Relazione di progetto: LoanRanger

Leonardo Toccafondi

2025-11-14

UNIVERSITÀ
DEGLI STUDI
FIRENZE

Indice

1 Introduzione

La seguente relazione riguarda l'elaborato per il superamento dell'esame di Ingegneria del Software, appartenente al modulo Basi di Dati / Ingegneria del Software del corso di Laurea Triennale in Ingegneria Informatica dell'Università degli Studi di Firenze.
Il progetto è stato sviluppato da Leonardo Toccafondi, matricola 7003929, durante il periodo di Agosto - Ottobre 2025 (a.a. 2024/2025).
Il codice sorgente è disponibile su *GitHub* al seguente indirizzo: https://github.com/ltocca/loanranger.

1.1 Obiettivo e descrizione del progetto

Con questo progetto si vuole realizzare un software in grado di gestire i prestiti e le prenotazioni di copie di libri all'interno di una **rete bibliotecaria**, come potrebbe essere quella del comune di Firenze.

Il sistema dovrà supportare la creazione e la gestione di account utente, suddivisi in tre attori principali: il membro (*Member*), il bibliotecario (*Librarian*) e l'amministratore del sistema (*Admin*). I primi dovranno poter cercare, prenotare e prendere in prestito copie disponibili in tutta la rete; i bibliotecari saranno responsabili della gestione delle operazioni di prestito, restituzione e manutenzione dei volumi della propria biblioteca, oltre all'aggiunta di nuove copie; mentre gli amministratori avranno la possibilità di sovrintendere l'intero sistema, modificando dati attinenti a libri, biblioteche e utenti. A tutte le tipologie di utenti dovrà esser data la possibilità di aggiornare le proprie credenziali.

Il programma, oltre a catalogare i titoli da rendere disponibili, dovrà tracciare ogni copia individualmente: ogni prestito viene registrato con una data di inizio e una di scadenza, permettendo un **monitoraggio costante dei prestiti**.

Il software dovrà includere un servizio di notifica via email, configurato per l'invio automatico di comunicazioni nel momento in cui una copia prenotata diventa disponibile e viene riservata all'utente in attesa. In tal caso, l'utente riceverà un messaggio che lo inviterà a recarsi presso la biblioteca per ritirare la copia entro il termine stabilito.

L'applicativo è usufruibile come interfaccia a riga di comando (CLI) per consentire l'utilizzo e la verifica delle funzionalità senza la necessità di un frontend grafico.

1.2 Architettura del progetto

La figura (1.1) mostra lo schema architetturale del progetto, che sarà affrontato in maniera più dettagliata nei capitoli successivi:

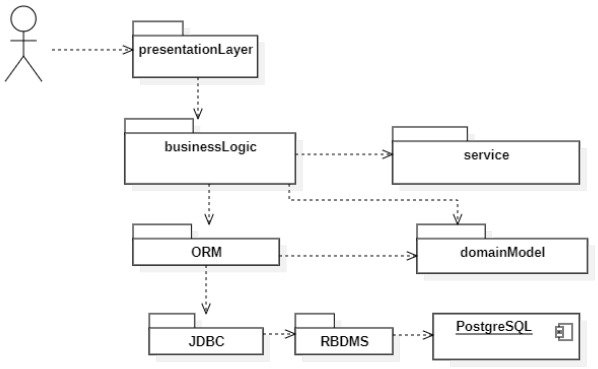

Figura 1.1: Diagramma di architettura

Il software è stato sviluppato in **Java** utilizzando il framework **Spring Boot**, il quale semplifica la configurazione e lo sviluppo di applicazioni stand-alone (improntato allo sviluppo di applicazioni web-based). Per la gestione dei dati è stato utilizzato un database PostgreSQL. Nello sviluppo è stato implementato il pattern DAO (segnalando a Spring Boot che le classi sono delle `@Repository`) per l'accesso ai dati e utilizzando JDBC puro per l'esecuzione delle query. Inoltre Spring Boot viene sfruttato per la gestione delle dipendenze, delle transazioni (tramite l'annotazione `@Transaction`) e del ciclo di vita di questi componenti DAO.

Per mantenere una netta separazione delle responsabilità, il progetto è stato organizzato in più livelli (layers) per garantire una chiara separazione delle responsabilità. Il "punto di ingresso" dell'applicazione è il **presentationLayer**, il quale gestisce l'interazione con l'utente tramite interfaccia a riga di comando. Il package **businessLogic** contiene i controller e la classe `LibraryFacade`, che coordinano le operazioni, applicano le regole di business e fungono da intermediari tra la presentazione e l'accesso ai dati. **Service** fornisce servizi di *supporto* alla logica di business, come l'invio di notifiche o la ricerca dei libri. Il package **Domain Model** definisce le entità principali del sistema, le loro relazioni e gli stati. Infine, lo **strato di persistenza**, sviluppato nel package ORM, gestisce la comunicazione con il database PostgreSQL tramite JDBC, utilizzando classi DAO per eseguire query SQL e mappare i risultati sugli oggetti del modello di dominio.

Per la progettazione sono stati utilizzati diagrammi UML (Unified Modeling Language). Per la fase di testing sono stati impiegati **JUnit 5** e **Mockito** per i test unitari, e **Testcontainers** per i test di integrazione, garantendo un ambiente di test isolato e riproducibile.

2 Progettazione

2.1 Casi d'uso

Il sistema è stato progettato per supportare le operazioni di tre attori principali:

- **Membri**: possono cercare libri, visualizzare la disponibilità delle copie, effettuare prenotazioni di specifiche copie e consultare lo stato dei propri prestiti.
- **Bibliotecari**: gestiscono le operazioni di circolazione come prestiti e restituzioni, l'inventario delle copie dei libri (anche con l'aggiunta di nuove copie), ed il monitoraggio delle prenotazioni e dei prestiti scaduti.
- **Amministratori**: hanno il controllo completo sul sistema, inclusa la gestione e creazioni di utenti, libri, biblioteche e operazioni a livello di database, quali il reset dello schema dello stesso e il seeding con dati predefiniti.

È presente anche un quarto attore "implicito" detto **Guest**, che rappresenta un utente ancora non registrato. La Figura 2.1 mostra i casi d'uso principali per ciascun ruolo, evidenziando le interazioni e le funzionalità a loro disposizione.

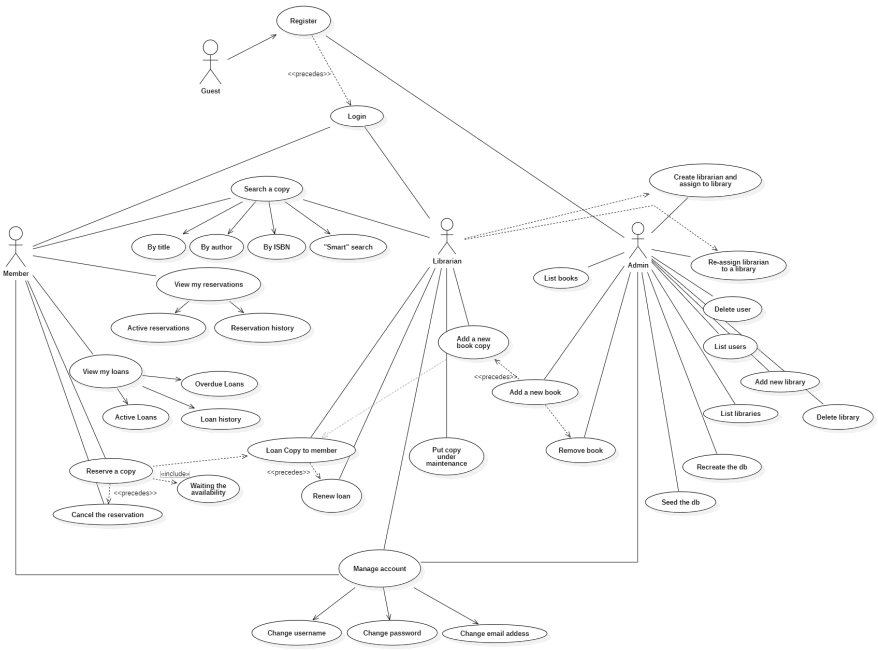

Figura 2.1: Diagramma dei casi d'uso

Use Case 1: Ricercare un libro	
ID	UC-MEM-01
Nome Use Case	Ricercare un libro nel catalogo
Attore Primario	Utente (Membro)
Breve Descrizione	L'utente cerca nel catalogo i libri di suo interesse per verificarne la disponibilità e la collocazione nelle varie biblioteche.
Precondizioni	1. L'utente ha effettuato l'accesso al sistema.
Postcondizioni	1. Il sistema mostra una lista di libri che corrispondono ai criteri, indicando per ogni copia la disponibilità e la biblioteca di appartenenza.
Flusso Principale	1. L'utente avvia la funzione di ricerca. 2. Il sistema richiede i criteri di ricerca (es. per titolo, autore). 3. L'utente inserisce la sua richiesta. 4. Il sistema interroga il catalogo in base ai criteri. 5. Il sistema presenta i risultati all'utente.
Flussi Alternativi	4a. Nessun risultato trovato: il sistema informa l'utente. 4b. Richiesta di ricerca non valida (es. troppo generica).

Use Case 2: Prenotare una copia di un libro	
ID	UC-MEM-02
Nome Use Case	Prenotare una copia di un libro
Attore Primario	Utente (Membro)
Breve Descrizione	L'utente prenota una copia specifica di un libro. Se non è disponibile, viene messo in lista d'attesa.
Precondizioni	1. L'utente ha effettuato l'accesso. 2. L'utente ha identificato la copia da prenotare.
Postcondizioni	1. Viene creata una nuova prenotazione. 2. Se la copia è libera, viene riservata per l'utente. 3. Altrimenti, l'utente viene aggiunto alla lista d'attesa.
Flusso Principale	1. L'utente seleziona la copia da prenotare. 2. Il sistema verifica la disponibilità della copia. 3. Se disponibile, la copia viene riservata e la prenotazione confermata. 4. Se non disponibile, l'utente viene aggiunto alla coda e notificato.
Flussi Alternativi	2a. La copia selezionata non esiste o non è valida.

Tabella 2.1: Templates di usecase per l'utente di tipo Membro.

Use Case 3: Registrare un prestito	
ID	UC-LIB-01
Nome Use Case	Registrare un prestito
Attore Primario	Bibliotecario
Breve Descrizione	Il bibliotecario registra l'uscita di una copia per un prestito a un utente.
Precondizioni	1. Il bibliotecario ha effettuato l'accesso. 2. L'utente e la copia sono identificati. 3. La copia appartiene alla biblioteca di competenza.
Postcondizioni	1. Viene registrato un nuovo prestito. 2. La copia risulta "in prestito". 3. Se l'utente aveva una prenotazione attiva su quella copia, questa viene contrassegnata come "soddisfatta".
Flusso Principale	1. Il bibliotecario identifica utente e copia. 2. Il sistema valida che l'operazione sia permessa (es. stato della copia idoneo). 3. Il sistema registra il prestito, aggiorna lo stato della copia e conferma l'operazione.
Flussi Alternativi	2a. L'utente o la copia non sono validi. 2b. La copia non è disponibile per il prestito. 2c. La copia non appartiene alla biblioteca.

