UNIVERSITÀ DEGLI STUDI FIRENZE

UNIVERSITÀ DEGLI STUDI DI FIRENZE
Dipartimento di Ingegneria dell'Informazione

Biblioteca Universitaria

Software per la gestione di una rete di biblioteche

Autori:

Filippo Taiti
Hui Tao Hu

N° Matricola:

7114729
7110925

Corso:

Ingegneria del Software

Docente:

Enrico Vicario

Contents

1 Introduzione

1.1 Statement

Il progetto consiste in un software di gestione di una rete di biblioteche universitarie.
Il sistema permette di accedere al catalogo (unico per l'intera rete di biblioteche) che contiene articoli di diversa natura sia in forma fisica che digitale.
Per la prenotazione di articoli l'utente deve autenticarsi con le proprie credenziali: se appartenente all'ateneo, effettua l'accesso attraverso il sistema di autenticazione universitario, altrimenti attraverso il sistema di autenticazione bibliotecario, dovendo essersi registrato presso una delle sedi.
Gli amministratori possono effettuare registrazioni e ricerche di utenti, aggiunta, modifica, cancellazione di articoli nel catalogo, conferme del ritiro di articoli prenotati, registrazioni di prestiti e restituzione di articoli.

1.2 Struttura del progetto

Il software è stato sviluppato in Java. I dati vengono salvati e gestiti attraverso un database relazionale su PostgreSQL utilizzando la libreria JDBC (Java DataBase Connectivity).

La struttura del progetto è suddivisa in tre package principali: Domain Model, Business Logic e ORM:

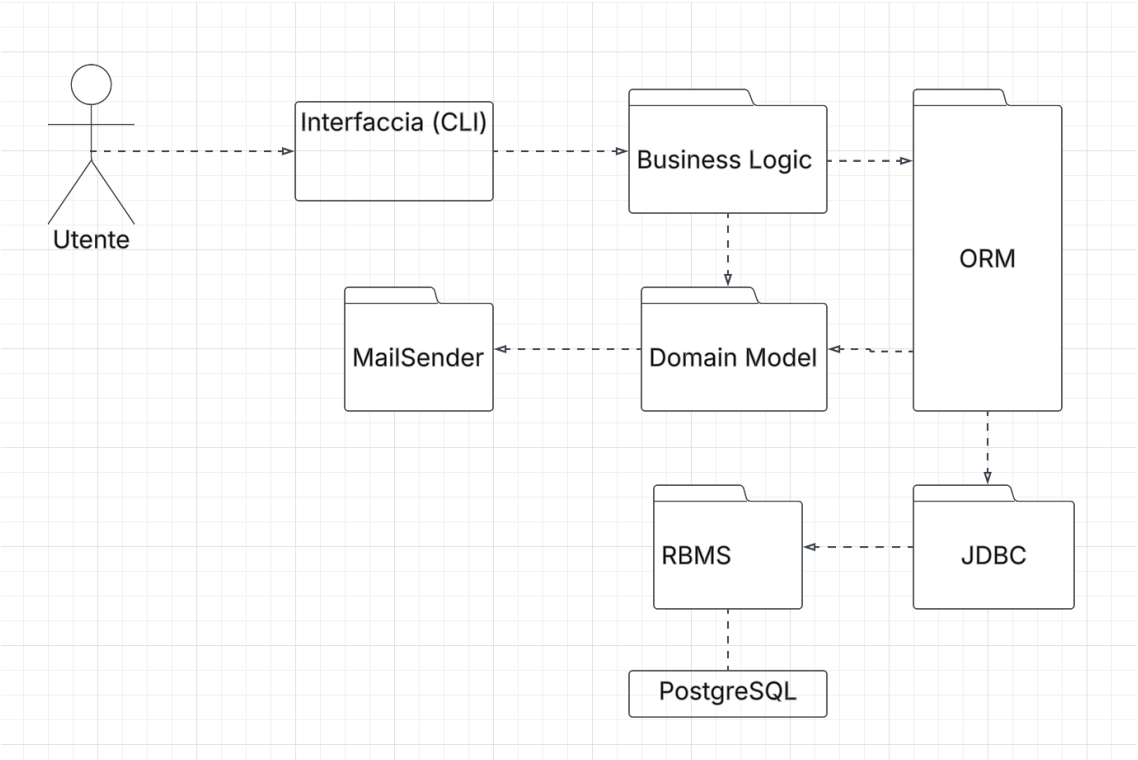
- **Domain Model:** rappresentazione delle entità del dominio bibliotecario;
- **Business Logic:** logica di esecuzione delle funzionalità del sistema;
- **ORM:** implementazione dell'Object-Relational-Mapping al fine di garantire la persistenza dei dati ed il loro recupero dal database.

Inoltre sono stati sviluppati:

- **Templates:** tabelle che specificano le funzionalità del software;
- **Mockups:** prototipi dell'interfaccia di utilizzo;
- **CLI:** interfaccia a linea di comando che permette l'interazione tra utente e software.

L'architettura del software prevede che l'utente, tramite la CLI, richiami i metodi della Business Logic eseguiti sugli oggetti del Domain Model creati dall'ORM partendo dai dati all'interno di un RDBMS utilizzando JDBC.

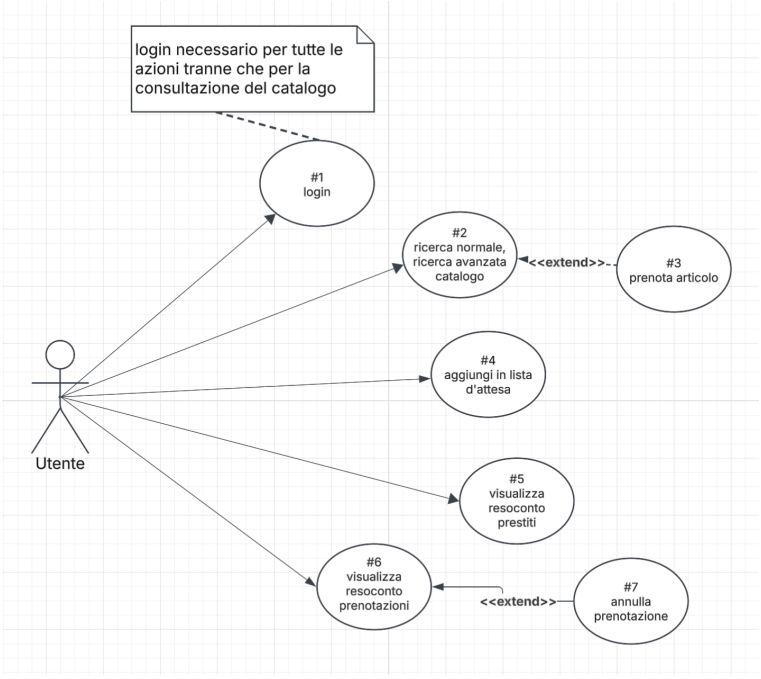
Inoltre è presente anche un MailSender che permette di inviare notifiche agli utenti.

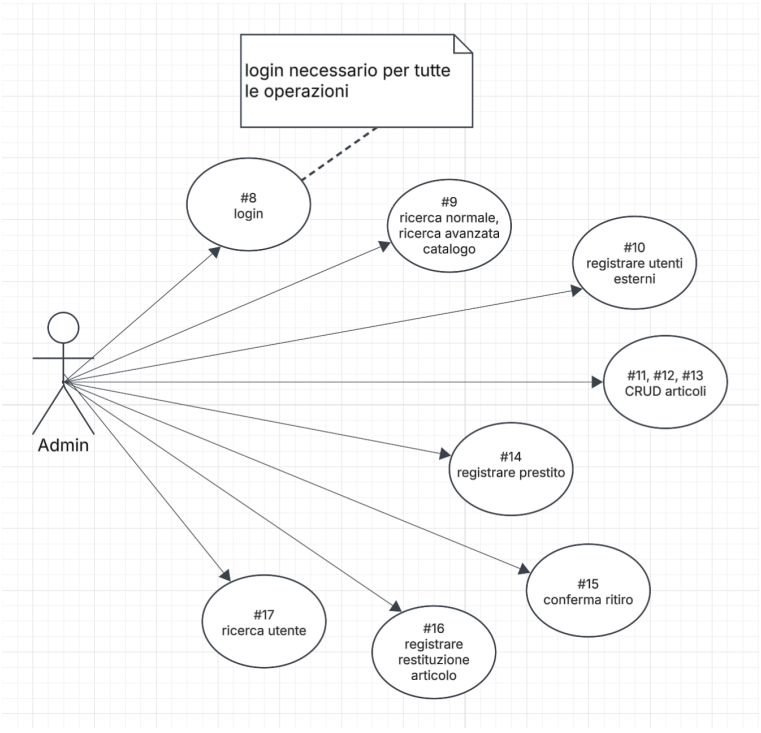


2 Progettazione

I Mockups sono stati realizzati con Balsamiq Wireframes mentre il resto dei diagrammi con Lucidchart.

2.1 Use Case Diagram





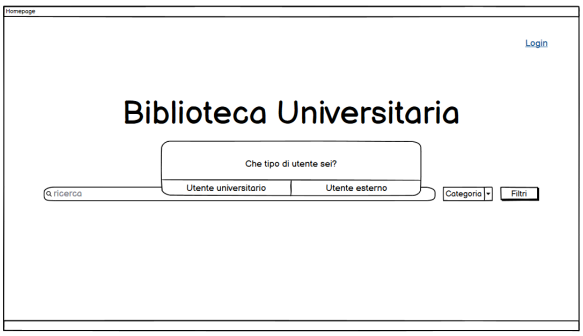
2.2 Templates e Mockups

In questa sezione verranno riportati i Templates del progetto accompagnati dai Mockups per una visualizzazione più concreta.
Nei Templates più importanti saranno riportati dei riferimenti ai test, mentre i restanti sono consultabili accedendo al link di Github disponibile nell'ultima sezione.

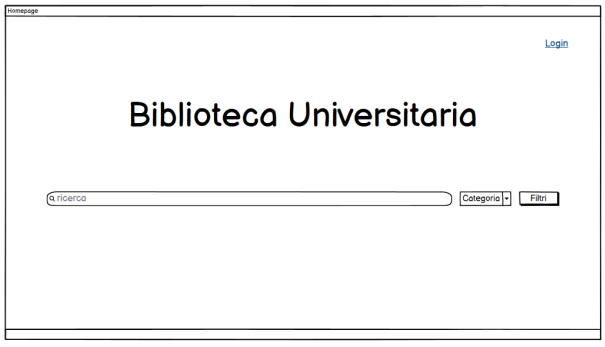
2.2.1 Templates relativi ad Hirer

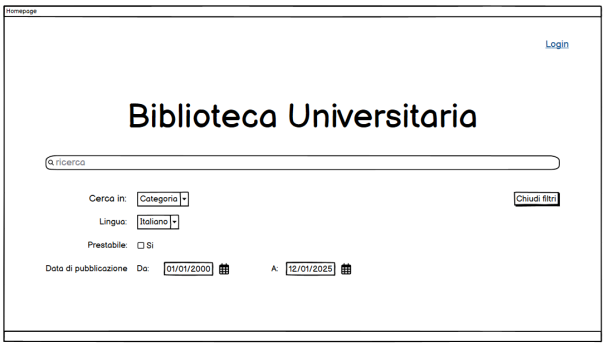
Questo primo template, nonostante sia riportato in questa sezione, in realtà è relativo sia a Hirer che ad Admin. Non verrà riportato di nuovo nella sezione dedicata ai templates relativi ad Admin per evitare ridondanza.

Use Case #1, #8	Login
Brief Description	L'utente accede al sito tramite credenziali per identificarsi ed ottenere l'autorizzazione per i servizi.
Level	Function
Actors	Admin, Utente
Pre-Conditions	L'utente deve trovarsi in una pagina in cui è presente il bottone di login.
Basic Flow	1. L'utente/admin seleziona "Login" nella pagina home; 2. L'utente/admin sceglie tra autenticazione tramite sistema universitario oppure tramite credenziali per utenti esterni autorizzati (Mockup #3); 3. L'utente/admin inserisce le proprie credenziali (Mockups #4-#6); 4. L'utente/admin invia le proprie credenziali; 5. Se l'utente/admin aveva scelto di autenticarsi tramite sistema universitario ne viene controllata l'esistenza nel database dell'ateneo; 5bis. Se l'utente aveva scelto di autenticarsi tramite credenziali per utenti esterni ne viene controllata nel database della biblioteca; 6. L'utente/admin accede al suo account. Test Login Admin, Test Login External Hirer
Alternative Flow	4bis. Se le credenziali inserite sono errate, comparirà una schermata che informerà l'utente/admin dell'errore. 5ter. Se è il primo login, avviene una registrazione automatica nel database bibliotecario.
Post-Conditions	L'utente/admin può accedere ai servizi ai quali è interessato.



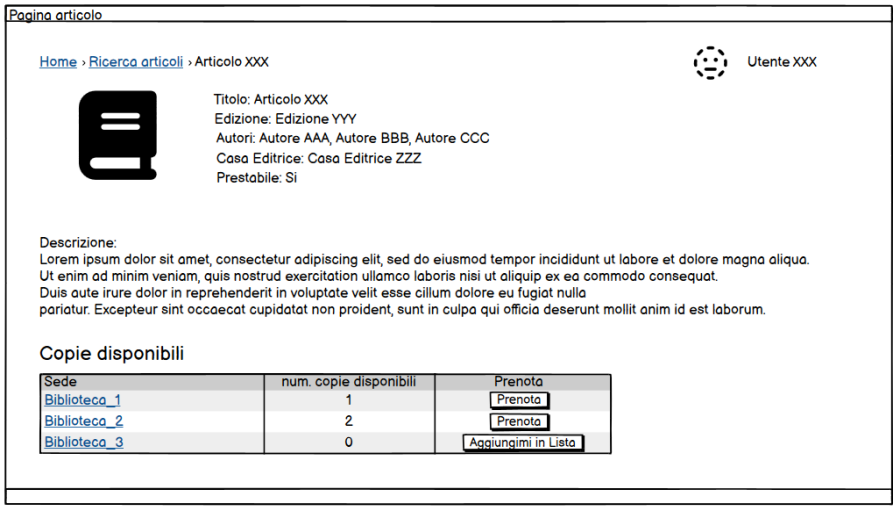
Use Case #2	Ricerca catalogo
Brief Description	L'utente consulta il catalogo della biblioteca.
Level	User Goal
Actors	Utente
Pre-Conditions	L'utente si trova nella pagina contenente la barra di ricerca.
Basic Flow	1. L'utente inserisce il contenuto che vuole cercare facendo una ricerca per parole chiavi; 2. L'utente clicca "cerca". Test Ricerca Articolo
Alternative Flow	1bis. L'utente inserisce il contenuto che vuole cercare facendo una ricerca per parole chiavi avanzata; 2bis. L'utente applica dei filtri del tipo "per categoria", "per lingua", "per data di pubblicazione" e "noleggiabile";
Post-Conditions	L'utente, eventualmente, ha trovato l'articolo che cercava e può decidere di prenotarlo.





Use Case #3	Prenota articolo
Brief Description	L'utente seleziona l'articolo di interesse e lo prenota.
Level	User Goal
Actors	Utente
Pre-Conditions	L'utente deve aver effettuato il login con successo ed essere nella pagina in cui è presente l'articolo.

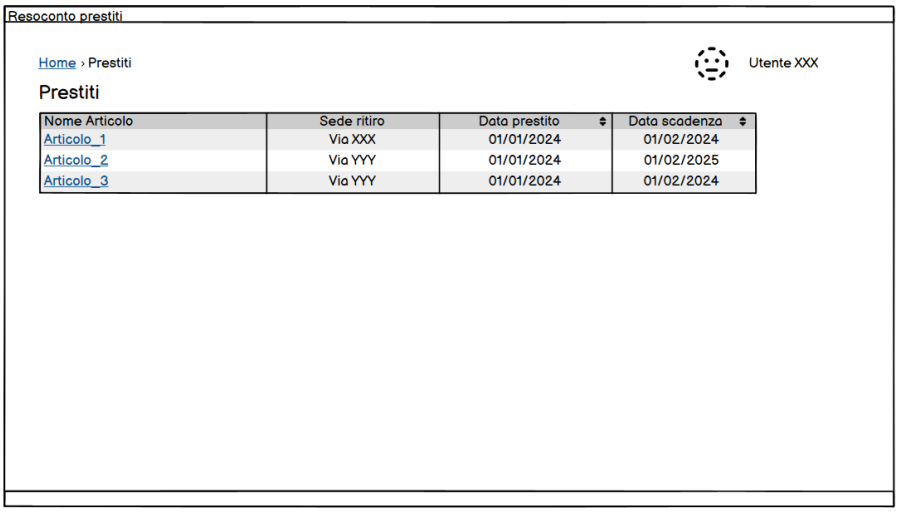
Basic Flow	1. L'utente clicca su "prenota" accanto alla sede di suo gradimento che ha l'articolo disponibile; 2. La prenotazione viene registrata nel database. Test Reserve Item
Alternative Flow	2bis. Il software mostra un messaggio di errore in quanto l'utente è bloccato.
Post-Conditions	1. L'utente possiede una nuova prenotazione.



