



UNIVERSITÀ
DEGLI STUDI
FIRENZE

UNIVERSITÀ DEGLI STUDI DI FIRENZE

Library Management System

Docente:

Vicario Enrico

Corso:

Ingegneria del Software

Studente:

Chirli Gabriele
Gopalakrishnan Nirushan
Berisha Matias Dardan
Donadoni Mattia

Matricola:

7110168
7109672
7111646
7109851

Indice

Indice	1
1 Introduzione	3
1.1 Statement	3
1.2 Architettura e Tecnologie Implementative	3
2 Progettazione	4
2.1 Use-Case Diagram	4
2.2 Use-Case Templates	5
2.3 Page Navigation	7
2.4 Mockup	7
2.5 Database	11
2.5.1 Modello ER	11
2.5.2 Modello Relazionale	12
3 Implementazione	13
3.1 Domain Model	14
3.2 ORM	15
3.3 Business Logic	16
3.4 Controller	17
3.5 View	18
3.6 Codice	19
3.6.1 Organizzazione Cartelle	19
3.6.2 Singleton	19
3.6.3 Gerarchia delle Responsabilità	20
3.6.4 Gestione dipendenze tra DAO	21
3.6.5 Gestione tabelle Commento	21
3.7 Test e Gestione sicurezza	22
3.7.1 Test Unit	23
3.7.2 Test UI	26
3.8 Gestione Errori e Sicurezza	28
3.8.1 Gestione degli Errori	28
3.8.2 Metodi di Validazione dei campi	33
3.8.3 Accesso concorrente	34
4 Conclusione	36
4.1 Strumentazione Esterna	36
4.1.1 Intelligenza Artificiale	36
4.2 Valutazioni Finali	37

Elenco delle figure

1 Diagramma Casi d'Uso	4
2 UC-1: Modifica Account (Registrazione)	5
3 UC-2: Summary Prestito (Effettua)	5
4 UC-3: Summary Libro (Aggiunge)	6
5 UC-4: Conclude Prestito	6
6 Page Navigation Diagram	7
7 MCK.1 Registrazione Utente	7

8	MCK.2 Login	8
9	MCK.3 Prenotazione Libro	8
10	MCK.4 Pagina Commenti	9
11	MCK.5 Aggiunta Libro	9
12	MCK.6 Conclusione Prestito	10
13	MCK.7 Catalogo Volumi	10
14	Modello ER	11
15	Diagramma dei Pacchetti	13
16	Diagramma UML Domain Model	14
17	Diagramma UML ORM	15
18	Diagramma UML Business Logic	16
19	Diagramma UML Controller	17
20	Diagramma UML View	18
21	Cartelle del Progetto	19
22	Implementazione del Singleton	19
23	Metodo Controller che richiama un metodo Service	20
24	Metodo Service che richiama un metodo DAO	20
25	Metodo DAO che interroga il database	20
26	Metodo Service del Prestito che richiama un altro DAO	21
27	Metodo del DAO Libro che estrae l'informazione necessaria dal database	21
28	Struttura tabelle commento nel Database	22
29	Test JUnit per la prenotazione di un libro e l'annullamento del prestito	23
30	Query per la prenotazione di un libro	23
31	Query per l'annullamento di un prestito	24
32	Test JUnit per il recupero dell'isbn corretto	24
33	Query per il recupero di un isbn	24
34	Test JUnit per la verifica della disponibilità di un'opera	25
35	Query per la verifica della disponibilità di un'opera	25
36	TestFX per il Login	26
37	TestFX per la prenotazione di un libro	26
38	TestFX per il prolungamento di un prestito	27
39	TestFX per la verifica dei campi in Registrazione	27
40	Query per il Login nell'UserDAO	28
41	Verifica esito Query da parte del Controller	29
42	Login errato alert	29
43	Campo non valido alert	30
44	Verifica del prestito sul DAO	31
45	Gestione dell'eccezione sul prestito non valido	31
46	Prestito non consentito alert	32
47	Errore caricamento dati	32
48	Errore di connessione al Database	33
49	Regex che definiscono i vincoli per i campi Utente	34
50	Metodi di verifica per i vincoli sui campi	34
51	Inizializzazione Session via login	34
52	Classe Session	35
53	Reset Session via logout	35

1 Introduzione

1.1 Statement

Si propone la realizzazione di un servizio di gestione libri per una biblioteca, tramite applicazione in Java sviluppata su IDE *IntelliJ* e dotata di interfaccia grafica realizzata con *JavaFX*; dotato di un sistema centralizzato per la prenotazione, il ritiro e la gestione catalogo della biblioteca. L'obiettivo principale è semplificare e digitalizzare il processo di prenotazione, permettendo agli utenti di visualizzare da remoto il catalogo libri proposto dalla biblioteca di riferimento, con la possibilità di individuare i libri di interesse, ed eventualmente prenotarli per il ritiro in loco. Inoltre, il sistema consente anche all'utente di visualizzare la lista dei libri presi in prestito, vederne l'eventuale scadenza (entro la quale effettuare la restituzione) e quindi prolungarne la durata.

1.2 Architettura e Tecnologie Implementative

Il progetto è suddiviso in **packages**, ognuno dei quali aventi specifiche responsabilità. Tale suddivisione ci garantisce l'isolamento delle operazioni e la gestione dislocata e partizionata dei compiti, rendendo l'intero sistema più dinamico, organizzato e facilmente ampliabile e manutenibile. Basterà infatti aggiungere o modificare classi all'interno dei vari **package** per aggiornare le logiche di funzionamento, senza intaccare le parti non interessate.

Nello specifico: il **package View** consente l'interazione tra utente e sistema, attraverso l'uso di un'interfaccia grafica intuitiva e semplice. Le richieste di operazione dell'utente, eseguite tramite la GUI, vengono poi gestite dalle classi all'interno del **package Controller**, che richiama i metodi necessari e si interfaccia con gli altri pacchetti per soddisfare tali richieste, e che consente inoltre la navigazione all'interno delle pagine. Ogni pagina ha un suo Controller che la gestisce.

Il **package Business Logic**, invece, elabora tutto ciò che riguarda le logiche più complesse del sistema, che non rientrano strettamente nella gestione dei dati o nell'interfaccia utente (come per esempio controlli di validazione delle operazioni o elaborazione di dati e interazioni tra diverse entità). Questo lavora a stretto contatto con il **package Model**, che raggruppa e definisce le entità principali del progetto, e con il **package ORM**, che si interfaccia direttamente con il *database* per gestirne i dati e assicurarne la persistenza. La dipendenza tra le varie entità del package ORM è gestita direttamente dalla Business Logic.

Per lo sviluppo del programma sono stati utilizzati diversi *Design Pattern*, descritti di seguito:

- **Singleton**: assicura l'unicità di specifiche istanze di classi come **DatabaseConnection** (vedi Sez. 3.6.2).
- **MVC** (Model-View-Controller): separa le logiche di presentazione, gestione e dati. Nello specifico, è stato implementato il **Page Controller Pattern** per poter gestire le funzioni di ogni pagina su uno specifico controller.
- **DAO** (Data Access Object): si occupa dell'accesso ai dati tramite connessione a database.

Il Model e le sue entità sono mappate sul DBMS relazionale *PostgreSQL* attraverso un Object-Relational Mapping (**ORM**) Tool realizzato tramite il pattern DAO.

L'architettura generale del progetto è stata pensata e progettata esclusivamente per garantire l'elevata manutenibilità e favorirne l'espandibilità nel tempo in base alle necessità future. Il codice è stato realizzato con *IntelliJ*, l'interfaccia grafica con *JavaFX* e le classi di test su interfaccia grafica con il relativo *TextFX*, mentre per i test da linea di codice *JUnit* (per questi vedi Sez. 3.7).

2 Progettazione

2.1 Use-Case Diagram

Il programma prevede due attori principali: "Utente" e "Biblioteca". Ognuno di questi ha la possibilità di effettuare determinate operazioni specifiche al proprio ruolo.

L'**Utente** può modificare le informazioni relative al proprio account o cancellarlo del tutto, può richiedere, estendere o cancellare un prestito per un libro a scelta, la prenotazione arriverà poi (se approvata dal sistema) alla biblioteca che predisporrà il libro per il ritiro in loco. Il prestito partirà automaticamente al momento della richiesta e starà all'utente andare a ritirarlo fisicamente. L'utente può poi scegliere di lasciare un commento prima, durante o dopo aver effettuato un prestito. Il commento può riguardare il volume fisico del libro preso in prestito (es. condizioni e/o difetti) oppure l'opera in generale (es. recensione e pareri). Per lasciare un commento su di un'opera non è necessario aver ottenuto quel libro in prestito, mentre lo è per i commenti riguardanti gli specifici volumi. Ogni utente può lasciare al massimo un commento per opera e uno per volume.

La **Biblioteca**, invece, può modificare il catalogo dei libri visualizzabili sul sito, aggiungendoli ed eliminandoli in base alle esigenze. La Biblioteca dovrà anche concludere i prestiti terminati, attraverso il sistema, una volta che l'Utente avrà effettuato la restituzione in loco. Ciò è fatto esclusivamente per facilitare la gestione delle copie date in prestito ed assicurarsi delle condizioni di esse, valutando personalmente eventuali danni e difetti e applicando, se necessario, le dovute sanzioni al detentore. Quindi, l'**Utente** potrà visualizzare i libri sul catalogo, cercarli per titolo, autore, genere o casa editrice e verificarne la disponibilità. Potrà, poi, visualizzare i propri prestiti, cancellarli o richiederne un'estensione. La **Biblioteca** allo stesso modo avrà la possibilità di visualizzare la lista intera dei prestiti per ogni Utente, in modo da poterne tener traccia.

Tutte le operazioni riportate nel diagramma dei casi d'uso sottostante sono da considerarsi effettuabili solo dopo aver effettuato il log-in.

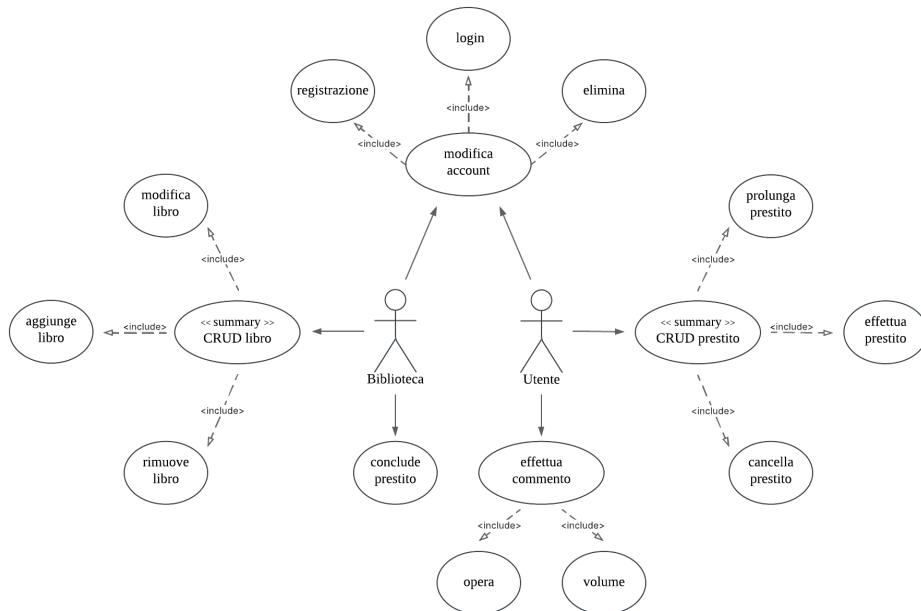


Figura 1: Diagramma Casi d'Uso

2.2 Use-Case Templates

Di seguito sono descritti 4 dei casi d'uso più importanti.

UC-1	Registrazione
Description	Registrare un nuovo utente
Main Actor	Utente
Basic Course	<ol style="list-style-type: none"> 1. L'utente apre la pagina di registrazione (MCK 1, fig. 7) 2. L'utente inserisce i dati personali (CF, email, password, ecc.) 3. L'utente preme 'registrati' 4. Il sistema verifica che i dati inseriti siano corretti 5. Il sistema reindirizza l'utente alla pagina di login (MCK 2, fig. 8)
Alternative Course	<ol style="list-style-type: none"> 4a. Il sistema comunica che i dati inseriti non sono corretti o che l'utente è già registrato e permette di ritentare

Figura 2: UC-1: Modifica Account (Registrazione)

UC-2	Effettua prestito
Description	Richiedere un prestito alla biblioteca
Main Actor	Utente
Basic Course	<ol style="list-style-type: none"> 1. L'utente visualizza il catalogo dei libri 2. L'utente cerca il libro nel catalogo 3. L'utente prenota il libro desiderato (MCK 3, fig. 9) 4. Il sistema verifica la disponibilità del libro 5. Il sistema verifica se l'utente ha superato il limite di prestiti 6. Il sistema accetta il prestito e lo mostra nella sezione prenotazioni attive
Alternative Course	<ol style="list-style-type: none"> 4a. Il sistema comunica che il libro non è disponibile 5a. Il sistema comunica che l'utente ha superato il limite massimo temporaneo di prenotazioni consentite

Figura 3: UC-2: Summary Prestito (Effettua)

UC-3	Aggiunge libro
Description	Aggiungere un nuovo libro al catalogo
Main Actor	Bibliotecario
Basic Course	<ol style="list-style-type: none"> 1. Bibliotecario apre il catalogo libri 2. Apre la pagina di aggiunta libro (MCK 5, fig. 11) 3. Inserisce le informazioni del libro da aggiungere 4. Il sistema controlla che non sia già stato inserito 5. Il sistema lo aggiunge correttamente
Alternative Course	4a. Il libro è già presente nella collezione e il sistema lo notifica

Figura 4: UC-3: Summary Libro (Aggiunge)

UC-4	Conclude Prestito
Description	Terminare un prestito effettuato dall'utente
Main Actor	Bibliotecario
Basic Course	<ol style="list-style-type: none"> 1. Bibliotecario seleziona il prestito nella lista dei prestiti attivi 2. Clicca su termina prestito (MCK 6, fig: 12) 3. Il sistema rende il libro nuovamente disponibile agli utenti
Alternative Course	

Figura 5: UC-4: Conclude Prestito

2.3 Page Navigation

Il Page Navigation diagram seguente mostra in dettaglio tutti i possibili passaggi di flusso nella navigazione tra le varie finestre del programma.