Use Case 4: Processare una restituzione	
ID	UC-LIB-02
Nome Use Case	Processare una restituzione
Attore Primario	Bibliotecario
Breve Descrizione	Il bibliotecario registra il rientro di una copia precedentemente in prestito.
Precondizioni	1. Il bibliotecario ha effettuato l'accesso. 2. La copia restituita è identificata.
Postcondizioni	1. Il prestito viene chiuso. 2. Se non ci sono utenti in attesa, la copia torna "disponibile". 3. Se ci sono utenti in attesa, la copia viene riservata per il primo della coda e quest'ultimo viene notificato.
Flusso Principale	1. Il bibliotecario identifica la copia restituita. 2. Il sistema chiude il prestito associato. 3. Il sistema controlla la lista d'attesa per quella copia e aggiorna lo stato di conseguenza (disponibile o riservata). 4. Il sistema conferma l'operazione.
Flussi Alternativi	2a. La copia non risulta essere in prestito.

Tabella 2.2: Templates di usecase per l'utente di tipo Bibliotecario.

Use Case 5: Aggiungere un nuovo libro	
ID	UC-ADM-01
Nome Use Case	Aggiungere un nuovo libro
Attore Primario	Amministratore
Breve Descrizione	L'amministratore inserisce un nuovo titolo nel catalogo generale del sistema.
Precondizioni	1. L'amministratore ha effettuato l'accesso.
Postcondizioni	1. Un nuovo titolo è stato aggiunto al catalogo generale del sistema.
Flusso Principale	1. L'amministratore inserisce i dati del libro (ISBN, titolo, etc.). 2. Il sistema valida i dati (es. unicità dell'ISBN). 3. Il libro viene salvato nel catalogo. 4. Il sistema conferma l'operazione.
Flussi Alternativi	2a. Dati obbligatori mancanti. 2b. L'ISBN inserito è già presente nel catalogo.

Use Case 6: Registrare un nuovo bibliotecario	
ID	UC-ADM-02
Nome Use Case	Registrare un nuovo bibliotecario
Attore Primario	Amministratore
Breve Descrizione	L'amministratore crea un account per un bibliotecario e lo assegna a una sede.
Precondizioni	1. L'amministratore ha effettuato l'accesso. 2. La biblioteca di destinazione esiste.
Postcondizioni	1. Esiste un nuovo account utente con privilegi da bibliotecario. 2. L'account è collegato a una biblioteca specifica.
Flusso Principale	1. L'amministratore fornisce i dati del nuovo utente e la biblioteca di assegnazione. 2. Il sistema verifica che i dati siano validi (es. email unica, biblioteca esistente). 3. Il nuovo account viene creato e associato alla biblioteca. 4. Il sistema conferma l'operazione.
Flussi Alternativi	2a. La biblioteca specificata non esiste. 2b. L'email o l'username sono già in uso.

Tabella 2.3: Templates di usecase per l'utente di tipo Admin.

2.2 Class Diagram

Il diagramma di classe mostrato nella figura 2.2 alla pagina seguente presenta le classi implementate e le loro interazioni.

Il progetto è stato articolato in packages, i quali seguono la suddivisione in layer citata nell'introduzione, ognuno con scopi specifici:

- **Presentation Layer**: È il punto di ingresso dell'applicazione e gestisce l'interazione con l'utente. Nel nostro caso, è rappresentato dalla classe `MainCLI`, che implementa un'interfaccia a riga di comando. In base alla tipologia di utente offre diverse opzioni.
- **Business Logic Layer**: Contiene i controller, i quali specificano delle operazioni che ogni tipologia di utente può fare (es. `MemberAccountController` per la gestione dell'account di un membro) e la classe `LibraryFacade`[1], che si occupa di fornire tutte le possibili "attività" riguardanti libri e copie. Questo layer orchestra le operazioni, applica le regole di business e funge da intermediario tra il livello di presentazione e quello di accesso ai dati.
- **Service Layer**: Fornisce funzionalità specializzate e trasversali utilizzate dalla Business Logic. Ne sono un esempio `EmailService` per l'invio di notifiche e `BookCopySearchService`, che implementa la logica di ricerca dei libri con diverse strategie, utilizzando il pattern `strategy`.
- **Domain Model**: Definisce le entità fondamentali del sistema, le loro relazioni e i loro stati. Contiene le classi principali come `Book`, `Loan`, `User` e le sue sottoclassi, oltre a componenti del modello come gli stati, applicazione del design pattern `State` (`AvailableState`, `LoanedState`, etc.).
- **Persistence Layer**: Sviluppato all'interno del package `ORM`, contiene le classi DAO (Data Access Object) che si occupano della persistenza dei dati. Questo strato comunica direttamente con il database PostgreSQL tramite JDBC, astraendo le operazioni SQL dal resto dell'applicazione. Ogni DAO è responsabile della scrittura di query SQL esplicite e della mappatura manuale dei risultati sugli oggetti del Domain Model.

[1] Implementazione design pattern strategy.

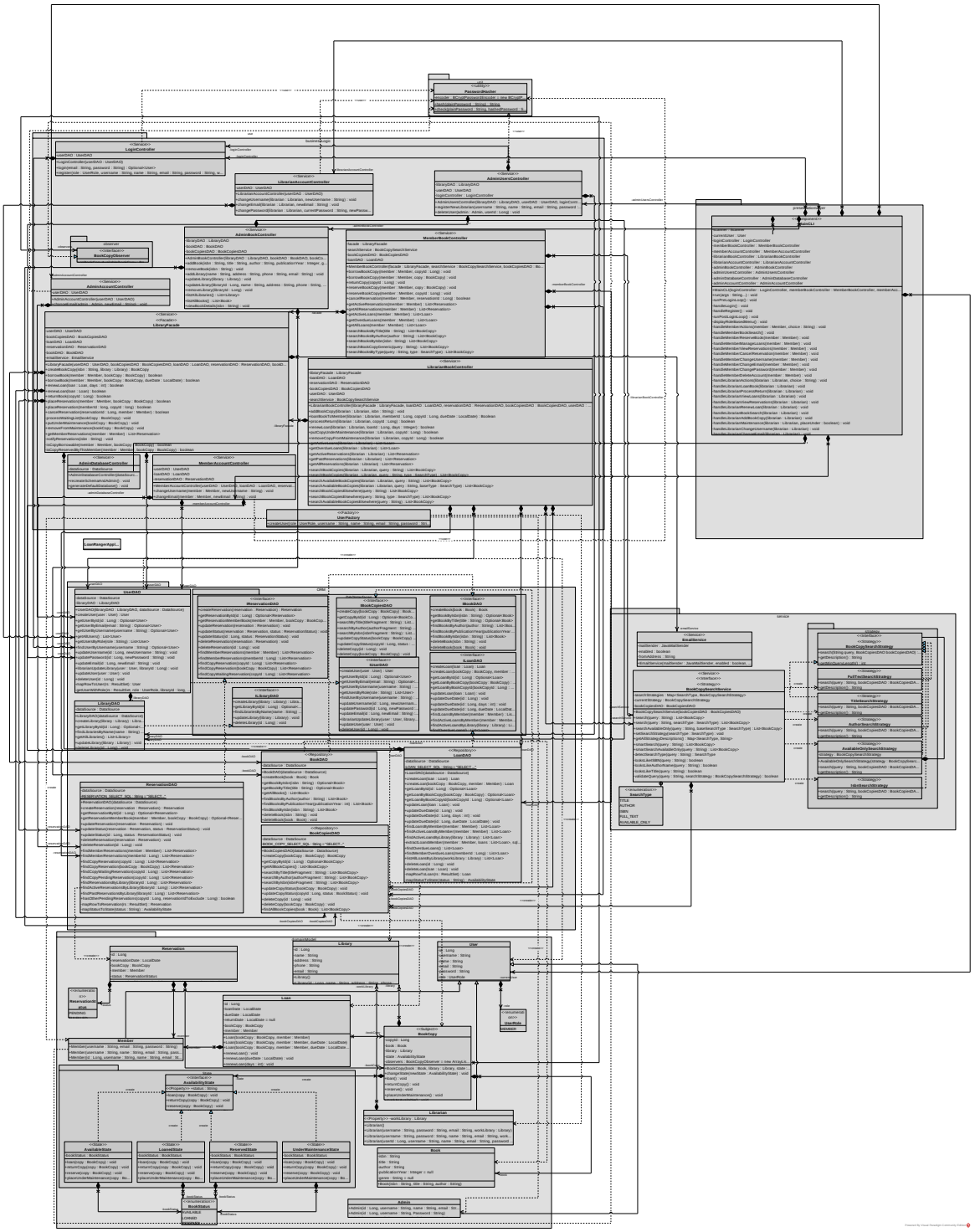

Figura 2.2: Diagramma delle classi

2.3 Diagramma ER e Modello Relazionale

La progettazione del database è partita da un diagramma Entità-Relazione (Figura 2.3) che definisce le entità, gli attributi e le relazioni. Da questo è stato derivato la versione alternativa (Figura 2.4) che riprende la struttura a "tabelle" del database PostgreSQL. Le tabelle principali sono: `users`, `libraries`, `books`, `book_copies`, `loans` e `reservations`.

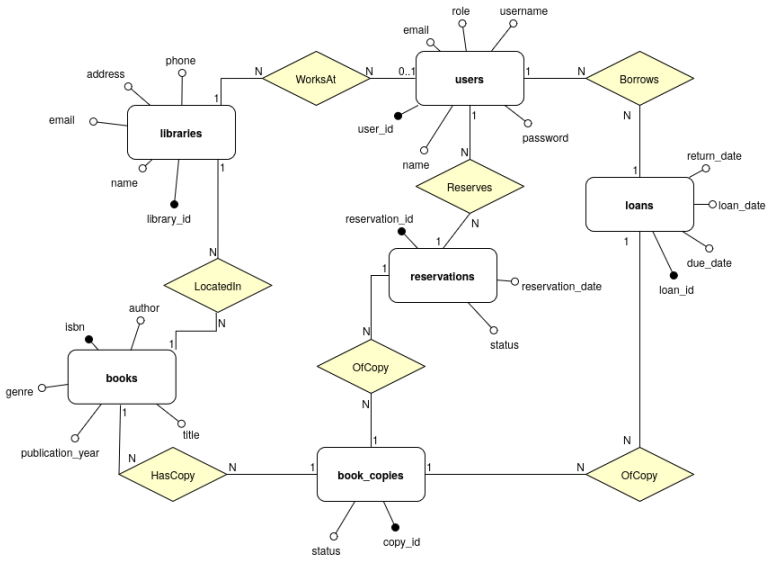

Figura 2.3: Schema ER del progetto

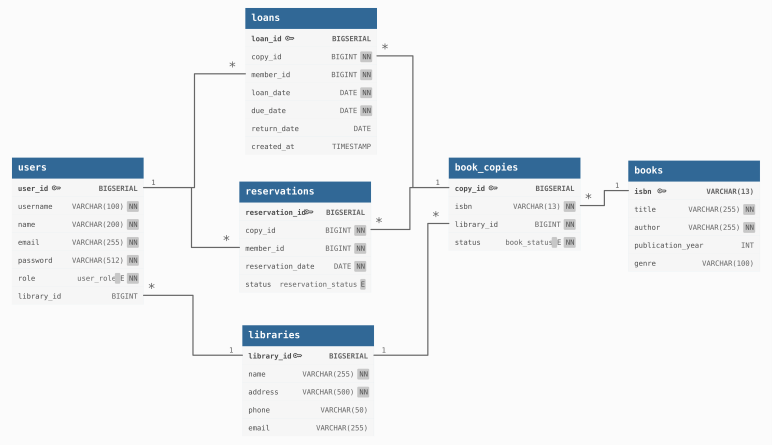

Figura 2.4: Schema ER alternativo

- **Libraries** (library_id, name, address, phone, email)
- **Users** (user_id, username, name, email, password, role, library_id)
- **Books** (isbn, title, author, publication_year, genre)
- **Book_Copies** (copy_id, isbn, library_id, status)
- **Loans** (loan_id, copy_id, member_id, loan_date, due_date, return_date, created_at)
- **Reservations** (reservation_id, copy_id, member_id, reservation_date, status)

Figura 2.5: Schema relazionale del progetto

2.4 Design Patterns

Durante la fase di progettazione del progetto sono stati utilizzati alcuni design pattern, al fine di migliorare la flessibilità del codice (soprattutto in previsione della fase di test), disaccoppiare i vari componenti e cercare di irrobustire quanto possibile il codice. Di seguito saranno introdotti i pattern implementati.

