Relazione sul Sistema di Prenotazione Cinema in Java

Niccolò Leoncini - Leonardo Petroni

Settembre 2024

Ingegneria del Software

Indice

1 Introduzione

1.1 Motivazione e descrizione

La scelta di sviluppare un sistema di prenotazione dei posti in un cinema è stata motivata dalla volontà di affrontare un problema reale. Il progetto consente di gestire in maniera semplice ed efficiente la prenotazione di posti, operazione che coinvolge numerosi aspetti come la gestione della disponibilità, l'assegnazione dei posti e l'interazione con l'utente, rendendolo un contesto ideale per applicare concetti fondamentali della programmazione software.

L'uso di Java come linguaggio di sviluppo si è rivelato particolarmente adatto per diverse ragioni. Inoltre, per garantire una gestione efficiente e centralizzata delle informazioni, è stato integrato un database, che permette di archiviare e manipolare i dati relativi alle sale, ai posti disponibili e alle prenotazioni in modo persistente. Questo approccio facilita la gestione delle variabili del progetto e rende il sistema più robusto e scalabile.

Il progetto segue un'architettura orientata ai servizi, in cui la business logic è separata dall'accesso ai dati tramite l'uso di **DAO (Data Access Object)**. I DAO si occupano di gestire le operazioni CRUD sul database, mentre i servizi gestiscono le operazioni tra la business logic e i repository. L'uso di pattern come il **Singleton**, implementato per la gestione centralizzata del database, e l'**Observer**, per gestire l'eliminazione in cascata tra le entità correlate, ha contribuito a migliorare la manutenibilità e la scalabilità del sistema.

2 Analisi

2.1 Problem Statement

Il sistema di prenotazione dei posti al cinema deve affrontare la sfida di gestire in modo efficace e automatizzato la selezione e la prenotazione dei posti da parte degli utenti, garantendo che le informazioni sui posti disponibili siano sempre aggiornate e che le prenotazioni siano processate in modo sicuro e affidabile. L'obiettivo principale è fornire un'interfaccia semplice (per ora via terminale) per gli utenti finali, che permetta loro di visualizzare la disposizione dei posti, selezionare quelli desiderati e completare la prenotazione grazie a dei crediti.

2.2 Requisiti Funzionali

I requisiti funzionali per il sistema includono:

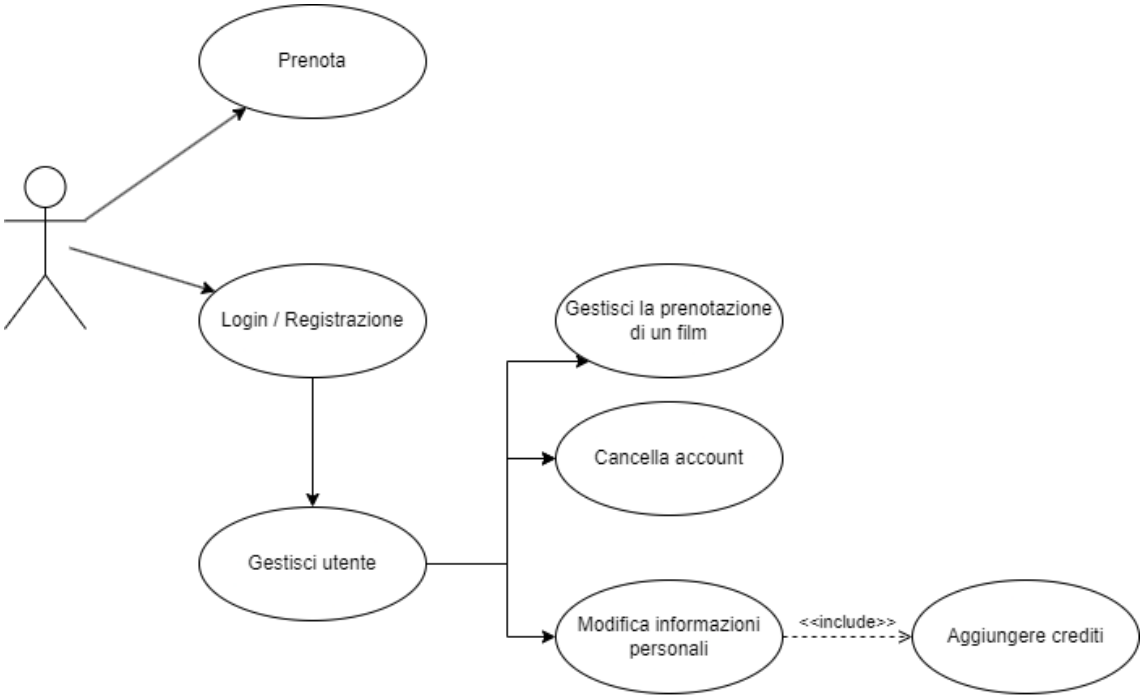
- **Profilo utente**: Un utente può creare il proprio account, ricaricare i propri crediti e gestire le proprie prenotazioni.
- **Visualizzazione dei posti**: L'utente può vedere la disposizione dei posti in una sala cinema.
- **Prenotazione di un posto**: L'utente può selezionare e prenotare un posto disponibile.
- **Interfaccia**: L'interazione dell'utente con l'applicativo avviene tramite terminale.