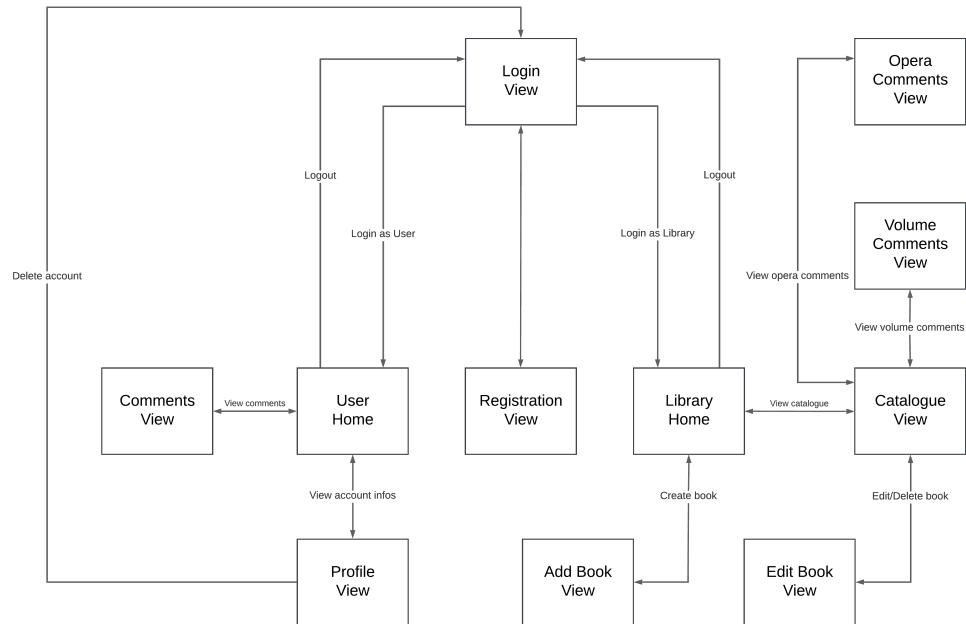


Figura 6: Page Navigation Diagram

2.4 Mockup

Di seguito sono presentate delle istantanee prese dall’interfaccia grafica del programma, utili a illustrare in modo chiaro e immediato i casi d’uso e le operazioni effettuabili precedentemente descritte, oltre alla navigazione tra le schermate principali del sistema. Sono mostrate 7 tra tutte le interfacce fondamentali sia per l’Utente che per la Biblioteca presenti nel programma.

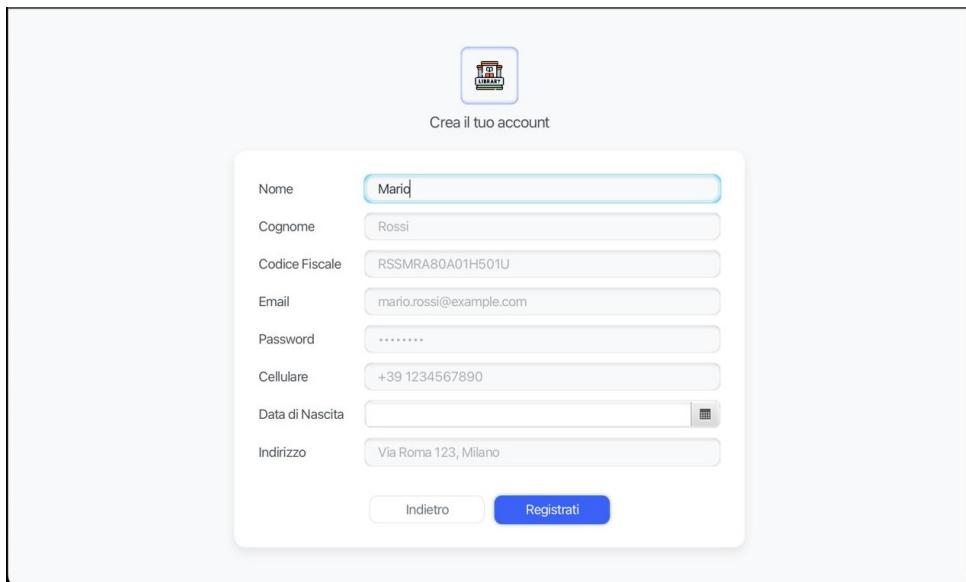


Figura 7: MCK.1 Registrazione Utente

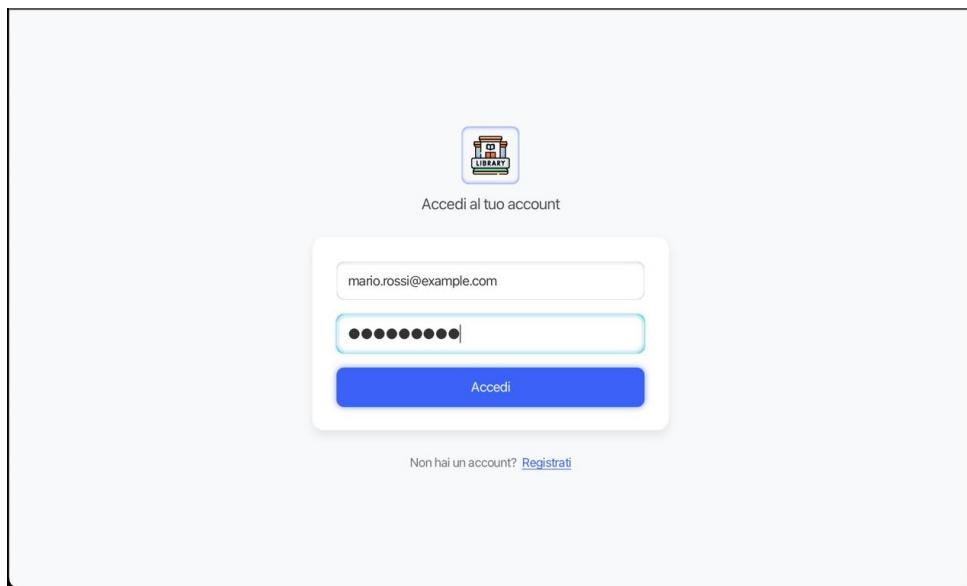


Figura 8: MCK.2 Login

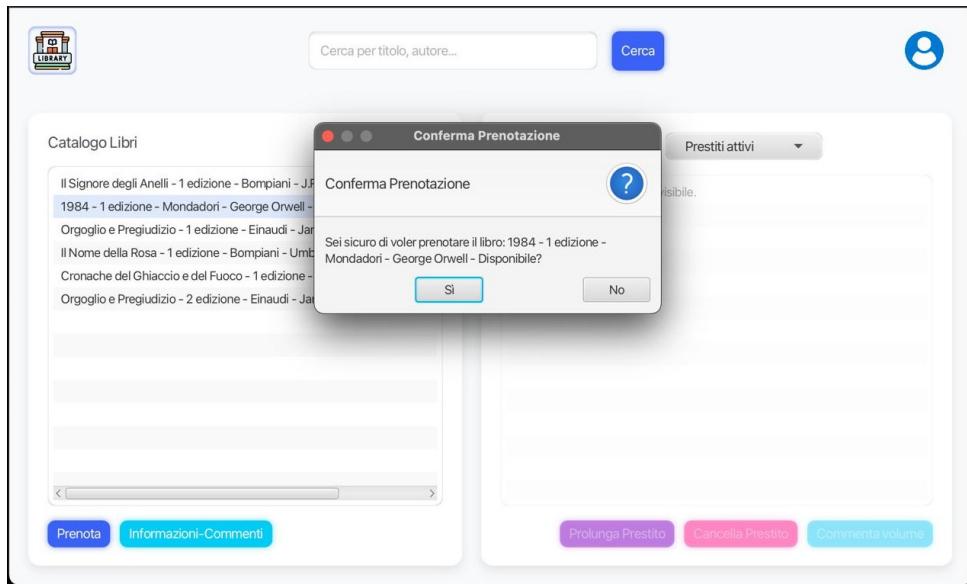
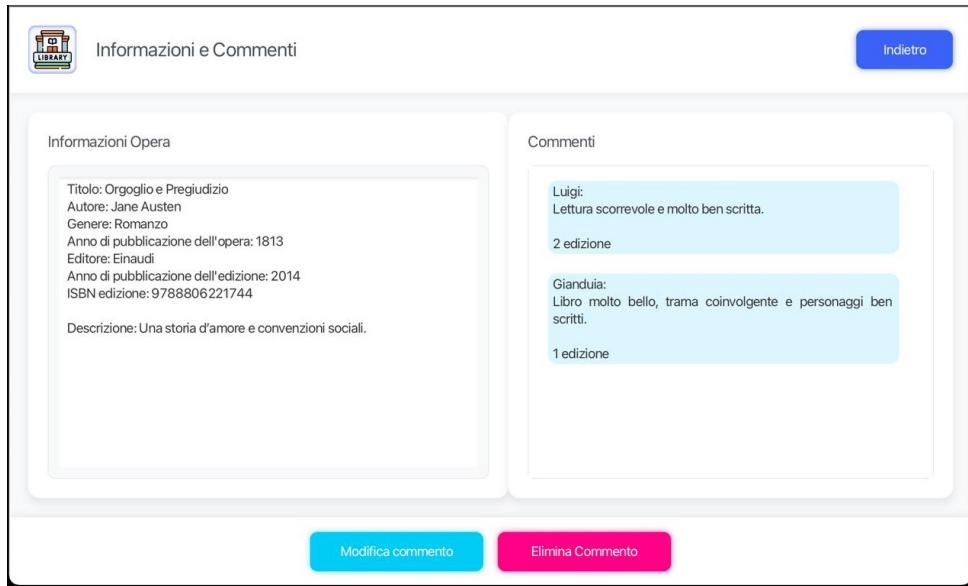
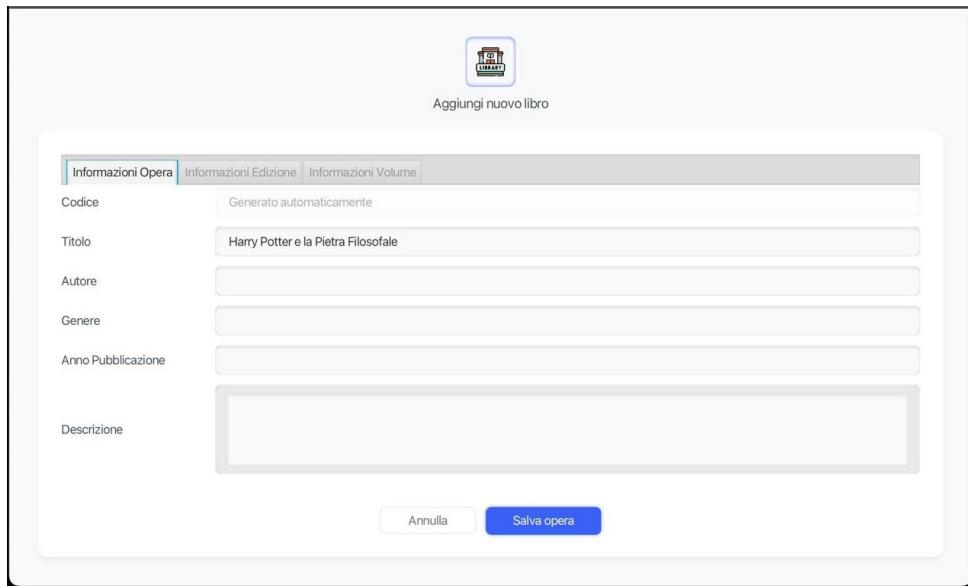


Figura 9: MCK.3 Prenotazione Libro



The screenshot shows a web page titled "Informazioni e Commenti" (Information and Comments). On the left, under "Informazioni Opera" (Book Information), there is a summary of the book "Orgoglio e Pregiudizio" by Jane Austen, published in 1813 by Einaudi, with ISBN 9788806221744. The description notes it's a story of love and social conventions. On the right, under "Commenti" (Comments), two comments are displayed: one from Luigi praising the reading experience, another from Gianduia describing the book as very beautiful with an engaging plot and well-written characters, and a third comment slot labeled "1 edizione". At the bottom are buttons for "Modifica commento" (Edit comment) and "Elimina Commento" (Delete comment).

Figura 10: MCK.4 Pagina Commenti



The screenshot shows a form titled "Aggiungi nuovo libro" (Add new book). It includes tabs for "Informazioni Opera" (Book Information), "Informazioni Edizione" (Edition Information), and "Informazioni Volume" (Volume Information). The "Informazioni Opera" tab is active. The form fields are: Codice (Code) - "Generato automaticamente" (Generated automatically); Titolo (Title) - "Harry Potter e la Pietra Filosofale" (Harry Potter and the Philosopher's Stone); Autore (Author); Genere (Genre); Anno Pubblicazione (Publication Year); and Descrizione (Description). At the bottom are "Annulla" (Cancel) and "Salva opera" (Save book) buttons.