2.4.1 DAO (Data Access Object)

Il DAO, acronimo di *Data Access Object* è un pattern architetturale[2], il quale fornisce un'interfaccia "astratta" per la gestione della persistenza dei dati in un database. Infatti questo pattern offre, alle classi che ne hanno bisogno, tutte le operazioni CRUD ("Create, Read, Update, Delete") che devono svolgere sui dati persistenti, ma senza esporre direttamente tutti i dettagli implementativi del database. In questo modo si aderisce al *principio di singola responsabilità*.

Nel caso del progetto ogni singola entità del dominio ha la propria interfaccia, che definisce tutte le possibili operazioni, e una classe che andrà ad implementare tutti i metodi definiti in precedenza. Le singole classi non gestiscono direttamente le connessioni al database, ma sfruttano i `DataSource` forniti dal framework Spring Boot: in questo modo ne sfruttiamo il pool delle connessioni e rendiamo più semplice lo sviluppo dei test. È importante notare che, data la natura relazionale del dominio, alcuni DAO collaborano tra loro; ad esempio, `UserDAO`, per costruire un oggetto `Librarian` completo, necessita di interagire con `LibraryDAO` per recuperare l'oggetto `Library` associato (tramite un metodo "helper" privato).

[2] Per pattern architetturale (o strutturale) si intendono quelle "soluzioni" che si concentrano sull'organizzazione di classi ed oggetti per formare strutture più grandi.

2.4.2 State

Il design pattern **State** è un pattern comportamentale[3] che permette ad un oggetto di alterare il suo comportamento quando il suo "stato" interno cambia. Ciò appare come un cambio di classe da parte dell'oggetto stesso. Viene utilizzato per evitare grandi costrutti condizionali.

All'interno del progetto il suddetto pattern è stato implementato per rappresentare i vari stati delle singole `BookCopy`, al fine di gestirne la disponibilità. Viene "descritta" un'interfaccia (`AvailabilityState`) che definisce tutte le operazioni possibili relative allo stato della copia: queste saranno poi implementate concretamente in classi singole, una per stato (`AvailableState`, `LoanedState`, `ReservedState`, `UnderMaintenanceState`). Tornando alla classe `BookCopy`, ovvero al contesto, viene mantenuto un riferimento all'oggetto stato, delegandogli l'esecuzione dei relativi metodi. Ad esempio, invocare il metodo `loan()` ad una copia che è nello stato `AvailableState`, comporterà una transizione allo stato `LoanedState`.

2.4.3 Observer

Observer è un altro pattern comportamentale[3] che consente di definire un meccanismo di sottoscrizione per notificare a più oggetti eventuali eventi che si verificano sull'oggetto che stanno osservando, aggiornandoli automaticamente.

All'interno del progetto, questo pattern è stato utilizzato per implementare il sistema di notifiche per le prenotazioni. La classe `BookCopy` rappresenta il soggetto: mantiene una lista di `BookCopyObserver`, fornendo metodi per aggiungerli e rimuoverli. La classe `Member` agisce da osservatore concreto, implementando l'interfaccia `BookCopyObserver` e il suo metodo `onBookCopyAvailable()`. Quando un membro prova a prenotare una copia non disponibile, viene registrato come osservatore di quella copia. Non appena la copia torna disponibile (ad esempio, dopo una restituzione o dopo il rientro dalla manutenzione), il metodo `notifyAvailabilityToWatchers()` della `BookCopy` viene invocato. Questo esegue un ciclo su tutti gli osservatori registrati e chiama il loro metodo `onBookCopyAvailable()`, innescando la logica di notifica (che può essere un messaggio a console e, se abilitato con l'apposito flag, l'invio di un'email).

2.4.4 Strategy

Anche **Strategy** è un design pattern comportamentale[3]: permette di definire una famiglia di algoritmi, "incapsularli" in varie classi separate e rendere i loro oggetti intercambiabili. In questo modo è possibile variare l'algoritmo utilizzato dal client indipendentemente da quest'ultimo.

Nel nostro caso è stato utilizzato per realizzare ed offrire modalità di ricerca di *libri* all'interno di tutto il sistema bibliotecario, attraverso `BookCopySearchService`. Questa interfaccia definisce il metodo `search()`, che poi sarà implementato concretamente nelle classi `TitleSearchStrategy`,

[3] Con pattern comportamentale ci si riferisce a tutti quei design pattern che si occupano di algoritmi o di assegnazione di responsabilità tra oggetti. In generale trattano dell'interazione tra oggetti.

`AuthorSearchStrategy` e `IsbnSearchStrategy`: così facendo viene delegata la query al DAO più appropriato. È stata implementata anche una logica di ricerca "smart": questa si occupa di interpretare la query inserita dall'utente, per poi selezionare la strategia più adatta (ad esempio se la query sembra un ISBN, ovvero è una stringa di almeno sei numeri, usa `IsbnSearchStrategy`). In definitiva `BookCopySearchService` riesce a cambiare *dinamicamente* la strategia da utilizzare.

2.4.5 Factory

Il design pattern **Factory** è un pattern creazionale[4] che fornisce un'interfaccia per la creazione di oggetti in una superclasse, ma consente alle sottoclassi di modificarne il tipo, al fine di "svincolare" le classi che ne fanno uso dalla creazione diretta degli oggetti.

Per quanto riguarda LoanRanger, ne è stata adottata una variante chiamata ***Static Factory***, implementata nella classe `UserFactory`. Il metodo statico `createUser()` implementata in quest'ultima, centralizza di fatto l'intera logica di creazione dei diversi tipi di utente (`Member`, `Librarian`, `Admin`). In base al `UserRole` passato come parametro, il metodo istanzia la sottoclasse corretta di `User`. Questo approccio è vantaggioso perché incapsula la logica di istanziazione, compresa la gestione di precondizioni (come la validazione dell'input e la presenza di una `workLibrary` per i bibliotecari) e operazioni comuni a tutte le creazioni, come l'hashing della password tramite `PasswordHasher`, garantendo che ogni oggetto utente venga creato in modo consistente e corretto.

2.4.6 Facade

Questo pattern è invece un pattern ***strutturale***[5]: infatti `Facade` svolge il ruolo di intermediario tra le classi specifiche degli utenti fornendo un punto d'accesso unico a un insieme di classi che operano in un sottosistema. In questo modo nasconde la complessità interna e consente ai client di interagire con il sistema tramite un'unica classe.

Questo pattern è stato implementato attraverso la classe `LibraryFacade`, che offre i metodi per prendere in prestito un libro oppure ad esempio restituirlo, non comunicando più con i singoli DAO, ma invocando un metodo "unificato". Questo approccio semplifica il codice, ne migliora la leggibilità e riduce l'accoppiamento con il livello di persistenza.

2.4.7 Dependency Injection (via Spring Boot):

La Dependency Injection è un pattern architetturale[2] in cui un oggetto riceve le altre istanze di oggetti di cui ha bisogno (le sue "dipendenze") da una fonte esterna, invece di crearle internamente. Il fra-

[4] I pattern creazionali rientrano nelle tecniche di sviluppo software volte alla creazione di oggetti, permettendo il riutilizzo del codice. L'obiettivo quindi è l'astrazione e l'incapsulamento della logica di istanziazione degli oggetti.

[5] Per design pattern strutturale s'intendono quei pattern che risolvono problemi relativi all'organizzazione e alla composizione delle classi e degli oggetti all'interno di un sistema software. Quindi si occupano di definire come le strutture delle classi e degli oggetti possono essere combinate per formare strutture più grandi e significative.

mework Spring Boot si basa massicciamente su questo principio, gestendo il ciclo di vita degli oggetti (chiamati "bean") e "iniettandoli" dove necessario.

Nel progetto LoanRanger questo pattern viene sfruttato tramite l'annotazione `@Autowired` di Spring. Invece di istanziare manualmente le dipendenze (ad esempio con `UserDAO userDAO = new UserDAO()` all'interno del costruttore di una classe), le classi di servizio (`@Service`) e di persistenza (`@Repository`) dichiarano le dipendenze come campi e lasciano che il container di Spring le risolva e le fornisca al momento della creazione del bean. Ad esempio, un controller come `MemberBookController` riceve direttamente tramite il suo costruttore tutte le istanze dei DAO e dei servizi di cui ha bisogno per funzionare. Questo approccio, noto come "Inversion of Control" serve ad ottenere a un codice più disaccoppiato e più facile da testare. Durante la fase di unit testing, infatti, è possibile "iniettare" versioni mock delle dipendenze (utilizzando le annotazioni `@Mock` e `@InjectMocks` di Mockito) invece di quelle reali, permettendo di testare una classe nella maniera più "isolata" possibile.

3 Implementazione

L'implementazione segue un'architettura a più livelli, cercando di essere il più possibile coerenti con il "paradigma" del ***Domain Driven Design***, applicando i concetti espressi nella fase di progettazione. Il codice sorgente è suddiviso in packages che riflettono questa struttura, descritti nelle successive sezioni.

3.1 Domain Model

Il package `domainModel` contiene le **entità** che descrivono i concetti fondamentali del sistema di gestione bibliotecaria. Le classi sono progettate come classici oggetti e seguono principi di incapsulamento, immutabilità logica e coerenza semantica. Molte di esse implementano pattern strutturali e comportamentali per gestire in modo chiaro le transizioni di stato e le notifiche. Inoltre, questa classi fanno uso delle annotazioni della libreria Lombok, che tramite `@Getter` `@Setter` o `@Data` generano automaticamente i metodi getter e setter, oltre che i metodi `toString()` nel caso di `@Data`.

- **User**: classe astratta che rappresenta un generico utente del sistema. Contiene attributi comuni come `id`, `name`, `email`, `username`, `password` e `role`. Definisce i metodi base di accesso e modifica dei dati e viene estesa da `Member`, `Librarian` e `Admin`.
  - **Member**: rappresenta un utente registrato come lettore. Implementa il metodo `onBookCopyAvailable(BookCopy bookCopy)` dell'interfaccia `BookCopyObserver`, che consente di ricevere notifiche quando una copia *osservata* diventa disponibile.