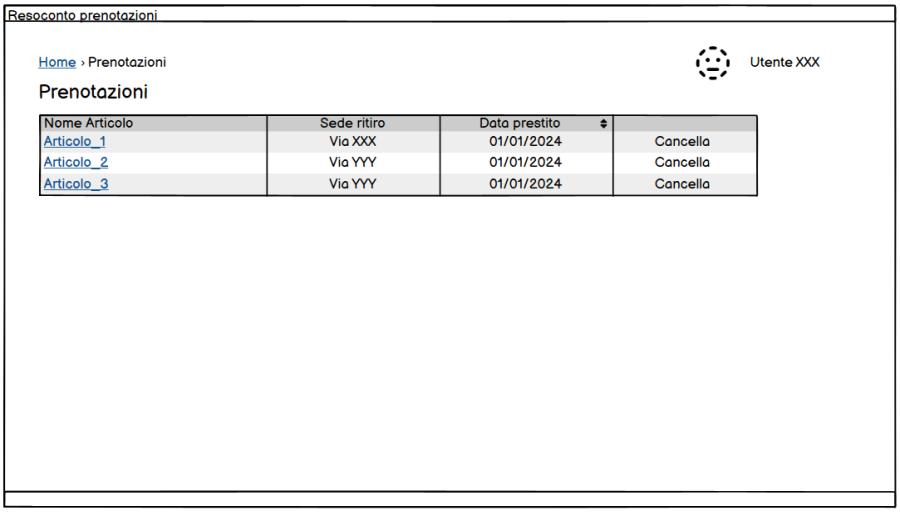
Use Case #4	Aggiungi in lista di attesa
Brief Description	Un utente viene aggiunto nella lista di attesa relativa a un articolo.
Level	Function
Actors	Hirer
Pre-Conditions	L'utente ha effettuato il login e si trova nella pagina di un articolo.
Basic Flow	1. L'utente clicca su "Aggiungimi in lista". Test Aggiungi In Lista Di Attesa
Post-Conditions	L'utente è stato aggiunto alla lista di attesa e verrà notificato (e rimosso da essa) quando l'articolo tornerà disponibile.

Use Case #5	Visualizza resoconto prestiti
Brief Description	L'utente visualizza il resoconto dei suoi prestiti.
Level	User Goal
Actors	Utente
Pre-Conditions	L'utente deve aver effettuato il login con successo e deve trovarsi sul suo profilo.
Basic Flow	1. L'utente clicca su "Resoconto Prestiti".

Post-Conditions	L'utente visualizza i dettagli dei suoi prestiti.



Use Case #6	Visualizza resoconto prenotazioni
Brief Description	L'utente visualizza il resoconto delle sue prenotazioni.
Level	User Goal
Actors	Utente
Pre-Conditions	L'utente deve aver effettuato il login con successo e deve trovarsi sul suo profilo.
Basic Flow	1. L'utente clicca su "Resoconto Prenotazioni".
Post-Conditions	L'utente visualizza i dettagli delle sue prenotazioni.

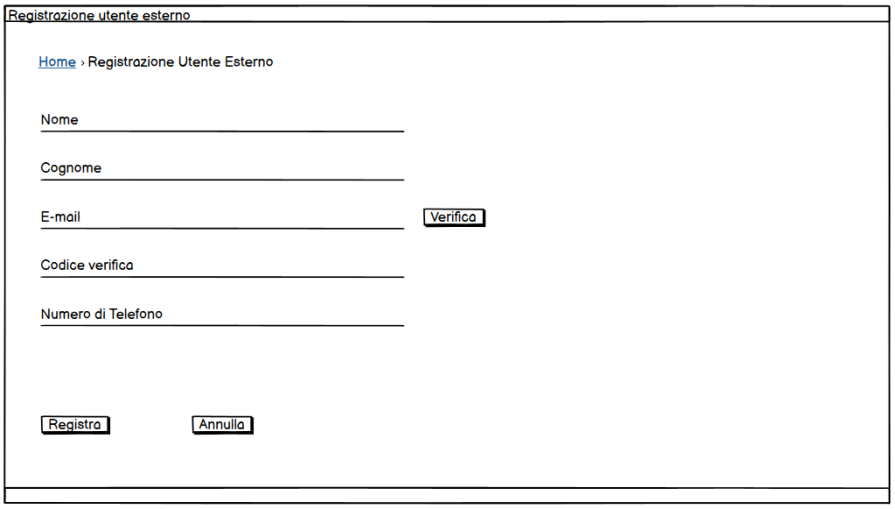


Use Case #7	Cancellazione di una prenotazione
Brief Description	Un utente cancella una prenotazione che ha effettuato.
Level	User Goal
Actors	Utente
Pre-Conditions	L'utente deve aver effettuato il login e deve trovarsi in "Visualizza resoconto prenotazioni".

Basic Flow	1. L'utente seleziona una delle prenotazioni; 2. L'utente clicca su "Annulla prenotazione". 3. Gli utenti che si trovano nella lista di attesa di tale articolo vengono notificati (e rimossi dalla lista). Test Cancellazione Prenotazione
Post-Conditions	La prenotazione selezionata dall'utente è stata cancellata.

2.2.2 Templates relativi ad Admin

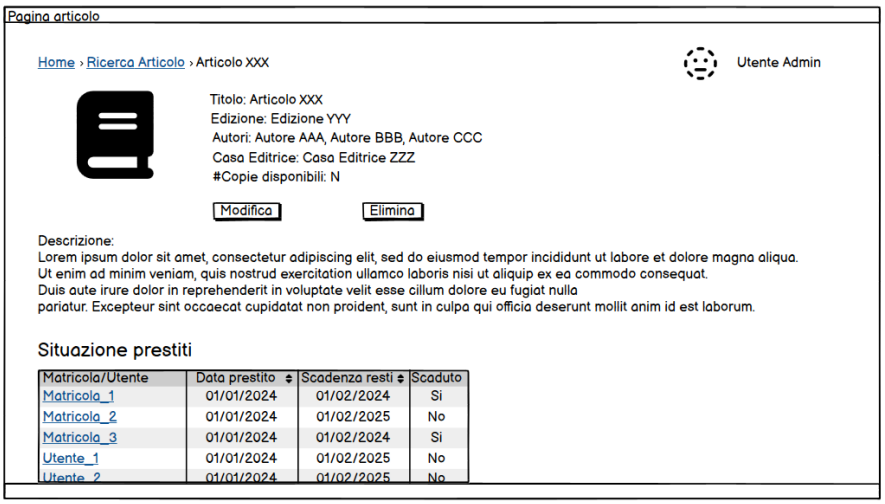
Use Case #10	Registrare utenti esterni
Brief Description	L'admin registra un utente esterno non presente nel database degli utenti esterni della biblioteca.
Level	User Goal
Actors	Admin
Pre-Conditions	L'admin ha effettuato il login e si trova sulla sua homepage.
Basic Flow	1. L'admin seleziona "Registrazione utente esterno"; 2. L'admin compila i dati dell'utente esterno e controlla la validità dell'email; 3. L'admin preme "Registra"; 4. Le credenziali per l'accesso vengono mandate all'e-mail fornita in fase di registrazione. Test Registrazione Utenti Esterni
Post-Conditions	L'utente esterno è registrato e può accedere al suo profilo personale.



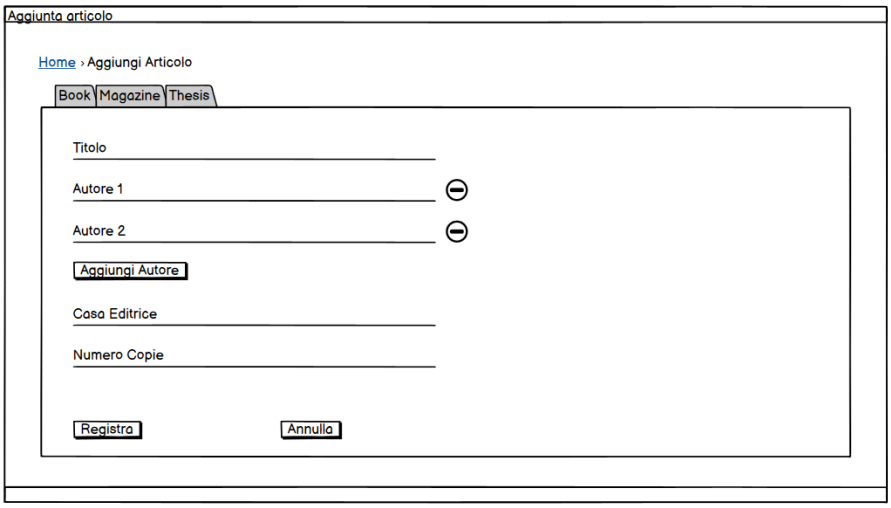
Use Case #11	Aggiunta di un articolo
Brief Description	Un admin accede al catalogo per aggiungere un articolo
Level	User Goal
Actors	Admin
Pre-Conditions	L'admin ha effettuato il login e si trova sulla sua homepage.
Basic Flow	1. L'admin clicca sul bottone "Aggiungi Articolo"; 2. L'admin decide che tipo articolo aggiungere; 3. L'admin inserisce i dati inerenti al nuovo articolo; 4. L'admin salva le modifiche cliccando su "Registra". Test Aggiunta Articolo
Alternative Flow	3bis. Se l'articolo è già presente nel catalogo viene aggiornato solo il numero di copie.
Post-Conditions	Nel catalogo è presente il nuovo articolo aggiunto.

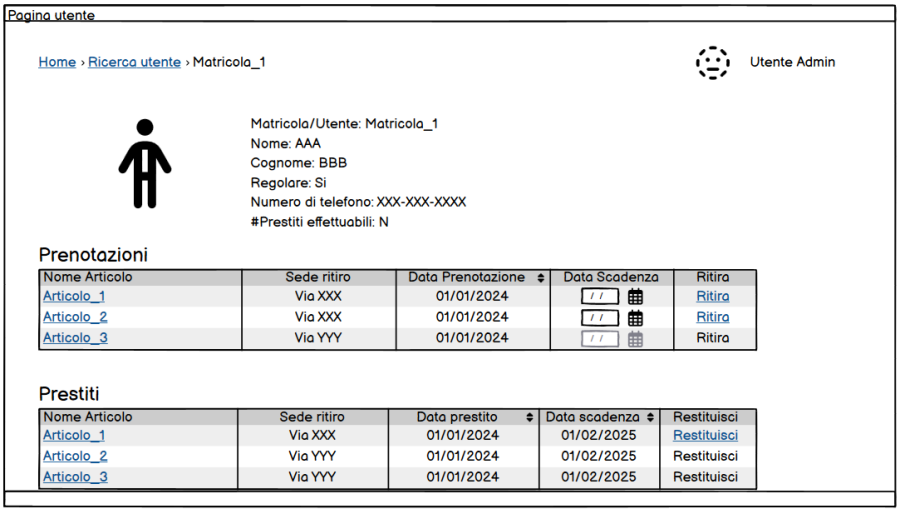
Use Case #12	Modifica di un articolo
Brief Description	Un admin accede al catalogo per aggiornare alcune informazioni di un articolo.
Level	User Goal

Actors	Admin
Pre-Conditions	L'admin ha effettuato il login e si trova sulla sua homepage.
Basic Flow	1. L'Admin esegue la ricerca di un articolo; 2. L'admin seleziona l'articolo; 3. L'admin clicca su "Modifica"; 4. L'admin modifica le informazioni interessate; 5. L'admin salva le modifiche applicate. Test Modifica Articolo
Post-Conditions	Le informazioni inerenti a quell'articolo nel catalogo sono state aggiornate.



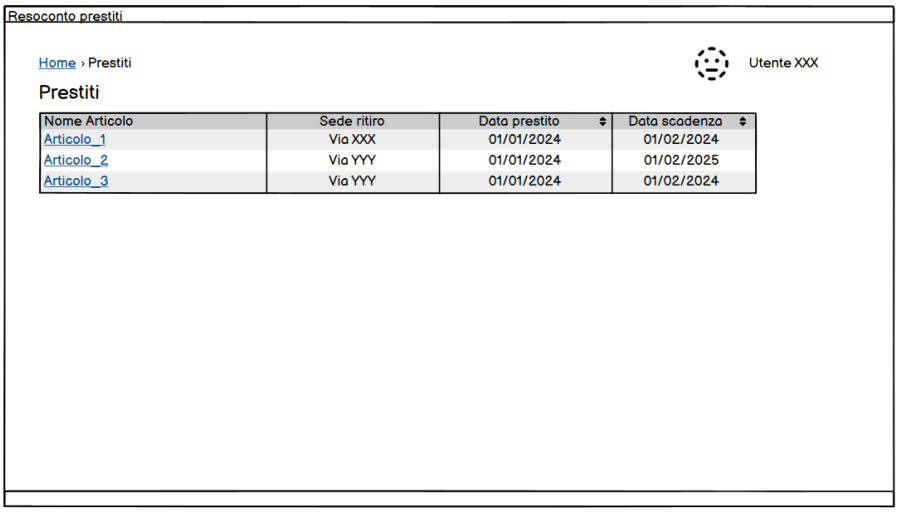
Use Case #13	Cancellazione di un articolo
Brief Description	Un admin accede al catalogo per eliminare un articolo.
Level	User Goal
Actors	Admin
Pre-Conditions	L'admin ha effettuato il login e si trova sulla sua homepage.
Basic Flow	1. L'Admin esegue la ricerca di un articolo; 2. L'admin seleziona l'articolo; 3. L'admin clicca su "Elimina"; 4. L'admin conferma l'operazione. Test Cancellazione Articolo
Alternative Flow	4bis. L'admin riceve un messaggio di errore in quanto ci sono prestiti o prenotazioni attive legate all'articolo. 4ter. Se l'articolo ha una versione digitale o è presente in altre sedi, l'articolo non viene cancellato completamente, ma solamente le copie fisiche nella sede in cui l'admin lavora.
Post-Conditions	Nel catalogo non è più presente l'articolo appena cancellato.





Use Case #14	Registrare prestito
Brief Description	Un admin registra il prestito di un utente.
Level	User Goal
Actors	Admin
Pre-Conditions	L'admin deve aver fatto l'accesso e deve trovarsi nella home page L'utente deve essere registrato nel database.
Basic Flow	1. L'admin inserisce il codice dell'articolo che l'utente vuole prendere in prestito, il codice dell'utente e la sede del ritiro; 2. L'admin registra la scadenza del prestito dell'articolo cliccando su "Registra"; 3. L'utente riceve una notifica. Test Registrazione Prestito
Alternative Flow	2bis. Se l'utente è in un periodo di penalità o l'articolo non è noleggiabile oppure è disponibile una sola copia, viene segnalato un errore e non viene registrato il prestito.

Post-Conditions	L'articolo compare nella lista dei prestiti dell'utente.

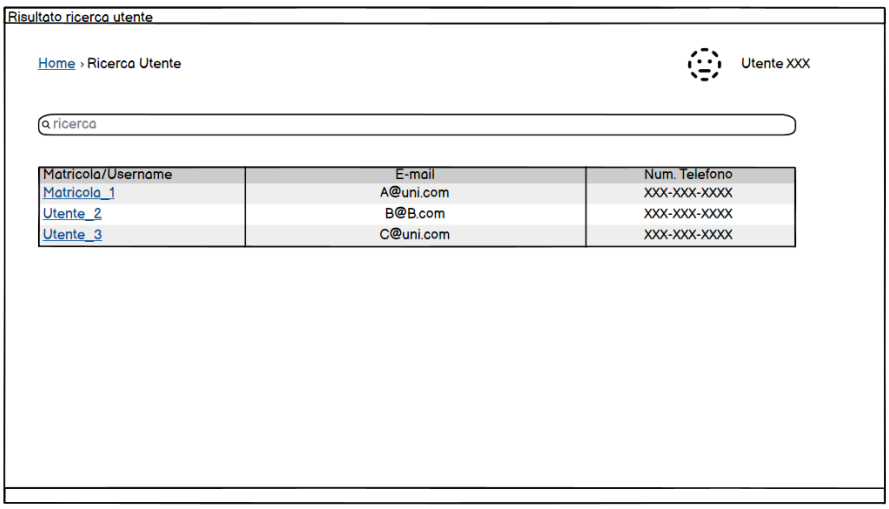


Use Case #15	Conferma ritiro
Brief Description	Un admin conferma il ritiro dell'articolo da parte di un utente con prenotazione.
Level	User Goal
Actors	Admin
Pre-Conditions	L'admin ha effettuato il login e si trova sul suo profilo. L'utente deve essere registrato nel database e deve aver effettuato una prenotazione.
Basic Flow	1. L'admin visualizza la lista di prenotazioni dell'utente. 2. L'admin conferma il prestito dell'articolo cliccando su "Ritira". Test Conferma Ritiro
Alternative Flow	2bis. Se l'utente è in un periodo di penalità viene segnalato e non viene registrato il prestito.
Post-Conditions	La prenotazione dell'articolo viene eliminata e l'articolo compare nella lista dei prestiti.