Figura 11: MCK.5 Aggiunta Libro

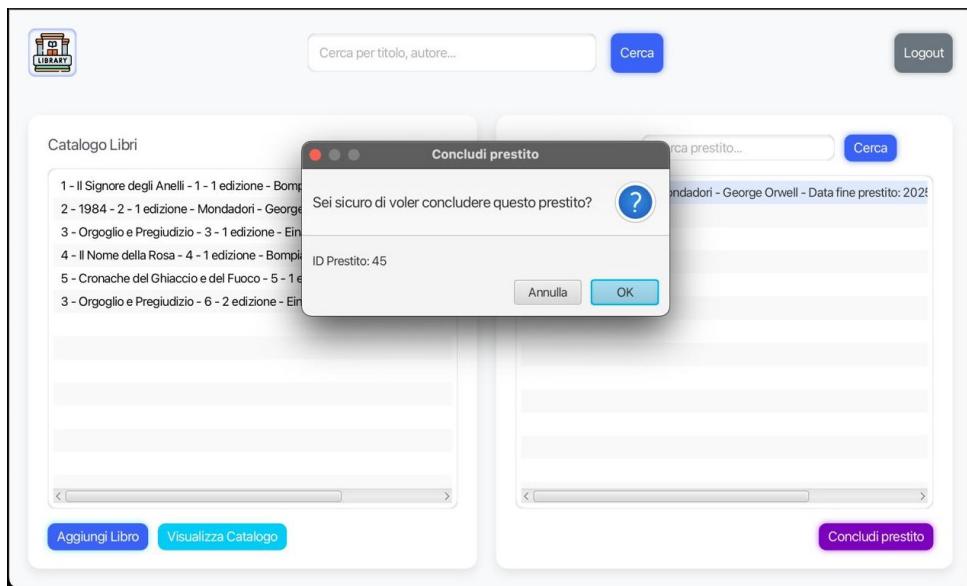


Figura 12: MCK.6 Conclusione Prestito

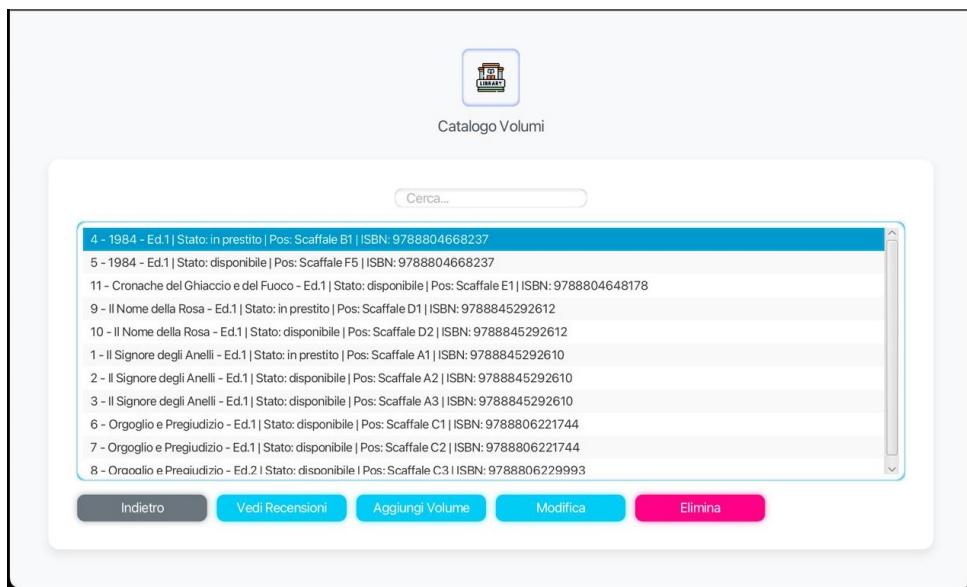


Figura 13: MCK.7 Catalogo Volumi

2.5 Database

Per la gestione della persistenza dei dati è stato utilizzato il DBMS *PostgreSQL*. Nella sezione 2.5.1 (ER) è presente il modello grafico Entity-Relationship, mentre nella sezione 2.5.2 è presente quello relazionale.

2.5.1 Modello ER

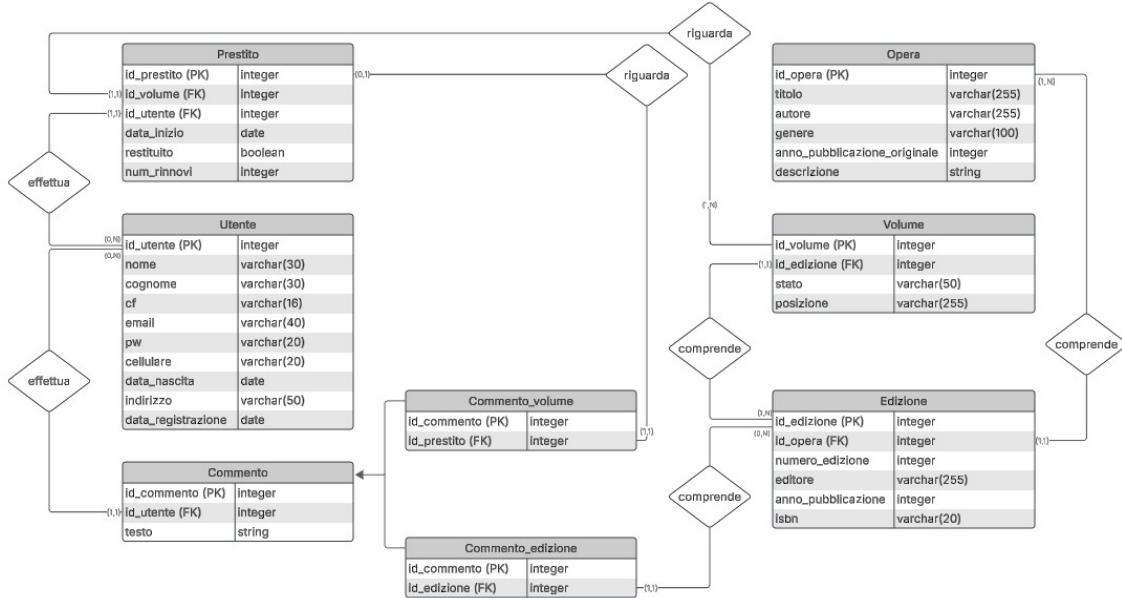


Figura 14: Modello ER

Le relazioni del modello sono così descritte:

- **Effettua**: Relazione uno a molti, indica le operazioni effettuate dall'*Utente*; quindi *Prestiti* attivati e *Commenti* lasciati.
- **Riguarda**: Relazione uno a molti, unisce ogni *Prestito* al *Volume* interessato e al relativo eventuale *Commento* su di esso.
- **Comprende**: Relazione uno a molti, lega ogni *Volume* alla relativa *Edizione* e ogni *Edizione* alla relativa *Opera*. Oltre che ogni *Edizione* ai propri *Commenti*.

2.5.2 Modello Relazionale

UTENTE(PK (id_utente), nome, cognome, cf, email, pw, cellulare, data_nascita, indirizzo, data_registrazione)

OPERA(PK (id_opera), titolo, autore, genere, anno_pubblicazione_originale, descrizione)

EDIZIONE(PK (id_edizione), FK (id_opera) → OPERA(id_opera), numero_edizione, editore, anno_pubblicazione, isbn)

VOLUME(PK (id_volume), FK (id_edizione) → EDIZIONE(id_edizione), stato, posizione)

PRESTITO(PK (id_prestito), FK (id_utente) → UTENTE(id_utente),

FK (id_volume) → VOLUME(id_volume), data_inizio, num_rinnovi, restituito)

COMMENTO(PK (id_commento), FK (id_utente) → UTENTE(id_utente), testo)

COMMENTO_EDIZIONE(PK (id_commento) → COMMENTO(id_commento), FK (id_edizione) → EDIZIONE(id_edizione))

COMMENTO_VOLUME(PK (id_commento) → COMMENTO(id_commento),

FK (id_prestito) → PRESTITO(id_prestito))

3 Implementazione

Il progetto è suddiviso in vari package per permettere una corretta separazione dei compiti e delle responsabilità, consentendo facile manutenzione, ampliabilità e una corretta interazione tra questi. Il package **View** rappresenta l'interfaccia grafica attraverso cui gli attori (Utente e Biblioteca) interagiscono con il sistema. Ogni view realizzata presenta ciascuna delle pagine navigabili. Le richieste qui effettuate vengono elaborate dal package **Controller**, tramite apposite classi (una per ognuna delle pagine) che raccolgono le richieste e richiamano le funzioni necessarie. Ciò avviene tramite l'interazione con il **Business Logic** e le relative classi **Service** che, effettuati i dovuti controlli di validazione, si interfaceranno con l'**ORM** per recuperare o modificare i dati necessari dal database. L'interazione tra il programma e la "memoria" è garantita in modo sicuro dall'utilizzo delle entità incluse all'interno del **Domain Model** che funge da tramite tra le due parti (vedi Sez.3.6.3).

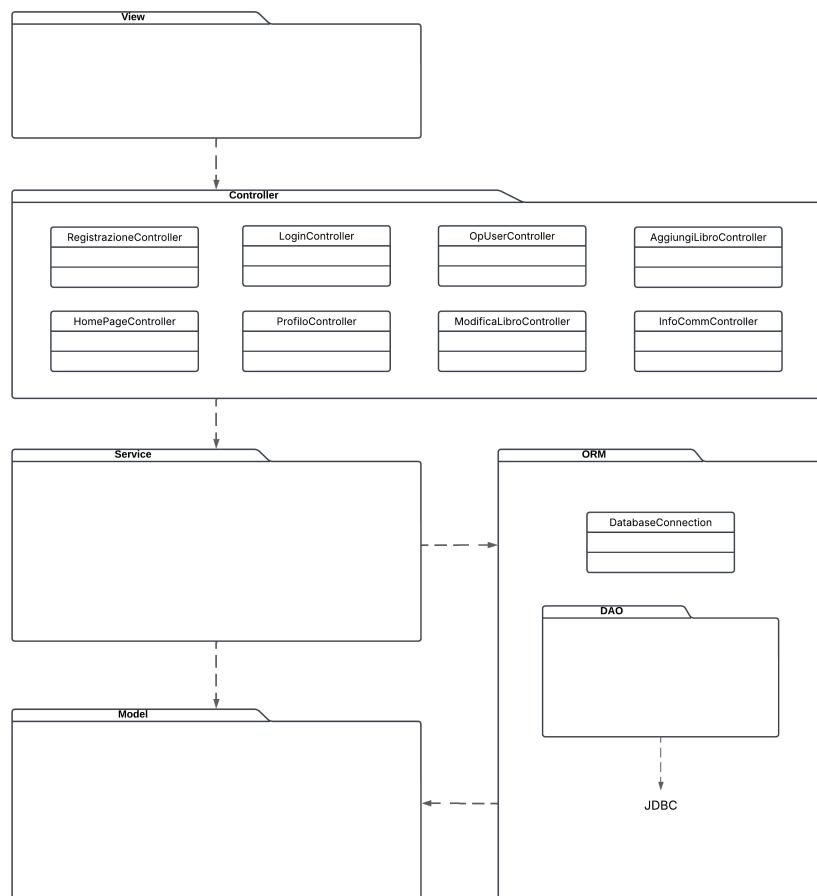


Figura 15: Diagramma dei Pacchetti

3.1 Domain Model

Le classi del **Domain Model** rappresentano le entità fondamentali del programma e contengono tutti gli attributi e i metodi utili alla loro gestione e utilizzo. La classe **Utente** modella una persona digitale che, tramite il log-in, può essere profilata come facente parte dell'attore Biblioteca o Utente comune; esso può quindi effettuare tutte le operazioni precedentemente descritte nella sezione 2.1 in base al proprio ruolo.

La classe **Opera** rappresenta nel catalogo l'entità concettuale "libro" disponibile, mentre **Volume** rappresenta la copia fisica di tale libro presente in biblioteca; la classe **Edizione** indica appunto l'edizione specifica di un determinato volume. Queste tre classi vengono poi gestite insieme e inserite nel catalogo digitale, tramite la classe contenitore **Catalogo**, appunto, attraverso un'unica rappresentazione del libro; si può quindi dire che le tre classi rappresentano le specifiche fisiche e astratte del singolo libro. La classe **Prestito**, invece, esplicita il legame tra Utente e Volume, modellando i prestiti e rendendoli quindi visualizzabili e gestibili da entrambi gli attori. La classe **Session** serve per rendere visibili le informazioni dell'utente attualmente loggato a tutte le classi del progetto.

Ogni **Prestito** è collegato a un **Utente** e a un **Volume**, l'entità **Volume** può essere associata a più prestiti in base alle istanze (copie) disponibili, mentre un **Utente** può avere fino a 3 **Prestiti** attivi contemporaneamente. Comunque, lo storico dei prestiti effettuati da un **Utente** viene salvato quindi, difatto, ogni **Utente** può essere legato dagli 0 agli n **Prestiti**.

