di uno specifico elemento
Actors	Cliente, Manager
Pre-conditions*	1. Essere connessi al database 2. Aver effettuato il login
Post-conditions*	Utente visualizza la pagina dell'elemento
Normal flow*	1. Utente clicca su un elemento 2. L'elemento viene visualizzato correttamente
Variations	1. Un cliente può accedere ai dettagli di un elemento dalla pagina prestiti 2. L'elemento può essere preso in prestito dal client 3. L'elemento può essere rimosso o modificato dal manager
Alternative flows	1A. Se non c'è connessione al database, compare un messaggio di errore

Table 2: Dettagli Use Case

Use Case	**Logout**
Level*	Function
Description*	Utente effettua il logout
Actors	Cliente, Manager
Pre-conditions*	Disporre di un account Avere eseguito il login
Post-conditions*	Utente viene disconnesso correttamente
Normal flow*	1. Utente preme il tasto Logout 3. Logout avviene con successo
Alternative flows	1A. Se viene a mancare la connessione con il database, dopo alcuni tentativi di riconnessione il sistema effettua il logout

Table 3: Dettagli Use Case - Logout

Use Case	**signup**
Level*	Function
Description*	Utente crea credenziali per account personale
Actors	Cliente, Manager
Pre-conditions*	1. Disporre di una e-mail 2. Essere connessi al database
Post-conditions*	Utente crea un account correttamente o viene segnalato errore al signup
Normal flow*	1. Utente inserisce e-mail e password 2. Utente preme il tasto Signup 3. Signup avviene con successo
Alternative flows	3A. Se e-mail nel database, compare scritta "E-mail già in uso" 3B. Se non c'è connessione al database, compare scritta "Impossibile connettersi con il database" 3C. Se password contiene meno di 8 caratteri o non ha almeno una lettera minuscola, una maiuscola, una cifra e un carattere speciale (!, #, $, %, &, =, ?, @), compare scritta "Password non valida"

Table 4: Dettagli del caso d'uso Signup

Use Case	**Ricerca di elementi**
Level*	User goal
Description*	Utente utilizza filtri per ottenere una lista di elementi con le caratteristiche desiderate
Actors	Cliente, Manager
Pre-conditions*	1. Disporre di un account 2. Essere connessi al database
Post-conditions*	Vengono mostrati uno o più elementi cercati o viene segnalato errore al login
Normal flow*	1. Utente inserisce parametri di ricerca 2. Vengono visualizzati a schermo gli elementi che corrispondono ai parametri inseriti
Alternative flows	3A. Se l'elemento non è presente nel database allora non verranno mostrati elementi

Table 5: Dettagli del caso d'uso Ricerca nel database

Use Case	**Sfoglia catalogo**
Level*	Function
Description*	Utente visualizza una tabella contenente gli elementi presenti nel catalogo
Actors	Cliente, Manager
Pre-conditions*	1. Disporre di un account 2. Essere connessi al database 3. Aver effettuato il login
Post-conditions*	Utente visualizza una serie di elementi¡br¿Utente può interagire con un singolo elemento per vederlo nel dettaglio
Normal flow*	1. Utente accede e visualizza tutti gli elementi presenti nel catalogo 2. Utente può selezionare un elemento per vederlo nel dettaglio
Alternative flows	3A. Se non sono presenti elementi allora la tabella del catalogo sarà vuota

Table 6: Dettagli del caso d'uso Sfoglia catalogo

2.1.4 Cliente

Il cliente è l'attore che si connetterà al sistema con il solo scopo di visualizzare il catalogo, prenotare uno o più elementi e di restituirli. Questo attore non avrà nessun potere sul database, ma potrà valutare e recensire gli elementi da lui presi in prestito. Qui di seguito saranno presentati solo gli **Use Case** che prevedono il **Cliente** come unico attore:

Use Case	**Richesta prestito**
Level*	User goal
Description*	Utente prende un elemento in prestito
Actors	Cliente
Pre-conditions*	Disporre di un account Essere connessi al database Aver effettuato il login
Post-conditions*	Utente riceve in prestito l'elemento o viene segnalato errore
Normal flow*	1. Utente visualizza un elemento 2. Utente preme il tasto per il prestito 3. Utente riceve l'elemento in prestito
Alternative flows	3A. L'utente potrebbe non avere l'elemento in prestito perché ha già preso il massimo di elementi possibili

Table 7: Tabella Use Case Richesta prestito

Use Case	**Restituzione prestito**
Level*	User goal
Description*	Cliente restituisce uno o più elementi che ha preso in prestito
Actors	Cliente
Pre-conditions*	Disporre di un account Essere connessi al database Aver preso in prestito almeno un elemento
Post-conditions*	Elemento/i viene/vengono reso/i nuovamente disponibile/i per il prestito o viene segnalato errore
Normal flow*	1. Utente sceglie dalla lista degli elementi presi in prestito quello/i da restituire 2. Utente preme il tasto "Restituisci" 3. Per ognuno degli elementi da restituire, il sistema chiede al Cliente di lasciare una breve recensione composta da una valutazione da 1 a 5 stelle e un breve commento opzionale. 4. L'elemento/Gli elementi viene/vengono restituito/i correttamente e le eventuali recensioni vengono aggiunte al database
Variations	1. Utente deve scegliere almeno un elemento da restituire o annullare l'operazione
Alternative flows	3A. Se l'utente clicca il tasto "No, grazie", la recensione dell'elemento attuale viene ignorata 4A. Se non c'è connessione al database, compare scritta "Impossibile connettersi con il database"

Table 8: Tabella Use Case Restituzione prestito

Use Case	**Elementi in prestito**
Description*	Visione degli elementi attualmente presi in prestito
Actors	Cliente
Pre-conditions*	1. Connessione al database 2. Avere un account 3. Aver effettuato il login
Normal flow*	1. Si accede al proprio account 2. Una volta aperta l'interfaccia sarà possibile vedere che elementi abbiamo in prestito 3. Scegliere se restituire o visualizzare l'elemento
Alternative flows	2A. L'utente potrebbe non aver preso in prestito nessun elemento e la pagina risulterà vuota

Table 9: Dettagli del caso d'uso Elementi in prestito

2.1.5 Amministratore

L'amministratore è quell'attore che si occupa di gestire il sistema e il suo database. Tra i suoi compiti troviamo quelli di aggiungere, rimuovere e modificare i vari elementi e anche quello di aggiungere o rimuovere account amministratori.
Qui saranno presentati tutti gli **Use Case** che prevedono l'amministratore come unico attore in gioco:

Use Case	**Aggiungere elemento al database**
Level*	User goal
Description*	Manager aggiunge un elemento nel database
Actors	Manager
Pre-conditions*	Avere effettuato il login Essere connessi al database
Post-conditions*	Elemento viene aggiunto nel database correttamente o viene segnalato un errore
Normal flow*	0. Manager clicca su bottone "Aggiungi elemento" 1. Manager sceglie tipologia di elemento da aggiungere 2. Manager compila il form relativo all'elemento selezionato con le relative informazioni 3. Manager clicca su "Save" 4. L'elemento viene aggiunto al database correttamente
Variations	1. Tipologia dell'elemento può essere libro, rivista o DVD/Blu-Ray
Alternative flows	4A. Se le informazioni aggiunte al punto 2 non sono esaustive (e.g. non sono stati compilati tutti i campi obbligatori del form), allora compare la scritta "Informazioni insufficienti" 4B. Se non c'è connessione al database, compare scritta "Impossibile connettersi con il database" 4C. L'operazione può essere annullata premendo il bottone "cancel"