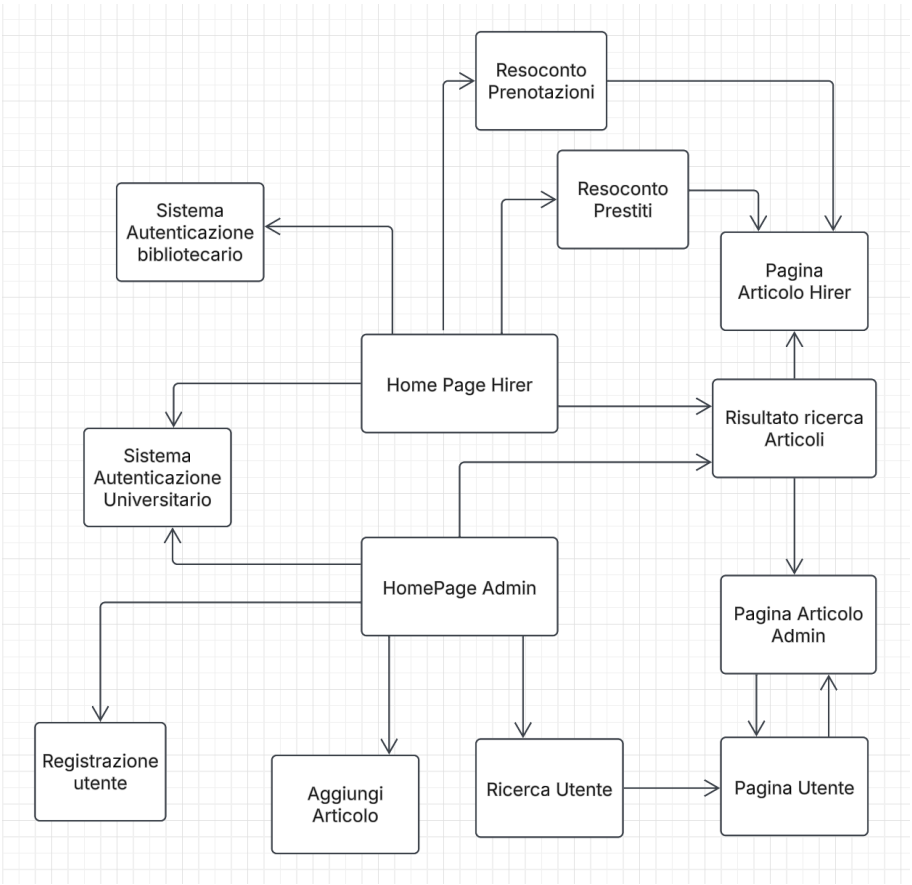
Use Case #16	Registrare restituzione articolo
Brief Description	Un admin registra la restituzione di un articolo da parte di un utente.
Level	User Goal
Actors	Admin
Pre-Conditions	L'admin deve aver fatto il login e deve trovarsi nella pagina di ricerca di un utente. L'utente deve essere registrato nel database e deve aver effettuato un prestito.

Basic Flow	1. L'admin visualizza la lista di prestiti dell'utente. 2. L'admin conferma la restituzione dell'articolo cliccando su "Restituisci". 3. L'utente riceve una mail che attesta l'avvenuta restituzione dell'articolo. 4. Gli utenti nella lista di attesa dell'articolo vengono notificati (e rimossi da essa). Test Restituzione Articolo
Post-Conditions	L'articolo preso in prestito non compare più nella lista di articoli presi in prestito da questo utente.

Use Case #17	Ricerca utente
Brief Description	Un Admin accede al catalogo per ottenere le informazioni di un utente.
Level	User Goal
Actors	Admin
Pre-Conditions	L'admin ha effettuato il login e si trova sulla sua homepage.
Basic Flow	1. L'admin clicca su "Ricerca Utente"; 2. L'admin inserisce i dati relativi all'utente; 3. L'admin clicca "cerca". Test Ricerca Utente
Post-Conditions	Nella schermata vengono visualizzati la matricola, la mail e il numero di telefono dell'utente cercato.

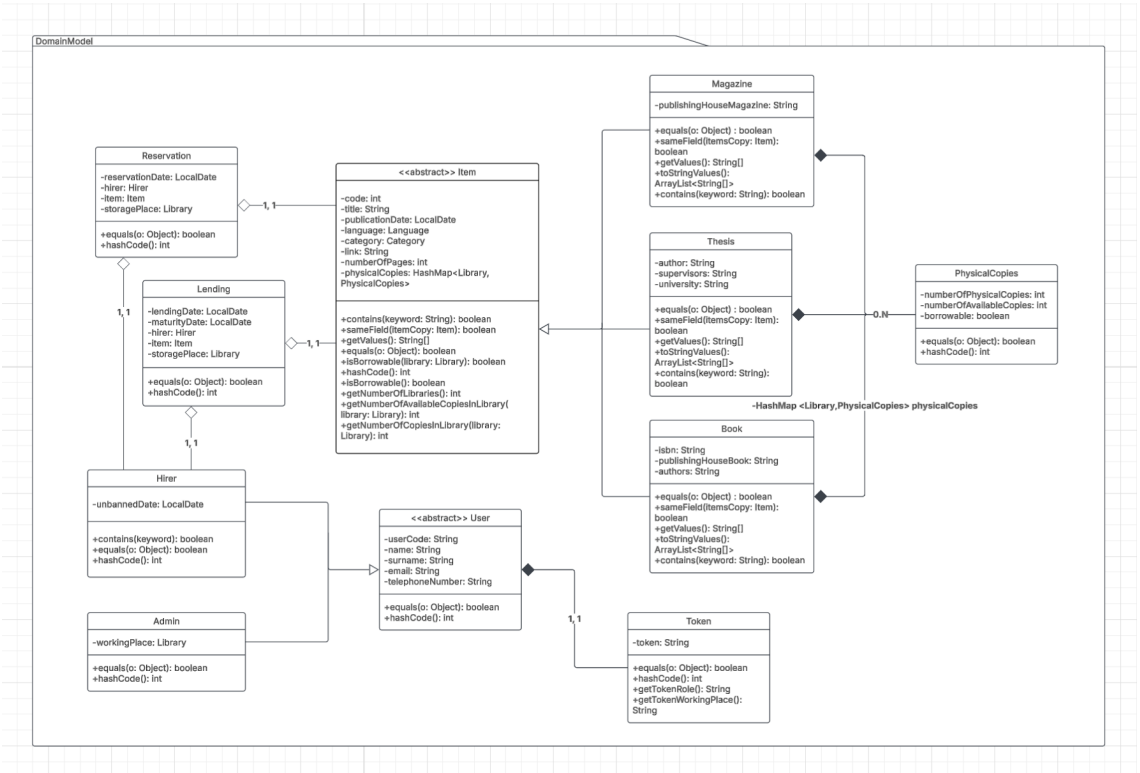


2.3 Navigation Diagram

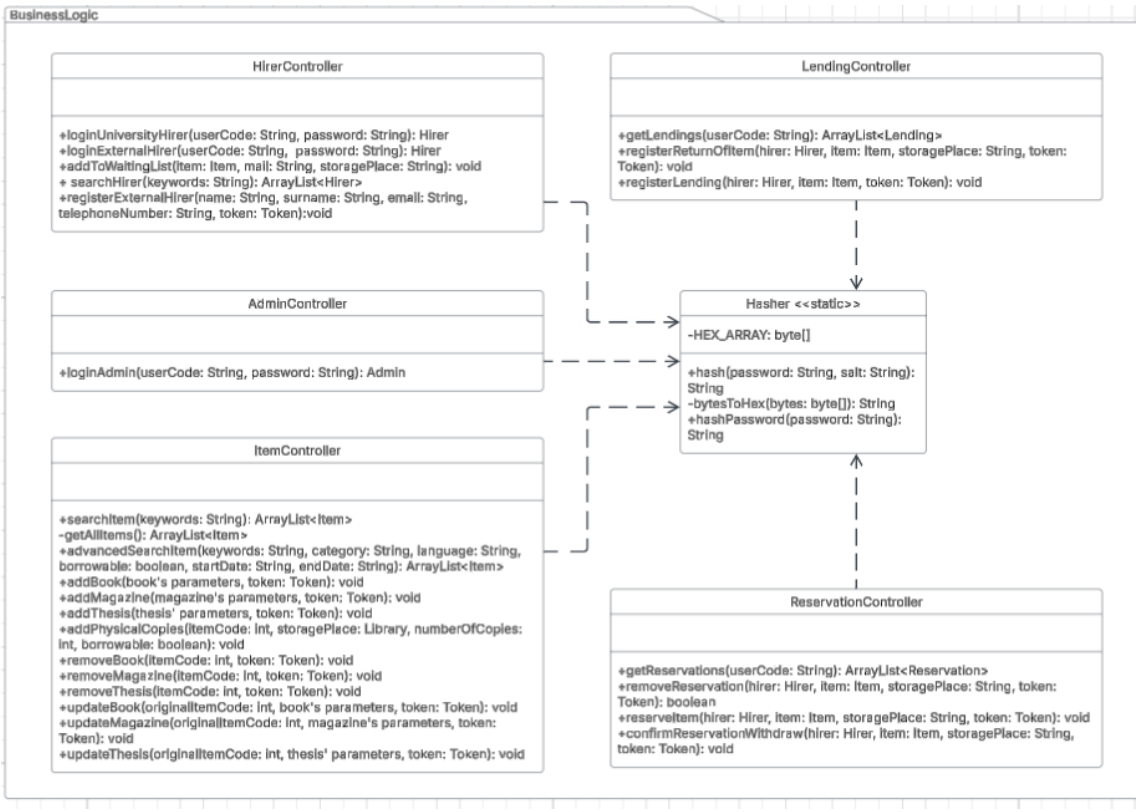


2.4 Class Diagram

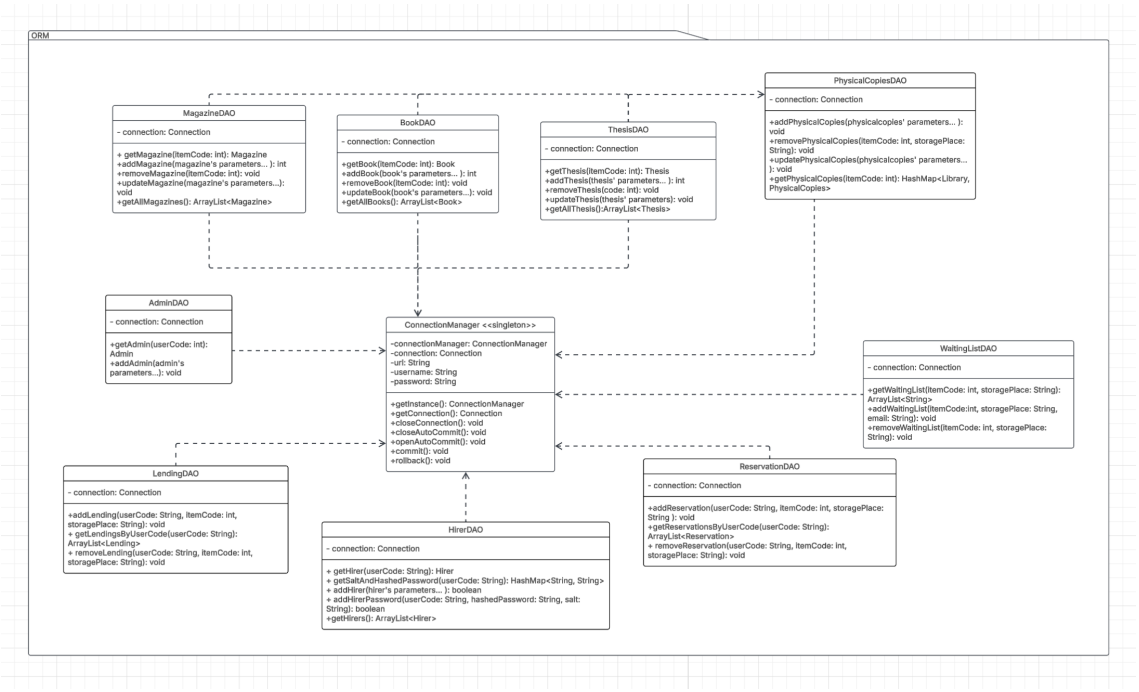
2.4.1 Domain Model



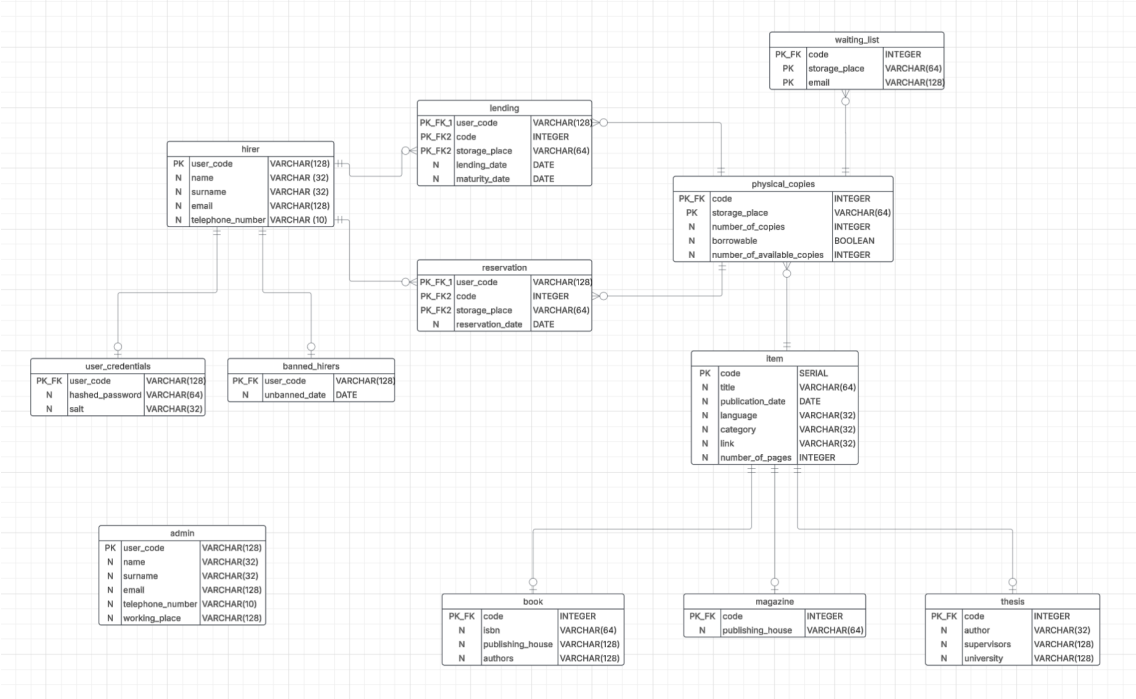
2.4.2 Business Logic



2.4.3 ORM



2.5 Diagramma ER e Modello Relazionale



Admin(<u>user_code</u>, name, surname, email, telephone_number, working_place)

Hirer(<u>user_code</u>, name, surname, email, telephone_number)

Banned_hirers(<u>user_code</u>, unbanned_date)
user_code → *Hirer*(user_code)

Item(<u>code</u>, title, publication_date, language, category, link, number_of_pages)

Book(<u>code</u>, isbn, publishing_house, authors)
code → *Item*(code)

Magazine(<u>code</u>, publishing_house)
code → *Item*(code)

Thesis(<u>code</u>, author, supervisors, university)
code → *Item*(code)

Physical_copies(<u>code</u>, <u>storage_place</u>, number_of_copies, borrowable, number_of_available_copies)
code → *Item*(code)

Lending(<u>code</u>, <u>storage_place</u>, <u>user_code</u>, lending_date, maturity_date)
(code, storage_place) → *Physical_copies*(code, storage_place)
user_code → *Hirer*(user_code)

Reservation(<u>code</u>, <u>storage_place</u>, <u>user_code</u>, reservation_date)
(code, storage_place) → *Physical_copies*(code, storage_place)
user_code → *Hirer*(user_code)

User_credentials(<u>user_code</u>, hashed_password, salt)
user_code → *Hirer*(user_code)

Waiting_list(<u>code</u>, storage_place, <u>email</u>)
code → *PhysicalCopies*(code)

3 Implementazione

3.1 Domain Model

Il *Domain Model* è composto da diverse classi, ognuna delle quali rappresenta un'entità logica del sistema:

Library, Language, Category
Queste entità sono definite tramite enumerazioni, utili a rappresentare valori predefiniti e ad aiutare durante la ricerca di un articolo.

User
Classe base astratta che rappresenta un generico utente del sistema. Ogni istanza possiede un identificativo univoco, nome, cognome, email, numero di telefono e un riferimento a un oggetto `Token`. Le classi `Admin` e `Hirer` derivano da essa.

Admin
Rappresenta un amministratore bibliotecario. Deriva dalla classe `User` aggiungendo l'attributo `workingPlace`, che indica la sede bibliotecaria di competenza.

Hirer
Anch'essa derivata da `User`, rappresenta un utente che può essere temporaneamente bloccato. L'attributo `unbannedDate` specifica la data di sblocco. Nel database è presente una tabella che tiene traccia degli utenti attualmente sospesi e una tabella che contiene le credenziali degli utenti esterni.

Item
Classe astratta che rappresenta un articolo bibliotecario. Contiene attributi comuni come codice identificativo, titolo, numero di pagine, data di pubblicazione, lingua, categoria, link, e una mappa tra sedi bibliotecarie e le relative copie fisiche (`PhysicalCopies`). Le sottoclassi concrete sono `Book`, `Magazine` e `Thesis`.