Listing 3.1: Metodo implementazione observer: crea messaggio che informa l'utente che il libro che stava aspettando è ora disponibile.

```
1  @Override
2  public void onBookCopyAvailable(BookCopy bookCopy) {
3      System.out.println("Book Copy Available - " + bookCopy.getBook().getTitle()
           );
4      String message = "Dear " + this.getName() + ",\n" +
5              "The book '" + bookCopy.getBook().getTitle() + "' by " + bookCopy.
                   getBook().getAuthor() +
6              " (Copy ID: " + bookCopy.getCopyId() + ") that you reserved is now
                   available.\n" +
7              "Please visit the library " + bookCopy.getLibrary().getName() + "
                   soon to borrow it!";
8     System.out.println(message);
9  }
```

  - **Librarian**: rappresenta il bibliotecario responsabile di una specifica biblioteca. Specializza ed estende `User` aggiungendo l'attributo `private Library workLibrary` per memorizzare la biblioteca di appartenenza, imponendo che tale associazione sia sempre presente. La logica è rafforzata nel metodo `setWorkLibrary(Library workLibrary)`, che contiene una clausola di guardia per lanciare un'`IllegalArgumentException` qualora si tenti di associare una biblioteca nulla o priva di ID, garantendo così l'integrità dei dati.
  - **Admin**: rappresenta l'amministratore del sistema. Estende `User` e definisce costruttori semplificati per la creazione di account amministrativi. Non introduce nuovi attributi ma imposta automaticamente il ruolo a `ADMIN`, svolgendo il ruolo di "type marker".

- **Library**: modella una biblioteca fisica. Contiene attributi identificativi come `id` ed informativi come `name`, `address`, `phone` e `email`. È utilizzata per associare a una sede specifica e per organizzare l'inventario dei libri, (per le singole copie).
- **Book**: rappresenta un'opera bibliografica. Specifica informazioni come `isbn`, `title`, `author`, `publicationYear` e `genre` (questi ultimi due possono essere nulli). Ogni istanza rappresenta un titolo univoco nel sistema e non è modificabile dopo la creazione.
- **BookCopy**: rappresenta una singola copia fisica di un libro in una biblioteca. Contiene riferimenti al libro, alla biblioteca e al proprio stato corrente (`private AvailabilityState state`). Come già illustrato utilizza il **pattern State**: i suoi metodi (`loan()`, `returnCopy()`, `reserve()`, etc.) delegano l'esecuzione al metodo corrispondente dell'oggetto `state` corrente (es. `state.loan(this)`). Agisce anche come soggetto nell' **Observer**: gestisce una lista di osservatori tramite i metodi `addObserver()` e `removeObserver()` e li notifica invocando `notifyAvailabilityToWatchers()`.
- **Loan**: rappresenta un prestito di una copia a un membro. Contiene gli attributi `id`, `bookCopy`, `member`, `loanDate`, `dueDate` e `returnDate` (questi ultimi tre attributi sono di tipo `LocalDate`). Implementa metodi di rinnovo, chiusura e verifica della validità del prestito attraverso metodi come `renewLoan()` (con *overloading* per diverse modalità di rinnovo), `isExpired()` (per verificare se la data di scadenza è passata) e `getRemainingDays()` (per calcolare i giorni rimanenti alla scadenza)..
- **Reservation**: modella una prenotazione effettuata da un membro su una copia di un libro. Contiene gli attributi `id`, `bookCopy`, `member`, `reservationDate` e `status`. Quest'ultimo (di tipo `ReservationStatus`) è fondamentale per tracciare il ciclo di vita di una prenotazione, dallo stato di attesa per una copia non disponibile (`WAITING`), a quello in cui il membro aspetta l'approvazione del prestito da parte del bibliotecario (`PENDING`) a quello di completamento (`FULFILLED`) o cancellazione (`CANCELLED`).
- ***Subpackage*** `state`
  - **AvailabilityState**: interfaccia che definisce i comportamenti generali di una copia in base al suo stato. Le operazioni principali (`loan`, `returnCopy`, `reserve`, `markAvailable`) vengono implementate dalle classi concrete.

– **AvailableState**: rappresenta una copia attualmente disponibile per il prestito. Gestisce il passaggio dallo stato di disponibilità a quello di prestito o prenotazione.

– **LoanedState**: rappresenta una copia attualmente in prestito. Impedisce nuove prenotazioni e gestisce il passaggio allo stato di disponibilità al momento della restituzione.

Listing 3.2: Contenuto della classe loanedState.

```
1   public class LoanedState implements AvailabilityState {
2   private final BookStatus bookStatus = BookStatus.LOANED;
3   @Override
4   public void loan(BookCopy copy) {
5       System.err.println("Error: Book is already loaned out.");
6   }
7   @Override
8   public void returnCopy(BookCopy copy) {
9       copy.changeState(new AvailableState());
10      copy.notifyAvailabilityToWatchers();
11      System.out.println("Book returned successfully.");
12  }
13  @Override
14  public void reserve(BookCopy copy) {
15      System.err.println("Error: Cannot reserve a book that is already loaned out.")
            ;
16  }
17  @Override
18  public void placeUnderMaintenance(BookCopy copy) {
19      System.err.println("Error: Cannot place a loaned book under maintenance.");
20  }
21  @Override
22  public void markAsAvailable(BookCopy copy) { // ONLY RETURNING CAN MARK AS
        AVAILABLE THE COPY
23      System.err.println("Error: Cannot mark loaned book as available (after
            maintenance method).");
24  }
25  @Override
26  public String getStatus() {
27      return bookStatus.toString();
28  }
29  }
```

– **ReservedState**: rappresenta una copia prenotata da un utente. Impedisce nuovi prestiti fino all'annullamento della prenotazione o al suo completamento.

– **UnderMaintenanceState**: rappresenta una copia temporaneamente indisponibile perché in manutenzione. Gestisce il ritorno allo stato disponibile una volta completata la procedura.

Ad esempio, `LoanedState` implementa `returnCopy()` per cambiare lo stato della `BookCopy` in `AvailableState` e notificare gli osservatori, mentre la sua implementazione di `loan()` si limita a stampare un messaggio di errore, poiché l'azione non è valida in quello stato.

- **BookStatus**: enumerativo che elenca i possibili stati di una copia (`AVAILABLE`, `LOANED`, `RESERVED`, `UNDER_MAINTENANCE`).
- **UserRole**: enumerativo che definisce i ruoli possibili degli utenti (`MEMBER`, `LIBRARIAN`, `ADMIN`).
- **ReservationStatus**: enumerativo che descrive lo stato di una prenotazione (`PENDING`, `FULFILLED`, `CANCELLED`, `WAITING`).

3.2 Business Logic

Il package `businessLogic` contiene i componenti che implementano le regole operative del sistema e coordinano le interazioni tra gli oggetti del dominio e il livello di persistenza. Ogni classe è gestita da Spring come `@Service` e, quando necessario, integra il supporto transazionale: indicando i metodi con l'annotazione sempre gestita da Spring `@Transactional`, tutte le operazioni che hanno a che fare indirettamente con operazioni di scrittura sul database saranno trattate come transazioni, in modo tale da poter effettuare operazioni di rollback se vi sono problemi.

Le classi denominate "controller" sono state pensate come pagine web indipendenti, al fine di separare sia per ogni tipologia di utente che per funzionalità le operazioni del software.

- **LoginController**: gestisce i processi di autenticazione e registrazione. Durante l'esecuzione del metodo `login()` interagisce con il `UserDAO` per recuperare l'utente utilizzando il metodo statico `PasswordHasher.check()` per confrontare in modo sicuro la password fornita con l'hash salvato. È presente il metodo `register(...)`, il quale gestisce la creazione di un nuovo utente: prima invoca `validateRegistrationParameters()` per controlli preliminari, poi delega la creazione dell'oggetto alla `UserFactory` e infine lo persiste tramite `UserDAO`.
- **MemberAccountController**: permette ai membri di aggiornare le proprie informazioni personali (username, email e password) e di eliminare l'account. La classe espone metodi come `changeUsername()`, `changeEmail()` e `changePassword()`, ognuno dei quali contiene una logica di validazione stringente (es. controllo di unicità per l'email, formato, lunghezza minima della password), prima di invocare il metodo di aggiornamento corrispondente sul `UserDAO`.
- **LibrarianAccountController**: consente ai bibliotecari di gestire le proprie credenziali e informazioni di contatto. Simile al `MemberAccountController`, fornisce i metodi `changeUsername()`, `changeEmail()` e `changePassword()`.
- **AdminAccountController**: gestisce i dati dell'amministratore. Come per le altre tipologie di utenti la classe espone i metodi `changeEmail()` e `changePassword()`, contenenti la stessa logica di validazione.
- **MemberBookController**: fornisce ai membri i metodi per esplorare il catalogo, effettuare prenotazioni e visualizzare lo storico personale dei prestiti. Per le ricerche (es. `searchBooksByTitle()`, `searchBookCopyGeneric()`), delega le operazioni al `BookCopySearchService`, sfruttando così il pattern Strategy. Per le operazioni che modificano lo stato del sistema, come `reserveBookCopy()`

o `cancelReservation()`, si appoggia alla `LibraryFacade`, gestendo le interazioni complesse tra i vari DAO. I metodi di visualizzazione (`getActiveLoans()`, `getAllReservations()`) interrogano direttamente i DAO competenti.

- **LibrarianBookController**: gestisce l'attività quotidiana del bibliotecario, come l'emissione-approvazione dei prestiti, la registrazione delle restituzioni e la gestione dello stato delle copie. Contiene metodi come `loanBookToMember()` o `processReturn()`, che utilizzano metodi di utility privati (`checkCopyBelongsToLibrary()`) per assicurarsi che il bibliotecario stia operando solo su copie appartenenti alla propria `workLibrary`. Invece, per le operazioni complesse (prestito, restituzione, manutenzione) si affida alla `LibraryFacade`.

Listing 3.3: Metodo che dimostra la logica dietro al prestito di una copia.

```
1   @Transactional
2   public Boolean loanBookToMember(Librarian librarian, Long memberId, Long copyId,
        LocalDate dueDate) {
3       User user = null;
4       try {
5           user = userDAO.getUserById(memberId).orElse(null);
6       } catch (Exception e) {
7           System.err.println("Error fetching member: " + e.getMessage());
8           return false;
9       }
10      if (!(user instanceof Member member)) {
11          System.err.println("The user provided (" + memberId + ") is not a user!
                try again");
12          return false;
13      }
14
15      BookCopy copy;
16      try {
17          copy = bookCopiesDAO.getCopyById(copyId).orElse(null);
18      } catch (Exception e) {
19          System.err.println("Error fetching book copy: " + e.getMessage());
20          return false;
21      }
22      if (copy == null) {
23          System.err.println("No copy inserted! Try again.");
24          return false;
25      } else if (!checkCopyBelongsToLibrary(librarian, copy)) {
26          System.err.printf("This book copy with id %d is not in this Library, but
                it is in %s!%n", copy.getCopyId(), copy.getLibrary().getName());
27          return false;
28      }
29
30      if (dueDate == null) {
31          return libraryFacade.borrowBook(member, copy);
32      } else {
33          return libraryFacade.borrowBook(member, copy, dueDate);
34      }
35  }
```

- **AdminBookController**: gestisce il catalogo generale e le informazioni relative alle biblioteche.

I suoi metodi, come `addBook()`, `removeBook()`, `addLibrary()` e `updateLibrary()`, interagiscono direttamente con `BookDAO` e `LibraryDAO`. In particolare il metodo `removeBook()` prima di eliminare un libro controlla tramite `BookCopiesDAO` che non esistano più copie di quel libro nel sistema, prevenendo così la creazione di record orfani.