Table 10: Tabella Use Case Aggiungere elemento al database

Use Case	**Rimuovere elemento**
Level*	User goal
Description*	Manager elimina un elemento dal database
Actors	Manager
Pre-conditions*	Avere effettuato il login Essere connessi al database Il database deve contenere almeno un elemento
Post-conditions*	Elemento viene eliminato dal database o viene segnalato un errore
Normal flow*	1. Manager sceglie elemento da eliminare 2. Manager preme il tasto "Elimina" 3. Eliminazione avviene con successo
Variations	
Alternative flows	3A. Se l'elemento è al momento in prestito, compare la scritta "Elemento non in deposito" 3B. Se non c'è connessione al database, compare scritta "Impossibile connettersi con il database"

Table 11: Tabella Use Case Rimuovere elemento

Use Case	**Modifica un elemento nel database**
Level*	User goal
Description*	Manager Modifica un elemento selezionato presente nel database
Actors	Manager
Pre-conditions*	Avere effettuato il login Essere connessi al database
Post-conditions*	Manager vede i dettagli di un elemento e può scegliere se e quali modificare
Normal flow*	1. Manager clicca su "Modifica" 2. Pagina della modifica viene caricata 3. Il manager effettua le modifiche necessarie 4. Viene premuto il tasto "conferma" 5. Le modifiche vengono aggiunte al database
Variations	
Alternative flows	3A. Manager può cliccare il tasto "cancel" e annullare l'operazione

Table 12: Tabella Use Case Modifica un elemento nel database

2.2 Diagramma delle classi

Il diagramma delle classi è il seguente:

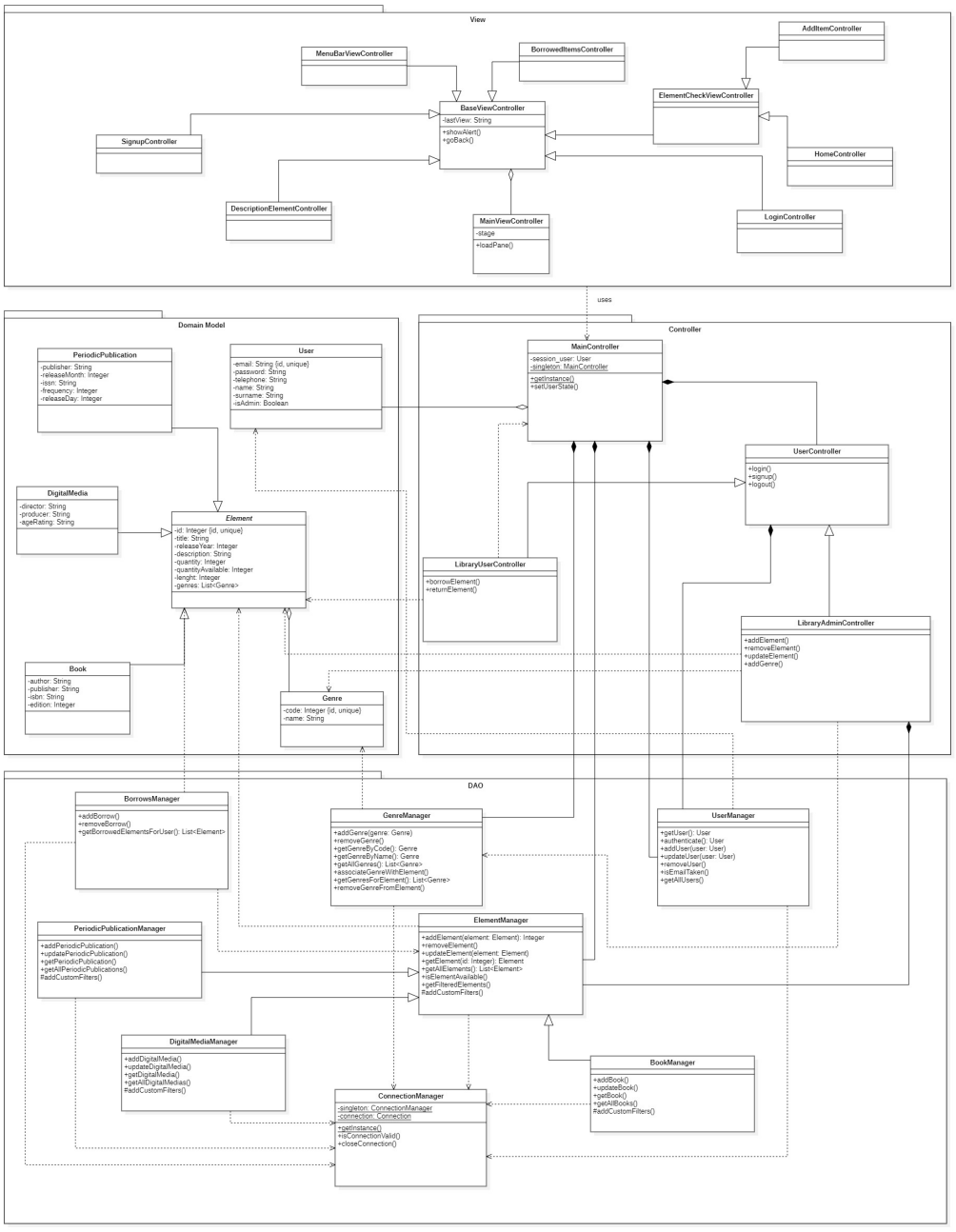


Figure 3: Diagramma delle classi

2.3 Design Patterns

Un design pattern è una soluzione riutilizzabile a un problema comune che si verifica durante lo sviluppo del software.

2.3.1 Model View Controller

: Il Model View Controller è un pattern in grado di separare la logica di presentazione dei dati dalla logica di business.
Nel nostro caso lo possiamo vedere come l'insieme dei package **DomainModel** e **DAO**, con l'aggiunta del database stesso.

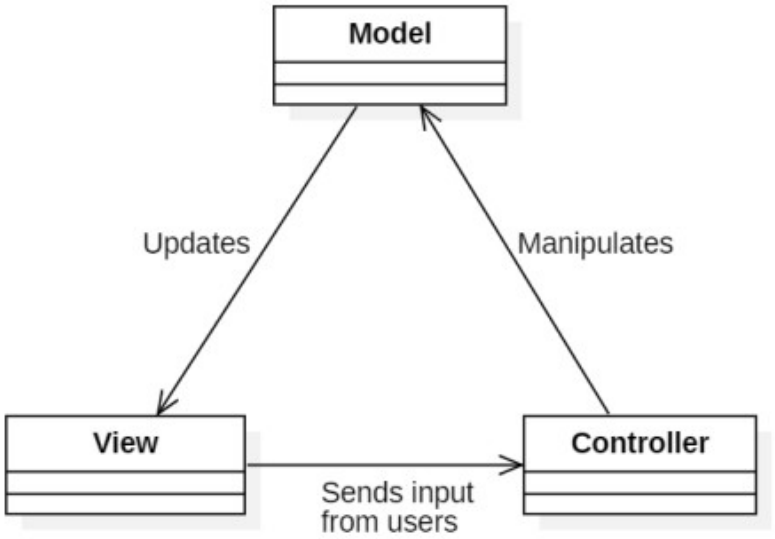


Figure 4: Diagramma UML di un generico pattern MVC

Nel nostro caso la parte del codice del controller è quella inclusa nell'ononimo package, composto da 4 classi.
La view è costituita unicamente dalla classe MainViewController che gestisce tutte le interazioni dell'utente con l'interfaccia. La scelta di usare una singola classe è stata dettata dalle esigenze delle librerie di JavaFX che prevedono l'esistenza di un unico controller dell'interfaccia.