Book
Deriva da `Item` e aggiunge gli attributi `ISBN`, casa editrice e autori.

Magazine
Deriva da `Item` aggiungendo la casa editrice.

Thesis
Anch'essa deriva da `Item`, aggiungendo autore, relatore (o relatori) e nome dell'università.

PhysicalCopies
Classe che rappresenta le copie fisiche di un articolo in una specifica sede. Include il numero totale di copie, quelle disponibili per il prestito e un flag che indica la noleggiabilità.

Lending
Rappresenta un prestito bibliotecario. Gli attributi principali sono le date di inizio e fine, l'utente, l'articolo e la sede in cui è stato effettuato il prestito.

Reservation
Rappresenta una prenotazione di un articolo. Contiene la data della prenotazione, l'utente, l'articolo e la sede bibliotecaria associati.

Token
Classe utilizzata per la gestione dei permessi e dei privilegi legati al ruolo dell'utente.

In tutte le classi sono stati ridefiniti i metodi `equals` e `hashCode` per semplificare la fase di test, mentre nelle classi derivate da `Item` è stato introdotto anche un metodo `toStringValues`, utile a fornire una rappresentazione leggibile dell'oggetto.

3.2 Business Logic

La Business Logic è composta da controller il cui compito è effettuare operazioni sui dati della classe correlata.

AdminController:

Il metodo `loginAdmin` esegue il controllo che l'`Admin` sia presente all'interno del database dell'università. Successivamente si controlla che la password inserita dall'utente sia corretta. Se l'`Admin` effettua il login per la prima volta, viene aggiunto nel database della biblioteca. Viene restituita l'istanza dell'admin.

```
public Admin loginAdmin(String userCode, String password)
       throws AccessDeniedException {
```

```
    UniversityAuthenticationSystem authenticationSystem = new UniversityAuthenticationSystem();
    //ottengo informazioni dal database universitario se password corretta
    HashMap<String, String> adminInfo = authenticationSystem.getLibraryAdmin(userCode, password);

    if (adminInfo.isEmpty()) { //password errata
        throw new AccessDeniedException("Errore: accesso come ruolo Admin rifiutato, controlla userCode
            e password.");
    }

    //controllo se presente in database bibliotecario
    AdminDAO adminDAO = new AdminDAO();
    Admin admin;
    try{
        admin = adminDAO.getAdmin(userCode);
    }catch(IdNotFoundException e){ //registrazione automatica
        adminDAO.addAdmin(...parametri Admin...);
        admin = new Admin(...parametri Admin...);
    }
    admin.setToken(new Token(admin));
    return admin;
}
```

La classe UniversityAuthenticationSystem è il sistema di autenticazione universitario: `getLibraryAdmin` effettua un controllo delle credenziali inviate: se queste sono corrette, vengono ritornati i dati dell'admin utili per la registrazione automatica durante il primo login, altrimenti viene restituita un'HashMap vuota che causa il lancio dell'eccezione di `AccessDeniedException`.

Nella Business Logic è presente anche una classe Hasher che viene utilizzata per aggiungere un `salt` e poi crittografare tramite `hash` le password per evitare il passaggio e la memorizzazione diretta della password e garantirne l'univocità.

HirerController: classe che gestisce: login di un hirer interno all'università (la cui logica è analoga a `loginAdmin`), login di un utente esterno all'università, aggiunta alla lista di attesa, ricerca di un hirer e registrazione di un utente esterno.

```
public Hirer loginExternalHirer(String userCode, String password) throws AccessDeniedException {
    HirerDAO hirerDAO = new HirerDAO();
    HashMap<String, String> saltAndHashedPassword = hirerDAO.getSaltAndHashedPassword(userCode);
    //controllo password
    String hashedPassword = Hasher.hashPassword(password,saltAndHashedPassword.get("salt");
    String storedPassword = saltAndHashedPassword.get("hashedPassword";
    if(!hashedPassword.equals(storedPassword)){
        throw new AccessDeniedException("Errore: accesso come ruolo Hirer rifiutato, controlla userCode
            e password.");
    }

    Hirer hirer = hirerDAO.getHirer(userCode);
    hirer.setToken(new Token(hirer));
    return hirer;
}
```

Il metodo `addToWaitingList` aggiunge un certo `Hirer` (tramite la mail) nella lista di attesa di un determinato articolo non disponibile. Se l'articolo non è noleggiabile presso la sede specificata viene lanciata un'eccezione.

```
public void addToWaitingList(Item item, String mail, String storagePlace){
       if(!item.isBorrowable(Library.valueOf(storagePlace))){
           throw new ActionDeniedException("Errore: L'Item con itemCode [" + item.getCode() + "] non
               noleggiabile nella sede [" + storagePlace + "].");
       }
       if(item.getNumberOfAvailableCopiesInLibrary(Library.valueOf(storagePlace)) > 1) {
           throw new ActionDeniedException("Errore: L'Item con itemCode [" + item.getCode() + "] ha
               abbastanza copie nella sede [" + storagePlace + "], puoi effettuare una prenotazione.")
```

```
                ;
        }

        WaitingListDAO waitingListDAO = new WaitingListDAO();
        waitingListDAO.addToWaitingList(item.getCode(), storagePlace, mail);
    }
```

Il metodo `searchHirer` ritorna gli hirers che hanno almeno un campo corrispondente a una delle keyword in ingresso.

```
public ArrayList<Hirer> searchHirer(String keywords, Token token) {
        if(!token.getTokenRole().equals(Hasher.hash("Admin"))){
            throw new ActionDeniedException("Errore: Questa operazione  eseguibile solo da un Admin.");
        }
        HirerDAO hirerDAO = new HirerDAO();
        ArrayList<Hirer> hirers = hirerDAO.getHirers_();
        String[] splittedKeywords = keywords.split(" ");
        ArrayList<Hirer> result = new ArrayList<>();
        for(Hirer h : hirers){
            for (String keyword : splittedKeywords){
                if (h.contains(keyword)) {
                    result.add(h);
                }
            }
        }
        return result;
    }
```

Il metodo `registerExternalHirer` genera un codice utente valido con cui i dati dell'`Hirer` e la password crittografata vengono registrati. Viene infine mandata una mail di conferma dell'avvenuta registrazione.

```
public String registerExternalHirer(String name, String surname, String email, String telephoneNumber,
     Token token) {
    if(!token.getTokenRole().equals(Hasher.hash("Admin"))){
        throw new ActionDeniedException("Errore: Operazione eseguibile solo da un Admin.");
    }
    //generazione userCode e password random
    HirerDAO hirerDAO = new HirerDAO();
    String password = "Password" + Math.round((Math.random() * 1000000));
    String userCode = "";
    try { //validazione userCode
        do {
            userCode = "E" + Math.round((Math.random() * 1000000));
            hirerDAO.getHirer(userCode);
        } while (true);
    } catch (IdNotFoundException e) {
    }
    String salt = String.valueOf(Math.round(Math.random()*100000));
    String hashedPassword = Hasher.hashPassword(password, salt);

    /*gestione transazione per garantire aggiunta in tabella Hirer e tabella User_Credentials*/
    ConnectionManager connectionManager = ConnectionManager.getInstance();
    connectionManager.closeAutoCommit();
    try{
        hirerDAO.addHirer(userCode, name, surname, email, telephoneNumber);
         hirerDAO.addHirerPassword(userCode, hashedPassword, salt);
        connectionManager.commit();
        connectionManager.openAutoCommit();
    } catch (Exception e){
        connectionManager.rollback();
        connectionManager.openAutoCommit();
        throw e;
    }
    MailSender.sendRegisterCredentials(email, userCode, password);
    return userCode;
```

```
}
```

ItemController: questa classe gestisce operazioni di ricerca ed aggiornamento (inserimento, modifica, cancellazione) delle varie tipologie di articoli. Di seguito saranno riportate solo quelle relative a `Book` dal momento che per `Magazine` e `Thesis` la logica è la stessa.

Sono presenti la ricerca "ordinaria" e la ricerca "avanzata", dove per "avanzata" si intende l'andare a eseguire una ricerca più specifica mediante maggiori vincoli.

```
public ArrayList<Item> searchItem(String keywords) {
       ArrayList<Item> items = getAllItems();

       String[] splittedKeyword = keywords.split(" ");
       ArrayList<Item> result = new ArrayList<>();
       for (Item i : items) {
           for (String keyword : splittedKeyword) {
               if (i.contains(keyword)) {
                   result.add(i);
               }
           }
       }
       return result;
   }
```

```
public ArrayList<Item> advanceSearchItem(String keywords, String category, String language, boolean
    borrowable, String startDate, String endDate) {
   ArrayList<Item> items = getAllItems();
   String[] splittedKeyword = keywords.split(" ");
   ArrayList<Item> result = new ArrayList<>();
   for (Item i : items) {
       for (String keyword : splittedKeyword){
           if (i.getCategory().equals(Category.valueOf(category)) &&
           i.getLanguage().equals(Language.valueOf(language)) &&
           i.isBorrowable() == borrowable &&
           i.getPublicationDate().isAfter(LocalDate.parse(startDate)) &&
           i.getPublicationDate().isBefore(LocalDate.parse(endDate)) &&
           i.contains(keyword)) {
               result.add(i);
           }
       }
   }
   return result;
}
```

Il metodo `addBook` va ad aggiungere un `Book` al database se esso non è già presente, altrimenti va ad aggiornare il suo numero di copie.

```
public void addBook(... parametri Book ...,Token token) {
   if(!token.getTokenRole().equals(Hasher.hash("Admin"))){
       throw new ActionDeniedException("Errore: Operazione eseguibile solo da un Admin.");
   }
   Book bookCopy = new Book(... parametri Book ...);
   BookDAO bookDAO = new BookDAO();
   ArrayList<Item> items = getAllItems();
   for (Item i : items) { //controllo book già esistente
       if(i.sameField(bookCopy)) {
           addPhysicalCopies(i.getCode(), Library.valueOf(token.getTokenWorkingPlace()),
               numberOfCopies, borrowable);
           return;
       }
   }
```

```
    /*garanzia aggiunta nelle tabelle Item, Book e Physical_copies*/
    ConnectionManager connectionManager = ConnectionManager.getInstance();
    try {
        connectionManager.closeAutoCommit();
        int itemCode = bookDAO.addBook(... parametri Book ...);
        if(numberOfCopies >= 0) {
            PhysicalCopiesDAO physicalCopiesDAO = new PhysicalCopiesDAO();
            physicalCopiesDAO.addPhysicalCopies(itemCode, token.getTokenWorkingPlace(), numberOfCopies,
                 borrowable);
        }
        connectionManager.commit();
        connectionManager.openAutoCommit();
    } catch(Exception e){
        connectionManager.rollback();
        connectionManager.openAutoCommit();
        throw e;
    }
}
```

Il metodo `addPhysicalCopies` va ad aggiornare il numero di copie di un articolo in una certa sede.

```
private void addPhysicalCopies(int itemCode, Library storagePlace, int numberOfCopies, boolean
     borrowable) {
    PhysicalCopiesDAO physicalCopiesDAO = new PhysicalCopiesDAO();
    int physicalCopies = physicalCopiesDAO.getPhysicalCopies(itemCode).get(storagePlace).
         getNumberOfPhysicalCopies();
    if (physicalCopies == 0){
        physicalCopiesDAO.addPhysicalCopies(itemCode, storagePlace.toString(), numberOfCopies,
             borrowable);
    }else {
        physicalCopiesDAO.updatePhysicalCopies(itemCode, storagePlace.toString(), (physicalCopies +
             numberOfCopies), borrowable);
    }
}
```

Il metodo `removeBook` va a rimuovere le copie del libro presenti all'interno della sede dove lavora l'`Admin` oppure lo rimuove direttamente dal database se in nessuna sede sono presenti copie di quel libro e non è presente la versione digitale.
Ci sono 2 casi di errore: quando non ci sono copie di quel libro nella sede lavorativa dell'`Admin` e quando ci sono prestiti o prenotazioni associate a quel libro in corso in quella sede.

```
public void removeBook(int itemCode, Token token) {
        if (!token.getTokenRole().equals(Hasher.hash("Admin"))) {
            throw new ActionDeniedException("Errore: Questa operazione  eseguibile solo da un Admin.");
        }
        BookDAO bookDAO = new BookDAO();
        Book book = bookDAO.getBook(itemCode);
        PhysicalCopiesDAO physicalCopiesDAO = new PhysicalCopiesDAO();
        book.setPhysicalCopies(physicalCopiesDAO.getPhysicalCopies(itemCode));

        ConnectionManager connectionManager = ConnectionManager.getInstance();
        try {
            connectionManager.closeAutoCommit();
            if (book.getNumberOfLibraries() == 0) {
                if (book.getLink().isEmpty()) {
                    bookDAO.removeBook(itemCode);
                }
            } else {
                if (book.getNumberOfCopiesInLibrary(Library.valueOf(token.getTokenWorkingPlace())) ==
                     0) {
                    throw new ActionDeniedException("Errore: questo Book con itemCode [" + itemCode + "]
                         non  presente nella tua sede [" + token.getTokenWorkingPlace() + "]");
                }
```

```
            if (book.getNumberOfCopiesInLibrary(Library.valueOf(token.getTokenWorkingPlace())) !=
                    book.getNumberOfAvailableCopiesInLibrary(Library.valueOf(token.
                        getTokenWorkingPlace()))) {
                throw new ActionDeniedException("Errore: questo Book con itemCode [" + itemCode + "]
                     ha ancora prenotazioni/prestiti nella tua sede [" +
                       token.getTokenWorkingPlace() + "]");
            }
            physicalCopiesDAO.removePhysicalCopies(itemCode, token.getTokenWorkingPlace());
        }
        connectionManager.commit();
        connectionManager.openAutoCommit();
    } catch (Exception e) {
        connectionManager.rollback();
        connectionManager.openAutoCommit();
        throw e;
    }
}
```

Il metodo `updateBook` va ad aggiornare i dati relativi a un `Book`.

```
public void updateBook(int itemCode,... nuovi parametri Book ..., Token token) {

    if(!token.getTokenRole().equals(Hasher.hash("Admin"))){
        throw new ActionDeniedException("Errore: Operazione eseguibile solo da un Admin.");
    }
    LocalDate.parse(publicationDate);
    Language.valueOf(language);
    Category.valueOf(category);
    ConnectionManager connectionManager = ConnectionManager.getInstance();
    try {
        connectionManager.closeAutoCommit();
        BookDAO bookDAO = new BookDAO();
        bookDAO.updateBook(... nuovi parametri Book ...);
        if(numberOfCopies > 0){
            PhysicalCopiesDAO physicalCopiesDAO = new PhysicalCopiesDAO();
            physicalCopiesDAO.updatePhysicalCopies(itemCode,token.getTokenWorkingPlace(),
                numberOfCopies, borrowable);
        }
        connectionManager.commit();
        connectionManager.openAutoCommit();
    } catch (Exception e){
        connectionManager.rollback();
        connectionManager.openAutoCommit();
        throw e;
    }
}
```

LendingController: classe che implementa la logica relativa ai prestiti. Ci sono 3 operazioni: ottenimento di una lista che contiene tutti i prestiti di un certo `Hirer`, la restituzione di un articolo all'opportuna sede bibliotecaria e la registrazione di un prestito:

Il metodo `getLendings` restituisce una lista contenente tutti i prestiti associati a un `Hirer`.

```
public ArrayList<Lending> getLendings(String userCode) {
    LendingDAO lendingDAO = new LendingDAO();
    return lendingDAO.getLendingsByUserCode(userCode);
}
```

Il metodo `registerReturnOfItem` registra la restituzione di un articolo alla sede opportuna, eliminando il prestito ed inviando una mail di conferma all'`Hirer`; inoltre, viene mandata una notifica a tutti gli `hirers` nella lista di attesa di quell'articolo e vengono rimossi da quest'ultima.

```
public void registerReturnOfItem(Hirer hirer, Item item, String storagePlace, Token token) {
    if (!token.getTokenRole().equals(Hasher.hash("Admin"))) {
```

```
        throw new ActionDeniedException("Errore: Operazione eseguibile solo da un Admin.");
    }
    if (!token.getTokenWorkingPlace().equals(storagePlace)) {
        throw new ActionDeniedException("Errore: Operazione non registrabile, prestito registrato in
            una sede diversa.");
    }
    Library.valueOf(storagePlace);

    LendingDAO lendingDAO = new LendingDAO();
    ConnectionManager connectionManager = ConnectionManager.getInstance();
    connectionManager.closeAutoCommit();
    try {
        lendingDAO.removeLending(hirer.getUserCode(), item.getCode(), token.getTokenWorkingPlace());
        MailSender.sendReturnSuccessMail(hirer.getEmail(), hirer.getUserCode(), item.getCode(), item.
            getTitle());
        WaitingListDAO waitingListDAO = new WaitingListDAO();
        ArrayList<String> emails = waitingListDAO.getWaitingList(item.getCode(), storagePlace);
        for (String email : emails) {
            //notifica Item disponibile
            MailSender.sendNotifyWaitingListMail(email, item.getCode(), item.getTitle(), storagePlace);
        }
        waitingListDAO.removeWaitingList(item.getCode(), storagePlace);
        connectionManager.commit();
        connectionManager.openAutoCommit();
    } catch (Exception e){
        connectionManager.rollback();
        connectionManager.openAutoCommit();
        throw e;
    }
}
```

Il metodo `registerLending` va a registrare un prestito. L'utente viene notificato tramite mail dell'avvenuta registrazione.

```
public void registerLending(Hirer hirer, Item item, Token token) {
    if (!token.getTokenRole().equals(Hasher.hash("Admin"))) {
        throw new ActionDeniedException("Errore: Operazione eseguibile solo da un Admin.");
    }
    if (hirer.getUnbannedDate() != null) {
        throw new ActionDeniedException("Errore: l'Hirer con userCode [" + hirer.getUserCode() +"]
            bloccato, non puo' eseguire un prestito.");
    }
    if (!item.isBorrowable(Library.valueOf(token.getTokenWorkingPlace()))) {
        throw new ActionDeniedException("Errore: articolo con itemCode [" + item.getCode() +"] non
            noleggiabile.");
    }
    if (item.getNumberOfAvailableCopiesInLibrary(Library.valueOf(token.getTokenWorkingPlace())) <= 1)
        {
        throw new ActionDeniedException("Errore: l'articolo con itemCode [" + item.getCode() +"] non ha
             abbastanza copie nella sede [" + token.getTokenWorkingPlace() + "].");
    }
    LendingDAO lendingDAO = new LendingDAO();
    lendingDAO.addLending(hirer.getUserCode(), item.getCode(), token.getTokenWorkingPlace());
    MailSender.sendLendingSuccessMail(hirer.getEmail(), hirer.getUserCode(),
        item.getCode(), item.getTitle(), token.getTokenWorkingPlace(), LocalDate.now().plusMonths(1));
}
```

ReservationController: classe che implementa la logica delle prenotazioni. Anche qui sono presenti 4 operazioni: ricerca di tutte le prenotazioni di un utente (la logica è analoga a `getLendings`), la cancellazione, la registrazione ed il ritiro di una prenotazione, dove per "ritiro di una prenotazione" si intende il fatto che l'utente si rechi nella sede scelta a ritirare l'articolo prenotato.