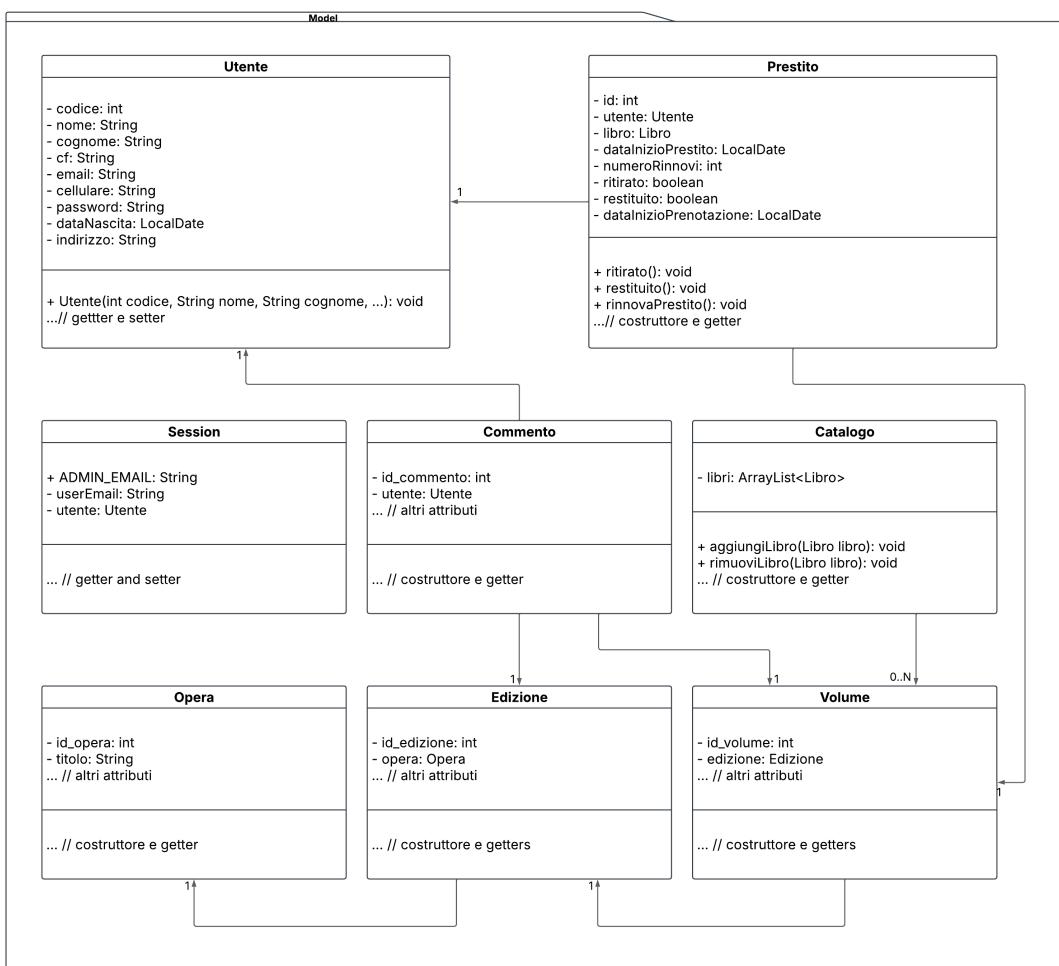


Figura 16: Diagramma UML Domain Model

3.2 ORM

Le classi del package **ORM** rappresentano il collegamento diretto tra il sistema e il database, garantendo la persistenza dei dati e l'esecuzione delle operazioni CRUD. La classe principale **DatabaseConnection** gestisce la connessione al database permettendoci l'accesso alla struttura dati, consentendo, nell'effettivo, di poter eseguire le query direttamente da codice. Questa classe implementa il pattern *Singleton* per garantire che l'accesso al database sia univoco.

Le restanti classi sono poi più specifiche e specializzate in singoli compiti, ancora una volta in modo da avere un codice di più facile gestione e organizzazione. Sono state implementate, quindi, le classi **UserDAO**, **BookDAO** e **LoanDAO** che estendono le funzionalità di **DatabaseConnection** e si occupano rispettivamente delle operazioni di registrazione, log-in e log-out la prima, della stampa del catalogo e la ricerca dei libri all'interno di esso la seconda, e infine della gestione dei prestiti l'ultima. Sebbene le operazioni sui vari libri vengano gestite direttamente dalla classe **BookDAO**, comunque sono presenti le specifiche classi **VolumeDAO**, **OperaDAO** e **EditionDAO** utili a recuperare dal database le informazioni riguardanti le varie specifiche del Libro.

Ogni DAO permette di connettersi al database e garantisce l'esecuzione di operazioni sui dati tramite query e attraverso l'utilizzo delle entità fornite dal *Domain Model*.

L'**ORM** è alle strette dipendenze delle classi **Controller**, assicurando che ogni accesso o modifica al database non sia arbitraria ma bensì coordinata in base alle richieste, rendendo la gestione dei dati più sicura. Inoltre, la *dipendenza reciproca* tra tutti i DAO è gestita direttamente dai Service (Sezione 3.3).

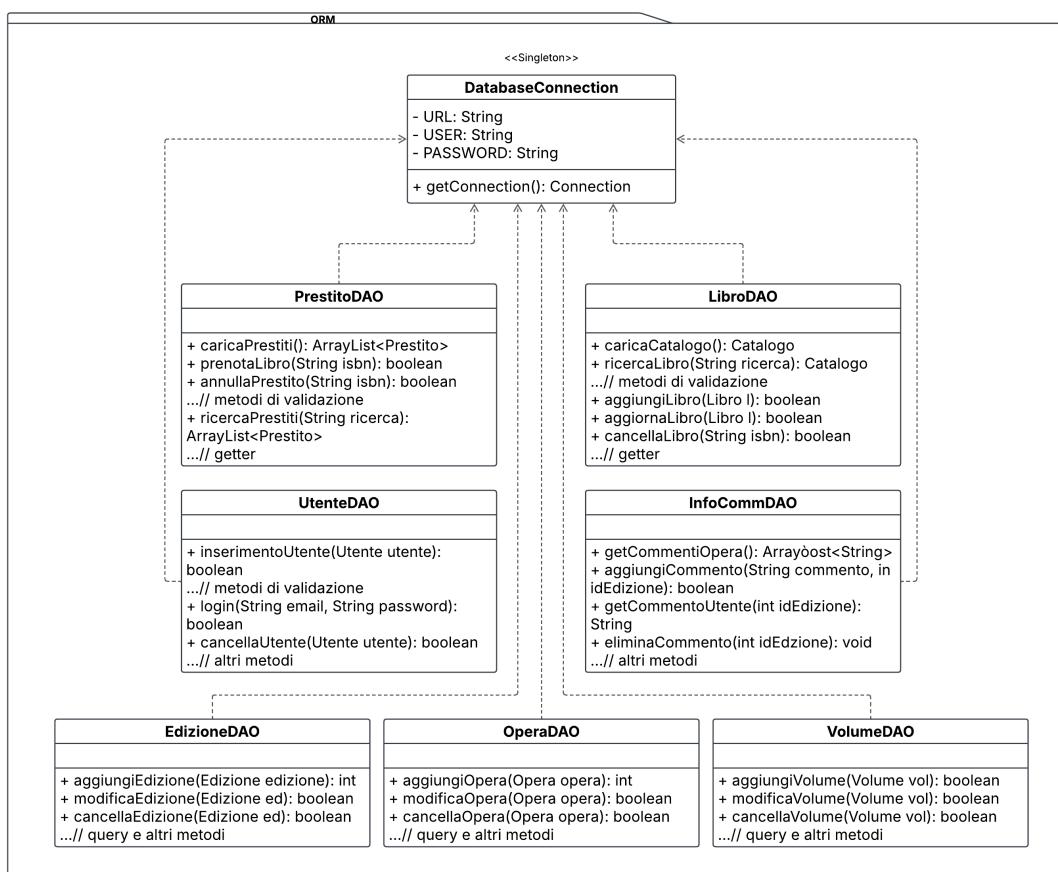


Figura 17: Diagramma UML ORM

3.3 Business Logic

La **Business Logic** è organizzata in classi **Service** che permettono la comunicazione gerarchica tra utente e sistema. I servizi, infatti, ricevono le richieste dall'interfaccia grafica (tramite i **Controller**) e le gestiscono attraverso l'appoggio all'**ORM** per assicurare la persistenza delle modifiche effettuate. I Service gestiscono anche la dipendenza tra DAO.

Le classi **UserService**, **LoanService** e **BookService** servono proprio per gestire i servizi relative alle specifiche entità. Come nei DAO, anche nella Business Logic le proprietà dei vari libri sono state divise in classi separate **VolumeService**, **OperaService** e **EditionService** per maggiore organizzazione di codice.

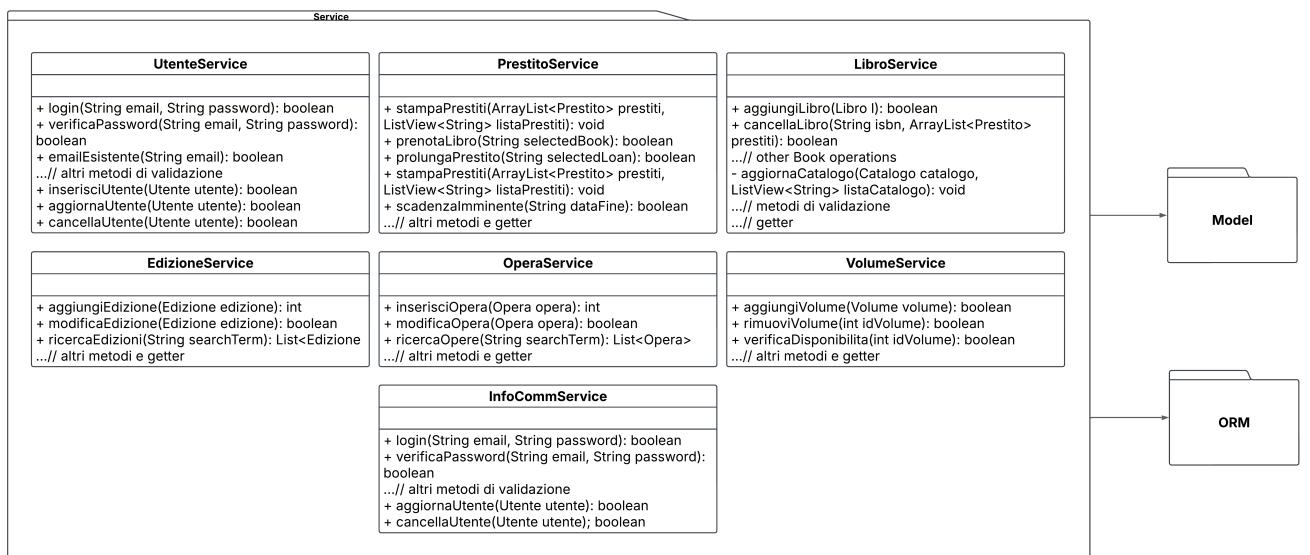


Figura 18: Diagramma UML Business Logic

3.4 Controller

Il **Controller** funge da tramite tra l'interfaccia grafica del **View** e le entità del **Model**. Sono stati realizzati un controller per ogni pagina navigabile dall'utilizzatore. Nello specifico, **HomePageController** contiene tutte le funzioni richiamabili dalla pagina principale dell'interfaccia grafica, alla quale si accede attraverso la registrazione prima e il log-in poi, entrambi gestiti dalle rispettive classi **RegistrationController** e **LoginController**. **OpUserController** è invece il controller che gestisce la home page a disposizione della biblioteca mentre gli altri Controller gestiscono le operazioni eseguite sulle relative pagine.