2.3.2 Singleton

Nella gestione della connessione al database e nel mantenimento dello stato del programma è stato implementato il pattern Singleton. Questo garantisce l'esistenza di un unico **ConnectionManager** responsabile di gestire la connessione al database, e di un solo **MainController**, responsabile invece del mantenimento dello stato del sistema.

2.4 Page Navigation Diagram

L'interazione prevista che l'utente deve avere con l'interfaccia è la seguente:

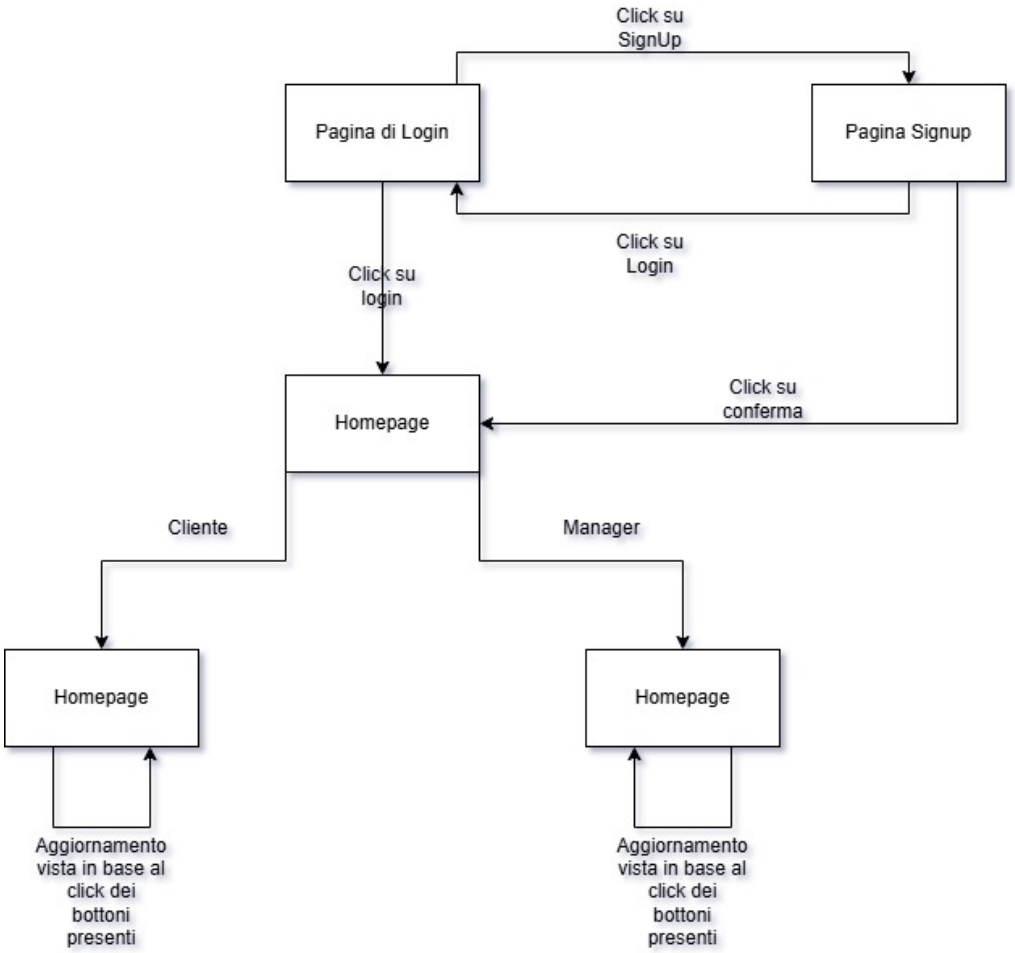


Figure 5: Diagramma di Navigazione

2.5 Entity Reletionship Diagram

Qui di seguito è presentato il diagramma ER:

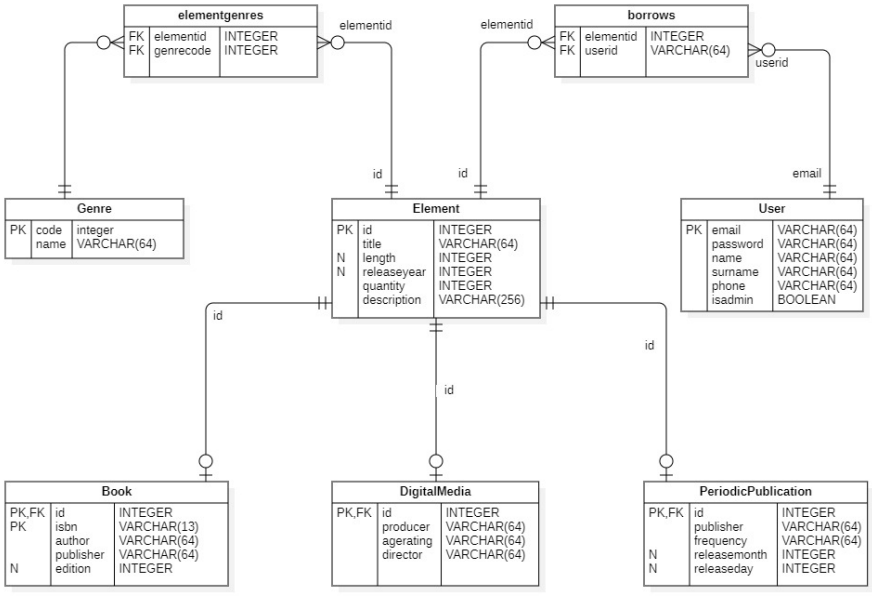


Figure 6: Diagramma ER

3 Implementazione delle classi

Il codice sorgente è articolato nel modo seguente:

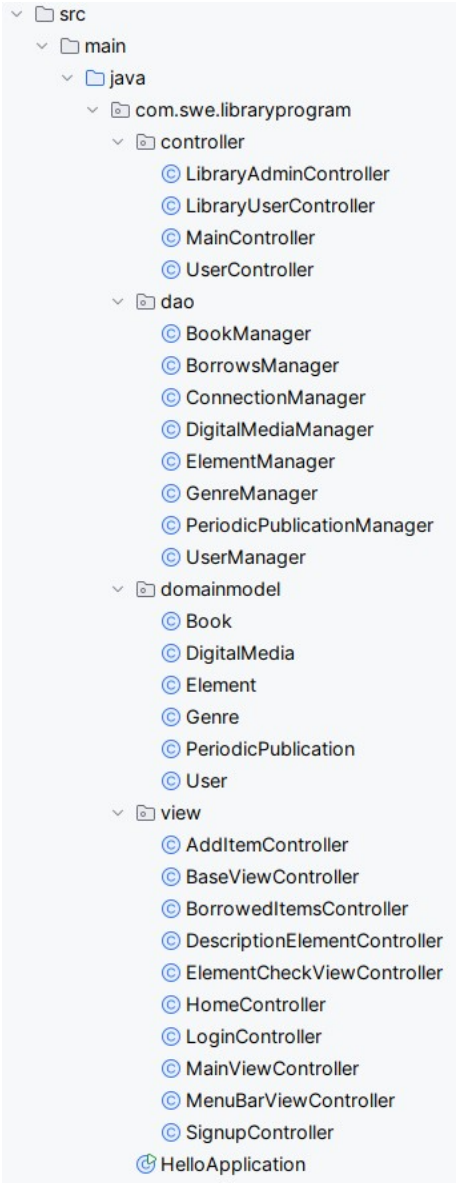


Figure 7: Stuttura del codice sorgente

3.1 DomainModel

Il DomainModel è il package che si occupa di definire un modello di composizione di classi, livello **Model** su **JavaFX**, su cui è possibile eseguire i casi d'uso espressi nello **Use Case Diagram**. Questo package si articola nelle seguenti classi, che descrivono le **entità del dominio** dell'applicazione:

3.1.1 Element

Questa classe si occupa di fornire attributi e metodi comuni tra i vari elementi presenti all'interno del sistema. Gli elementi condivisi tra le varie classi sono:

1. id
2. title
3. releaseYear
4. description
5. quantity
6. quantityAvailable
7. lenght
8. genres

3.1.2 Book

Questa classe mantiene i dati riguardanti i **libri** presenti all'interno del catalogo della biblioteca.
Tra questi dati, oltre a quelli ereditati dalla classe **Element** abbiamo:

- isbn
- author
- publisher
- edition

Ogni libro è identificato anche dal suo **isbn**.

3.1.3 DigitalMedia

Questa classe si occupa di mentenere le informazioni relative ai **media digitali** presenti all'interno della biblioteca.
Tra questi dati, oltre a quelli ereditati dalla classe **Element** abbiamo:

- producer
- ageRating
- director

3.1.4 Genre

Questa classe viene utilizzata per mantenere informazioni sui vari generi che caratterizzano un elemento.
Al suo interno troviamo gli attributi **name** e **code**

3.1.5 Periodic Publication

Questa classe è stata progettata per mantenere le informazioni relative alle pubblicazioni di riviste o altri elementi periodici. Oltre alle informazioni ereditate dalla classe **Element**, al suo interno troviamo:

- publisher
- frequency
- releaseMonth
- releaseDay

3.1.6 User

Questa classe si occupa di mantenere le informazioni relative agli utenti all'interno del sistema. Tra queste informazioni è presente un attributo **Boolean** utilizzato per identificare se un utente è **Amministratore**, o se non lo è.
Al suo interno troviamo:

- email
- password
- name
- surname
- phone
- isAdmin

L'**email** è utilizzata per effettuare le operazioni di login e signup, oltre ad essere l'**identificativo** di un user.

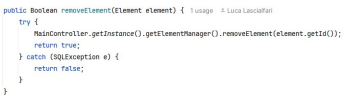
3.2 Controller

In questo **package** si trovano tutte le classi che si occupano di fornire i servizi richiesti da un utente tramite la **GUI**.

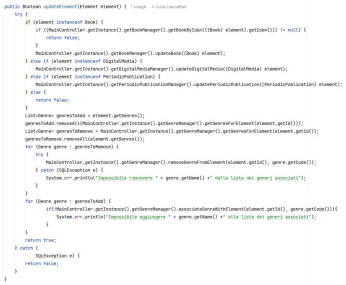
3.2.1 LibraryAdminController

Questo controller estende la classe **UserController** e si occupa di fornire a un Amministratore tutti i metodi e gli attributi necessari a svolgere i suoi compiti, quindi rimuovere, aggiungere e modificare un elemento nel database.

(a) Aggiungi elemento



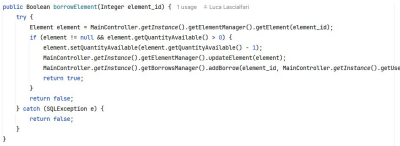
(b) Rimuovi elemento



(c) Modifica elemento

3.2.2 LibraryUserController

Questo controller estende la classe **UserController** e si occupa di fornire agli utenti non amministratori i metodi per prendere in prestito e per restituire un elemento.
Al suo interno troviamo **borrowElement**, per prendere in prestito un elemento e **returnItem** per restituirlo.



(a) borrowElement

(b) returnElement

3.2.3 MainController

Questo controller è un **Singleton**, di conseguenza esiste una sola sua istanza in tutta l'applicazione, della quale forma la base per il suo funzionamento. Infatti si occupa di gestire lo stato dell'utente connesso e gestisce l'accesso ai vari manager del sistema.
Esso si occupa quindi di:

- **Gestire il login dell'utente** con `setUserState()`, differenziando tra amministratore e utente normale.
- **Tenere traccia degli elementi presi in prestito** (`borrowedElements`).
- **Esporre metodi per accedere ai vari manager** per la gestione di libri, utenti, prestiti, ecc.
- **Permettere il logout** con `resetUserState()`, ripulendo i dati utente e i prestiti.

3.2.4 UserController

Questa classe controller si occupa di gestire le operazioni di **Login**, **Logout** e **Signup**.
Il controller si occupa in fase di login di richiamare un metodo per verificare la correttezza delle credenziali inserite, in fase di signup di controllare se tutti i dati sono stati inseriti correttamente e se rispettano i criteri prefissati e la fase di logout che disconnette l'utente dall'applicazione.

3.3 DAO

Questo package si occupa di gestire le comunicazioni con il database sia in lettura che in scrittura.

3.3.1 ConnectionManager

La classe si occupa della gestione della connessione al database PostgreSQL per l'applicazione e garantisce che essa sia **unica** attraverso il design pattern **singleton**.
La classe contiene: La classe fornisce quindi un'interfaccia sicura e centralizzata

Metodo	Descrizione
`getInstance()`	Restituisce l'unica istanza della classe, in quanto Singleton.
`getConnection()`	Restituisce una connessione attiva al database, creandola se necessario.
`setDbUser(String dbUser)`	Imposta dinamicamente il nome utente del database.
`setDbPass(String dbPass)`	Imposta dinamicamente la password del database.
`isConnectionValid()`	Verifica se la connessione al database è ancora valida.
`closeConnection()`	Chiude la connessione al database se è attiva.

Table 13: Metodi principali della classe `ConnectionManager`

per la gestione della connessione al database, seguendo il pattern Singleton e implementando metodi per configurare, verificare e chiudere la connessione in modo efficiente.

3.3.2 ElementManager

Questa classe si occupa di fornire tutti i metodi necessari per la gestione di un elemento all'interno del sistema. Da essa ereditano le classi:

- BookManager
- DigitalMediaManager
- PeriodicPublicationManager

3.3.3 BookManager

La classe BookManager gestisce le operazioni relative ai libri nel database, estendendo **ElementManager**. Si occupa di aggiungere, aggiornare, recuperare ed effettuare ricerche sui libri.

3.4 BorrowsManager

La classe BorrowsManager gestisce i prestiti di elementi nella biblioteca, interagendo con la tabella borrows del database dalla quale possiamo ricavare informazioni che collegano i vari utenti agli elementi che questi hanno preso in prestito.

3.4.1 DigitalMediaManager

Questa classe si occupa della gestione di oggetti di tipo **DigitalMedia**. Essa estende la classe ElementManager e interagisce con il database tramite query SQL per eseguire operazioni di inserimento, aggiornamento, recupero e filtraggio di media digitali.