Il metodo `removeReservation` va a cancellare una prenotazione. L'`Hirer` riceve una mail che specifica l'esecuzione dell'operazione. Inoltre vengono notificati tutti gli `hirers` presenti nella lista di attesa e rimossi da essa.

```
public void removeReservation(Hirer hirer, Item item, String storagePlace, Token token) {
       if(!token.getTokenWorkingPlace().equals("NONE") && !token.getTokenWorkingPlace().equals(
           storagePlace)){
           throw new ActionDeniedException("Errore: Non puoi registrare questa operazione, il libro
               non  stato prenotato nella sede in cui lavori.");
       }
       Library.valueOf(storagePlace);
       ReservationDAO reservationDAO = new ReservationDAO();

       reservationDAO.removeReservation(hirer.getUserCode(), item.getCode(), storagePlace);
       WaitingListDAO waitingListDAO = new WaitingListDAO();
       ArrayList<String> emails = waitingListDAO.getWaitingList(item.getCode(), storagePlace);
       for (String email : emails) {
           MailSender.sendNotifyWaitingListMail(email, item.getCode(), item.getTitle(), storagePlace);
       }
       waitingListDAO.removeWaitingList(item.getCode(), storagePlace);
   }
```

Il metodo `reserveItem` aggiunge una prenotazione e invia una notifica di conferma all'utente.

```
public void reserveItem(Hirer hirer, Item item, String storagePlace, Token token) {
       if(!token.getTokenRole().equals(Hasher.hash("Hirer"))){
           throw new ActionDeniedException("Errore: Questa operazione  eseguibile solo da un Hirer.");
       }
       if(hirer.getUnbannedDate() != null){
           throw new ActionDeniedException("Errore: l'Hirer con userCode [" + hirer.getUserCode() +"]
                bannato, non pu eseguire un prestito.");
       }

       if (!item.isBorrowable(Library.valueOf(storagePlace))) {//
           throw new ActionDeniedException("Errore: l'articolo con itemCode [" + item.getCode() +"]
               non  noleggiabile, non pu essere eseguita la prenotazione.");
       }
       if (item.getNumberOfAvailableCopiesInLibrary(Library.valueOf(storagePlace)) <= 1) {
           throw new ActionDeniedException("Errore: l'articolo con itemCode [" + item.getCode() +"]
               non ha abbastanza copie nella sede [" +
                  storagePlace + "].");
       }

       Library.valueOf(storagePlace);
       ReservationDAO reservationDAO = new ReservationDAO();

       reservationDAO.addReservation(hirer.getUserCode(), item.getCode(), storagePlace);
       MailSender.sendReservationSuccessMail(hirer.getEmail(), hirer.getUserCode(), item.getCode(),
           item.getTitle(), storagePlace, LocalDate.now().plusDays(7));
   }
```

Il metodo `confirmReservationWithdraw` cancella la prenotazione di un articolo ed inserisce il prestito. L'`Hirer` riceve una mail di conferma dell'avvenuta operazione.

```
public void confirmReservationWithdraw(Hirer hirer, Item item, String storagePlace, Token token) {
   if(!token.getTokenRole().equals(Hasher.hash("Admin"))){
       throw new ActionDeniedException("Errore: Operazione eseguibile solo da un Admin.");
   }
   if(!token.getTokenWorkingPlace().equals(storagePlace)){
       throw new ActionDeniedException("Errore: Non puoi registrare questa operazione, articolo non
           prenotato nella sede in cui lavori.");
   }
   Library.valueOf(storagePlace);

   ConnectionManager connectionManager = ConnectionManager.getInstance();
   connectionManager.closeAutoCommit();
```

```
    ReservationDAO reservationDAO = new ReservationDAO();
    LendingController lendingController = new LendingController();
    try {
        reservationDAO.removeReservation(hirer.getUserCode(), item.getCode(), storagePlace);
    } catch (Exception e){
        connectionManager.rollback();
        connectionManager.openAutoCommit();
        throw e;
    }
    try {
        lendingController.registerLending(hirer, item, token);
        connectionManager.commit();
        connectionManager.openAutoCommit();
    }catch (Exception e){
        connectionManager.rollback();
        connectionManager.openAutoCommit();
        throw e;
    }
}
```

3.2.1 Eccezioni Business Logic

Come si può notare dai vari metodi della Business Logic esposti precedentemente, sono state create delle eccezioni `AccessDeniedException` e `ActionDeniedException` per distinguere i motivi che causano l'impossibilità di effettuare con successo l'operazione.

Di seguito verrà riportata solo la prima delle due (la seconda è analoga alla prima)

```
public class AccessDeniedException extends RuntimeException {
    public AccessDeniedException(String message) {
        super(message);
    }
}
```

3.3 ORM

Questa sezione riguarda l'integrazione con il database tramite ORM (Object–Relational Mapping) e comprende il `ConnectionManager` e le classi DAO (Data Access Object). Non vengono riportate le implementazioni dettagliate dei metodi, trattandosi per la maggior parte di operazioni CRUD ripetitive ed analoghe.

Nel Database sono implementati anche dei triggers per l'inserimento e la cancellazione sulla tabella `physical_copies` in seguito a una nuova prenotazione o alla cancellazione di una prenotazione esistente e all'aggiornamento del numero di copie di un certo `Item` in una determinata sede:

```
    CREATE OR REPLACE FUNCTION increment_number_of_available_copies()
    RETURNS TRIGGER AS $$
    BEGIN
    UPDATE physical_copies
    SET number_of_available_copies = number_of_available_copies + 1
    WHERE itemCode = OLD.itemCode AND storage_place = OLD.storage_place;
    END;
    $$ LANGUAGE plpgsql;

    CREATE OR REPLACE FUNCTION decrement_number_of_available_copies()
    RETURNS TRIGGER AS $$
    BEGIN
    UPDATE physical_copies
    SET number_of_available_copies = number_of_available_copies - 1
    WHERE itemCode = NEW.itemCode AND storage_place = NEW.storage_place;
    END;
    $$ LANGUAGE plpgsql;
```

```
CREATE OR REPLACE FUNCTION update_number_of_available_copies_on_num_of_copies_change()
RETURNS TRIGGER AS $$
BEGIN
UPDATE physical_copies
SET number_of_available_copies = number_of_available_copies + (NEW.number_of_copies - OLD.
    number_of_copies)
WHERE itemCode = OLD.itemCode AND storage_place = OLD.storage_place;
END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE TRIGGER update_available_copies_on_delete
AFTER DELETE
ON reservation
FOR EACH ROW
EXECUTE FUNCTION increment_number_of_available_copies();

CREATE OR REPLACE TRIGGER update_available_copies_on_insert
AFTER INSERT
ON reservation
FOR EACH ROW
EXECUTE FUNCTION decrement_number_of_available_copies();

CREATE OR REPLACE TRIGGER update_available_copies_on_update_of_number_of_copies
AFTER UPDATE
ON physical_copies
FOR EACH ROW
EXECUTE FUNCTION update_number_of_available_copies_on_num_of_copies_change();
```

ConnectionManager Classe responsabile della gestione della connessione al database. È stata implementata secondo il *Singleton Pattern*, garantendo un'unica istanza condivisa in tutto il sistema.

Di seguito l'implementazione:

```
public class ConnectionManager {
    private static ConnectionManager connectionManager;
    private Connection connection;
    private String url = "";
    private String username = "";
    private String password = "";

    private ConnectionManager() {
        try {
            List<String> righe = Files.readAllLines(Paths.get("./src/main/resources/credenziali"));
            url = righe.get(2);
            username = righe.get(3);
            password = righe.get(4);
        } catch (IOException e) {
            e.printStackTrace();
        }
        try {
            connection = DriverManager.getConnection(url, username, password);
        } catch (SQLException e) {
            throw new DatabaseConnectionException(e);
        }
    }

    public static ConnectionManager getInstance() {
        if (connectionManager == null) {
            connectionManager = new ConnectionManager();
        }
        return connectionManager;
    }

}
```

Le classi DAO sviluppate sono le seguenti:

- `AdminDAO`
- `HirerDAO`
- `BookDAO`
- `MagazineDAO`
- `ThesisDAO`
- `PhysicalCopiesDAO`
- `LendingDAO`
- `ReservationDAO`
- `WaitingListDAO`

Dal momento che è stata usata nei metodi della Business Logic, di seguito si riporta l'implementazione della classe `BookDAO`. Le altre classi non vengono riportate perché sono concettualmente analoghe.

```
public class BookDAO {

    private Connection connection;

    public BookDAO() {
        this.connection = ConnectionManager.getInstance().getConnection();
    }

    public Book getBook(int itemCode) throws IdNotFoundException, DatabaseConnectionException {
        try {
            String query = """
                    SELECT *
                    FROM Item I JOIN Book B ON I.code = B.code
                    WHERE I.code = ?;
                    """;
            PreparedStatement ps = connection.prepareStatement(query);
            ps.setInt(1, itemCode);
            ResultSet resultSet = ps.executeQuery();

            if(!resultSet.next()) {
                throw new IdNotFoundException("Errore: Book con itemCode [" + itemCode + "] non
                    presente nel DB.");
            }

            Book book = new Book(
                    itemCode,
                    resultSet.getString("title"),
                    LocalDate.parse(resultSet.getString("publication_date")),
                    Language.valueOf(resultSet.getString("language")),
                    Category.valueOf(resultSet.getString("category")),
                    resultSet.getString("link"),
                    resultSet.getString("isbn"),
                    resultSet.getString("publishing_house"),
                    resultSet.getInt("number_of_pages"),
                    resultSet.getString("authors"));
            return book;
        } catch (SQLException e) {
            throw new DatabaseConnectionException(e);
        }
    }

    public int addBook(... parametri Book ...)
            throws DatabaseConnectionException {
        try {
            //Creazione Item e Book
            String query = """
                    INSERT INTO Item (title, publication_date, language, category, link, number_of_pages
                        )
                    VALUES (?, ?, ?, ?, ?, ?)
                    RETURNING code;
                    """;
```

```
        PreparedStatement ps = connection.prepareStatement(query);
        ps.setString(1, title);
        ps.setDate(2, Date.valueOf(publicationDate));
        ps.setString(3, language);
        ps.setString(4, category);
        ps.setString(5, link);
        ps.setInt(6, numberOfPages);

        ResultSet resultSet = ps.executeQuery();
        resultSet.next();
        int itemCode = resultSet.getInt("code");

        String query_2 = """
                INSERT INTO Book (code, isbn, publishing_house, authors)
                VALUES (?, ?, ?, ?);
                """;
        PreparedStatement ps2 = connection.prepareStatement(query_2);
        ps2.setInt(1, itemCode);
        ps2.setString(2, isbn);
        ps2.setString(3, publishingHouse);
        ps2.setString(4, authors);
        ps2.executeUpdate();

        return itemCode;
    } catch (SQLException e) {
        throw new DatabaseConnectionException(e);
    }
}

public void removeBook(int itemCode) throws IdNotFoundException, ConstraintViolationException,
    DatabaseConnectionException {
    try {
        String query = """
                DELETE FROM Book
                WHERE code = ?;
                """;
        PreparedStatement ps = connection.prepareStatement(query);
        ps.setInt(1, itemCode);

        if(ps.executeUpdate() != 1) {
            throw new IdNotFoundException("Errore: Il Book con itemCode [" + itemCode + "] non
                presente nel DB.");
        }

        String query_2 = """
                DELETE FROM Item
                WHERE code = ?;
                """;
        ps = connection.prepareStatement(query_2);
        ps.setInt(1, itemCode);

        if(ps.executeUpdate() != 1) {
            throw new IdNotFoundException("Errore: L'Item con ItemCode [" + itemCode + "] che vuoi
                rimuovere non e' un Book. ");
        }
    } catch (SQLException e) {
        if(e.getSQLState().equals("23503")){
            throw new ConstraintViolationException("Errore: Book con itemCode [" + itemCode + "]
                non pue' essere eliminato " +
                   "perche' sono ancora presenti Copie/Prenotazioni/Prestiti. ");
        }
        throw new DatabaseConnectionException(e);
    }
}

public void updateBook(int originalItemCode,...parametri Book...)
```

```
        throws IdNotFoundException, ConstraintViolationException, DatabaseConnectionException{
    try {
        String query = """
               UPDATE Item
               SET title = ?, publication_date = ?, language = ?, category = ?, link = ? ,
                   number_of_pages = ?
               WHERE code = ?;
               """;
        PreparedStatement ps = connection.prepareStatement(query);
        ps.setString(1, title);
        ps.setDate(2, Date.valueOf(publicationDate));
        ps.setString(3, language);
        ps.setString(4, category);
        ps.setString(5, link);
        ps.setInt(6, numberOfPages);
        ps.setInt(7, originalItemCode);

        if(ps.executeUpdate() != 1) {
            throw new IdNotFoundException("Errore: Book con ItemCode [" + originalItemCode + "] non
                 presente nel DB.");
        }

        query = """
               UPDATE Book
               SET isbn = ?, publishing_house = ?, authors = ?
               WHERE code = ?;
               """;
        ps = connection.prepareStatement(query);
        ps.setString(1, isbn);
        ps.setString(2, publishingHouse);
        ps.setString(3, authors);
        ps.setInt(4, originalItemCode);

        if(ps.executeUpdate() != 1) {
            throw new IdNotFoundException("Errore: Item con ItemCode [" + originalItemCode + "] che
                 vuoi aggiornare non e' un Book.");
        }
    } catch (SQLException e) {
        throw new DatabaseConnectionException(e);
    }
}


public ArrayList<Book> getAllBooks() throws DatabaseConnectionException {
    ArrayList<Book> books = new ArrayList<>();
    try {
        String query = """
               SELECT *
               FROM Item I JOIN Book B ON I.code = B.code;
               """;
        PreparedStatement ps = connection.prepareStatement(query);
        ResultSet resultSet = ps.executeQuery();
        while (resultSet.next()) {
            books.add(new Book(
                   resultSet.getInt("code"),
                   resultSet.getString("title"),
                   LocalDate.parse(resultSet.getString("publication_date")),
                   Language.valueOf(resultSet.getString("language")),
                   Category.valueOf(resultSet.getString("category")),
                   resultSet.getString("link"),
                   resultSet.getString("isbn"),
                   resultSet.getString("publishing_house"),
                   resultSet.getInt("number_of_pages"),
                   resultSet.getString("authors")));
        }
        return books;
```

```
        } catch (SQLException e) {
            throw new DatabaseConnectionException(e);
        }
    }
}
```

3.3.1 Eccezioni ORM

Per la gestione delle anomalie a livello di ORM sono state implementate alcune eccezioni personalizzate:

- `ConstraintViolationException`
- `DatabaseConnectionException`
- `IdAlreadyExistsException`
- `IdNotFoundException`

Di seguito si riporta, a titolo esemplificativo, l'implementazione della prima (le altre seguono una struttura simile).

```
public class ConstraintViolationException extends RuntimeException {
    public ConstraintViolationException(String message) {
        super(message);
    }
}
```

3.4 MailSender

È stato implementato un sistema di notifica automatico via email, per poter fornire una funzione di promemoria e comunicazione di svolgimento con successo delle operazioni.
Il sistema `MailSender` invia notifiche nei seguenti casi:

- Conferma di prenotazione o prestito avvenuto con successo;
- Invio codice di verifica per mail (Figure 1);
- Scadenza di una prenotazione;
- Rinnovo di un noleggio;
- Blocco o sblocco dell'account utente;
- Invio credenziali ad utente esterno;
- Restituzione dell'articolo alla sede;
- Inserimento dell'utente in lista d'attesa;
- Notifica agli utenti in lista d'attesa quando un articolo diventa disponibile.