- **AdminUsersController**: consente la gestione degli utenti da parte dell'amministratore. Con il metodo `registerNewLibrarian()` permette la creazione di un bibliotecario, verificando prima l'esistenza della biblioteca tramite `LibraryDAO` e poi delegando la registrazione al `LoginController`. Altri metodi come `deleteUser()` e `assignLibrarianToLibrary()` interagiscono direttamente con il `UserDAO` per eseguire le modifiche.
- **AdminDatabaseController**: fornisce strumenti per la manutenzione del database, come la ricreazione dello schema o il ripristino dei dati iniziali, quindi funzionalità a *basso livello* per la gestione del database. I metodi `recreateSchemaAndAdmin()` e `generateDefaultDatabase()` utilizzano un `DataSource` iniettato da Spring per ottenere una connessione al database ed eseguire script SQL contenuti nei file di risorse (`reset.sql`, `default.sql`), offrendo la possibilità di gestire lo stato del database..
- **UserFactory**: applica il pattern *Factory* per la creazione delle istanze di `User`. Il suo unico metodo pubblico, `createUser()`, riceve come parametri un `UserRole` e i dati dell'utente. Al suo interno, uno **`switch`** sul ruolo determina quale classe concreta (`Member`, `Librarian`, o `Admin`) istanziare. Centralizza anche la logica di validazione dell'input (es. campi non nulli) e l'hashing della password, garantendo che ogni oggetto utente sia creato in uno stato valido.
- **LibraryFacade**: implementa il pattern *Facade* per semplificare le operazioni che coinvolgono più componenti del sistema. Rappresenta un punto di accesso unificato per le operazioni complesse del sistema bibliotecario. Metodi come `borrowBook()`, `returnBook()` e `placeReservation()` nascondono la complessità di coordinare le varie chiamate a `LoanDAO`, `BookCopiesDAO` e `ReservationDAO`. Ad esempio, `returnBook()` non solo aggiorna il `Loan`, ma cambia lo stato della `BookCopy` e invoca `processWaitingList()`, che a sua volta può creare una nuova prenotazione e notificare un utente tramite `EmailService`.

Listing 3.4: Metodo di LibraryFacade che espone la logica dietro al prestito di una copia.

```
1   @Transactional
2   public Boolean loanBookToMember(Librarian librarian, Long memberId, Long copyId,
        LocalDate dueDate) {
3       User user = null;
4       try {
5           user = userDAO.getUserById(memberId).orElse(null);
6       } catch (Exception e) {
7           System.err.println("Error fetching member: " + e.getMessage());
8           return false;
9       }
10      if (!(user instanceof Member member)) {
11          System.err.println("The user provided (" + memberId + ") is not a user!
                try again");
12          return false;
13      }
```

```
14
15          BookCopy copy;
16          try {
17              copy = bookCopiesDAO.getCopyById(copyId).orElse(null);
18          } catch (Exception e) {
19              System.err.println("Error fetching book copy: " + e.getMessage());
20              return false;
21          }
22          if (copy == null) {
23              System.err.println("No copy inserted! Try again.");
24              return false;
25          } else if (!checkCopyBelongsToLibrary(librarian, copy)) {
26              System.err.printf("This book copy with id %d is not in this Library, but
                    it is in %s!%n", copy.getCopyId(), copy.getLibrary().getName());
27              return false;
28          }
29
30          if (dueDate == null) {
31              return libraryFacade.borrowBook(member, copy);
32          } else {
33              return libraryFacade.borrowBook(member, copy, dueDate);
34          }
35      }
```

3.3 Package `util`

- **PasswordHasher**: fornisce funzioni di *hashing e verifica* delle password basate su algoritmi sicuri della libreria `spring-security-crypto`, nel caso specifico sono criptate utilizzando BCrypt2, con un istanza statica di `BCryptPasswordEncoder`.Il metodo `hash(String plainPassword)` prende una password in chiaro e restituisce il suo hash BCrypt, che include un salt generato casualmente. Invece `check(String plainPassword, String hashedPassword)` confronta una password in chiaro con un hash esistente in modo sicuro, senza esporre l'hash, garantendo che le password non siano mai salvate in chiaro nel database.

3.4 Package `service`

- **EmailService**: gestisce l'invio delle email di notifica agli utenti. È utilizzato principalmente da `LibraryFacade` per comunicare la disponibilità di una copia prenotata o altre variazioni di stato rilevanti. Il suo comportamento è condizionato dalla proprietà `mail.enabled` contenuta nel file `application.properties`. Se essa è **`true`**, utilizza un'istanza di `JavaMailSender` (iniettata da Spring) per inviare una vera email tramite un server SMTP [1]. Se **`false`**, il metodo `sendEmail()` stampa l'email sulla console, fornendo una simulazione (mock) utile per gli ambienti di sviluppo e test senza la necessità di configurare un server di posta.

[1] Sarà approfondito in seguito.

Listing 3.5: In questo metodo viene specificato come sono inviate le notifiche agli utenti.

```java
public void sendEmail(String to, String subject, String body) {
    if (!enabled) {
        System.out.println("Mock email (sending disabled):");
        System.out.println("To: " + to);
        System.out.println("Subject: " + subject);
        System.out.println("Body: " + body);
        return;
    }

    try {
        MimeMessage message = mailSender.createMimeMessage();
        MimeMessageHelper helper = new MimeMessageHelper(message, true);
        helper.setFrom(fromAddress);
        helper.setTo(to);
        helper.setSubject(subject);
        helper.setText(body, false);
        mailSender.send(message);
        System.out.println("Email sent successfully to: " + to);
    } catch (MailException e) {
        System.err.println("Failed to send email to " + to + ": " + e.getMessage()
            );
    } catch (MessagingException | RuntimeException e) {
        System.err.println("Failed to send email to " + to + ": " + e.getMessage()
            );
    }
}
```

- **BookCopySearchService**: implementa le logiche di ricerca avanzata di libri e copie. Contiene una `Map` che associa un `SearchType` (enum) a un'implementazione concreta di `BookCopySearchStrategy`. Il metodo `search(String query, SearchType type)` seleziona la strategia appropriata dalla mappa e la esegue. Inoltre il metodo `smartSearch(String query )` aggiunge un livello di complessità : analizza la query con metodi privati (`looksLikeISBN`, `looksLikeAuthorName`) per dedurre il tipo di ricerca più probabile e applicare la strategia corrispondente in automatico.

- ***Subpackage*** `service.strategy`

  Questo subpackage contiene l'implementazione del **design pattern Strategy**, utilizzato per diversificare gli algoritmi di ricerca dei libri.

  - **BookCopySearchStrategy**: è un'interfaccia `sealed`[2] che definisce il contratto per tutte le strategie di ricerca. Dichiara i metodi `search(String query, BookCopiesDAO bookCopiesDAO)`, `getDescription()` e un metodo **`default`** `getMinQueryLength()` che fornisce una lunghezza minima di default per la query.

  - **TitleSearchStrategy**: è una classe **`final`** che implementa `BookCopySearchStrategy` per la ricerca basata sul titolo del libro. Il metodo `search()` viene implementato delegando la chiamata al metodo `bookCopiesDAO.searchByTitle(query)`. Rappresenta la logica speci-

[2] Tipologia di interfaccia che limita le classi che la possono implementare, descrivendo un insieme predefinito.

fica per la ricerca per titolo, compresa la validazione sulla lunghezza minima della query (sovrascrivendo `getMinQueryLength()` per richiedere almeno 2 caratteri), andando ad astrarre questa logica dal `BookCopySearchService` che la utilizza.

- **AuthorSearchStrategy**: è una classe `final` che implementa `BookCopySearchStrategy` per la ricerca basata sull'autore. Analogamente a `TitleSearchStrategy`, questa classe implementa il metodo `search()` invocando `bookCopiesDAO.searchByAuthor(query)`. Isola l'algoritmo di ricerca per autore.
- **IsbnSearchStrategy**: è una classe `final` che implementa `BookCopySearchStrategy` per la ricerca basata sul codice ISBN. La sua implementazione del metodo `search()` "sanifica" la query in input tramite la regex `replaceAll("[^0-9X]", "")` per rimuovere trattini o spazi,dopodiché esegue la ricerca tramite `bookCopiesDAO.searchByIsbn(cleanQuery)`. Definisce anche una lunghezza minima della query più restrittiva (`getMinQueryLength()` restituisce 6) per evitare ricerche ambigue.

Listing 3.6: Esempio di implementazione del metodo search() per la ricerca per ISBN.

```java
@Override
public List<BookCopy> search(String query, BookCopiesDAO bookCopiesDAO) {
  if (query == null || query.trim().isEmpty()){
      System.err.println("Error: the query is null or empty");
      return List.of();
  }
  // Regex added to remove all characters apart from the numbers and the X
      for ISBN10
  String cleanQuery = query.trim().replaceAll("[^0-9X]", "");
  try{
      return bookCopiesDAO.searchByIsbn(cleanQuery);
  } catch (Exception e) { // broad exception check
      throw new RuntimeException("Error searching book copies by isbn", e);
  }
}
```

- **FullTextSearchStrategy**: è una classe `final` che implementa `BookCopySearchStrategy` per una ricerca generica su più campi. La classe combina i risultati di altre strategie. Il suo metodo `search()` esegue tre chiamate separate al DAO: `searchByTitle()`, `searchByAuthor()` e `searchByIsbn()`. Successivamente, unisce i risultati delle tre liste in un'unica lista, utilizzando un `HashSet` per garantire che ogni `BookCopy` appaia una sola volta, anche se è stata trovata da più criteri di ricerca.

3.5 Object-Relational Mapping (ORM)

Il package `ORM` gestisce l'interazione diretta con il database PostgreSQL tramite JDBC. Ogni classe DAO è sviluppata con una specifica entità del dominio: è presente un subpackage `DAOInterfaces` che contiene tutte le interfacce che definiscono le operazioni da utilizzare. Per evitare la possibilità di SQL in-

jection, vengono utilizzate delle query ***parametriche***[3] tramite oggetti della classe `PreparedStatement`. Inoltre è stato implementato in modo sistematico il costrutto `try-with-resources` in ogni metodo che coinvolga risorse del tipo `AutoCloseable`, come `Connection`, `Statement` e `ResultSet`: così facendo ogni risorsa "sensibile" viene chiusa in modo deterministico al termine del blocco try. Le connessioni al database sono basate su un `DataSource` configurato da Spring Boot (basato su HikariCP) per la gestione efficiente del connection pooling. Ciò consente una migliore integrazione con l'ambiente di test `Testcontainers`.