3.4.2 GenreManager

Questa classe fornisce metodi per aggiungere, rimuovere e recuperare i generi e per associare generi agli elementi nel sistema. Utilizza il ConnectionManager per ottenere le connessioni al database e le query SQL per manipolare i dati.
Al suo interno abbiamo:

Metodo	**Descrizione**
`addGenre(Genre genre)`	Aggiunge un nuovo genere al database. Il metodo inserisce il nome del genere nella tabella `genres` usando una query SQL di tipo `INSERT`. Restituisce `true` se l'inserimento ha successo.
`removeGenre(Integer code)`	Rimuove un genere dal database utilizzando il codice del genere. La query `DELETE` viene eseguita sulla tabella `genres`. Restituisce `true` se la rimozione ha successo.
`getGenreByCode(Integer code)`	Recupera un genere dal database in base al codice specificato. La query `SELECT` restituisce il genere che corrisponde al codice. Se non viene trovato nessun genere, restituisce `null`.
`getGenreByName(String name)`	Recupera un genere dal database in base al nome specificato. La query `SELECT` cerca il genere corrispondente nel database e restituisce un oggetto `Genre` se trovato, altrimenti `null`.
`getAllGenres()`	Recupera tutti i generi dal database. La query `SELECT` eseguita senza filtri restituirà tutti i generi presenti nella tabella `genres`.
`associateGenreWithElement(Integer elementId, Integer genreCode)`	Associa un genere a un elemento. La query `INSERT` inserisce una riga nella tabella di associazione `elementgenres` utilizzando l'ID dell'elemento e il codice del genere. Restituisce `true` se l'associazione ha successo.
`getGenresForElement(Integer elementId)`	Recupera tutti i generi associati a un elemento. La query `SELECT` unisce le tabelle `genres` ed `elementgenres` per ottenere i generi per l'elemento specificato.
`removeGenreFromElement(Integer elementId, Integer genreCode)`	Rimuove l'associazione tra un elemento e un genere. La query `DELETE` elimina la riga corrispondente nella tabella `elementgenres`, che collega l'elemento al genere. Restituisce `true` se l'operazione ha successo.

Table 14: Metodi della classe `GenreManager` e descrizione

3.4.3 PeriodicPublicationManager

La classe PeriodicPublicationManager è una sottoclasse di ElementManager ed è responsabile della gestione delle pubblicazioni periodiche.

3.4.4 UserManager

Questa classe si occupa della gestione degli utenti all'interno del database e offre la possibilità di: creare, aggiungere, rimuovere e aggiornare gli utenti.
Inoltre, questa classe si occupa anche di controllare l'unicità degli attributi utili all'identificazione dell'utente, insieme a un controllo che permetta di vedere se l'attributo inserito esiste o meno all'interno del sistema.

3.5 Altre classi

3.5.1 HelloApplication

Questa è la classe main del sistema, che si occupa di creare l'interfaccia grafica, con la quale gli utenti si connettono all'applicazione e si occupa anche del ciclo di vita del software.
Questa classe di **javaFX** estende **Application**.
Quindi il suo scopo è quello di gestire la configurazione della finestra, il caricamento della vista e l'interazione con il database, specificando anche le dimensioni della **GUI**.
La funzione main si occupa di avviare il programma.

3.5.2 View

Nel progetto sono presenti varie viste, realizzate utilizzando JavaFX che, grazie all'applicazione SceneBuilder, consente di realizzare interfacce in modo rapido e semplice. Di seguito verranno mostrate e brevemente spiegate alcune viste:

Figure 10: Viste dell'applicativo

3.5.3 SignUp-view

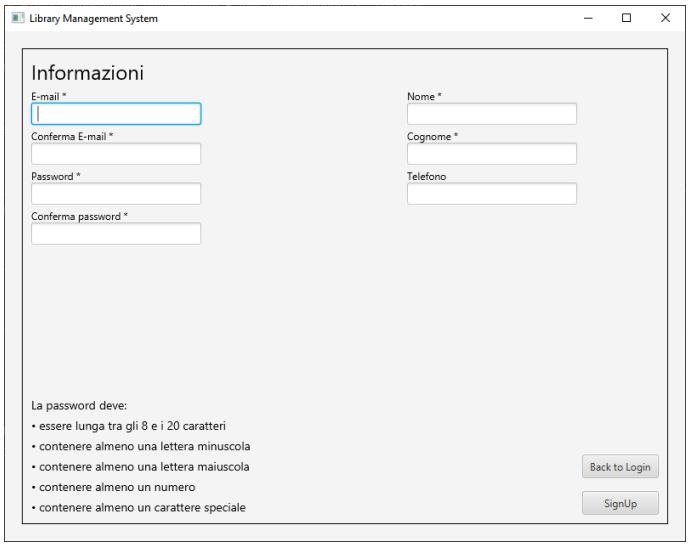


Figure 11: Questa immagine mostra comè stata sviluppata su **Scene Builder** l'interfaccia grafica la creazione di un account utile per accedere al sistema di prenotazione

3.5.4 AddItem-view

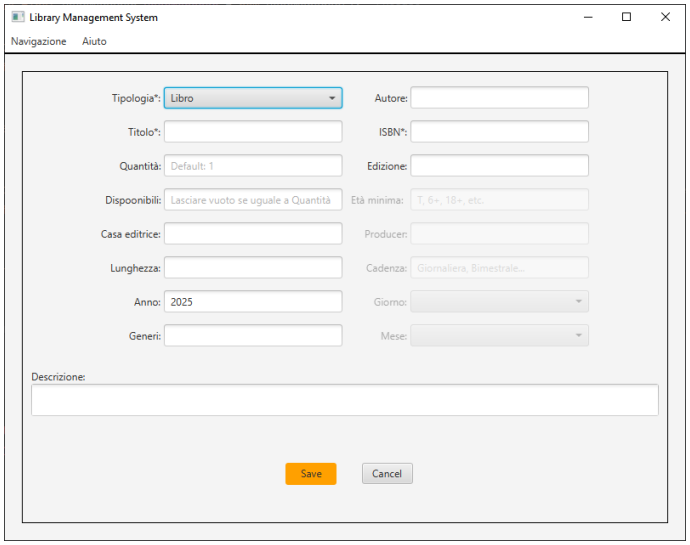


Figure 12: Questa immagine mostra comè stata sviluppata su **Scene Builder** l'interfaccia grafica per aggiungere un elemento al catalogo

3.5.5 ElementDetails-view

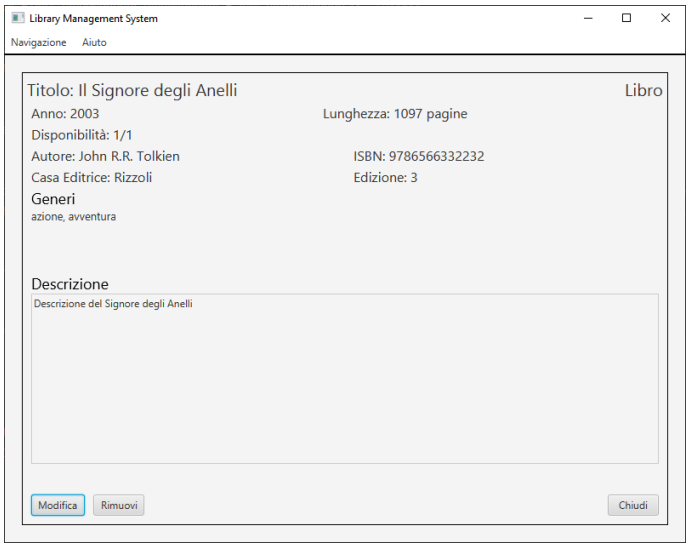


Figure 13: Questa immagine mostra comè stata sviluppata su **Scene Builder** l'interfaccia grafica per descrivere un elemento presente catalogo