Per l'implementazione abbiamo utilizzato la libreria `Jakarta`, ed essendo una classe che non tiene conto dello stato interno del software abbiamo deciso di implementarla come classe statica. Per mantenere l'utilizzo di una singola connessione `SMTP` è stata messa un'istanza statica di `Session` e `Transport`.
È implementato un blocco static con il ruolo di costruttore ed esegue le operazioni di setup: ottenere le credenziali da un file "credenziali" nella sezione `resources` per evitare di esporre direttamente email e password; creazione delle istanze di `Session` e `Transport`.

```
static {
    ..
        List<String> righe = Files.readAllLines(Paths.get("./src/main/resources/credenziali"));
        myAccountEmail = righe.get(0);
        password = righe.get(1);
    ..

    ..
    session = Session.getInstance(properties, new Authenticator() {
```

Figure 1: verifica validità email durante registrazione utente esterno

```
        @Override
        protected PasswordAuthentication getPasswordAuthentication() {
            return new PasswordAuthentication(myAccountEmail, password);
        }
    ..

    ..
        transport = session.getTransport("smtp");
        transport.connect();
    ..
}
```

Preparazione messaggio:

```
    if (!transport.isConnected()) {
        transport.connect();
    }
    Message message = new MimeMessage(session);
    message.setFrom(new InternetAddress(myAccountEmail));
    message.setRecipients(Message.RecipientType.TO, InternetAddress.parse(recipient));
    message.setSubject(subject);
    message.setContent(content, "text/html");
    transport.sendMessage(message, message.getAllRecipients());
```

4 Test

4.1 Introduzione

In questa sezione viene riportata e descritta l'implementazione dei test della *Business Logic* e dell'*ORM*.
Per quanto riguarda invece il *Domain Model*, dal momento che implementa principalmente soltanto costruttori, *getter/setter* e gli override dei metodi `equals` ed `hashCode`, i test non vengono descritti né riportati, nonostante siano stati effettuati.

4.2 Business Logic

Con i test della *Business Logic* si va a simulare l'interazione con l'utente (i test si riferiscono ai Templates).

I metodi della *Business Logic* richiamano a loro volta metodi dell'*ORM*, che per ora si assume funzionino correttamente. Una giustificazione a questa assunzione sarà fornita nella sezione successiva, dedicata ai test dell'*ORM*.
Dal momento che è coinvolto anche il database, in ogni classe test c'è un metodo `@BeforeEach` che va a svuotare le tabelle utilizzate e un metodo `@AfterAll` che va a chiudere la connessione.

4.2.1 AdminControllerTest

Funzione di setup:

```
private void setUpLoginRecognized() throws SQLException {
    PreparedStatement ps = connection_university_db.prepareStatement("INSERT INTO library_admin VALUES
         (?, ?, ?, ?, ?, ?, ?, ?)");
    ps.setString(1, "E256743");
    ps.setString(2, "Marco");
    ps.setString(3, "Verdi");
    ps.setString(4, "marco.verdi@unimail.com");
    ps.setString(5, "00001");
    ps.setString(6, "LIBRARY_1");
    ps.setString(7, "345234");
    ps.setString(8, "1722c3266324344fa1dbf0c156d299a26ce14fd5d16b1f38e447da831fcaf7e9");
    ps.executeUpdate();
}
```

Listing 1: Login effettuato con successo da un admin già presente nel sistema bibliotecario.

```
@Test
public void testLoginAdmin_SuccessAndNotFirstLogin() throws SQLException {
    //inserisco l'admin nel database dell'università
    setUpLoginRecognized();
    //inserisco l'admin nel database della biblioteca
    adminDAO.addAdmin("E256743", "Marco", "Verdi", "marco.verdi@unimail.com", "00001", "LIBRARY_1");
    //controllo che loginAdmin ritorni l'admin corretto
    assertEquals(adminDAO.getAdmin("E256743"), adminController.loginAdmin("E256743", "abcd1234"));
}
```

Listing 2: Login effettuato con successo da un admin che effettua il login per la prima volta.

```
@Test
public void testLoginAdmin_SuccessAndFirstLogin() throws SQLException {
    setUpLoginRecognized();

    Admin admin = adminController.loginAdmin("E256743", "abcd1234");

    assertEquals(admin, adminDAO.getAdmin("E256743"));
}
```

Listing 3: Caso di un admin non riconosciuto in quanto non presente nel database dell'università.

```
@Test
public void testLoginAdmin_NotRecognized() throws SQLException{
    assertThrows(AccessDeniedException.class, () -> adminController.loginAdmin("wrong_usercode",
        "wrong_password"));
}
```

Clicca qui per tornare ai templates

Si riporta anche un test funzionale che copre diverse operazioni per quanto riguarda gli amministratori.
Il test comprende:

- Login dell'amministratore;
- Aggiunta di un libro al catalogo;
- Modifica dello stesso libro aggiunto poco prima;
- Registrazione di un utente esterno.

```
@Test
public void FunctionalTestAdmin() throws SQLException {
    setUpLoginRecognized();

    Admin admin = adminController.loginAdmin("E256743", "abcd1234");
    Token admin_token = new Token(admin);

    itemController.addBook("Programmazione", LocalDate.of(2020, 3,4).toString(),
        Language.LANGUAGE_1.toString(), Category.CATEGORY_1.toString(), "link", "isbn", "Mondadori",
        500, "autori", 8, true, admin_token);

    Book expected_book = new Book(1, "Programmazione", LocalDate.of(2020, 3,4), Language.LANGUAGE_1,
         Category.CATEGORY_1, "link", "isbn", "Mondadori", 500, "autori");

    HashMap<Library, PhysicalCopies> expected_pcs = new HashMap<>();
    PhysicalCopies pc = new PhysicalCopies(8, 8, true);
    expected_pcs.put(Library.LIBRARY_1, pc);

    assertEquals(expected_book, bookDAO.getBook(1));
    assertEquals(expected_pcs, pcDAO.getPhysicalCopies(1));


    itemController.updateBook(1, "Fondamenti di informatica", LocalDate.of(2020, 3,4).toString(), true
         , Language.LANGUAGE_1.toString(), Category.CATEGORY_1.toString(), "link", "isbn", "Mondadori",
          500, "autori", 8, admin_token );

    Book book = bookDAO.getBook(1);
    book.setPhysicalCopies(expected_pcs);

    assertEquals(1, book.getCode());
    assertEquals("Fondamenti di informatica", book.getTitle());
    assertEquals(LocalDate.of(2020, 3,4).toString(), book.getPublicationDate().toString());
    assertTrue(book.isBorrowable());
    assertEquals(Language.LANGUAGE_1.toString(), book.getLanguage().toString());
    assertEquals(Category.CATEGORY_1.toString(), book.getCategory().toString());
    assertEquals("link", book.getLink());
    assertEquals("isbn", book.getIsbn());
    assertEquals("Mondadori", book.getPublishingHouse());
    assertEquals(500, book.getNumberOfPages());
    assertEquals("autori", book.getAuthors());
    assertEquals(8, book.getNumberOfCopiesInLibrary(Library.LIBRARY_1));


    String usercode = hirerController.registerExternalHirer("Filippo", "Taiti",
        "filippotaiti@studuni.com", "00000", admin_token);

    assertEquals("Filippo", hirerDAO.getHirer(usercode).getName());
```

```
    assertEquals("Taiti", hirerDAO.getHirer(usercode).getSurname());
    assertEquals("filippotaiti@studuni.com", hirerDAO.getHirer(usercode).getEmail());
    assertEquals("00000", hirerDAO.getHirer(usercode).getTelephoneNumber());
}
```

4.2.2 HirerControllerTest

In questa sezione si riportano i test relativi alla registrazione di un utente esterno, login di un utente esterno, ricerca utente e aggiunta nella lista di attesa.

Listing 4: Funzione di setup per il riconoscimento dell'Hirer.

```
private void setUpLoginRecognized() throws SQLException {
    PreparedStatement ps = connection_university_db.prepareStatement("INSERT INTO university_people
        VALUES (?, ?, ?, ?, ?, ?, ?)");
    ps.setString(1, "E256743");
    ps.setString(2, "Marco");
    ps.setString(3, "Verdi");
    ps.setString(4, "marco.verdi@studuni.com");
    ps.setString(5, "00001");
    ps.setString(6, "345234");
    ps.setString(7, "1722c3266324344fa1dbf0c156d299a26ce14fd5d16b1f38e447da831fcaf7e9");
    ps.executeUpdate();
}
```

Listing 5: Utente esterno registrato con successo.

```
@Test
public void testRegisterExternalHirer_Success(){
    adminDAO.addAdmin("uc1", "name", "surname", "email", "00000", Library.LIBRARY_1.toString());
    Admin admin = adminDAO.getAdmin("uc1");
    Token admin_token = new Token(admin);
    admin.setToken(admin_token);
    String expected_usercode = hirerController.registerExternalHirer("nome", "cognome", "email",
        "01234", admin_token);

    assertEquals(expected_usercode, hirerDAO.getHirer(expected_usercode).getUserCode());
}
```

Listing 6: Fallimento dovuto al tentativo di registrazione da parte di un Hirer, che non possiede i permessi necessari per eseguire l'operazione.

```
@Test
public void testRegisterExternalHirer_Fail(){
    hirerDAO.addHirer("E256743", "Marco", "Verdi", "marco.verdi@studuni.com", "00001");
    Hirer hirer = hirerDAO.getHirer("E256743");
    Token token = new Token(hirer);
    hirer.setToken(token);

    assertThrows(ActionDeniedException.class, () -> hirerController.registerExternalHirer(hirer.
        getName(), hirer.getSurname(), hirer.getEmail(), hirer.getTelephoneNumber(),
        hirer.getToken()));
}
```

Clicca qui per tornare ai templates

Listing 7: L'Hirer viene aggiunto con successo alla lista di attesa.

```
@Test
public void testAddToWaitingList_Success(){
    int book_code = bookDAO.addBook("titolo", LocalDate.of(2000, 6,3).toString(),
        Language.LANGUAGE_1.toString(), Category.CATEGORY_1.toString(), "link", "isbn",
        "publishing house", 200, "authors");
```

```
    physicalCopiesDAO.addPhysicalCopies(book_code, Library.LIBRARY_1.toString(), 2, true);
    hirerDAO.addHirer("usercode", "name", "surname", "mail", "00001");
    reservationDAO.addReservation("usercode", book_code, Library.LIBRARY_1.toString());

    Book book = bookDAO.getBook(book_code);
    HashMap<Library, PhysicalCopies> pcs = physicalCopiesDAO.getPhysicalCopies(book_code);
    book.setPhysicalCopies(pcs);

    hirerController.addToWaitingList(book, hirerDAO.getHirer("usercode").getEmail(),
        Library.LIBRARY_1.toString());

    ArrayList<String> emails = waitingListDAO.getWaitingList(book_code, Library.LIBRARY_1.toString());

    assertEquals(emails.size(), 1);
    assertEquals(emails.get(0), hirerDAO.getHirer("usercode").getEmail());
}
```

Listing 8: Fallimento causato dal fatto che l'articolo selezionato non è presente in alcuna sede.

```
@Test
   public void testAddToWaitingList_Fail_1(){
       hirerDAO.addHirer("usercode", "name", "surname", "mail", "00001");
       int book_code = bookDAO.addBook("titolo", LocalDate.of(2000, 6,3).toString(), Language.
           LANGUAGE_1.toString(),
              Category.CATEGORY_1.toString(),
              "link", "isbn", "publishing house", 200, "authors");

       assertThrows(ActionDeniedException.class, () -> hirerController.addToWaitingList(bookDAO.getBook
           (book_code), "mail", Library.LIBRARY_1.toString()));
   }
```

Listing 9: Fallimento causato dal fatto che l'articolo selezionato è disponibile nella sede specificata.

```
@Test
    public void testAddToWaitingList_Fail_2(){
        hirerDAO.addHirer("usercode", "name", "surname", "mail", "00001");
        int book_code = bookDAO.addBook("titolo", LocalDate.of(2000, 6,3).toString(), Language.
            LANGUAGE_1.toString(),
               Category.CATEGORY_1.toString(),
               "link", "isbn", "publishing house", 200, "authors");
        physicalCopiesDAO.addPhysicalCopies(book_code, Library.LIBRARY_1.toString(), 2, true);

        assertThrows(ActionDeniedException.class, () -> hirerController.addToWaitingList(bookDAO.
            getBook(book_code), "mail", Library.LIBRARY_1.toString()));
    }
```

Clicca qui per tornare ai templates

Listing 10: Test relativo alla ricerca di Hirer (o Hirers) all'interno del database.

```
@Test
public void testSearchHirer(){
    hirerDAO.addHirer("uc1", "Marco", "Bianchi", "marco.bianchi@unimail.com", "02121");
    hirerDAO.addHirer("uc2", "Luca", "Bianchi", "luca.bianchi@unimail.com", "09876");
    hirerDAO.addHirer("uc3", "Mario", "Rossi", "mario.rossi@unimail.com", "34563");

    ArrayList <Hirer> expected_hirers = new ArrayList<>();
    expected_hirers.add(hirerDAO.getHirer("uc1"));
    expected_hirers.add(hirerDAO.getHirer("uc2"));

    ArrayList <Hirer> notExpected_hirers = new ArrayList<>();
    notExpected_hirers.add(hirerDAO.getHirer("uc1"));
    notExpected_hirers.add(hirerDAO.getHirer("uc2"));
    notExpected_hirers.add(hirerDAO.getHirer("uc3"));
```

```
    assertEquals(expected_hirers, hirerController.searchHirer("Bianchi"));
    assertNotEquals(notExpected_hirers, hirerController.searchHirer("Bianchi"));
}
```

Clicca qui per tornare ai templates

Listing 11: Login effettuato con successo da parte di un utente esterno.

```
@Test
public void testLoginExternalHirer_Success() throws SQLException {
    setUpLoginRecognized();
    hirerDAO.addHirer("E256743", "Marco", "Bianchi", "marco.bianchi@studuni.com", "00001");
    hirerDAO.addHirerPassword("E256743",
        "1722c3266324344fa1dbf0c156d299a26ce14fd5d16b1f38e447da831fcaf7e9", "345234");

    assertEquals(hirerDAO.getHirer("E256743"), hirerController.loginExternalHirer("E256743",
        "abcd1234"));
}
```

Listing 12: Fallimento del login da parte di un utente esterno per credenziali errate.

```
@Test
public void testLoginExternalHirer_Fail() throws SQLException {
    setUpLoginRecognized();
    hirerDAO.addHirer("E256743", "Marco", "Bianchi", "marco.bianchi@studuni.com", "00001");
    hirerDAO.addHirerPassword("E256743",
        "1722c3266324344fa1dbf0c156d299a26ce14fd5d16b1f38e447da831fcaf7e9", "345234");

    assertThrows(AccessDeniedException.class, () ->
        hirerController.loginExternalHirer("E256743", "abcd12345"));
}
```

Clicca qui per tornare ai templates

Anche per quanto riguarda `HirerController` è stato riportato un test funzionale che copre diverse operazioni:

- Login;
- Prenotazione;
- Ritiro della prenotazione presso la sede specificata.

```
@Test
public void FunctionalTestHirer() throws SQLException {
    //inizio pre-set
    int book_code = bookDAO.addBook("Fondamenti di informatica", LocalDate.of(2015, 3, 2).toString(),
        Language.LANGUAGE_1.toString(), Category.CATEGORY_1.toString(), "link", "isbn", "Mondadori",
        300, "autori");

    physicalCopiesDAO.addPhysicalCopies(book_code, Library.LIBRARY_1.toString(), 5, true);

    //fine pre-set
    setUpLoginRecognized();

    Hirer hirer = hirerController.loginUniversityHirer("E256743", "abcd1234");

    Token hirer_token = new Token(hirer);
    hirer.setToken(hirer_token);

    reservationDAO.addReservation("E256743", book_code, Library.LIBRARY_1.toString());

    Reservation reservation = new Reservation(LocalDate.now(), hirer, bookDAO.getBook(book_code),
        Library.LIBRARY_1);

    assertEquals(reservation, reservationDAO.getReservationsByUserCode("E256743").get(0));
```

```
    ReservationController rc = new ReservationController();

    adminDAO.addAdmin("M23234", "Filippo", "Taiti", "ft0011ft@gmail.com", "034567", Library.LIBRARY_1.
        toString());
    Admin admin = adminDAO.getAdmin("M23234");
    Token admin_token = new Token(admin);
    HashMap<Library, PhysicalCopies> pcs = physicalCopiesDAO.getPhysicalCopies(book_code);
    Book book = bookDAO.getBook(book_code);
    book.setPhysicalCopies(pcs);

    rc.confirmReservationWithdraw(hirer, book, Library.LIBRARY_1.toString(), admin_token);

    Lending lending = new Lending(LocalDate.now(), LocalDate.now().plusMonths(1), hirer, bookDAO.
        getBook(book_code), Library.LIBRARY_1);

    assertTrue(reservationDAO.getReservationsByUserCode("E256743").isEmpty());
    assertEquals(lending, lendingDAO.getLendingsByUserCode("E256743").get(0));

}
```

4.2.3 ItemControllerTest

Questa sezione riporta i test relativi alla ricerca, aggiunta, cancellazione e modifica di un articolo nel catalogo. Gli articoli considerati possono essere di tipo *Book*, *Magazine* o *Thesis*. Tuttavia, i test di aggiunta, cancellazione e modifica sono stati implementati esclusivamente per gli articoli di tipo *Book*, poiché la logica risulta analoga anche per le altre due tipologie.