- **UserDAO**: si occupa della persistenza degli utenti del sistema. Implementa l'interfaccia `IUserDAO`. Il metodo `createUser()` gestisce la logica di inserimento per le diverse sottoclassi di `User`, immettendo correttamente il parametro `library_id` esclusivamente nel caso in cui l'utente sia un `Librarian`. Il metodo `mapRowToUser()` è fondamentale: legge un `ResultSet` e, in base al valore della colonna `role`, procede ad istanziare la corretta tipologia di utente (`Member`, `Librarian` o `Admin`), recuperando se necessario la `Library` associata per i bibliotecari. Gestisce la cifratura e verifica delle password in collaborazione con `PasswordHasher`. Sono presenti inoltre i metodi `getUserById()`, `getUserByEmail()` e `getUserByUsername()` per il recupero di singoli utenti, `getAllUsers()` per ottenerne un elenco completo, e altri metodi di aggiornamento specifici come `updateUsername()`, `updatePassword()`, `updateEmail()` e `librarianUpdateLibrary()` per modifiche atomiche dei parametri.
- **BookDAO**: gestisce le operazioni relative ai libri nel catalogo generale. Implementa il metodo `createBook()`, che gestisce correttamente l'inserimento di valori `NULL` per gli attributi opzionali come `publication_year` e `genre`. I metodi di ricerca come `findBooksByAuthor()` e `findBookByIsbn()` utilizzano query SQL con clausole `WHERE` e `ORDER BY` per recuperare e ordinare i dati direttamente dal database. Anche questa classe possiede un metodo `mapRowToBook()` per convertire un `ResultSet` in un oggetto di tipo `Book`. Vengono forniti anche metodi di recupero di oggetti `Book` come `getBookByIsbn()`, `getBookByTitle()`, `getAllBooks()` e `findBooksByPublicationYear()`.È presente anche `deleteBook()`, quest'ultimo con overloading per accettare sia un ISBN (quindi una `String` che rappresenta univocamente il libro) sia un oggetto `Book`.
- **BookCopiesDAO**: classe responsabile della persistenza degli oggetti `BookCopy` nel database. È presente una costante `BOOK_COPY_SELECT_SQL` che definisce la query `JOIN` principale, riutilizzata da quasi tutti i metodi di lettura per recuperare non solo i dati della copia, ma anche quelli del `Book` e della `Library` associati con un'unica chiamata. Il metodo `mapRowToBookCopy()` è complesso: ricostruisce l'intera struttura degli oggetti (`BookCopy`, `Book`, `Library`), invocando contestualmente il metodo "helper" `mapStatusToState()` al fine di convertire lo stato testuale del database (es. "LOANED") nell'oggetto di tipo `State` corretto (es. `new LoanedState()`). La classe inoltre espone metodi di ricerca come `searchByTitle()`, `searchByAuthor()` e `searchByIsbn()`, i quali sfruttano `ILIKE` per ricerche parziali[4]. Sono presenti anche metodi "utility" come

[3] Tramite le query parametriche è possibile creare prima una stringa che definisce la query, specificando in seguito i valori desiderati i quali andranno a sostituire dei segnaposto all'interno della stringa. Per questo viene detta parametrica

[4] Non è necessario fornire ad esempio il nome completo dell'autore per effettuare le ricerca.

`findAllBookCopies(Book book)` per trovare tutte le copie di un dato libro e `updateCopyStatus()` per modificare lo stato di una copia.

Listing 3.7: Metodo per ricreare un oggetto di tipo BookCopy da una riga del database.

```
private BookCopy mapRowToBookCopy(ResultSet rs) throws SQLException {
      // 1. Build the nested Book object
      Book book = new Book(
              rs.getString("isbn"),
              rs.getString("title"),
              rs.getString("author"),
              rs.getInt("publication_year"),
              rs.getString("genre")
      );

      // 2. Build the nested Library object
      Library library = new Library(
              rs.getLong("library_id"),
              rs.getString("library_name"),
              rs.getString("address"),
              rs.getString("phone"),
              rs.getString("library_email")
      );

      // 3. Build the main BookCopy object
      BookCopy copy = new BookCopy();
      copy.setCopyId(rs.getLong("copy_id"));

      // 4. Assemble the object graph
      copy.setBook(book);
      copy.setLibrary(library);

      // 5. Convert status string to concrete State object
      copy.setState(mapStatusToState(rs.getString("status")));

      return copy;
  }
```

- **LoanDAO**: si occupa della gestione dei prestiti. Come `BookCopiesDAO`, la classe utilizza una query `JOIN` predefinita (`LOAN_SELECT_SQL`) per recuperare tutte le informazioni correlate a un prestito. Fornisce metodi di ricerca specializzati come `findActiveLoansByMember()` e `findOverdueLoans()`, che saranno utilizzati all'interno delle classi in `businessLogic` (es. `WHERE l.return_date IS NULL` o `WHERE l.due_date < CURRENT_DATE`).

Listing 3.8: Esempio di metodo in LoanDAO per trovare i prestiti scaduti.

```
@Override
public List<Loan> findOverdueLoans() {
    List<Loan> loans = new ArrayList<>();
    String sql = LOAN_SELECT_SQL + " WHERE l.due_date < CURRENT_DATE AND l.
        return_date IS NULL ORDER BY l.due_date ASC";
    try (Connection connection = dataSource.getConnection();
        Statement stmt = connection.createStatement();
```

```
7           ResultSet rs = stmt.executeQuery(sql)) {
8           while (rs.next()) {
9               loans.add(mapRowToLoan(rs));
10          }
11      } catch (SQLException e) {
12          throw new RuntimeException("Error finding overdue loans", e);
13      }
14      return loans;
15  }
```

Oltre ai metodi principali quali `createLoan()`, `getLoanById()`, `updateLoan()` e `deleteLoan()`, sono stati implementati metodi di interrogazione del database a supporto dei casi d'uso del software, quali `findLoansByMember()`, `findActiveLoansByLibrary()`, `findOverdueLoans()` e `findMemberOverdueLoans(Long memberId)`. Ognuno di questi definisce una query SQL specifica in modo tale da filtrare efficientemente i risultati.

- **ReservationDAO**: gestisce le prenotazioni dei membri. Come nelle classi precedenti è stata create una query `JOIN` di default (`RESERVATION_SELECT_SQL`) per recuperare tutti i dati correlati. Offre metodi di ricerca come `findCopyWaitingReservation()`, il quale recupera le prenotazioni per una data copia in stato `WAITING` e le ordina per data, implementando la logica della "coda di attesa" direttamente a livello di database.

  La classe definisce metodi CRUD standard come `createReservation()`, `getReservationById()` e `deleteReservation()`. Altri metodi più specifici comprendono `findMemberReservations(Member member)` per mostrare lo storico di un utente, `findReservationsByLibrary(Long libraryId)` per il monitoraggio delle prenotazioni da parte dei bibliotecari, e il cruciale `hasOtherPendingReservations()`, il quale esegue una query ottimizzata (`SELECT 1 ... LIMIT 1`) per verificare rapidamente se esistono altre prenotazioni attive su una copia.

Listing 3.9: Esempio di metodo in reservationDAO per creare/inserire un oggetto all'interno del database.

```
1  @Override
2    public Reservation createReservation(Reservation reservation) {
3        String sql = "INSERT INTO reservations (copy_id, member_id, reservation_date,
             status) VALUES (?, ?, ?, ?::reservation_status)";
4        try (Connection connection = dataSource.getConnection();
5             PreparedStatement pstmt = connection.prepareStatement(sql, Statement.
                  RETURN_GENERATED_KEYS)) {
6            pstmt.setLong(1, reservation.getBookCopy().getCopyId());
7            pstmt.setLong(2, reservation.getMember().getId());
8            pstmt.setDate(3, Date.valueOf(reservation.getReservationDate()));
9            pstmt.setString(4, reservation.getStatus().name());
10
11           int affectedRows = pstmt.executeUpdate();
12           if (affectedRows > 0) {
13               try (ResultSet rs = pstmt.getGeneratedKeys()) {
14                   if (rs.next()) {
15                       reservation.setId(rs.getLong("reservation_id"));
16                   }
```

```
17                 }
18             }
19             return reservation;
20         } catch (SQLException e) {
21             throw new RuntimeException("Error creating reservation", e);
22         }
23     }
```

- **LibraryDAO**: gestisce la persistenza delle biblioteche. Consente l'inserimento di nuove biblioteche, la modifica dei dati e la rimozione di record, oltre al recupero dell'elenco completo delle strutture registrate. Ad esempio il metodo `findLibrariesByName()` utilizza l'operatore `ILIKE` di PostgreSQL per eseguire una ricerca testuale case-insensitive, una scelta che migliora l'usabilità della ricerca.

  Oltre ai metodi CRUD `createLibrary()`, `updateLibrary()` e `deleteLibrary()`, fornisce i metodi di lettura `getLibraryById()` per recuperare una singola entità tramite `Optional`, e `getAllLibraries()` per ottenere l'elenco completo delle biblioteche presenti nel sistema, ordinate per nome.

Listing 3.10: Esempio di metodo in LibraryDAO per la ricerca per nome.

```
1   @Override
2   public List<Library> findLibrariesByName(String name) {
3       List<Library> libraries = new ArrayList<>();
4       String sql = "SELECT * FROM libraries WHERE name ILIKE ?"; // ILIKE used for
            case-insensitivity
5       try (Connection connection = dataSource.getConnection();
6           PreparedStatement pstmt = connection.prepareStatement(sql)) {
7
8           pstmt.setString(1, "%" + name + "%");
9           try (ResultSet rs = pstmt.executeQuery()) {
10              while (rs.next()) {
11                  libraries.add(mapRowToLibrary(rs));
12              }
13          }
14      } catch (SQLException e) {
15          throw new RuntimeException(String.format("No library contains: %s", name),
                e);
16      }
17      return libraries;
18  }
```

3.6 Package `presentationLayer` (interfaccia CLI)

L'interfaccia utente è sviluppata nella classe `MainCLI`, che implementa l'interfaccia `CommandLineRunner` di Spring Boot. Questa classe si occupa di gestire il ciclo di vita dell'applicazione, mostrando menu contestuali basati sul ruolo dell'utente loggato (`Member`, `Librarian`, `Admin`), invocando i metodi appropriati dei controller della Business Logic in base all'input dell'utente.

Usa l'annotazione `@Component` di Spring Boot, oltre ad usare `@ConditionalOnProperty` per specificare delle proprietà dell'applicazione.

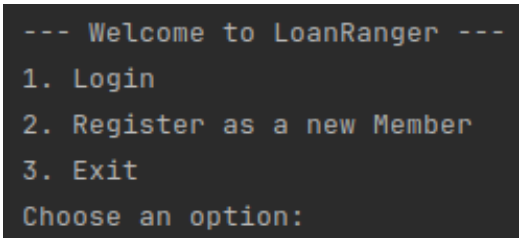

Figura 3.1: Menù principale all'avvio dell'applicazione.

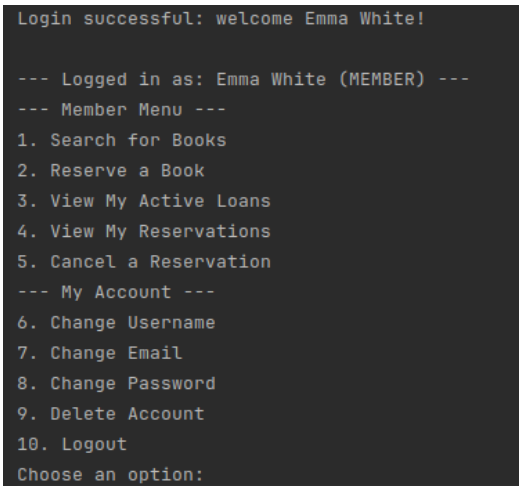

Figura 3.2: Menù contestuale per l'utente di tipo Membro.