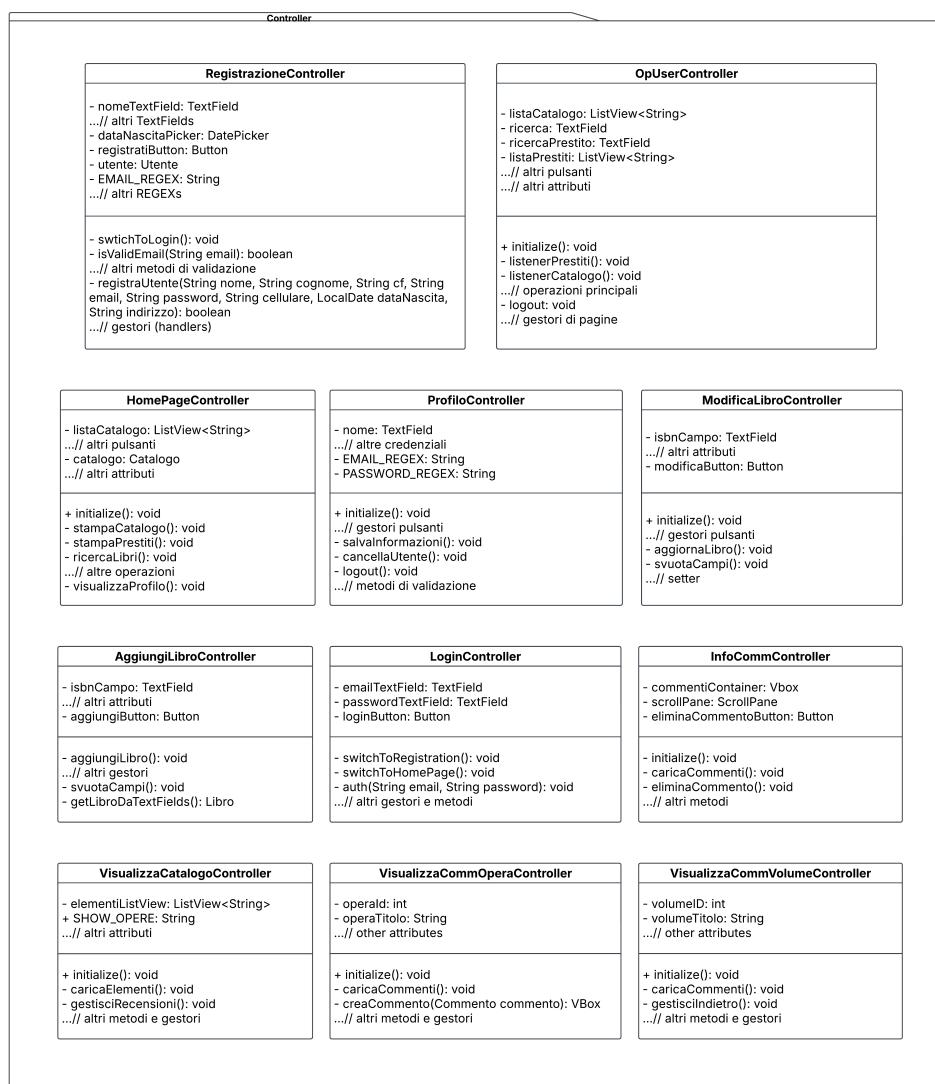


Figura 19: Diagramma UML Controller

3.5 View

La **View** è responsabile della presentazione dei dati all'utente (sia Utente che Biblioteca). Le viste sono state implementate utilizzando *JavaFX*, un framework per la creazione di applicazioni desktop mediante interfacce grafiche in Java, che fornisce componenti per l'interfaccia utente come bottoni, caselle di testo, pannelli etc. Ciò ci permette di creare un'interfaccia grafica intuitiva e interattiva senza dover ricorrere a codici complessi o interazioni macchinose per l'utente. In tutto sono state create 11 View, una per ciascuna pagina del programma.

Qui sotto è mostrato il diagramma delle classi UML del package View:

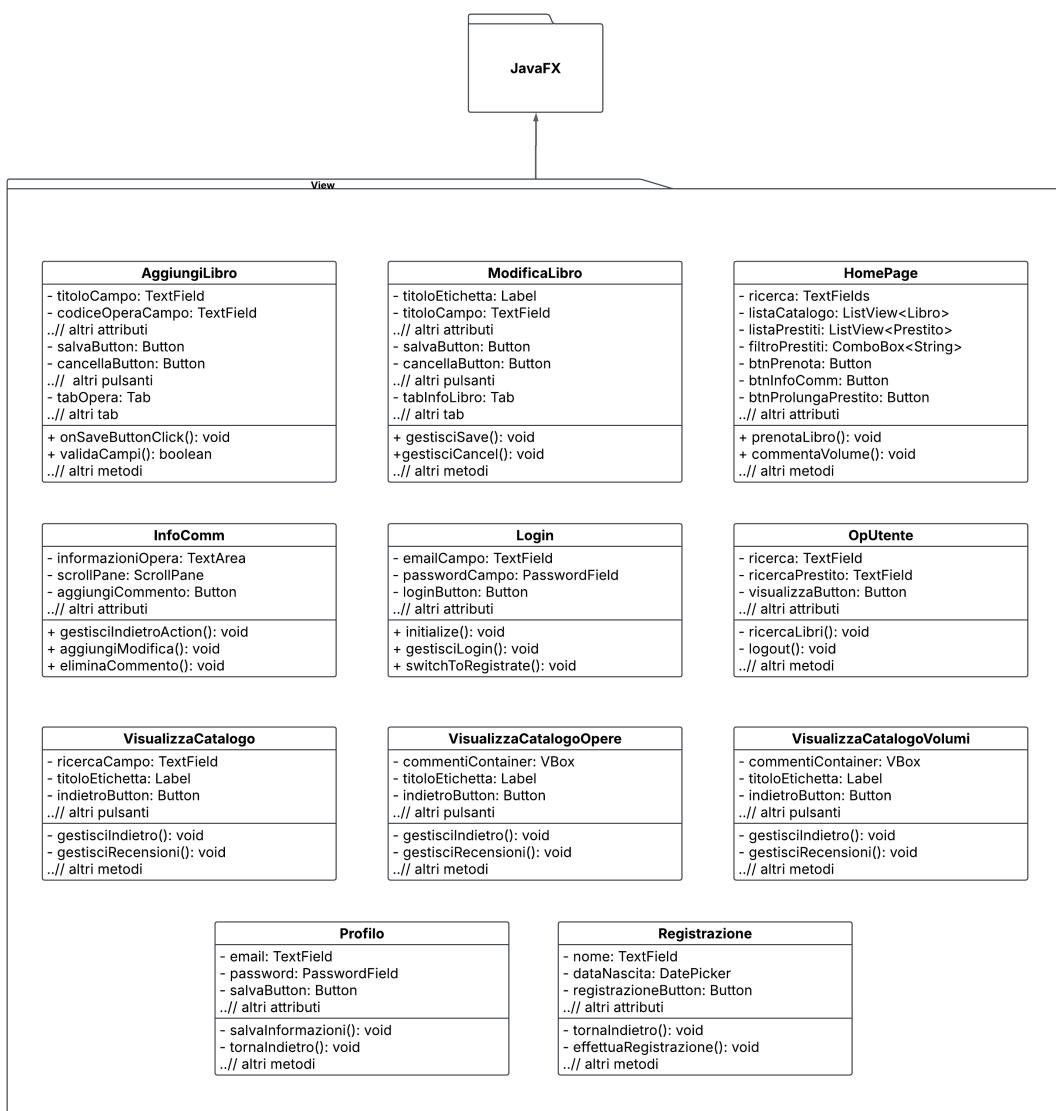


Figura 20: Diagramma UML View

3.6 Codice

Di seguito sono presentate delle porzioni di codice riguardanti le sezioni più importanti del progetto, in modo da mostrare l'implementazione effettiva di ciò che è stato già precedentemente spiegato nella relazione. Verrà mostrata la struttura generale del progetto, l'implementazione di pattern come *Singleton*, la gestione delle interazioni tra i vari package **Controller**, **Business Logic** e **DAO** e alcuni esempi di *testing* automatici e inline sulle funzioni principali.

3.6.1 Organizzazione Cartelle

Il codice del progetto è stato suddiviso, per motivi di facilità organizzativa, in cartelle, ognuna per ogni package, contenenti le relative classi. Ci sono inoltre cartelle aggiuntive come **utilities**, **resources** e **test** contenenti classi utili a gestire separatamente risorse utili come immagini, alert o classi di test.

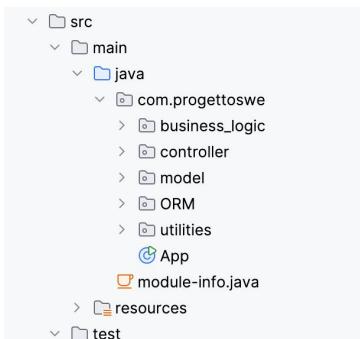


Figura 21: Cartelle del Progetto

3.6.2 Singleton

Di seguito è mostrata l'implementazione pratica del pattern Singleton per gestire la connessione al database (tramite la classe **DatabaseConnection**) in modo da generare una singola istanza di connessione *statica* e unica e non consentire molteplici connessioni diverse che potrebbero generare conflitti e problemi di sincronizzazione dei dati.

```

public class DatabaseConnection {

    private static DatabaseConnection instance;
    private Connection connection;

    private static final String DB_URL = "jdbc:postgresql://localhost:5432/ProvaSWE";
    private static final String USER = "postgres";
    private static final String PASSWORD = "1234";

    // Costruttore privato per impedire l'istanziazione
    private DatabaseConnection() {
        // Inizializzazione lasciata vuota
    }

    // Metodo pubblico per ottenere l'istanza singleton
    public static DatabaseConnection getInstance() {
        if (instance == null) {
            instance = new DatabaseConnection();
        }
        return instance;
    }

    public Connection getConnection() throws SQLException {
        if (connection == null || connection.isClosed()) {
            try {
                connection = DriverManager.getConnection(DB_URL, USER, PASSWORD);
            } catch (SQLException e) {
                throw new SQLException("Errore durante la connessione al database.", e);
            }
        }
        return connection;
    }
}
  
```

Figura 22: Implementazione del Singleton

Come si può vedere nell’immagine sopra, il pattern Singleton è stato implementato mediante l’uso di attributi statici che creano un’istanza della connessione al Database, con costruttore *privato* così da impedire la creazione di altre istanze, e un metodo `getInstance` che recupera la suddetta e ne crea una nuova SE E SOLO SE non ne esiste già una. In questo modo il Singleton impedisce in qualsiasi modo la creazione di ulteriori istanze o connessioni al medesimo database.

3.6.3 Gerarchia delle Responsabilità

Come già spiegato nelle sezioni precedenti, ogni package ha una specifica responsabilità ad esso attribuita, in modo da dividerne i compiti. Ciò implica la necessità di implementare un metodo di comunicazione e di passaggio di informazioni tra ognuno di questi. Nello specifico le classi *Controller* raccolgono le informazioni dall’interfaccia grafica, richiamano i metodi delle classi *Service* della Business Logic, passandogli tali informazioni, che a loro volta richiamano le classi *DAO* che useranno queste informazioni per interagire direttamente con la base dati. In questo modo si viene a creare un sistema di divisione delle responsabilità ordinato e **gerarchico**. Di seguito un esempio di interazioni tra package.

```
private void stampaCatalogo() {
    BookService.stampaCatalogo(catalogo, listaCatalogo);
}
```

Figura 23: Metodo Controller che richiama un metodo Service

```
public static void stampaCatalogo(Catalogo catalogo, ListView<String> listaCatalogo) {
    catalogo = BookDAO.caricaCatalogo();
    aggiornaCatalogo(catalogo, listaCatalogo);
}
```

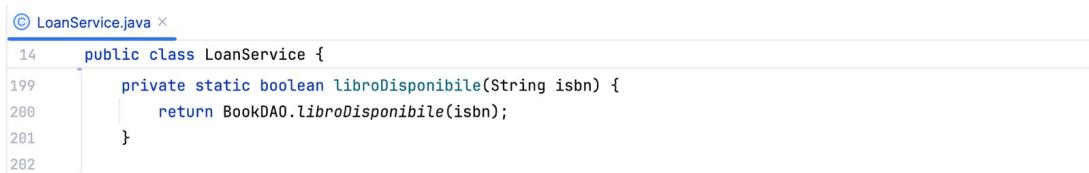
Figura 24: Metodo Service che richiama un metodo DAO

```
public static Catalogo caricaCatalogo() {
    Catalogo catalogo = new Catalogo();
    String query = "SELECT DISTINCT ON (edizione.id_edizione) "
        + "opera.id_opera, "
        + "opera.titolo, "
        + "opera.autore, "
        + "opera.genere, "
        + "edizione.id_edizione, "
        + "edizione.isbn, "
        + "edizione.editore, "
        + "edizione.numero_edizione, "
        + "volume.id_volume, "
        + "volume.stato, "
        + "volume.posizione, "
        + "opera.anno_pubblicazione_originale, "
        + "edizione.anno_pubblicazione, "
        + "opera.descrizione "
        + "FROM opera "
        + "JOIN edizione ON opera.id_opera = edizione.id_opera "
        + "JOIN volume ON edizione.id_edizione = volume.id_edizione";
```

Figura 25: Metodo DAO che interroga il database

3.6.4 Gestione dipendenze tra DAO

Nel progetto, le dipendenze tra i Data Access Object (DAO) vengono gestite a livello di layer service, il quale funge da intermediario tra la logica applicativa e il livello di persistenza. Questo approccio consente di centralizzare la coordinazione tra più DAO all'interno di metodi transazionali, migliorando la coesione del codice e separando le responsabilità tra accesso ai dati e logica di business.



```

⑤ LoanService.java ×
14  public class LoanService {
199     private static boolean libroDisponibile(String isbn) {
200         return BookDAO.libroDisponibile(isbn);
201     }
202

```

Figura 26: Metodo Service del Prestito che richiama un altro DAO



```

public static boolean libroDisponibile(String isbn) {
    String query = "SELECT stato FROM volume WHERE id_edizione = (SELECT id_edizione FROM edizione WHERE isbn = ?);";
    try(Connection connection = DatabaseConnection.getInstance().getConnection()){
        PreparedStatement statement = connection.prepareStatement(query);
        statement.setString( parameterIndex: 1, isbn);
        ResultSet resultSet = statement.executeQuery();
        if(resultSet.next()){
            String stato = resultSet.getString( columnLabel: "stato");
            return stato.equals("disponibile");
        }
    }catch(SQLException e){
        e.printStackTrace();
    }
    return false;
}

```

Figura 27: Metodo del DAO Libro che estrae l'informazione necessaria dal database

Nell'immagine sopra, per esempio, viene mostrato come un service (Prestito) richiami un altro DAO (Libro), diverso da quello di cui si occupa, per recuperare delle informazioni utili allo svolgimento delle operazioni sul proprio livello di persistenza, stabilendo difatto una dipendenza tra il DAO di Prestito e quello di Libro.