Listing 13: Test relativo alla ricerca di un articolo nel catalogo.

```
@Test
public void testSearchItem(){
    int book_code_1 = bookDAO.addBook("Fondamenti di informatica", LocalDate.of(2023,4,1).toString(),
        Language.LANGUAGE_1.toString(), Category.CATEGORY_1.toString(), "link1", "isbn1",
       "publishing house 1", 200, "authors1");
    int book_code_2 = bookDAO.addBook("Anatomia", LocalDate.of(2023,4,2).toString(), Language.
        LANGUAGE_1.toString(), Category.CATEGORY_1.toString(), "link2", "isbn2", "publishing house 2",
         220, "authors2");

    ArrayList<Item> expected_items = new ArrayList<>();

    expected_items.add(bookDAO.getBook(book_code_1));

    ArrayList<Item> notExpected_items = new ArrayList<>();
    notExpected_items.add(bookDAO.getBook(book_code_2));

    assertEquals(expected_items, itemController.searchItem("Fond", Category.CATEGORY_1.toString()));
    assertNotEquals(notExpected_items, itemController.searchItem("Fond",
       Category.CATEGORY_1.toString()));
}
```

Clicca qui per tornare ai templates

Listing 14: Aggiunta con successo di un Book all'interno del catalogo.

```
@Test
public void testAddBook_Success(){
    adminDAO.addAdmin("uc1", "name", "surname", "mail", "09876", Library.LIBRARY_1.toString());
    Admin admin = adminDAO.getAdmin("uc1");
    Token admin_token = new Token(admin);
    admin.setToken(admin_token);

    itemController.addBook("Fondamenti di informatica", LocalDate.of(2015,2, 3).toString(),
       Language.LANGUAGE_1.toString(), Category.CATEGORY_1.toString(), "link", "isbn", "Mondadori",
       300, "autori", 5, true, admin_token);
```

```
    Book expected_book = new Book(1, "Fondamenti di informatica", LocalDate.of(2015,2, 3),
        Language.LANGUAGE_1, Category.CATEGORY_1, "link", "isbn", "Mondadori", 300, "autori");

    assertEquals(expected_book, bookDAO.getBook(1));
}
```

Listing 15: Fallimento dell'aggiunta di un Book nel catalogo perché ad eseguire l'operazione è un Hirer.

```
@Test
public void testAddBook_Fail(){
    Hirer hirer = new Hirer("usercode", "name", "surname", "email", "00001", null, null);
    Token token = new Token(hirer);
    hirer.setToken(token);

    assertThrows(ActionDeniedException.class, () -> itemController.addBook("title",
        LocalDate.of(2004, 2, 13).toString(), Language.LANGUAGE_1.toString(),
        Category.CATEGORY_1.toString(), "link", "isbn", "publishing house", 200, "authors", 5, true
        , token));
}
```

Clicca qui per tornare ai templates

Listing 16: Cancellazione di un Book dal catalogo effettuata con successo. In questo primo caso sono state cancellate solo le copie del libro dalla rispettiva tabella.

```
@Test
public void removeBook_Success_1(){
    adminDAO.addAdmin("uc1", "name", "surname", "mail", "09876", Library.LIBRARY_1.toString());
    Admin admin = adminDAO.getAdmin("uc1");
    Token admin_token = new Token(admin);
    admin.setToken(admin_token);
    int book_code = bookDAO.addBook("Fondamenti di informatica", LocalDate.of(2015,2, 3).toString(),
        Language.LANGUAGE_1.toString(), Category.CATEGORY_1.toString(), "link", "isbn", "Mondadori",
        300, "autori");
    pcDAO.addPhysicalCopies(book_code, Library.LIBRARY_1.toString(), 5, true);
    HashMap<Library, PhysicalCopies> pcs = pcDAO.getPhysicalCopies(book_code);
    bookDAO.getBook(book_code).setPhysicalCopies(pcs);

    itemController.removeBook(book_code, admin_token);
    assertEquals(Collections.EMPTY_MAP, pcDAO.getPhysicalCopies(book_code));
}
```

Listing 17: Cancellazione di un Book dal catalogo effettuata con successo. In questo secondo caso, dal momento che non ci sono copie di quel libro in alcuna sede, è stato cancellato direttamente l'articolo dalla tabella a cui appartiene.

```
@Test
public void removeBook_Success_2(){
    adminDAO.addAdmin("uc1", "name", "surname", "mail", "09876", Library.LIBRARY_1.toString());
    Admin admin = adminDAO.getAdmin("uc1");
    Token admin_token = new Token(admin);
    admin.setToken(admin_token);
    int book_code = bookDAO.addBook("Fondamenti di informatica", LocalDate.of(2015,2, 3).toString(),
        Language.LANGUAGE_1.toString(), Category.CATEGORY_1.toString(), "", "isbn", "Mondadori", 300,
        "autori");

    itemController.removeBook(book_code, admin_token);
    assertThrows(IdNotFoundException.class, () -> bookDAO.getBook(book_code));
}
```

Listing 18: Fallimento della cancellazione di un Book perché ad eseguire l'operazione è un Hirer.

```
@Test
public void testRemoveBook_Fail1(){
```

```
    Hirer hirer = new Hirer("usercode", "name", "surname", "email", "00001", null, null);
    Token hirer_token = new Token(hirer);
    hirer.setToken(hirer_token);

    int book_code_1 = bookDAO.addBook("Fondamenti di informatica", LocalDate.of(2023,4,1).toString(),
        Language.LANGUAGE_1.toString(), Category.CATEGORY_1.toString(), "link1", "isbn1",
        "publishing house 1", 200, "authors1");


    assertThrows(ActionDeniedException.class, () -> itemController.removeBook(book_code_1,
        hirer_token));
}
```

Listing 19: Fallimento della cancellazione di un Book perché non è presente nella sede dove lavora l'Admin.

```
@Test
public void testRemoveBook_Fail2(){
    Admin admin = new Admin("uc1", "nome", "cognome", "mail", "00000", Library.LIBRARY_1,null);
    Token admin_token = new Token(admin);
    admin.setToken(admin_token);

    itemController.addBook("Anatomia", LocalDate.of(2023,4,2).toString(),
        Language.LANGUAGE_1.toString(), Category.CATEGORY_1.toString(), "link2", "isbn2",
        "publishing house 2", 220, "authors2", 1, true, admin_token);

    Admin admin_2 = new Admin("uc2", "nome2", "cognome2", "mail2", "00002", Library.LIBRARY_2,null);
    Token admin_token_2 = new Token(admin_2);
    admin.setToken(admin_token_2);

    assertThrows(ActionDeniedException.class, () -> itemController.removeBook(1, admin_token_2));
}
```

Listing 20: Fallimento della cancellazione di un Book perché c'è ancora una prenotazione attiva legata ad esso.

```
@Test
public void testRemoveBook_Fail3(){
    Admin admin = new Admin("uc1", "nome", "cognome", "mail", "00000", Library.LIBRARY_1,null);
    Token admin_token = new Token(admin);
    admin.setToken(admin_token);

    int book_code_1 = bookDAO.addBook("Fondamenti di informatica", LocalDate.of(2023,4,1).toString(),
        Language.LANGUAGE_1.toString(), Category.CATEGORY_1.toString(), "link1", "isbn1",
        "publishing house 1", 200, "authors1");

    pcDAO.addPhysicalCopies(book_code_1, Library.LIBRARY_1.toString(), 5, true);

    HashMap<Library, PhysicalCopies> pcs = pcDAO.getPhysicalCopies(book_code_1);
    bookDAO.getBook(book_code_1).setPhysicalCopies(pcs);

    hirerDAO.addHirer("usercode", "name", "surname", "email", "00001");
    reservationDAO.addReservation("usercode", book_code_1, Library.LIBRARY_1.toString());

    assertThrows(ActionDeniedException.class, () -> itemController.removeBook(book_code_1,
        admin_token));
}
```

Clicca qui per tornare ai templates

Listing 21: Modifica con successo di un Book all'interno del catalogo.

```
@Test
public void testUpdateBook_Success(){
    Admin admin = new Admin("uc1", "nome", "cognome", "mail", "00000", Library.LIBRARY_1,null);
    Token admin_token = new Token(admin);
    admin.setToken(admin_token);
```

```
    int book_code_1 = bookDAO.addBook("Anatomia", LocalDate.of(2023,4,2).toString(),
        Language.LANGUAGE_1.toString(), Category.CATEGORY_1.toString(), "link2", "isbn2",
        "publishing house 2", 220, "authors2");
    pcDAO.addPhysicalCopies(book_code_1, Library.LIBRARY_1.toString(), 5, true);

    itemController.updateBook(book_code_1, "Programmazione", LocalDate.of(2020, 4,2).toString(), false
        , Language.LANGUAGE_2.toString(), Category.CATEGORY_2.toString(), "link3", "isbn3",
        "Mondadori", 250, "authors3", 8, admin_token);

    Book expected_book = new Book(book_code_1, "Programmazione", LocalDate.of(2020, 4,2),
        Language.LANGUAGE_2, Category.CATEGORY_2, "link3", "isbn3", "Mondadori", 250, "authors3");


    PhysicalCopies pc = new PhysicalCopies(8, 8, false);
    HashMap<Library, PhysicalCopies> expected_pcs = pcDAO.getPhysicalCopies(book_code_1);
    expected_pcs.put(Library.LIBRARY_2, pc);


    assertEquals(expected_book, bookDAO.getBook(book_code_1));
    assertEquals(8, pcDAO.getPhysicalCopies(book_code_1).get(Library.LIBRARY_1).
        getNumberOfPhysicalCopies());
    assertFalse(pcDAO.getPhysicalCopies(book_code_1).get(Library.LIBRARY_1).isBorrowable());
}
```

Listing 22: Fallimento della modifica di un Book nel catalogo perché ad eseguire l'operazione è un Hirer.

```
@Test
public void testUpdateBook_Fail(){
    Hirer hirer = new Hirer("usercode", "name", "surname", "email", "00001", null, null);
    Token token = new Token(hirer);
    hirer.setToken(token);

    assertThrows(ActionDeniedException.class, () -> itemController.updateBook(1,"title", LocalDate.of
        (2004, 2, 13).toString(), true, Language.LANGUAGE_1.toString(), Category.CATEGORY_1.toString()
        , "link", "isbn", "publishing house", 200, "authors", 5, token));
}
```

Clicca qui per tornare ai templates

4.2.4 LendingControllerTest

In questa sezione vengono mostrati i test dei casi di fallimento della registrazione di un prestito (use case #14) e della restituzione di un articolo (use case #16).

Listing 23: Funzione di setup.

```
private int setup(){
    hirerDAO.addHirer("usercode", "name", "surname", "mail1", "00001");
    return bookDAO.addBook("Fondamenti di informatica", LocalDate.of(2023,4,1).toString(),
        Language.LANGUAGE_1.toString(), Category.CATEGORY_1.toString(), "link1", "isbn1",
        "publishing house 1", 200, "authors1");
}
```

Listing 24: Prestito registrato con successo.

```
@Test
public void registerLending_Success(){
    Book book = bookDAO.getBook(setup());
    pcDAO.addPhysicalCopies(book.getCode(), Library.LIBRARY_1.toString(), 5, true);
    HashMap<Library, PhysicalCopies> pcs = pcDAO.getPhysicalCopies(book.getCode());
    book.setPhysicalCopies(pcs);
    adminDAO.addAdmin("usercode1", "name", "surname", "mail", "00000", Library.LIBRARY_1.toString());
    Admin admin = adminDAO.getAdmin("usercode1");
```

```
    Token token = new Token(admin);
    admin.setToken(token);
    Hirer hirer = hirerDAO.getHirer("usercode");
    lendingController.registerLending(hirer, book, token);

    ArrayList<Lending> lendings = lendingDAO.getLendingsByUserCode("usercode");

    Lending l = lendings.get(0);

    assertTrue(l.getHirer().equals(hirer));
    assertTrue(l.getLendingDate().equals(LocalDate.now()));
    assertTrue(l.getMaturityDate().equals(LocalDate.now().plusMonths(1)));
    assertTrue(l.getItem().equals(book));
    assertTrue(l.getStoragePlace().equals(Library.LIBRARY_1));
}
```

Listing 25: Fallimento della registrazione del prestito perché ad eseguire l'operazione è un Hirer.

```
@Test
public void registerLending_Fail1(){
    Book book = bookDAO.getBook(setup());
    Hirer hirer = hirerDAO.getHirer("usercode");
    Token hirer_token = new Token(hirer);
    hirer.setToken(hirer_token);

    assertThrows(ActionDeniedException.class, () -> lendingController.registerLending(hirer, book,
        hirer_token));
}
```

Listing 26: Fallimento della registrazione del prestito perché l'Hirer è bloccato.

```
@Test
public void registerLending_Fail2(){
    Book book = bookDAO.getBook(setup());
    Hirer hirer = hirerDAO.getHirer("usercode");
    adminDAO.addAdmin("usercode1", "name", "surname", "mail", "00000", Library.LIBRARY_1.toString());
    Admin admin = adminDAO.getAdmin("usercode1");
    Token token = new Token(admin);
    admin.setToken(token);
    hirer.setUnbannedDate(LocalDate.now());

    assertThrows(ActionDeniedException.class, () -> lendingController.registerLending(hirer, book,
        token));
}
```

Listing 27: Fallimento della registrazione del prestito perché il libro non è presente in nessuna sede.

```
@Test
public void registerLending_Fail3(){
    hirerDAO.addHirer("usercode", "name", "surname", "mail", "00001");
    int book_code = bookDAO.addBook("Fondamenti di informatica", LocalDate.of(2023,4,1).toString(),
        Language.LANGUAGE_1.toString(), Category.CATEGORY_1.toString(), "link1", "isbn1",
        "publishing house 1", 200, "authors1");
    Hirer hirer = hirerDAO.getHirer("usercode");
    adminDAO.addAdmin("usercode1", "name", "surname", "mail", "00000", Library.LIBRARY_1.toString());
    Admin admin = adminDAO.getAdmin("usercode1");
    Token token = new Token(admin);
    admin.setToken(token);

    assertThrows(ActionDeniedException.class, () -> lendingController.registerLending(hirer,
        bookDAO.getBook(book_code), token));
}
```

Listing 28: Fallimento della registrazione del prestito perché il libro non ha abbastanza copie nella sede specificata.

```
@Test
public void registerLending_Fail4(){
    hirerDAO.addHirer("usercode", "name", "surname", "mail", "00001");
    int book_code = bookDAO.addBook("Fondamenti di informatica", LocalDate.of(2023,4,1).toString(),
        Language.LANGUAGE_1.toString(), Category.CATEGORY_1.toString(), "link1", "isbn1",
        "publishing house 1", 200, "authors1");
    pcDAO.addPhysicalCopies(book_code, Library.LIBRARY_1.toString(), 2, true);
    Hirer hirer = hirerDAO.getHirer("usercode");
    adminDAO.addAdmin("usercode1", "name", "surname", "mail", "00000", Library.LIBRARY_1.toString());
    Admin admin = adminDAO.getAdmin("usercode1");
    Token token = new Token(admin);
    admin.setToken(token);

    ReservationDAO reservationDAO = new ReservationDAO();
    reservationDAO.addReservation("usercode", book_code, Library.LIBRARY_1.toString());
    assertThrows(ActionDeniedException.class, () -> lendingController.registerLending(hirer,
       bookDAO.getBook(book_code), token));
}
```

Clicca qui per tornare ai templates

Listing 29: Libro restituito con successo alla sede bibliotecaria.

```
@Test
public void registerReturnOfItem_Success(){
    Book book = bookDAO.getBook(setup());
    pcDAO.addPhysicalCopies(book.getCode(), Library.LIBRARY_1.toString(), 5, true);
    Hirer hirer = hirerDAO.getHirer("usercode");
    Admin admin = new Admin("uc1", "nome", "cognome", "mail", "00002", Library.LIBRARY_1, null);

    Token admin_token = new Token(admin);
    admin.setToken(admin_token);

    lendingDAO.addLending("usercode", book.getCode(), Library.LIBRARY_1.toString());
    waitingListDAO.addToWaitingList(book.getCode(), Library.LIBRARY_1.toString(), hirer.getEmail());
    lendingController.registerReturnOfItem(hirer, book, Library.LIBRARY_1.toString(), admin_token);

    assertTrue(lendingDAO.getLendingsByUserCode("usercode").isEmpty());
    assertTrue(waitingListDAO.getWaitingList(book.getCode(), Library.LIBRARY_1.toString()).isEmpty());
}
```

Listing 30: Fallimento della registrazione della restituzione del libro perché ad eseguire l'operazione è un Hirer.

```
@Test
public void registerReturnOfItem_Fail1(){
    Book book = bookDAO.getBook(setup());
    Hirer hirer = hirerDAO.getHirer("usercode");
    Token hirer_token = new Token(hirer);
    hirer.setToken(hirer_token);

    assertThrows(ActionDeniedException.class, () -> lendingController.registerReturnOfItem(hirer, book
        , Library.LIBRARY_1.toString(), hirer_token));
}
```

Listing 31: Fallimento della registrazione della restituzione del libro perché non è stato noleggiato nella sede dove lavora l'admin.

```
@Test
public void registerReturnOfItem_Fail2(){
    Book book = bookDAO.getBook(setup());
    adminDAO.addAdmin("usercode1", "name", "surname", "mail", "00000", Library.LIBRARY_1.toString());
```

```
    Admin admin = adminDAO.getAdmin("usercode1");
    Token token = new Token(admin);
    admin.setToken(token);
    Hirer hirer = hirerDAO.getHirer("usercode");
    assertThrows(ActionDeniedException.class, () -> lendingController.registerReturnOfItem(hirer, book
        , Library.LIBRARY_2.toString(), token));
}
```

Clicca qui per tornare ai templates

4.2.5 ReservationControllerTest

In questa sezione si riportano i test relativi ai casi di fallimento della prenotazione di un articolo, cancellazione di una prenotazione e conferma ritiro di un articolo).