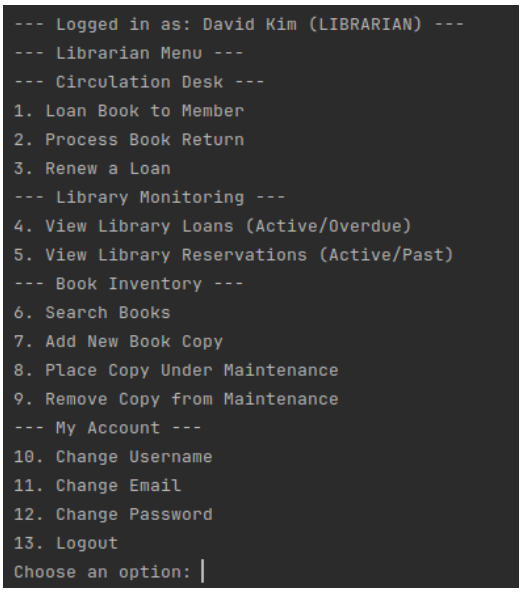

Figura 3.3: Menù contestuale per l'utente di tipo Bibliotecario.

```
Login successful: welcome Admin One!

--- Logged in as: Admin One (ADMIN) ---
--- Admin Menu ---
--- Library Management ---
1. Add New Library
2. Update Library Information
3. Remove Library
4. List All Libraries
--- Book Management ---
5. Add New Book
6. Remove Book
7. List All Books
8. View Book Details
--- User Management ---
9. Register New Librarian
10. Re-assign Librarian
11. Delete a User Account
12. List All Users
--- System ---
13. Seed Database with Default Data
14. Recreate Database (Schema + Admin)
--- My Account ---
15. Change Email
16. Change Password
17. Logout
Choose an option:
```

Figura 3.4: Menù contestuale per l'utente di tipo Amministratore.

3.7 Self-hosting server email

Per evitare di utilizzare un servizio di terze parti per l'invio di notifiche, si è deciso di provare ad implementare un server email autogestito. Il sistema utilizzato prevede un VPS ospitato da OVH con una macchina virtuale con OS Rocky Linux 9, utilizzando Mailcow per avere direttamente tutta l'architettura necessaria all'interno di un container Docker. In questo modo tutti i servizi più complessi sono già preconfigurati, come Postfix e Dovecot, incapsulati in ambienti isolati per semplificarne sia la gestione che l'aggiornamento.

Non è stato semplice ottenere immediatamente una buona reputazione per il server, difatti per i primi giorni tutte le email mandate dal server venivano bloccate dai server antispam dei vari servizi di posta più famosi. Dopo qualche giorno la configurazione dei record DNS del dominio mail.ltocca.dev si è propagata e le email arrivano passando il filtro spam. Tornando alla configurazione oltre al record `MX`, sono stati implementati tutti gli standard moderni di autenticazione del mittente:

- **SPF** (**Sender Policy Framework**), per dichiarare gli IP autorizzati a inviare email.
- **DKIM** (**DomainKeys Identified Mail**), per apporre una firma digitale a ogni email, verificandone l'integrità.
- **DMARC**, per istruire i server riceventi su come gestire i fallimenti di autenticazione.

La sicurezza della connessione tra l'applicazione e il server è garantita dalla cifratura TLS, con l'applicazione Spring Boot che si connette sulla porta `587` utilizzando STARTTLS.

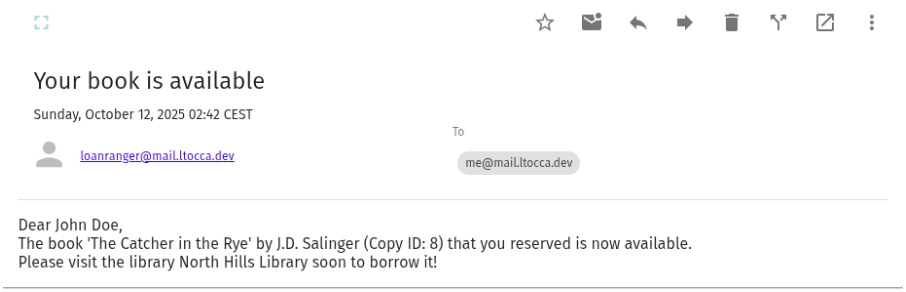

Figura 3.5: Esempio di notifica email mandata dall'indirizzo `loanranger@mail.ltocca.dev`

3.8 Implementazione ed utilizzo del framework Spring Boot

L'adozione del framework Spring Boot è stata fondamentale durante l'implementazione del progetto. Sebbene l'utilizzo sia stato previsto fin dall'inizio per creare più semplicemente l'ambiente di sviluppo con gli `spring-boot-starters`, al momento del testing è stato effettuato un refactoring della parte ORM. Questo perché applicare il pattern Inversion of Control era un passaggio necessario per semplificare la costruzione dei test applicando le classi create con Mockito.

Inoltre utilizzare una configurazione esterna in `application.properties` ha permesso di *isolare* dal codice informazioni sensibili come le crendeziali del database PostgreSQL e quelle dell'account email utilizzato, senza intervenire sul codice sorgente poi pubblicato su Github.

4 Testing

Per garantire la correttezza e la robustezza dell'applicazione sono stati implementati test unitari, di integrazione e end-to-end. In questo modo si mira ad assicurare che ogni layer del software funzioni come previsto.

Nel determinare una convenzione per i nomi dei metodi di test, è stato seguito il formato `metodoTestato_condizioneTestata_risultatoAtteso()`, in modo tale da descrivere il più possibile il comportamento del test. Questo stile, derivato dal *Behavior-Driven Development* (*BDD*), rende i test praticamente auto-documentati e migliora la manutenzione e l'individuazione di errori in caso di fallimento.

4.1 Unit Testing

Questi test sono stati scritti per le classi del package `businessLogic`, `service` e `utility`, per verificare la logica di business in isolamento dalle dipendenze esterne come il database. Per ottenere questo isolamento, ogni classe è stata testata simulando tutte le sue dipendenze esterne (come i DAO, gli helper o il servizio di hashing delle password) attraverso oggetti *mock* creati con il framework **Mockito**. In questo modo, nessun test unitario dipende dal database o da uno stato persistente: il comportamento atteso è stato verificato esclusivamente tramite la logica interna della classe.

4.1.1 package `businessLogic`

Sono state testate tutte le classi di "tipo Controller", creando svariati test al fine di coprire diversi scenari: sono stati testati tutti i percorsi logici che implementano le regole di business. Ad esempio, nella classe `LibrarianAccountControllerTest`, è presente il metodo di test `changeUsername_whenUsernameIsTaken_throwsIllegalArgumentException()` nel quale viene verificato che, il tentativo di modifica dell'username (da parte di un bibliotecario) con uno nuovo ma già presente nel database, sollevi correttamente un eccezione di tipo `IllegalArgumentException`.

Passando ad un altro esempio, contenuto nella classe `LibrarianBookController` (nel test `loanBookToMember_onBookInAnotherLibrary_fails`) viene simulato lo scenario in cui un bibliotecario prova ad effettuare un prestito per una copia di un libro registrato in un'altra biblioteca. Tramite Mockito, `BookCopiesDAO` viene configurato affinché restituisca un oggetto `BookCopy` associato a una biblioteca diversa da quella del bibliotecario. L'asserzione del test deve verificare quindi che il

metodo restituisca false e, cosa ancora più importante, che il metodo borrowBook sulla LibraryFacade non venga mai invocato.

Per componenti come `LibraryFacade`, i test sono stati sviluppati per verificare che le operazioni gestiscano correttamente le chiamate ai vari DAO. Infatti, in `LibraryFacadeTest` è presente `returnBook_whenLoanExists_updatesLoanAndProcessesWaitingList`: questo test simula lo scenario di restituzione di un libro per il quale esiste un altro utente in coda di attesa. Sempre facendo uso di Mockito, si configura il `LoanDAO` per restituire un oggetto `Loan` *"attivo"* e il `ReservationDAO` per fornire una prenotazione in stato `WAITING`. Il test verifica quindi che la classe esegua correttamente l'intera sequenza di operazioni.

```
```{#lst:test-username-taken .java language="Java" caption="Test unitario che verifica la gestione di un username duplicato." label="lst:test-username-taken"}
  @Test
  void changeUsername_whenUsernameIsTaken_throwsIllegalArgumentException() throws Exception {
      // Given
      String newUsername = "takenUser";
      when(userDAO.findUserByUsername(newUsername)).thenReturn(Optional.of(new Librarian()));

      // When & Then
      assertThatThrownBy(() -> librarianAccountController.changeUsername(librarian, newUsername))
          .isInstanceOf(IllegalArgumentException.class)
          .hasMessage("Error: This username is already taken by another user.");

      verify(userDAO, never()).updateUsername(anyLong(), anyString());
  }
```
```

Listing 4.1: Test che verifica la restituzione di un libro con utenti di attesa.

```
  @Test
  void returnBook_whenLoanExists_updatesLoanAndProcessesWaitingList() {
      // Given
      availableCopy.loan(); // Manually set state to Loaned
      Loan activeLoan = new Loan(availableCopy, member);
      activeLoan.setId(50L);

      Reservation waitingReservation = new Reservation(availableCopy, waitingMember);
      waitingReservation.setId(99L);
      waitingReservation.setStatus(ReservationStatus.WAITING);

      when(bookCopiesDAO.getCopyById(101L)).thenReturn(Optional.of(availableCopy));
      when(loanDAO.getLoanByBookCopyId(101L)).thenReturn(Optional.of(activeLoan));
      when(reservationDAO.findCopyWaitingReservation(101L)).thenReturn(List.of(waitingReservation));

      // When
      boolean success = libraryFacade.returnBook(101L);

      // Then
```

```
20        assertThat(success).isTrue();
21
22        // Verify loan is updated with a return date
23        ArgumentCaptor<Loan> loanCaptor = ArgumentCaptor.forClass(Loan.class);
24        verify(loanDAO).updateLoan(loanCaptor.capture());
25        assertThat(loanCaptor.getValue().getReturnDate()).isNotNull();
26
27        // Verify the waiting reservation is now PENDING
28        verify(reservationDAO).updateStatus(99L, ReservationStatus.PENDING);
29
30        // Verify the book copy is now RESERVED for the next person
31        assertThat(availableCopy.getState()).isInstanceOf(ReservedState.class);
32        verify(bookCopiesDAO).updateCopyStatus(availableCopy);
33
34        // Verify an email is sent to the waiting member
35        verify(emailService).sendEmail(eq("jane.doe@email.com"), anyString(), anyString());
36    }
```

4.1.2 package `util`

La classe `PasswordHasherTest` verifica il corretto funzionamento dell'utility di hashing delle password, basata su BCrypt. I test coprono gli scenari principali: la generazione di un hash valido, la verifica positiva con una password corretta e, soprattutto, la verifica negativa. Ad esempio, il test `check_withIncorrectPassword_returnsFalse` si assicura che il metodo di verifica restituisca **`false`** nel momento in cui viene fornita una password errata, confermando la robustezza del meccanismo di autenticazione.

Listing 4.2: Test unitario che verifica il fallimento del confronto con una password errata.

```
1  @Test
2  void check_withIncorrectPassword_returnsFalse() {
3      String plainPassword = "correctPassword";
4      String wrongPassword = "wrongPassword";
5      String hashedPassword = PasswordHasher.hash(plainPassword);
6
7      boolean result = PasswordHasher.check(wrongPassword, hashedPassword);
8
9      assertThat(result).isFalse();
10 }
```

4.1.3 package `service`

La classe `EmailServiceTest` si concentra sul comportamento del servizio di notifica tramite email. Come esempio prendiamo il test `sendEmail_whenDisabled_doesNotCallMailSender`: utilizzando un mock di `JavaMailSender`, si verifica che, quando la proprietà `mail.enabled` è impostata su **`false`**, nessun metodo di invio venga **mai invocato**, confermando che la logica di "mocking" per l'ambiente di sviluppo funzioni come previsto.