3.6.5 Gestione tabelle Commento

Per migliorare la correttezza della gestione dei commenti nel database, è stata creata una struttura ereditaria padre-figlio in cui il commento creato dall'utente può essere istanziato come uno dei due figli (in base al tipo di commento effettuato) ottenendo così le opportune foreign keys necessarie per far riferimento al Volume o all'Edizione interessata, pur ereditando dal padre le informazioni comuni come id utente, testo e id commento. In questo modo si evitano problemi di ambiguità tra tuple e presenze di molteplici valori nulli.

```

CREATE TABLE Commento (
    id_commento SERIAL PRIMARY KEY,
    id_utente INTEGER NOT NULL,
    testo TEXT NOT NULL,
    FOREIGN KEY (id_utente) REFERENCES Utente(id_utente) ON DELETE CASCADE,
);

CREATE TABLE Commento_edizione (
    id_commento SERIAL PRIMARY KEY REFERENCES Commento(id_commento),
    id_edizione INTEGER NOT NULL,
    FOREIGN KEY (id_edizione) REFERENCES Edizione(id_edizione) ON DELETE CASCADE
);

CREATE TABLE Commento_volume (
    id_commento SERIAL PRIMARY KEY REFERENCES Commento(id_commento),
    id_prestito INTEGER NOT NULL,
    FOREIGN KEY (id_prestito) REFERENCES Prestito(id_prestito) ON DELETE CASCADE
);
  
```

Figura 28: Struttura tabelle commento nel Database

3.7 Test e Gestione sicurezza

Anteprima dei Test La tabella seguente riepiloga i test automatici riportati nel dettaglio nelle sezioni successive. I test coprono sia sia la logica applicativa con **JUnit** (sez. 3.7.1), sia l’interfaccia utente tramite **TestFX** (sez. 3.7.2). Pur non rappresentando la totalità delle classi e dei metodi testati, questi esempi evidenziano i principali casi d’uso, inclusi gli scenari di errore e i comportamenti limite.

Tipo Test	Obiettivo	Metodo testato	Esito atteso
Unit (JUnit)	Prenotazione e annullamento prestito	creaPrestito() / annullaPrestito()	Stato coerente nel database (volume aggiornato)
Unit (JUnit)	Recupero ISBN tramite titolo/autore/edizione	ottieniIsbn()	Valore null o ISBN corretto
Unit (JUnit)	Verifica disponibilità opera	operaDisponibile()	true se disponibili, false altrimenti
UI (TestFX)	Login con credenziali errate	login()	Messaggio di errore
UI (TestFX)	Prenotazione libro dal catalogo	prenota()	Messaggio di successo
UI (TestFX)	Prolungamento di un prestito	prorogaPrestito()	Messaggio di conferma
UI (TestFX)	Validazione email in fase di registrazione	registrazione()	Messaggio di errore

Tabella 1: Riepilogo dei principali test

3.7.1 Test Unit

I test di correttezza sulle **query** sono stati eseguiti tramite **JUnit**, un framework di testing per Java utile per eseguire test automatici su unità di codice, come classi e metodi. I test su cui ci siamo concentrati riguardano le interrogazioni al database. Di seguito degli esempi.

Test 1 - Creazione e Annullamento Prestito Questo test verifica il corretto funzionamento dell'intero ciclo di vita di un prestito: dalla prenotazione alla sua successiva annullazione. Il test si assicura che la prenotazione e il suo successivo annullamento avvengano correttamente con un'asserzione positiva (tramite *assertTrue*). Questo test garantisce quindi l'affidabilità e la reversibilità delle operazioni fondamentali di prestito e restituzione all'interno del sistema.

```
@Test
void testPrenotaEAnnullaPrestito() {
    boolean prenotazioneSuccesso = LoanDAO.prenotaLibro(isbnDisponibile);
    assertTrue(prenotazioneSuccesso, message: "La prenotazione del libro dovrebbe andare a buon fine");

    boolean annullamentoSuccesso = LoanDAO.annullaPrestito(isbnDisponibile);
    assertTrue(annullamentoSuccesso, message: "L'annullamento del prestito dovrebbe andare a buon fine");
}
```

Figura 29: Test JUnit per la prenotazione di un libro e l'annullamento del prestito

Questo insieme di query sotto riportate viene utilizzato per gestire l'operazione di prenotazione di un libro da parte di un utente. La prima query seleziona il primo volume disponibile (stato = 'disponibile') associato all'edizione del libro corrispondente all'ISBN fornito, ordinando i risultati in modo crescente per garantire una logica FIFO (First In, First Out). La seconda query aggiorna lo stato del volume selezionato, impostandolo su 'in prestito' per indicare che il volume non è più disponibile per altre prenotazioni. Infine, la terza query registra il prestito nella tabella prestito, associando l'identificativo del volume a quello dell'utente che ha effettuato la prenotazione. Questo processo garantisce la tracciabilità dei prestiti e l'integrità dello stato del catalogo.

```
public static boolean prenotaLibro(String isbn) {
    //otteniamo l'id_volume disponibile
    String getIdVolumeQuery = "SELECT id_volume " +
        "FROM volume " +
        "WHERE id_edizione = (SELECT id_edizione FROM edizione WHERE isbn = ?)" +
        "AND stato = 'disponibile' " +
        "ORDER BY id_volume ASC LIMIT 1"; // Ottieni il primo volume disponibile

    String updateQuery = "UPDATE volume " +
        "SET stato = 'in prestito' " +
        "WHERE id_volume = ?; // Aggiorna lo stato del volume

    String insertPrestitoQuery = "INSERT INTO prestito (id_volume, id_utente) " +
        "VALUES (?, ?); // Inserisce il prestito
```

Figura 30: Query per la prenotazione di un libro

Le due query sotto illustrate, invece, sono impiegate nel processo di annullamento di un prestito da parte di un utente. La prima query (DELETE ... USING) permette di eliminare un record dalla tabella prestito, effettuando un join con le tabelle volume ed edizione per risalire al volume corretto tramite il codice ISBN e l'identificativo dell'utente. Grazie all'uso della clausola RETURNING, viene immediatamente recuperato l'identificativo del volume associato al prestito eliminato. La seconda query aggiorna poi lo stato del volume corrispondente, impostandolo nuovamente su 'disponibile'. Questo meccanismo garantisce la coerenza del sistema, assicurando che un volume torni prenotabile non appena un prestito viene revocato.

```

public static boolean annullaPrestito(String isbn){
    String deleteQuery = "DELETE FROM prestito " +
        "USING volume, edizione " +
        "WHERE prestito.id_volume = volume.id_volume " +
        "AND volume.id_edizione = edizione.id_edizione " +
        "AND edizione.isbn = ? " +
        "AND prestito.id_utente = ? " +
        "RETURNING prestito.id_volume";

    String updateQuery = "UPDATE volume " +
        "SET stato = 'disponibile' " +
        "WHERE id_volume = ?";
  
```

Figura 31: Query per l'annullamento di un prestito

Test 2 - Ricerca ISBN Questo test verifica il comportamento del metodo ottieniIsbn quando viene eseguita una ricerca con dati non presenti nel database (titolo, autore e numero edizione inesistenti). Il test è importante perché assicura che il metodo gestisca correttamente i casi in cui non esiste alcuna corrispondenza, restituendo null anziché sollevare eccezioni o restituire risultati errati. In questo modo si validano la robustezza e l'affidabilità della logica di interrogazione.

```

@Test
public void testOttieniIsbnEsistente() {
    String titolo = "Il Signore degli Anelli";
    String autore = "J.R.R. Tolkien";
    int numeroEdizione = 1;

    String isbn = BookDAO.ottieniIsbn(titolo, autore, numeroEdizione);

    assertNotNull(isbn, message: "L'ISBN non dovrebbe essere nullo per un libro esistente.");
    assertEquals( expected: "9788845292610", isbn);
}
  
```

Figura 32: Test JUnit per il recupero dell'isbn corretto

Il metodo recupera l'ISBN di una specifica edizione di un'opera, utilizzando una JOIN tra le tabelle opera ed edizione. I parametri di ricerca sono titolo, autore e numero edizione. Se la corrispondenza viene trovata, restituisce l'ISBN; altrimenti null. È utile per identificare univocamente un'edizione ai fini di prestiti o prenotazioni.

```

public static String ottieniIsbn (String nome, String autore, int num_edizione) {
    String query = "SELECT isbn FROM opera JOIN edizione ON opera.id_opera = edizione.id_opero" +
        "+ WHERE opera.titolo = ? AND opera.autore = ? AND numero_edizione = ?;";
  
```

Figura 33: Query per il recupero di un isbn

Test 3 - Verifica Disponibilità Questa suite di test copre in modo esaustivo il comportamento del metodo operaDisponibile della classe BookDAO. Viene verificato che il metodo restituisca true quando esistono effettivamente copie disponibili per un determinato ISBN, false quando tutte le copie sono esaurite ma l'edizione è presente nel catalogo, e ancora false nel caso in cui l'ISBN fornito non corrisponda a nessuna edizione registrata. In questo modo si assicura che la logica di disponibilità funzioni correttamente in tutti i casi d'uso più comuni e in quelli limite.

```

  @Test
  public void testOperaDisponibileConCopieDisponibili() {

    String isbnDisponibile = "9788845292610";

    boolean disponibile = BookDAO.operaDisponibile(isbnDisponibile);

    assertTrue(disponibile, message: "Dovrebbero esserci copie disponibili per questo ISBN.");
  }

  @Test
  public void testOperaDisponibileSenzaCopieDisponibili() {

    String isbnNonDisponibile = "9788806229993";

    boolean disponibile = BookDAO.operaDisponibile(isbnNonDisponibile);

    assertFalse(disponibile, message: "Non dovrebbero esserci copie disponibili per questo ISBN.");
  }

  @Test
  public void testOperaDisponibileIsbnInesistente() {

    String isbnInesistente = "00000000000000";

    boolean disponibile = BookDAO.operaDisponibile(isbnInesistente);

    assertFalse(disponibile, message: "Non dovrebbero esserci copie per un ISBN inesistente.");
  }
}

```

Figura 34: Test JUnit per la verifica della disponibilità di un'opera

La seguente query ha lo scopo di determinare la disponibilità di un'opera all'interno del sistema bibliotecario, partendo dall'ISBN fornito. Attraverso una sottoquery, viene ricavato l'id edizione corrispondente all'ISBN, che viene poi utilizzato per contare, nella tabella volume, quante copie risultano in stato 'disponibile'. Il risultato della query consente quindi di stabilire se esistono copie attualmente disponibili per il prestito, rappresentando un passaggio fondamentale per abilitare o meno l'operazione di prenotazione.

```

public static boolean operaDisponibile(String isbn) {
  String query = "SELECT COUNT(*) FROM volume WHERE id_edizione = (SELECT id_edizione " +
    "FROM edizione WHERE isbn = ?) AND stato = 'disponibile';";
  try (Connection connection = DatabaseConnection.getInstance().getConnection()) {
    PreparedStatement statement = connection.prepareStatement(query);
    statement.setString( parameterIndex: 1, isbn);
    ResultSet resultSet = statement.executeQuery();

    if (resultSet.next()) {
      int copieDisponibili = resultSet.getInt( columnIndex: 1);
      return copieDisponibili > 0;
    }
  } catch (SQLException e) {
    e.printStackTrace();
  }
  return false;
}

```

Figura 35: Query per la verifica della disponibilità di un'opera

3.7.2 Test UI

I test di correttezza automatici sulla **grafica** sono stati eseguiti tramite **TestFX**, una libreria Java utile per interfacciarsi proprio con le applicazioni realizzate tramite *JavaFX* e testarne in maniera **automatica** il funzionamento, simulando il comportamento dell'utente secondo i dati forniti alle classi. Risulta molto simile a *Selenium* come funzionamento, ma è adattato specificatamente per le applicazioni JavaFX desktop. Di seguito degli esempi di testing.