3.5.6 Login-view

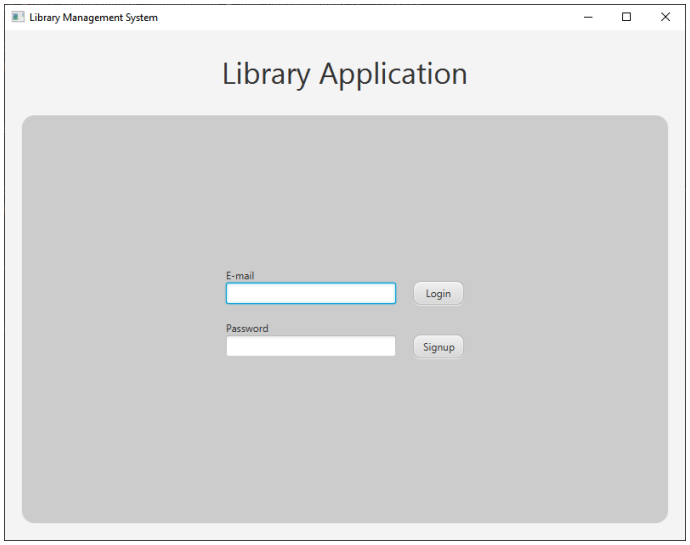


Figure 14: Questa immagine mostra comè stata sviluppata su **Scene Builder** l'interfaccia grafica fare il login al proprio account, per accedere al sistema

3.5.7 Home-view

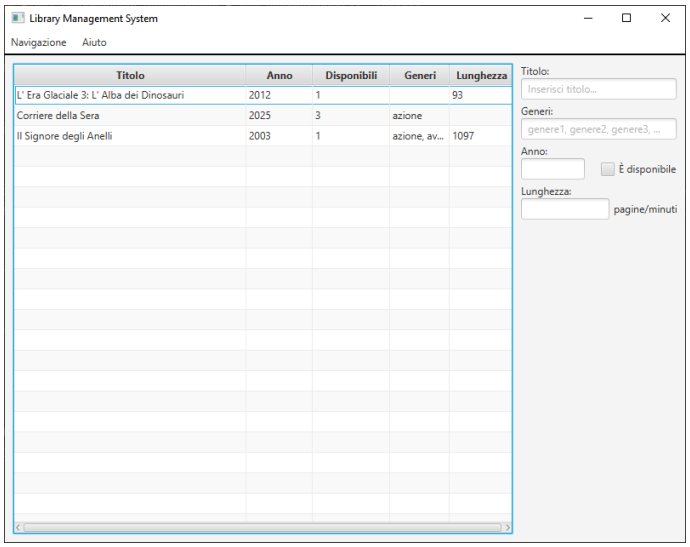


Figure 15: Questa immagine mostra comè stata sviluppata su **Scene Builder** l'interfaccia grafica dell'**Home Page** del sistema, con la possibilità di vedere l'elenco di elementi al suo interno ed eventualmente di ricercarli

3.5.8 Borrowed-view

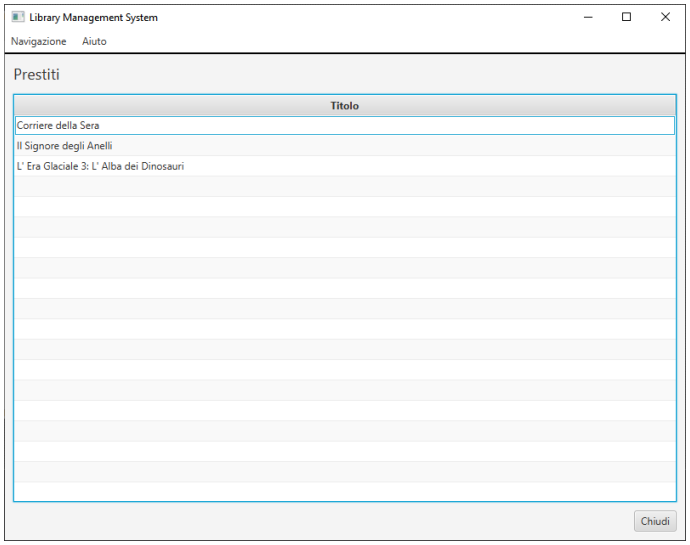


Figure 16: Questa immagine mostra comè stata sviluppata su **Scene Builder** l'interfaccia grafica della pagina relativa agli elementi presi in prestito da un utente

.

3.6 Struttura del database

Per la gestione dei dati sia degli utenti, degli elementi, delle prese in prestito e delle restituzioni si utilizza un database relazionale gestito attraverso **PostgreSQL**. In figura si mostra la struttura del database.

```
users(PK(email), password, name, surname, phone, isadmin)

books(PK(id,isbn), author, publisher, edition)
    FK(id) ref elements(id)

digitalmedias(PK(id), producer, agerating, director)
    FK(id) ref elements(id)

periodicpublication(PK(id), publisher, frequency, releasemonth, releaseday)
    FK(id) ref elements(id)

borrows(PK(elementid,userid))
    FK(elementid) ref elements(id)
    FK(userid) ref users(email)

elementgenres(PK(elementid,genrecode))
    FK(elementid) ref elements(id)
    FK(genrecode) ref genres(code)

genres(PK(code), name)

elements(PK(id), title, releaseyear, description, quantity, quantityavailable, length)
```

Figure 17: Struttura del database

4 Testing

In questa sezione verranno elencati i test che sono stati effettuati. Per fare i test sono stati utilizzati **Maven**, **JUnit5** e **Mockito-JUnit**.
I test sono di tipo strutturale e sono atti a verificare la corretta interazione con il database e sul suo utilizzo.
La struttura dei test è la seguente:

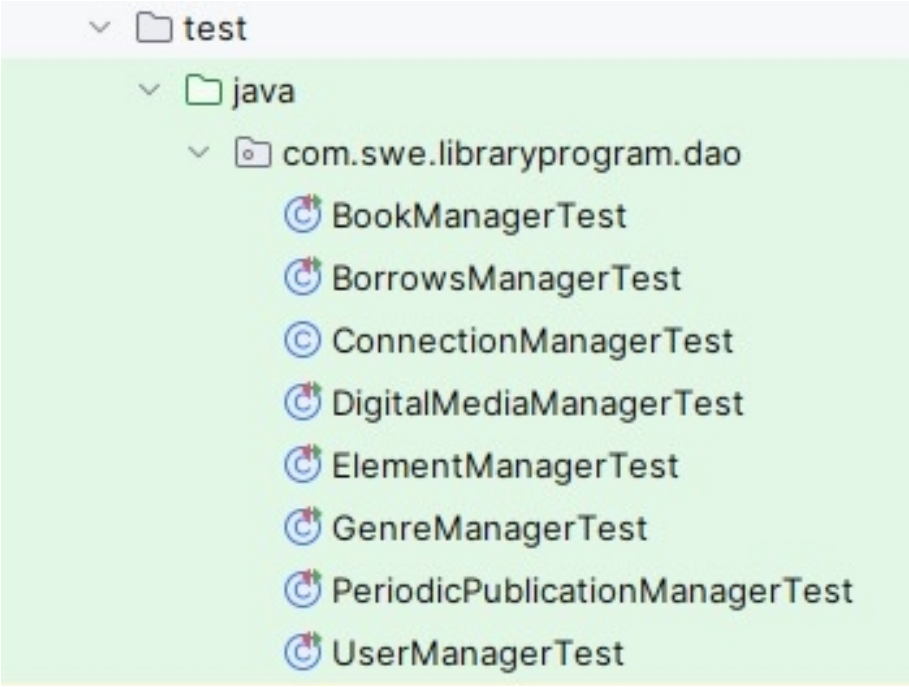


Figure 18: Struttura dei test

4.1 BookManagerTest

La classe **BookManagerTest** è una suite di test JUnit 5 che verifica il corretto funzionamento della classe BookManager, che si occupa della gestione dei libri nel database. I test utilizzano Mocking con Mockito per simulare il comportamento delle dipendenze, evitando di interagire direttamente con il database in alcuni casi.

Nome Test	**Scopo**	**Aspettativa**	**Possibili Fallimenti**
addBookTest	Verifica l'inserimento di un libro nel database	Deve restituire un ID valido dopo l'inserimento	Il libro non viene aggiunto e l'ID è `null`
updateBookTest	Verifica l'aggiornamento di un libro esistente	La funzione deve restituire `true` se l'aggiornamento ha successo	Il libro non viene aggiornato correttamente
getBookByIsbnTest	Verifica il recupero di un libro tramite ISBN	Il libro trovato deve avere lo stesso ISBN richiesto	Nessun libro trovato oppure ISBN errato

Table 15: Riepilogo dei test eseguiti su `BookManager`.

4.1.1 Conclusione

Questa classe testa correttamente le operazioni CRUD sulla gestione dei libri, verificando che l'aggiunta, l'aggiornamento e il recupero funzionino come previsto. I test utilizzano mocking per isolare il database e altre classi, garantendo che i risultati siano affidabili senza dipendere da una connessione attiva al database.

4.2 BorrowsManagerTest

Questa classe di test **BorrowsManagerTest** utilizza JUnit 5 e Mockito per verificare il corretto funzionamento della classe BorrowsManager, che presumibilmente gestisce il prestito di elementi.

Nome Test	**Scopo**	**Aspettativa**	**Possibili Fallimenti**
`addBorrowTest`	Verifica l'aggiunta di un prestito	Deve restituire `true` dopo aver aggiunto il prestito	Il prestito non viene registrato e il metodo restituisce `false`
`removeBorrowTest`	Verifica la rimozione di un prestito	Deve restituire `true` dopo aver rimosso il prestito	Il prestito non viene rimosso dal database
`getBorrowedElementsForUserTest`	Verifica il recupero dei prestiti per un utente	Deve restituire un elenco di prestiti non `null`	Il metodo restituisce `null`, indicando un problema

Table 16: Tabella riassuntiva dei test su `BorrowsManager`

4.2.1 Conclusione

I test effettuati sulla classe `BorrowsManager` hanno verificato le funzionalità principali relative alla gestione dei prestiti. In particolare:

- **Aggiunta di un prestito**: Il test `addBorrowTest` ha confermato che il metodo registra correttamente un nuovo prestito nel database, restituendo un valore positivo in caso di successo.
- **Rimozione di un prestito**: Il test `removeBorrowTest` ha dimostrato che è possibile eliminare un prestito esistente, garantendo che l'elemento venga effettivamente rimosso dal sistema.
- **Recupero dei prestiti di un utente**: Il test `getBorrowedElementsForUserTest` ha garantito che l'applicazione è in grado di restituire un elenco valido di elementi attualmente presi in prestito da un utente specifico.

I test hanno fornito risultati coerenti con le aspettative, garantendo che il sistema gestisca correttamente l'aggiunta, la rimozione e il recupero dei prestiti. Tuttavia, eventuali miglioramenti potrebbero includere test per verificare comportamenti in scenari limite, come tentativi di prestito con disponibilità insufficiente o rimozione di prestiti inesistenti.

4.3 ConnectionManagerTest

Questa classe, ConnectionManagerTest, gestisce la connessione a un database PostgreSQL per i test.

- **Connessione al database**: Il metodo `getConnection()` viene chiamato per assicurarsi che la connessione venga creata correttamente.
- **Verifica della validità**: `isConnectionValid()` garantisce che la connessione sia utilizzabile prima di procedere con altre operazioni.
- **Chiusura della connessione**: `closeConnection()` assicura che la connessione venga chiusa correttamente dopo l'uso.

Per questi test dobbiamo tenere di conto che:

- **Pattern Singleton**: Usato per mantenere un'unica connessione.
- **Gestione delle eccezioni**: Gli errori di connessione vengono catturati e stampati.
- **Metodi `setDbUser()` e `setDbPass()`**: Permettono di cambiare le credenziali senza ricompilare il codice.

4.4 DigitalMediaManagerTest

Questa classe testa le operazioni di gestione dei media digitali effettuati da **DigitalMediaManager**, verificando che i metodi funzionino correttamente con il database.Viene usato Mockito per il mocking di ConnectionManager e MainController, simulando l'accesso al database senza eseguire effettivamente query su un database reale.

Test	Scopo	Criterio di Successo
`addDigitalMediaTest()`	Verifica che un media digitale possa essere inserito nel database.	Il metodo restituisce un ID valido, non `null`.
`updateDigitalMediaTest()`	Controlla che i dati di un media digitale possano essere aggiornati.	Il metodo restituisce `true`.
`getDigitalMediaTest()`	Testa il recupero di un media digitale dal database.	Il metodo restituisce un oggetto valido, non `null`.

Table 17: Riassunto dei test effettuati sulla classe DigitalMediaManager

4.4.1 Conclusioni

I test condotti sulla classe `DigitalMediaManager` confermano il corretto funzionamento delle operazioni principali sulla gestione dei media digitali. In particolare:

- Il metodo di inserimento di un media digitale, `addDigitalMediaTest`, ha sempre restituito un identificativo valido, dimostrando la corretta interazione con il database.
- Il metodo di aggiornamento `updateDigitalMediaTest` ha sempre restituito `true`, segnalando che le modifiche ai dati vengono applicate correttamente.
- Il metodo di recupero dei media digitali `getDigitalMediaTest` ha restituito oggetti validi, confermando l'integrità dei dati salvati.

L'uso di **Mockito** per il mocking del `ConnectionManager` e del `MainController` ha permesso di isolare il codice e testare le funzionalità senza dipendere da un database reale. Inoltre, grazie all'uso del `rollback` alla fine dei test, i dati temporanei non vengono mantenuti nel database.