Per `BookCopySearchService`, i test convalidano la corretta implementazione del pattern Strategy. Il test `smartSearch_withIsbnLikeQuery_usesIsbnStrategy` simula l'input, da parte di un qualsiasi utente, di una stringa simile a un ISBN e verifica, tramite `verify()`, che il servizio selezioni e invochi correttamente la `IsbnSearchStrategy` appropriata, dimostrando la correttezza della logica di "smart detection" per la selezione automatica dell'algoritmo di ricerca.

Listing 4.3: Test unitario che verifica il non invio dell'email quando il servizio è disabilitato.

```
@Test
void sendEmail_whenDisabled_doesNotCallMailSender() {
    emailService.sendEmail("recipient@example.com", "Test Subject", "Test Body");

    verify(mailSender, never()).createMimeMessage();
    verify(mailSender, never()).send(any(MimeMessage.class));
}
```

Listing 4.4: Test unitario che verifica la selezione della strategia di ricerca corretta.

```
@Test
void smartSearch_withIsbnLikeQuery_usesIsbnStrategy() {
    // Given
    String query = "9780134685991";
    when(bookCopiesDAO.searchByIsbn(query)).thenReturn(Collections.singletonList(testCopy)
        );

    // When
    searchService.smartSearch(query);

    // Then
    verify(bookCopiesDAO).searchByIsbn(query);
}
```

4.2 Integration Testing

I test di integrazione hanno lo scopo di **validare** l'interazione reale tra l'applicazione e il database. Per questo livello è stato utilizzato **Testcontainers**, una libreria che permette di avviare un'istanza di PostgreSQL all'interno di un container Docker per tutta la durata dei test. Questo approccio garantisce un ambiente di test **isolato e riproducibile** eliminando la dipendenza da un database "reale".

4.2.1 package `ORM`

Per ogni classe DAO è stata creata una classe di test dedicata: abbiamo quindi `UserDAOTest`, `LibraryDAOTest`, `BookDAOTest`, `BookCopiesDAOTest`, `LoanDAOTest` e `ReservationDAOTest`. Ognuna di questa classi estende "a cascata" due interfaccie: `OrmIntegrationTestBase`, che a sua volta estende `PostgresIntegrationTestBase`. Quest'ultima rappresenta una classe "Singleton" utilizzata per separare il ciclo di vita del contenitore dal runner JUnit, avviandolo una sola volta e arrestandolo

all'uscita dalla JVM. In questo modo è possibile riutilizzare/condividere il container Docker tra i test. Tornando invece a `OrmIntegrationTestBase`, esso si occupa di istanziare la base di tutte le classi di test di ORM: in questo modo tutte le classi sono annotate con `@JdbcTest`, con `@Transactional` e con `@Testcontainers`. Inoltre la classe si occupa di collegare tramite `@Autowired` tutti i DAO necessari ed il Datasource che sarà utilizzato con testcontainers. Infine si occupa anche della creazione ed istanziazione di oggetti di test, che alcune classi condividono.

Viene definito anche un metodo `executeSchemaScript()` per caricare sul container lo schema del database, **identico** a quello del software "in produzione".

Listing 4.5: Metodo che si occupa di caricare sul database di test lo schema SQL.

```
1    protected void executeSchemaScript() throws SQLException {
2          try (Connection connection = dataSource.getConnection()) {
3              ScriptUtils.executeSqlScript(connection, new ClassPathResource("sql/schema.sql
                   "));
4              System.out.println("Schema loaded successfully using the main test DataSource.
                   ");
5          } catch (Exception e) {
6              throw new RuntimeException("Failed to execute schema script: " + e.getMessage
                   (), e);
7          }
8      }
```

In generale queste classi di test si concentrano esclusivamente sul layer di persistenza. L'obiettivo di questi test è da un lato verificare la correttezza sintattica e semantica delle query SQL eseguite tramite JDBC, mentre dall'altro assicurare che il mapping tra le tabelle del database e gli oggetti Java funzioni correttamente in entrambe le direzioni. Prima di eseguire tutti i metodi di test, ogni classe ha un proprio metodo `setUp()` (preceduto da un'annotazione `@BeforeEach`) per preparare l'ambiente del test.

In `LibraryDAOTest` prendiamo come esempio il test `findLibrariesByName_ShouldBeCaseInsensitive()`, il quale controlla che il metodo `findLibrariesByName` sfrutti correttamente l'operatore `ILIKE` di PostgreSQL per eseguire una ricerca case-insensitive.

Facendo un altro esempio in `LoanDAOTest` il test `findOverdueLoans_ShouldReturnOnlyOverdueLoans()` crea un prestito, ne modifica la data di scadenza per farlo scadere e infine invoca il metodo di ricerca. L'asserzione finale verifica che la query SQL filtri correttamente i risultati, restituendo solo il prestito che soddisfa le condizioni di "prestito scaduto", confermando così la corretta traduzione di una richiesta della business logic in una query per un database.

Listing 4.6: Test di integrazione che verifica la ricerca case-insensitive in LibraryDAO.

```
1  @Test
2  void findLibrariesByName_ShouldBeCaseInsensitive() {
3      createTestLibrary(); // Crea una libreria con nome "Test Library"
4      List<Library> foundLibraries = libraryDAO.findLibrariesByName("test library");
5      assertThat(foundLibraries).hasSize(1);
6      assertThat(foundLibraries.get(0).getName()).isEqualTo("Test Library");
7  }
```

Listing 4.7: Test di integrazione che valida la query per trovare i prestiti scaduti.

```
@Test
void findOverdueLoans_ShouldReturnOnlyOverdueLoans() {

    Loan loan = createTestLoan(testBookCopy, testMember);
    LocalDate pastDueDate = LocalDate.now().minusDays(1);
    loanDAO.updateDueDate(loan.getId(), pastDueDate);

    List<Loan> overdueLoans = loanDAO.findOverdueLoans();

    assertThat(overdueLoans).hasSize(1);
    assertThat(overdueLoans.get(0).getId()).isEqualTo(loan.getId());
    assertThat(overdueLoans.get(0).getDueDate()).isEqualTo(pastDueDate);
}
```

4.3 End-to-End Testing

Infine sono stati sviluppati dei test end-to-end (E2E), progettati con lo scopo di verificare l'intero funzionamento dell'applicazione dal punto di vista dell'utente finale. Attraverso la classe `MainCLITest`, viene simulata l'interazione di un utente con l'interfaccia a riga di comando, assicurando che i flussi completi, i quali coinvolgono tutti i layer dell'architettura, funzionino correttamente.

Anche in questo caso, viene utilizzato Testcontainers per fornire un ambiente di database realistico e popolato con dati di default. L'approccio del test consiste nel reindirizzare gli stream di input e output standard di Java: `System.in` viene sostituito con metodo `provideInput()` che simulano la digitazione dell'utente, mentre `System.out` e `System.err` vengono catturati con un'altro metodo `getOutput()` per analizzare l'output della console, ritrasformandolo in stringa.

È importante sottolineare come sia stato utilizzato un oggetto annotato con `@MockitoBean` per evitare che Spring utilizzasse subito `CommandLineRunner`, che altrimenti impediva la riuscita dei test.

Listing 4.8: Metodo per simulare l'input di un utente.

```
    private void provideInput(String data) {
        ByteArrayInputStream testIn = new ByteArrayInputStream(data.getBytes(
            StandardCharsets.UTF_8));
        System.setIn(testIn);
    }
```

Prendendo come esempio il test `cli_adminViewBookDetails_succeeds`, esso verifica l'intero processo: il funzionamento del `LoginController` per l'autenticazione, la capacità della `MainCLI` di interpretare l'input e invocare l'`AdminBookController`, l'interazione di quest'ultimo con il `BookDAO` ed infine la corretta formattazione dell'output sulla console. Questo garantisce che il sistema funzioni in modo coeso.

Listing 4.9: Test E2E che simula il login di un admin e la visualizzazione dei dettagli di un libro.

```java
@Test
void cli_adminViewBookDetails_succeeds() {
    // Given: una sequenza di input che simula le azioni dell'utente
    String input = "1\n" +          // 1. Login
            ADMIN_EMAIL + "\n" +
            PASSWORD + "\n" +
            "8\n" +          // 8. View Book Details
            GATSBY_ISBN + "\n" +
            "17\n" +         // 17. Logout
            "3\n";          // 3. Exit
    provideInput(input);

    // When: l'applicazione CLI viene eseguita
    cliTestInstance.run();

    // Then: si verifica che l'output contenga i messaggi chiave del flusso
    String output = getOutput();
    assertThat(output).contains("Login successful: welcome Admin One!");
    assertThat(output).contains("--- Book Details ---");
    assertThat(output).contains("The Great Gatsby");
}
```

5 Librerie di terze parti

Di seguito è riportata la lista delle principali librerie di terze parti utilizzate per lo sviluppo dell'applicazione.

- **Spring Boot** - Framework principale per la creazione, la configurazione e l'esecuzione dell'applicazione stand-alone.

  https://spring.io/projects/spring-boot

- **Spring Framework** - Fornisce i moduli fondamentali per la Dependency Injection (`@Autowired`), la gestione delle transazioni (`@Transactional`) e la sicurezza (`BCryptPasswordEncoder`).

  https://spring.io/projects/spring-framework

- **Project Lombok** - Utilizzato per ridurre il codice boilerplate nelle classi di dominio tramite annotazioni (es. `@Data`, `@Getter`, `@Setter`).

  https://projectlombok.org/

- **PostgreSQL JDBC Driver** - Driver che implementa l'API JDBC per consentire la comunicazione con il database PostgreSQL.

  https://jdbc.postgresql.org/

- **JUnit 5** - Framework per l'implementazione e l'esecuzione dei test unitari e di integrazione del progetto.

  https://junit.org/junit5/

- **Mockito** - Framework utilizzato per la creazione di oggetti mock e stub durante i test unitari, per isolare i componenti da testare.

  https://site.mockito.org/

- **Testcontainers** - Libreria per la gestione di container Docker (in questo caso, PostgreSQL) durante i test di integrazione, garantendo un ambiente di test pulito e riproducibile.

  https://www.testcontainers.org/

5.1 Strumenti di Sviluppo

Oltre alle librerie software, per la progettazione, lo sviluppo e la gestione del progetto sono stati utilizzati i seguenti strumenti.

- **IntelliJ IDEA** - Ambiente di Sviluppo Integrato (IDE) utilizzato per la scrittura e il debug del codice Java.

  https://www.jetbrains.com/idea/

- **Visual Paradigm** - Software utilizzato per la modellazione UML.

  https://www.visual-paradigm.com/

- **StarUML** - Altro software utilizzato per la modellazione degli usecase e per la creazione dei diagrammi architetturali.

  https://www.staruml.com

- **DBeaver** - Strumento client per l'interazione e la gestione del database PostgreSQL.

  https://dbeaver.io/

- **GitHub** - Piattaforma per il versioning e la gestione del codice sorgente tramite Git.

  https://github.com/

- **dbdiagram-oss-wrep**: software per la creazione del diagramma ER.

  https://github.com/NomadRazor/dbdiagram-oss-wrep

- **draw.io**: altro software utilizzato per la creazione di un diagramma ER.

  https://draw.io