Test 1 - Login errato Questo test verifica il corretto comportamento dell'interfaccia grafica in caso di credenziali errate durante il login. Simula l'inserimento di un'email e una password non valide nei rispettivi campi, clicca sul pulsante di login e attende il completamento degli eventi JavaFX asincroni. Infine, controlla che venga visualizzato un messaggio di errore (“Errore di accesso”), confermando che il sistema gestisce correttamente i tentativi di accesso falliti.

```

  @Test
  void testLoginFail() {
    clickOn(emailTextField).write( s: "wronguser@example.com");
    clickOn(passwordTextField).write( s: "wrongpassword");

    clickOn( query: "#loginButton");

    WaitForAsyncUtils.waitForFxEvents();

    verifyThat( nodeQuery: "Errore di accesso", hasText("Errore di accesso"));
  }
}
  
```

Figura 36: TestFX per il Login

Test 2 - Prenotazione libro Questo test automatizza il flusso di prenotazione di un libro tramite l'interfaccia grafica. Simula la selezione di un libro disponibile dalla lista del catalogo, l'interazione con il pulsante “Prenota”, e la conferma dell'azione tramite un dialog di conferma (“Sì”). Al termine, verifica che venga mostrato il messaggio di successo, confermando che il processo di prenotazione è stato completato correttamente lato UI.

```

  @Test
  void testPrenotaLibro() {
    // Simula la selezione di un libro
    clickOn(listaCatalogo).clickOn( query: "1984 - 1 edizione - Mondadori - George Orwell - Disponibile");

    // Simula la prenotazione di un libro
    clickOn(btnPrenota);

    clickOn( query: "Sì");

    WaitForAsyncUtils.waitForFxEvents();

    // Verifica che il libro sia stato prenotato con successo
    verifyThat( nodeQuery: "Libro prenotato con successo.", hasText("Libro prenotato con successo."));
  }
  
```

Figura 37: TestFX per la prenotazione di un libro

Test 3 - Prolungamento prestito Questo test simula il processo di prolungamento di un prestito tramite l'interfaccia grafica JavaFX. Dopo aver applicato i filtri sui prestiti e selezionato un elemento specifico della lista (identificato dal titolo dell'opera), esegue un clic sul pulsante per la proroga e conferma l'azione. Infine, verifica che l'operazione sia andata a buon fine controllando la presenza del messaggio “Successo”, confermando così il corretto funzionamento dell'interazione lato utente.

```

@Test
void testProlungaPrestito() {

    clickOn(filtroPrestiti);
    clickOn(query: "Tutti i prestiti");

    clickOn(filtroPrestiti);
    clickOn(query: "Prestiti attivi");

    ListView<?> lista = lookup(query: "#listaPrestiti").queryAs(ListView.class);
    String targetText = "1984 - 1 edizione - Mondadori - George Orwell";

    for (Node cell : lista.lookupAll(selector: ".list-cell")) {
        if (cell instanceof Labeled labeled) {
            String cellText = labeled.getText();
            if (cellText != null & cellText.contains(targetText)) {
                robot.clickOn(cell);
                break;
            }
        }
    }

    clickOn(btnProlungaPrestito);

    clickOn(query: "Sì");

    WaitForAsyncUtils.waitForFxEvents();

    // Verifica che il prestito sia stato prolungato
    verifyThat(nodeQuery: "Successo", hasText("Successo"));
}
  
```

Figura 38: TestFX per il prolungamento di un prestito

Test 4 - Email non valida Questo test simula la compilazione del form di registrazione con un indirizzo email non valido. Alla pressione del pulsante "Registrati", il sistema deve intercettare l'errore (tramite le eccezioni, vedi sez. 3.8.1) e mostrare un messaggio di dialogo con l'avviso "Email non valida". Il test verifica che tale messaggio venga effettivamente visualizzato, validando così il corretto comportamento della logica di validazione lato interfaccia utente.

```

@Test
public void testEmailNonValidaMostraErrore() throws InterruptedException {

    WaitForAsyncUtils.waitForFxEvents();

    clickOn(query: "#nomeTextField").write(s: "Mario");
    clickOn(query: "#cognomeTextField").write(s: "Rossi");
    clickOn(query: "#codiceFiscaleTextField").write(s: "RSSMRA80A01H501U");
    clickOn(query: "#emailTextField").write(s: "email-nonvalida");
    clickOn(query: "#passwordTextField").write(s: "P@ssw0rd");
    clickOn(query: "#cellulareTextField").write(s: "1234567890");
    clickOn(query: "#dataNascitaPicker").write(s: "01/01/2000").type(KeyCode.ENTER);
    clickOn(query: "#indirizzoTextField").write(s: "Via Roma 1");

    clickOn(query: "#registerButton");

    FxAssert.verifyThat(nodeQuery: ".dialog-pane .content", LabeledMatchers.hasText("Email non valida\n" +
    "\n" +
    "L'email deve essere valida e non deve contenere spazi (es. esempio@prova.com)."));
    clickOn(query: "OK");
}
  
```

Figura 39: TestFX per la verifica dei campi in Registrazione

3.8 Gestione Errori e Sicurezza

In questa sezione verranno analizzati i metodi di gestione di tutti i possibili errori sia lato Utente che Sistema, e tutta la gestione riguardante l'accesso al servizio mediante login, il salvataggio temporaneo dei dati e il relativo accesso concorrente da parte di più utenti contemporaneamente.

3.8.1 Gestione degli Errori

Gli errori che possono occorrere si dividono in **User Errors** e **System Errors**. I primi sono dovuti ad errori di inserimento o ad operazioni non consentite da parte dell'utente, mentre la seconda tipologia riguarda tutti quegli errori che possono occorrere indipendentemente dall'azione dell'utente, come per esempio errori di connessione al database o errori di inizializzazione delle variabili.

User Errors Gli errori Utente riguardano tutti quelli errori che dipendono direttamente da un'azione errata o non consentita eseguita dall'utente. Queste operazioni possono essere per esempio:

Accesso negato L'Utente o la Biblioteca, in fase di `login` inseriscono dei campi (email o password) non presenti o non accoppiati nel database, e quindi non associati a nessun utente precedentemente creato mediante `registrazione`. Il sistema quindi, richiama la query per il login sull'UserDAO...

```

public static boolean login(String email, String password) {
    String query = "SELECT count(*) FROM utente WHERE email = ? AND pw = ?";

    try (Connection conn = DatabaseConnection.getInstance().getConnection()) {
        PreparedStatement pstmt = conn.prepareStatement(query);
        pstmt.setString( parameterIndex: 1, email);
        pstmt.setString( parameterIndex: 2, password);

        try (ResultSet rs = pstmt.executeQuery()) {
            if (rs.next() && rs.getInt( columnIndex: 1) > 0) {
                return true;
            } else {
                return false;
            }
        }
    } catch (SQLException e) {
        e.printStackTrace();
        return false;
    }
}
  
```

Figura 40: Query per il Login nell'UserDAO

Una volta riscontrato esito negativo sulla query qui sopra, il metodo lancia un'`eccezione` che verrà raccolta dal `Service` e passata al `Controller` che, una volta ricevuta, darà esito negativo al suo di metodo di autenticazione e, attraverso ulteriori metodi di validazione (`checkPassword` presente in `UserService`) verifica se risulta errata solo la password o anche l'email...

```

public class LoginController {

    private void authenticate(String email, String password) {
        email = email.trim();

        if (!InputValidator.validateNotEmpty(emailTextField, fieldName: "Email")
            || !InputValidator.validateNotEmpty(passwordTextField, fieldName: "Password")) {
            return;
        }

        if(UserService.login(email, password)) {
            try {
                loginByEmailType(email);
                return;
            } catch (IOException e) {
                e.printStackTrace();
            }
        } else if(UserService.checkPassword(email, password)) {
            Alert a = new Alert(AlertType.ERROR, contentText: "La password inserita è errata");
            a.setHeaderText("Errore di accesso");
            a.setTitle("Errore durante l'accesso");
            a.showAndWait();
            passwordTextField.clear();
        } else{
            Alert a = new Alert(AlertType.ERROR, contentText: "L'email e la password non sono corretti\n\nSe non sei ancora registrato, fallo ora!");
            a.setHeaderText("Errore di accesso");
            a.setTitle("Errore durante l'accesso");
            a.showAndWait();
            emailTextField.clear();
            passwordTextField.clear();
        }
    }
}
  
```

Figura 41: Verifica esito Query da parte del Controller

Infine, il metodo, una volta verificata la natura dell'errore, fa apparire di conseguenza un opportuno alert all'utente per informarlo del problema e per consigliargli una soluzione possibile (suggerisce la creazione di un account se non ancora fatto).



Figura 42: Login errato alert

Tutta la procedura appena descritta è comunque testata nel **Test 1 - Login errato** sopra riportato.

Inserimento non valido A seconda dei **Metodi di Validazione** (sez. 3.8.2), i campi inseriti dall’Utente in fase di creazione o modifica del profilo (o di inserimento di libri nel caso della Biblioteca) vengono validati dal sistema tramite dei metodi che, usando delle espressioni regolari (REGEX), verificano se tali campi rispettano le specifiche di sintassi richieste. Nello specifico, per la registrazione per esempio, ogni campo richiesto (vedi Figura 7) ha delle espressioni regolari ben precise da dover rispettare; nel caso di mancato rispetto di tali regole, il sistema riceve l’eccezione relativa e verifica su quale campo si è verificato l’errore, mostrando all’Utente il relativo alert per guidarlo alla risoluzione del problema. Per esempio nel caso di email non valida inserita appare il seguente.



Figura 43: Campo non valido alert

Anche questa operazione è stata comunque testata nel **Test 4 - Email non valida** sopra.

Operazioni non consentite Possono occorrere qualora l’Utente stia provando ad effettuare delle operazioni a lui non consentite. Tra tutte le più comuni sono per esempio la prenotazione di un libro non disponibile, il rinnovo di un prestito oltre il limite massimo o la richiesta di un prestito ulteriore dopo aver già superato i 3 massimi effettuabili contemporaneamente. Quest’ultimo viene preso in analisi di seguito:

Il DAO, una volta effettuata la richiesta di prestito tramite l’UI, effettuerà l’opportuna query e, attraverso un sistema di `try` e `catch` verifica se tale operazione è consentita, effettuando le diverse verifiche del caso.

```

public static boolean prenotaLibro(String isbn) {
    try (Connection connection = DatabaseConnection.getInstance().getConnection()) {
        // 1. Ottenerne l'id_volume disponibile
        PreparedStatement getIdVolumeStatement = connection.prepareStatement(getIdVolumeQuery);
        getIdVolumeStatement.setString(parameterIndex: 1, isbn); // Imposta l'ISBN

        ResultSet getIdVolumeResultSet = getIdVolumeStatement.executeQuery();
        if (getIdVolumeResultSet.next()) {
            int idVolume = getIdVolumeResultSet.getInt(columnLabel: "id_volume"); // Ottieni l'ID del volume disponibili

            // 2. Aggiorna lo stato del volume selezionato come "in prestito"
            PreparedStatement updateStatement = connection.prepareStatement(updateQuery);
            updateStatement.setInt(parameterIndex: 1, idVolume); // Imposta l'ID del volume

            int rowsUpdated = updateStatement.executeUpdate(); // Esegui l'UPDATE
            if (rowsUpdated > 0) {
                // 3. Inserisci il prestito nel database
                PreparedStatement insertStatement = connection.prepareStatement(insertPrestitoQuery);
                insertStatement.setInt(parameterIndex: 1, idVolume); // Imposta l'ID del volume selezionato
                insertStatement.setInt(parameterIndex: 2, Session.getUtente().getId_utente()); // Imposta l'ID dell'utente

                int rowsInserted = insertStatement.executeUpdate();
                if (rowsInserted > 0) {
                    // Se l'inserimento del prestito è andato a buon fine, ritorna true
                    return true;
                }
            }
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
    return false; // Ritorna false in caso di errore
}
  
```

Figura 44: Verifica del prestito sul DAO

Una volta fatto ciò l'azione passerà al rispettivo **Controller** che gestirà il risultato ritornato di conseguenza: se tutto è andato a buon fine, il controller aggiorna sia il catalogo libri che la lista prestiti dell'utente in tempo reale sull'interfaccia grafica; se invece viene restituita un'eccezione, il controller non eseguirà nessuna di queste operazioni ma mostrerà invece all'Utente un alert di errore.

```

private void prenotaLibro(){
    confirmAlert.showAndWait().ifPresent(ButtonType.type -> {
        if (type == buttonTypeYes) {
            if (LoanService.prenotaLibro(selectedBook)) {
                listaCatalogo.getItems().clear();
                listaPrestiti.getItems().clear();
                //modificare lo stato del volume in prestito

                BookService.stampaCatalogo(catalogo, listaCatalogo);
                /stampaprestiti();
                filtraPrestiti();
                Alert successAlert = new Alert(Alert.AlertType.INFORMATION);
                successAlert.setContentText("Libro prenotato con successo.");
                successAlert.setHeaderText("Successo");
                successAlert.setTitle("Prenotazione avvenuta con successo");
                successAlert.showAndWait();
            } else {
                Alert errorAlert = new Alert(Alert.AlertType.ERROR);
                errorAlert.setContentText("Il libro non è stato selezionato correttamente.\n\n" +
                    "Il libro non è disponibile.\n\n" +
                    "Hai già un prestito attivo per questo libro.\n\n" +
                    "Il libro è appena stato prenotato da un altro utente.\n\n" +
                    "Hai raggiunto il numero massimo di prestiti.");
                errorAlert.setHeaderText("Errore");
                errorAlert.setTitle("Errore durante la prenotazione del libro");
                errorAlert.showAndWait();
            }
        }
    });
}
  
```

Figura 45: Gestione dell'eccezione sul prestito non valido

Il successivo alert mostrato indicherà all'Utente tutte le possibili motivazioni per le quali l'operazione da lui effettuata non è stata accettata, così che questo possa trovare il modo di riprovare senza errori.

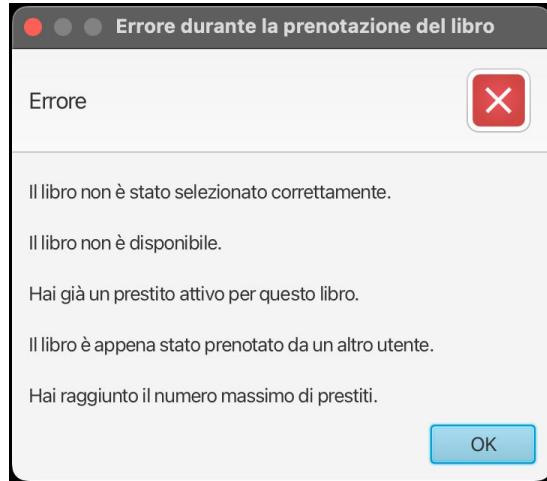


Figura 46: Prestito non consentito alert

System Errors Gli errori del Sistema riguardano tutte quelle problematiche che non dipendono direttamente da un'azione esplicita dell'Utente ma sono invece generati da errori, crash o bug di processi, codice o database. Questi possono essere per esempio variabili non correttamente inizializzate, errori di concorrenza sui dati in diverse sessioni, errori sulla gestione delle eccezioni, errori di recupero dati dal database o direttamente di connessione ad esso.