Listing 32: Funzione di setup.

```
private int setup(){
    hirerDAO.addHirer("usercode", "name", "surname", "biblioteca.SWE@gmail.com", "00001");
    return bookDAO.addBook("Fondamenti di informatica", LocalDate.of(2023,4,1).toString(),
        Language.LANGUAGE_1.toString(), Category.CATEGORY_1.toString(), "link1", "isbn1",
            "publishing house 1", 200, "authors1");
}
```

Listing 33: Prenotazione registrata con successo.

```
@Test
public void testReserveItem_Success(){
    Book book = bookDAO.getBook(setup());
    pcDAO.addPhysicalCopies(book.getCode(), Library.LIBRARY_1.toString(), 5, true);
    Hirer hirer = hirerDAO.getHirer("usercode");

    Token hirer_token = new Token(hirer);
    hirer.setToken(hirer_token);

    reservationController.reserveItem(hirer, book, Library.LIBRARY_1.toString(), hirer_token);

    ArrayList<Reservation> reservations = reservationDAO.getReservationsByUserCode("usercode");
    Reservation r = reservations.get(0);

    assertTrue(r.getHirer().equals(hirer));
    assertTrue(r.getItem().equals(book));
    assertTrue(r.getStoragePlace().equals(Library.LIBRARY_1));
    assertTrue(r.getReservationDate().equals(LocalDate.now()));
}
```

Listing 34: Fallimento della registrazione della prenotazione del libro perché ad eseguire l'operazione è un Admin.

```
@Test
public void testReserveItem_Fail1(){
    Book book = bookDAO.getBook(setup());
    adminDAO.addAdmin("usercode1", "name", "surname", "mail", "00000", Library.LIBRARY_1.toString());
    Admin admin = adminDAO.getAdmin("usercode1");
    Token token = new Token(admin);
    admin.setToken(token);

    assertThrows(ActionDeniedException.class, () -> reservationController.reserveItem(hirerDAO.
        getHirer("usercode"), book, Library.LIBRARY_1.toString(), token));
}
```

Listing 35: Fallimento della registrazione della prenotazione del libro perché l'Hirer è bloccato.

```
@Test
public void testReserveItem_Fail2(){
    Book book = bookDAO.getBook(setup());
    Hirer hirer = hirerDAO.getHirer("usercode");
    Token hirer_token = new Token(hirer);
    hirer.setToken(hirer_token);
    hirer.setUnbannedDate(LocalDate.now());

    assertThrows(ActionDeniedException.class, () -> reservationController.reserveItem(hirer, book,
        Library.LIBRARY_1.toString(), hirer_token));
}
```

Listing 36: Fallimento della registrazione della prenotazione del libro perché il libro non è prestabile.

```
@Test
    public void testReserveItem_Fail_3(){
        Book book = bookDAO.getBook(setup());
        pcDAO.addPhysicalCopies(book.getCode(), Library.LIBRARY_1.toString(), 5, false);
        HashMap<Library, PhysicalCopies> pcs = pcDAO.getPhysicalCopies(1);
        book.setPhysicalCopies(pcs);
        Hirer hirer = hirerDAO.getHirer("usercode");

        Token hirer_token = new Token(hirer);
        hirer.setToken(hirer_token);

        assertThrows(ActionDeniedException.class, () -> {reservationController.reserveItem(hirer, book,
             Library.LIBRARY_1.toString(), hirer_token);});
    }
```

Listing 37: Fallimento della registrazione della prenotazione del libro perché non ci sono abbastanza copie nella sede specificata.

```
@Test
    public void testReserveItem_Fail_4(){
        Book book = bookDAO.getBook(setup());
        pcDAO.addPhysicalCopies(book.getCode(), Library.LIBRARY_1.toString(), 1, true);
        HashMap<Library, PhysicalCopies> pcs = pcDAO.getPhysicalCopies(1);
        book.setPhysicalCopies(pcs);
        Hirer hirer = hirerDAO.getHirer("usercode");
        Token hirer_token = new Token(hirer);
        hirer.setToken(hirer_token);

        assertThrows(ActionDeniedException.class, () -> {reservationController.reserveItem(hirer, book,
             Library.LIBRARY_1.toString(), hirer_token);});
    }
```

Clicca qui per tornare ai templates

Listing 38: Prenotazione cancellata con successo.

```
@Test
public void testRemoveReservation_Success(){
    Book book = bookDAO.getBook(setup());
    pcDAO.addPhysicalCopies(book.getCode(), Library.LIBRARY_1.toString(), 5, true);
    Hirer hirer = hirerDAO.getHirer("usercode");
    Admin admin = new Admin("uc1", "nome", "cognome", "mail", "00002", Library.LIBRARY_1, null);

    Token admin_token = new Token(admin);
    admin.setToken(admin_token);

    reservationDAO.addReservation("usercode", book.getCode(), Library.LIBRARY_1.toString());

    reservationController.removeReservation(hirer, book, Library.LIBRARY_1.toString(), admin_token);
```

```
    assertTrue(reservationDAO.getReservationsByUserCode("usercode").isEmpty());
    assertTrue(waitingListDAO.getWaitingList(book.getCode(), Library.LIBRARY_1.toString()).isEmpty());
}
```

Listing 39: Fallimento della cancellazione della prenotazione del libro perché non è stata effettuata presso la sede dove lavora l'Admin.

```
@Test
public void testRemoveReservation_Fail1(){
    Book book = bookDAO.getBook(setup());
    Hirer hirer = hirerDAO.getHirer("usercode");
    adminDAO.addAdmin("usercode1", "name", "surname", "mail", "00000", Library.LIBRARY_1.toString());
    Admin admin = adminDAO.getAdmin("usercode1");
    Token token = new Token(admin);
    admin.setToken(token);

    assertThrows(ActionDeniedException.class, () -> reservationController.removeReservation(hirer,
        book, Library.LIBRARY_2.toString(), token));
}
```

Clicca qui per tornare ai templates

Listing 40: Registrazione del ritiro del libro prenotato effettuata con successo.

```
@Test
public void testConfirmReservationWithdraw_Success(){
    Book book = bookDAO.getBook(setup());
    Hirer hirer = hirerDAO.getHirer("usercode");
    pcDAO.addPhysicalCopies(book.getCode(), Library.LIBRARY_1.toString(), 5, true);
    HashMap<Library, PhysicalCopies> pcs = pcDAO.getPhysicalCopies(book.getCode());
    book.setPhysicalCopies(pcs);
    adminDAO.addAdmin("usercode1", "name", "surname", "mail", "00000", Library.LIBRARY_1.toString());
    Admin admin = adminDAO.getAdmin("usercode1");
    Token admin_token = new Token(admin);
    admin.setToken(admin_token);

    reservationDAO.addReservation("usercode", book.getCode(), Library.LIBRARY_1.toString());

    reservationController.confirmReservationWithdraw(hirer, book, Library.LIBRARY_1.toString(),
        admin_token);

    assertTrue(reservationDAO.getReservationsByUserCode("usercode").isEmpty());

    ArrayList<Lending> lendings = lendingDAO.getLendingsByUserCode("usercode");

    Lending l = lendings.get(0);


    assertTrue(l.getHirer().equals(hirer));
    assertTrue(l.getItem().equals(book));
    assertTrue(l.getLendingDate().equals(LocalDate.now()));
    assertTrue(l.getMaturityDate().equals(LocalDate.now().plusMonths(1)));
    assertTrue(l.getStoragePlace().equals(Library.LIBRARY_1));
}
```

Listing 41: Fallimento della registrazione della conferma del ritiro del libro perché ad eseguire l'operazione è un Hirer.

```
@Test
public void testConfirmReservationWithdraw_Fail1(){
    Book book = bookDAO.getBook(setup());
    Hirer hirer = hirerDAO.getHirer("usercode");
    Token hirer_token = new Token(hirer);
```

```
    hirer.setToken(hirer_token);

    assertThrows(ActionDeniedException.class, () -> reservationController.confirmReservationWithdraw(
        hirer, book, Library.LIBRARY_1.toString(), hirer_token));
}
```

Listing 42: Fallimento della registrazione della conferma del ritiro del libro perché non è stato prenotato presso la sede dove lavora l'admin.

```
@Test
public void testConfirmReservationWithdraw_Fail2(){
    Book book = bookDAO.getBook(setup());
    adminDAO.addAdmin("usercode1", "name", "surname", "mail", "00000", Library.LIBRARY_1.toString());
    Admin admin = adminDAO.getAdmin("usercode1");
    Token token = new Token(admin);
    admin.setToken(token);

    assertThrows(ActionDeniedException.class, () -> reservationController.confirmReservationWithdraw(
        hirerDAO.getHirer("usercode"), book, Library.LIBRARY_2.toString(), token));
}
```

Clicca qui per tornare ai templates

4.3 ORM

In questa sezione dedicata ai test dell'ORM si riportano solo i test relativi ai metodi della classe BookDAO dal momento che i test delle altre classi sono concettualmente analoghi.
I test sono stati effettuati sul DB, ripulendo le tabelle prima dell'esecuzione di ogni test.

4.3.1 BookDAOTest

Listing 43: Getter di un libro.

```
@Test
public void testGetBook(){

    int book_code = bookDAO.addBook("titolo", LocalDate.of(2023,4,5).toString(),
        Language.LANGUAGE_1.toString(), Category.CATEGORY_1.toString(), "link", "isbn",
        "publishing house", 200, "authors");
    Book book = new Book(book_code, "titolo", LocalDate.of(2023,4,5), Language.LANGUAGE_1, Category.
        CATEGORY_1, "link", "isbn", "publishing house", 200, "authors");

    assertEquals(book, bookDAO.getBook(book_code));
    assertThrows(IdNotFoundException.class, () -> bookDAO.getBook(book_code+1));

}
```

Listing 44: Aggiunta di un libro nel database.

```
@Test
public void testAddBook(){
    Book book_1 = new Book(1, "titolo1", LocalDate.of(2023,4,1), Language.LANGUAGE_1,
        Category.CATEGORY_1, "link1", "isbn1", "publishing house 1", 200, "authors1" );

    int itemCode = bookDAO.addBook("titolo1", LocalDate.of(2023,4,1).toString(), Language.LANGUAGE_1.
        toString(), Category.CATEGORY_1.toString(), "link1", "isbn1", "publishing house 1", 200,
        "authors1");
    book_1.setCode(itemCode);
    Book copy_of_book_1 = bookDAO.getBook(book_1.getCode());
    assertEquals(book_1, copy_of_book_1);
    assertNotEquals(3, bookDAO.addBook("titolo2", LocalDate.of(2023,4,2).toString(), Language.
        LANGUAGE_2.toString(), Category.CATEGORY_2.toString(), "link2", "isbn2", "publishing house 2",
         200, "authors2"));
```

```
}
```

Listing 45: Cancellazione di un libro dal database.

```
@Test
public void testRemoveBook(){

    int book_code = bookDAO.addBook("titolo1", LocalDate.of(2023,4,1).toString(), Language.LANGUAGE_1.
        toString(), Category.CATEGORY_1.toString(), "link1", "isbn1", "publishing house 1", 200,
        "authors1");
    HirerDAO hirerDAO = new HirerDAO();
    hirerDAO.addHirer("uc1", "name", "surname", "email", "00000");
    PhysicalCopiesDAO physicalCopiesDAO = new PhysicalCopiesDAO();
    physicalCopiesDAO.addPhysicalCopies(book_code, Library.LIBRARY_1.toString(), 2, true);
    LendingDAO lendingDAO = new LendingDAO();
    lendingDAO.addLending("uc1", book_code, Library.LIBRARY_1.toString());

    //id non presente
    assertThrows(IdNotFoundException.class, () -> bookDAO.removeBook(3));
    //cancellazione di book con ancora un prestito non restituito
    assertThrows(ConstraintViolationException.class, () -> bookDAO.removeBook(book_code));

    //cancellazione di un magazine tramite removeBook
    MagazineDAO magazineDAO = new MagazineDAO();
    int magazineCode = magazineDAO.addMagazine("titolo3", "2003-03-03", "lingua1", "categoria", "link"
        , "publishinghouse", 200);
    assertThrows(IdNotFoundException.class, () -> bookDAO.removeBook(magazineCode));

    //cancellazione con successo
    int book_code2 = bookDAO.addBook("titolo1", LocalDate.of(2023,4,1).toString(), Language.LANGUAGE_1
        .toString(), Category.CATEGORY_1.toString(), "link1", "isbn1", "publishing house 1", 200, "
        authors1");
    bookDAO.getBook(book_code2);
    bookDAO.removeBook(book_code2);
    assertThrows(IdNotFoundException.class, () -> bookDAO.getBook(book_code2));
}
```

Listing 46: Modifica di un libro all'interno del database.

```
@Test
public void testUpdateBook(){

    Book book_2 = new Book(2, "titolo2", LocalDate.of(2023,4,2), Language.LANGUAGE_2,
        Category.CATEGORY_2, "link2", "isbn2", "publishing house 2", 200, "authors2" );
    int itemCode = bookDAO.addBook("titolo1", LocalDate.of(2023,4,1).toString(),
        Language.LANGUAGE_1.toString(), Category.CATEGORY_1.toString(), "link1", "isbn1",
        "publishing house 1", 200, "authors1");
    //item code non presente nel database
    assertThrows(IdNotFoundException.class, () -> bookDAO.updateBook(99, "titolo2",
        LocalDate.of(2023,4,2).toString(), Language.LANGUAGE_2.toString(),
        Category.CATEGORY_2.toString(), "link2", "isbn2", "publishing house 2", "authors2", 200));

    //update con successo
    Book book_inserted = bookDAO.getBook(itemCode);
    book_2.setCode(itemCode);
    assertNotEquals(book_2, book_inserted);
    bookDAO.updateBook(itemCode, "titolo2", LocalDate.of(2023,4,2).toString(), Language.LANGUAGE_2.
        toString(), Category.CATEGORY_2.toString(), "link2", "isbn2", "publishing house 2", "authors2"
        , 200);
    book_inserted = bookDAO.getBook(itemCode);
    assertEquals(book_2, book_inserted);

    //update di un magazine tramite updateBook
    MagazineDAO magazineDAO = new MagazineDAO();
    int magazineCode = magazineDAO.addMagazine("titolo3", "2003-03-03", "lingua1", "categoria", "link"
```

```
        , "publishinghouse", 200);
    assertThrows(IdNotFoundException.class, ()-> bookDAO.updateBook(magazineCode, "titolo2",
        LocalDate.of(2023,4,2).toString(), Language.LANGUAGE_2.toString(),
        Category.CATEGORY_2.toString(), "link2", "isbn2", "publishing house 2", "authors2", 200));
}
```

Listing 47: Get di tutti i libri presenti nel database.

```
@Test
public void testGetAllBooks(){
    int code_1 = bookDAO.addBook("titolo1", LocalDate.of(2023,4,1).toString(),
        Language.LANGUAGE_1.toString(), Category.CATEGORY_1.toString(), "link1", "isbn1",
        "publishing house 1", 200, "authors1");
    int code_2 = bookDAO.addBook("titolo2", LocalDate.of(2023,4,2).toString(),
        Language.LANGUAGE_1.toString(), Category.CATEGORY_1.toString(), "link2", "isbn2",
        "publishing house 2", 220, "authors2");

    Book book_1 = bookDAO.getBook(code_1);
    Book book_2 = bookDAO.getBook(code_2);
    ArrayList<Book> expected = new ArrayList<>();
    expected.add(book_1);
    expected.add(book_2);

    ArrayList<Book> notExpected = new ArrayList<>();
    notExpected.add(book_1);

    assertEquals(expected, bookDAO.getAllBooks());
    assertNotEquals(notExpected, bookDAO.getAllBooks());
}
```

5 Conclusione

Per concludere, dal momento che alcune porzioni di codice non sono state riportate per ridondanza o se sono sorti dubbi, è disponibile il link di GitHub:

https://github.com/fili-taiz/Biblioteca-SWE-.git