4.5 ElementManagerTest

I test condottu sulla classe **ElementManagerTest** si occupano di verificare il corretto funzionamento di operazioni di aggiunta, rimozione e ricerca di elementi all'interno del database. I test della classe ElementManagerTest sono stati progettati per verificare il corretto funzionamento delle operazioni CRUD sugli "elementi" nel database. Di seguito sono descritti i test effettuati. Per la rimozione di un elemento, con lo scopo di verificare che un elemento possa essere rimosso correttamente dal database, si opera nel seguente modo:

- **Preparazione**: Viene inserito un elemento nel database con una query SQL. L'ID generato per l'elemento viene salvato per essere utilizzato nel test.
- **Test**: Viene chiamato il metodo `removeElement(generatedId)` per rimuovere l'elemento dal database. Successivamente, viene effettuata l'asserzione `assertTrue(removeElement(generatedId))`, che verifica che la rimozione sia stata eseguita correttamente.
- **Verifica**: Se il test passa, significa che l'elemento è stato correttamente rimosso dal database.

Invece per quanto il corretto recupero di un elemento:

- **Preparazione**: Viene inserito un nuovo elemento nel database, e l'ID generato per l'elemento viene salvato per essere utilizzato nel test.
- **Test**: Viene chiamato il metodo `getElement(generatedId)` per recuperare l'elemento dal database. Successivamente, viene effettuata l'asserzione `assertEquals(generatedId, element.getId())`, che verifica che l'ID dell'elemento recuperato corrisponda all'ID generato precedentemente.
- **Verifica**: Se l'asserzione passa, significa che il metodo `getElement()` funziona correttamente e recupera gli elementi dal database in modo corretto.

4.5.1 Conclusioni

I test effettuati hanno consentito di verificare il corretto funzionamento delle operazioni CRUD sugli "elementi" nel database. In particolare, i test hanno confermato che:

- Il metodo `removeElement()` rimuove correttamente un elemento dal database.
- Il metodo `getElement()` recupera correttamente un elemento dal database utilizzando l'ID.

4.6 GenreManagerTest

I test nella classe GenreManagerTest sono progettati per verificare il corretto funzionamento dei metodi relativi alla gestione dei generi associati agli elementi nel sistema. Ecco una descrizione dettagliata di ciascun test.

I test nella classe `GenreManagerTest` sono progettati per verificare il corretto funzionamento dei metodi relativi alla gestione dei generi associati agli elementi nel sistema. I test coprono vari scenari, tra cui l'associazione di generi agli elementi, l'aggiunta e la rimozione di generi, nonché il recupero dei generi per un dato elemento.

Test di Setup e Teardown:

- `@BeforeAll` e `@AfterAll`: Questi metodi vengono eseguiti prima e dopo tutti i test nella classe.
- `setUp()`: Crea un elemento e un genere nel database, i cui ID e codici vengono salvati per essere utilizzati nei test successivi.
- `tearDown()`: Rimuove i dati inseriti nel database durante i test, cancellando le associazioni e i dati generati.

Nel dettaglio i test:
Test: `associateGenreWithElementTest()`

- **Scopo**: Verificare che un genere possa essere correttamente associato a un elemento.
- **Preparazione**: L'ID dell'elemento `generatedElementId` e il codice del genere `generatedGenreCode` sono già presenti nel database.
- **Esecuzione**: Viene chiamato il metodo `genreManager.associateGenreWithElement(generat generatedGenreCode)`.

- **Verifica**: L'asserzione `assertTrue(result)` verifica che l'associazione sia riuscita.

Test: `addGenreTest()`

- **Scopo**: Verificare che un nuovo genere possa essere aggiunto al database.
- **Preparazione**: Non è necessaria alcuna preparazione in quanto il test si concentra sull'aggiunta di un nuovo genere.
- **Esecuzione**: Viene chiamato il metodo `genreManager.addGenre(new Genre("Insert Test"))`.
- **Verifica**: L'asserzione `assertTrue(result)` verifica che il genere sia stato aggiunto correttamente.

Test: `getAllGenresTest()`

- **Scopo**: Verificare che tutti i generi possano essere recuperati correttamente.
- **Preparazione**: I generi esistenti sono già nel database, incluso quello inserito nel test precedente.
- **Esecuzione**: Viene chiamato il metodo `genreManager.getAllGenres()` per recuperare tutti i generi.
- **Verifica**: L'asserzione `assertNotNull(result)` verifica che la lista di generi non sia vuota.

Test: `getGenresForElementTest()`

- **Scopo**: Verificare che i generi associati a un elemento possano essere recuperati correttamente.
- **Preparazione**: Un'associazione tra l'elemento `generatedElementId` e il genere `generatedGenreCode` viene inserita nella tabella `elementgenres`.
- **Esecuzione**: Viene chiamato il metodo `genreManager.getGenresForElement(generatedEle`
- **Verifica**: L'asserzione `assertNotNull(result)` verifica che la lista di generi associati all'elemento non sia vuota.

Test: `removeGenreFromElementTest()`

- **Scopo**: Verificare che un genere possa essere rimosso da un elemento.
- **Preparazione**: Un'associazione tra l'elemento `generatedElementId` e il genere `generatedGenreCode` viene inserita nella tabella `elementgenres`.
- **Esecuzione**: Viene chiamato il metodo `genreManager.removeGenreFromElement(generatedE generatedGenreCode)`.
- **Verifica**: L'asserzione `assertTrue(result)` verifica che la rimozione sia riuscita.

4.7 PeriodicPublicationManagerTest

I test progettati nella classe**PeriodicPublicationManagerTest** sono stati progettati con lo scopo di testare l'aggiunta di un elemento **PeriodicPublication** all'interno del database e di un suo eventuale aggiornamento.

4.8 UserManagerTest

I test nella classe **UserManagerTest** sono stati progettati per verificare il corretto funzionamento della classe **UserManager**. Per verificare che tutto sia corretto i test utilizzano il mocking per evitare la necessità di interagire direttamente con un database reale. Il mock della connessione al database simula l'interazione con il sistema senza effettuare modifiche reali e per quanto riguarda la connessione al database viene gestita esplicitamente, con commit e rollback, evitando che i test influiscano permanentemente sul database.

4.9 MainControllerTest

I test della classe **MainControllerTest** sono stati progettati per verificare il corretto funzionamento della classe **MainController**.

4.10 UserControllerTest

I test della classe **UserControllerTest** sono stati progettati per verificare il corretto funzionamento della classe **UserController**.

4.11 Risultati Maven

I risultati ottenuti tramite i test effettuati con **Maven** sono i seguenti:

```
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.966 s -- in com.swe.libraryprogram.dao.BookManagerTest
[INFO] Running com.swe.libraryprogram.dao.BorrowsManagerTest
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.550 s -- in com.swe.libraryprogram.dao.BorrowsManagerTest
[INFO] Running com.swe.libraryprogram.dao.DigitalMediaManagerTest
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.621 s -- in com.swe.libraryprogram.dao.DigitalMediaManagerTest
[INFO] Running com.swe.libraryprogram.dao.ElementManagerTest
Elemento con ID 989 rimosso correttamente.
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.715 s -- in com.swe.libraryprogram.dao.ElementManagerTest
[INFO] Running com.swe.libraryprogram.dao.GenreManagerTest
Generated Element ID: 991
Cleanup successful: Deleted test data.
[INFO] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.682 s -- in com.swe.libraryprogram.dao.GenreManagerTest
[INFO] Running com.swe.libraryprogram.dao.PeriodicPublicationManagerTest
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.569 s -- in com.swe.libraryprogram.dao.PeriodicPublicationManagerTest
[INFO] Running com.swe.libraryprogram.dao.UserManagerTest
Utente inserito correttamente.
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.421 s -- in com.swe.libraryprogram.dao.UserManagerTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 19, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```

Figure 19: Risultati dei test **Maven**