Nello specifico di seguito viene mostrata la gestione di questi ultimi due scenari.

```

public static void stampaCatalogo(Catalogo catalogo, ListView<String> listaCatalogo) {
    try {
        listaCatalogo.getItems().clear(); // Pulisce la lista prima del caricamento
        catalogo = BookDAO.caricaCatalogo();
        if (catalogo == null || catalogo.getVolumi().isEmpty()) {
            AlertUtil.showErrorAlert(
                title: "Errore di Caricamento",
                header: "Catalogo vuoto",
                content: "Non è stato possibile recuperare i dati del catalogo. Il database potrebbe essere vuoto."
            );
            return;
        }
        aggiornaCatalogo(catalogo, listaCatalogo);
    } catch (Exception e) {
        AlertUtil.showErrorAlert(
            title: "Errore di Sistema",
            header: "Impossibile caricare il catalogo",
            content: "Si è verificato un errore durante il caricamento dei dati: " + e.getMessage()
        );
        listaCatalogo.getItems().add("Errore nel caricamento del catalogo");
    }
}
  
```

Figura 47: Errore caricamento dati

Viene qui sopra mostrata una classica gestione dell'errore nel caso in cui una query di lettura nel database fallisca e finisca in eccezione. Qualora l'oggetto `catalogo` (in cui viene salvato il risultato della query dal `BookDAO`) sia vuoto del tutto o non contenga nessun Volume, il sistema mostra un alert di errore che notifica l'Utente del temporaneo errore; quindi riprova nuovamente a caricare il catalogo richiamando il metodo `aggiornaCatalogo`. Se invece il problema fosse nel flusso del programma e pure questo metodo dovesse finire in eccezione, un errore più generico viene mostrato.

```

public Connection getConnection() throws SQLException {
    if (connection == null || connection.isClosed()) {
        try {
            connection = DriverManager.getConnection(DB_URL, USER, PASSWORD);
        } catch (SQLException e) {
            throw new SQLException("Errore durante la connessione al database.", e);
        }
    }
    return connection;
}
  
```

Figura 48: Errore di connessione al Database

Qui invece il caso in cui avvenga un'errore di connessione con il Database. Come mostrato già nell'analisi del `Singleton` (vedi Figura 22), la classe `DatabaseConnection` è in grado di intercettare e gestire eventuali errori di connessione mostrando anche un apposito errore `SQLException`.

3.8.2 Metodi di Validazione dei campi

Come già accennato nella sezione precedente, all'interno del programma sono state implementate una serie di verifiche su tutte le operazioni di inserimento o modifica di dati nel Database, imponendo che ogni campo che venga aggiunto o cambiato rispetti delle precise regole stabilite in base alla tabella di appartenenza. In questo modo si garantisce ordine e correttezza del database, evitando inconsistenza e presenza di dati dissonanti dagli altri.

Questi controlli riguardano:

- **Registrazione** di un nuovo Utente o modifica di uno esistente. La verifica riguarda ogni campo della tabella `Utente` con diverse regole per ognuno. Nello specifico campi come *CF*, *e-mail*, *telefono* devono rispettare i normali standard di lunghezza e caratteri; il campo *Password* deve contenere almeno 1 carattere maiuscolo, 1 numero e 1 carattere speciale, oltre ad essere ristretto ad un range di lunghezza; la *data di nascita*, invece, è limitata a un margine inferiore (non si possono registrare utenti minori di 10 anni). Eventuali errori su questi campi vengono gestiti come nel caso di **Inserimento non valido**.
- **Inserimento o modifica di un Libro** da parte della Biblioteca. Ovviamente ogni libro deve rispettare delle specifiche comuni per tutti
- **Richiesta di Prestito** da parte di un Utente. Come già analizzato nel caso di **Operazioni non consentite**, la richiesta di prestito per un determinato libro può comportare diversi errori qualora questa sfiori i limiti consentiti di disponibilità o permessi. Ci saranno quindi dei metodi di validazione utili a verificare ciò come già spiegato.

Di seguito vengono mostrate le espressioni regolari (REGEX) e i metodi di validazione usati per verificare la correttezza nell'operazione di **Registrazione**.

```
//Espressioni di validazione dei campi
private static final String EMAIL_REGEX = "^[A-Za-z0-9+_.-]+@[A-Za-z0-9.-]+\.\.[A-Za-z]{2,6}$";
private static final String CF_REGEX = "^[A-Z]{6}\d{2}[A-Z]\d{2}[A-Z]\d{3}[A-Z]$";
private static final String PASSWORD_REGEX = "(?=.*\d)(?=.*[!@#$%^&]).{5,}$";
private static final String PHONE_REGEX = "\+\d{8,15}$";
```

Figura 49: Regex che definiscono i vincoli per i campi Utente

```
//Metodi di validazione
private boolean isValidEmail(String email){
    return Pattern.matches(EMAIL_REGEX, email);
}
private boolean isValidCf(String cf){
    return Pattern.matches(CF_REGEX, cf);
}
private boolean isValidPassword(String password){
    return Pattern.matches(PASSWORD_REGEX, password);
}
private boolean isValidPhone(String phone){
    return Pattern.matches(PHONE_REGEX, phone);
}
private boolean isValidDate(LocalDate dataNascita) {
    LocalDate today = LocalDate.now(); // Data attuale
    LocalDate limiteMinimo = today.minusYears( yearsToSubtract: 100 );
    LocalDate limiteMassimo = today.minusYears( yearsToSubtract: 10 );

    return !dataNascita.isAfter(limiteMassimo) && !dataNascita.isBefore(limiteMinimo);
}
```

Figura 50: Metodi di verifica per i vincoli sui campi

3.8.3 Accesso concorrente

Per gestire l'accesso concorrente alla piattaforma da parte di più Utenti contemporaneamente viene utilizzata la classe **Session**. Tale classe, al momento del login, tiene salvati i dati dell'Utente (solo i dati di riferimento, non le credenziali, per motivi di privacy) in modo da avere fissata una referenza all'Utente che sta utilizzando in quel momento l'applicazione, così da sapere a quale tupla della tabella Utente attribuire eventuali operazioni effettuate come per esempio la richiesta di Prestiti o l'inserimento di Commenti.

La classe Session è instanziabile più volte, così da consentire e l'accesso a più Utenti in contemporanea, ognuno con una propria sessione attribuita. Ciò consente anche di poter aggiornare in tempo reale il sistema e quindi l'interfaccia grafica (per esempio il Catalogo libri) così da mostrare subito agli utilizzatori le modifiche apportate dagli altri Utenti e poter agire di conseguenza. Di seguito viene mostrata la classe appena descritta.

```
@FXML
private void handleLogin() {
    String mail = emailTextField.getText();
    String password = passwordTextField.getText();
    Session.setUserEmail(mail);
    authenticate(mail, password);
}
```

Figura 51: Inizializzazione Session via login

Quando l'Utente effettua il **Login**, dentro la classe **LoginController** viene richiamato il metodo sopra che passa alla **Session** l'indirizzo e-mail inserito in modo da poterlo salvare temporaneamente e avere quindi un riferimento costante all'Utente loggato.

```

public class Session {
    public static final String ADMIN_EMAIL = "@biblioteca.it";
    private static String userEmail;
    private static Utente utente;
    private static String nomeOpera;
    private static int edizione;

    public static void setUserEmail(String email) {
        userEmail = email;
        utente = UserDAO.utente(email); // Aggiorna l'oggetto Utente quando viene impostata l'email
    }

    public static String getUserEmail() {
        return userEmail;
    }

    public static Utente getUtente() {
        return utente;
    }

    public static void setUtente(Utente user) {
        utente = user;
    }
}
  
```

Figura 52: Classe Session

Come si può vedere qui sopra, nel momento in cui viene passato alla Session tale indirizzo mail (tramite il `setUserMail`), viene anche richiamato lo `UserDAO` che, tramite la mail, recupera tutte le informazioni di tale Utente e le salva in un apposito oggetto. Questo è fondamentale per far funzionare per esempio tutte le operazioni effettuabili nella schermata *Profilo*, come per esempio la modifica dei dati personali o la cancellazione dell'account, e per poter direttamente utilizzare le informazioni relative all'utente senza dover interrogare costantemente il database.

```

@FXML
private void logout() {
    Session.setUserEmail(null);
    Session.setUtente(null);
    try {
        App.setRoot("login");
    } catch (IOException e) {
        e.printStackTrace();
    }
}
  
```

Figura 53: Reset Session via logout

Una volta che l'Utente autenticato decide di effettuare il **Logout**, tramite la classe `ProfileController`, viene richiamato un metodo apposito che effettuerà il `reset` della classe `Session`.

4 Conclusione

4.1 Strumentazione Esterna

Per la realizzazione di questo progetto sono state usate diverse piattaforme di generazione per facilitare il lavoro di progettazione e implementazione. Nello specifico è stata usata la piattaforma web **Lucidchart** per la creazione dei diagrammi e degli schemi presenti nella relazione riguardanti: il Diagramma dei Casi d’Uso (vedi sez. 2.1), la Page Navigation (vedi sez. 2.3), il Modello ER del Database (vedi sez. 2.5.1) e tutte le sotto sezioni di Implementazione (vedi sez. 3). Inoltre, è stato usato **Google Docs** per la realizzazione dei Casi d’Uso (vedi sez. 2.2).

Un'altra piattaforma esterna fondamentale allo sviluppo del progetto è stata **Github**, utile per consentire il lavoro coordinato in simultanea del progetto e per tenere sempre aggiornati tutti i componenti del gruppo sullo stato di avanzamento dello sviluppo, anche magari lavorando in remoto l'uno dall'altro.

4.1.1 Intelligenza Artificiale

Per la realizzazione del codice, invece, è stato fatto ricorso all'utilizzo di strumenti di generazione automatica, sia per generare le porzioni di codice stesse (poi manualmente e opportunamente modificate e corrette in base alle esigenze) che per risolvere dubbi strutturali sulle varie classi, come per esempio la struttura della comunicazione gerarchica tra i packages o l'implementazione coerente dei vari pattern. Abbiamo usato principalmente due fonti di Intelligenza Artificiale, *ChatGPT* e *DeepSeek*, notando alcune differenze di generazione e di approccio tra le due. Per esempio, nell'implementazione del pattern **Singleton**, nella classe *DatabaseConnection*, è stato chiesto a ChatGPT di realizzare l'implementazione nella classe ma il risultato non era propriamente ottimale e in linea con l'implementazione standard del pattern. Dopo aver dunque richiesto di rifare l'implementazione sulla classe a DeepSeek, dando anche un prompt molto più specifico sulla richiesta in input, il risultato è nettamente migliorato.

Oppure nella realizzazione delle view per l'interfaccia grafica in FXML sono state trovate delle difficoltà nel far comprendere all'IA esattamente dove andava posizionato a schermo un elemento o come andava disposto il layout della pagina, dovendo presentare dei prompt sempre più specifici prima di ottenere il risultato desiderato.

In generale, abbiamo notato che l'utilizzo di strumenti di generazione tramite Intelligenza Artificiale risultano molto utili ed efficienti in termini di risparmio di tempo e facilità di programmazione; ma, ovviamente, senza un utilizzo mirato e delle richieste estremamente specifiche sui risultati che si vogliono ottenere è molto facile farsi generare del codice errato, formalmente insatto o che non esegue le funzioni richieste. Il nostro codice è stato principalmente supportato da *DeepSeek*, mentre *ChatGPT* è stato più utile per fare domande sulla struttura esterna del progetto, delle relazioni tra packages e sulla gestione generale di tutte le possibili funzioni reali implementabili nel progetto.

4.2 Valutazioni Finali

Apprendimento personale : Lo sviluppo di questo progetto ha rappresentato un'esperienza didattica estremamente significativa, sia sul piano tecnico che su quello organizzativo. Il lavoro di gruppo, svolto in un team composto da quattro persone, ci ha permesso di confrontarci con la reale complessità della collaborazione tra più persone su un progetto software strutturato. Abbiamo imparato a suddividere in modo intelligente e funzionale i compiti, bilanciando le competenze e le preferenze individuali, ma soprattutto ci siamo confrontati quotidianamente su idee, soluzioni e priorità. Non sono mancate le difficoltà nel mettersi d'accordo o nel definire approcci condivisi, ma proprio questi momenti di confronto si sono rivelati preziosi per la crescita personale e professionale, rafforzando la capacità di comunicazione tecnica e di compromesso progettuale.

Difficoltà pratico implementative : Dal punto di vista pratico, abbiamo affrontato e superato diverse sfide, tra cui l'utilizzo di tecnologie mai affrontate prima, come JavaFX per la parte grafica e le logiche di impaginazione con FXML, non sempre intuitive e ben documentate. Inoltre, costruire un'applicazione completa e coerente, che includesse gestione del database, interfaccia grafica, validazioni, test automatizzati, e gestione degli errori, ha richiesto uno sforzo importante, ma necessario per giustificare un lavoro di gruppo a quattro persone. La difficoltà maggiore è stata forse trovare un equilibrio tra profondità tecnica e semplicità d'uso, in modo da consegnare un prodotto che fosse sia robusto che fruibile.

Considerazioni finali : Guardando al risultato raggiunto, possiamo dirci soddisfatti del progetto realizzato: l'applicazione risponde pienamente ai requisiti iniziali, è stabile, estensibile, ben documentata e arricchita da test che ne garantiscono l'affidabilità. Al di là del risultato tecnico, l'esperienza ci ha permesso di toccare con mano cosa significhi pianificare, costruire e rifinire un progetto software reale, fornendoci strumenti e consapevolezze che saranno utili nel nostro futuro professionale.