2.3 Requisiti Non Funzionali

I requisiti non funzionali includono:

- **Performance**: Il sistema deve rispondere in modo rapido, soprattutto durante la visualizzazione e la selezione dei posti.
- **Usabilità**: L'interfaccia utente deve essere semplice e intuitiva per garantire che gli utenti possano prenotare i posti senza difficoltà.
- **Manutenibilità**: Il codice deve essere strutturato in modo da facilitare eventuali modifiche o estensioni future.

2.4 Use Case Diagram



3 Progettazione e implementazione

3.1 Gestione dell'Interfaccia Utente

L'interfaccia utente nel sistema di prenotazione cinema è gestita dalla classe `InputOutputHandler`, situata nel pacchetto `ui`. Questa classe si occupa della comunicazione tra l'utente e il sistema attraverso il terminale. Utilizzando un sistema di enumerazioni (`Page`), la classe gestisce la navigazione tra le diverse pagine del sistema, offrendo all'utente la possibilità di gestire il proprio account, selezionare i film, gli orari e i posti, e confermare le prenotazioni.

- **Navigazione**: La classe permette all'utente di navigare tra diverse pagine, come la homepage, la pagina di gestione account e la gestione delle prenotazioni.
- **Interazione con l'utente**: Attraverso metodi come `cinemaSelectionPage()`, l'utente può inserire opzioni per scegliere cinema, film e orari di proiezione. La classe gestisce anche controlli di input per assicurare che vengano inserite scelte valide e coerenti.
- **Gestione delle Prenotazioni**: La classe guida l'utente nel processo di selezione dei film e dei posti, interfacciandosi con `CinemaService` per confermare le prenotazioni o gestire eventuali rimborsi in caso di cancellazione delle prenotazioni.

La classe `InputOutputHandler` rappresenta un elemento essenziale per garantire una corretta interazione tra l'utente e il sistema, mantenendo l'esperienza dell'utente semplice e intuitiva tramite l'uso del terminale.

3.2 Scelte di Progettazione

Il sistema di prenotazione è stato progettato utilizzando un'architettura orientata ai servizi, separando la business logic (gestita nei service e nei repository) dalla gestione dei dati e dall'interazione con l'utente.

Di seguito vengono descritti i componenti principali e i diversi design pattern coinvolti nel progetto e la loro implementazione:

- **Repositories**: Le classi `Repository` gestiscono l'interazione con i DAO e fungono da intermediari tra la business logic e il livello di accesso ai dati. I repository utilizzano i DAO per eseguire operazioni CRUD e nascondono i dettagli di accesso ai dati alla logica di business, mantenendo una chiara separazione dei livelli.

- **DAO (Data Access Object)**: Le classi DAO si occupano delle operazioni CRUD sui dati nel database. I DAO incapsulano la logica di accesso ai dati, fornendo metodi per interrogare, aggiornare e cancellare i record relativi alle sale, ai posti e alle prenotazioni. I DAO interagiscono con il database attraverso la singola istanza di `CinemaDatabase`, garantendo la persistenza dei dati.

- **Servizi**: La classe `CinemaService` e la sua implementazione `CinemaServiceImpl` incapsulano la business logic legata alla gestione delle sale e dei cinema. Questa classe si occupa di gestire le operazioni tra i repository e le classi che gestiscono i dati. `BookingService` gestisce la logica relativa alle prenotazioni, collegando i posti selezionati dagli utenti alle sale cinematografiche disponibili.

- **Observer**: Utilizzato per gestire l'eliminazione in cascata tra oggetti correlati nel sistema. La classe `Observer.java` notifica i componenti correlati quando un'entità viene rimossa, garantendo che, ad esempio, un `Seat` eliminato venga rimosso anche dalla `Hall` corrispondente. Questo assicura la coerenza dei dati tra le entità.

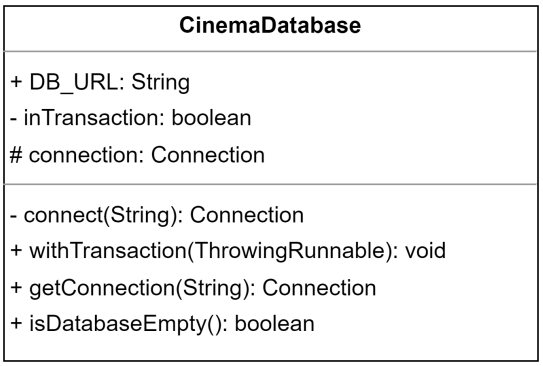
- **Singleton**: Il pattern Singleton è implementato nella classe `CinemaDatabase.java`, che garantisce un unico punto di accesso centralizzato per la gestione delle operazioni sul database. I DAO utilizzano questa singola istanza di `CinemaDatabase` per eseguire operazioni di accesso e gestione dei dati. Questo approccio assicura la consistenza dei dati e l'efficienza nel controllo dell'accesso ai dati.

- **Factory**: Il pattern Factory è implementato nella classe `HallFactory.java`, che si occupa della creazione di oggetti di tipo `Hall`. Questo pattern fornisce un'interfaccia per creare sale cinematografiche in base alle esigenze dell'applicazione, migliorando l'estensibilità del sistema.

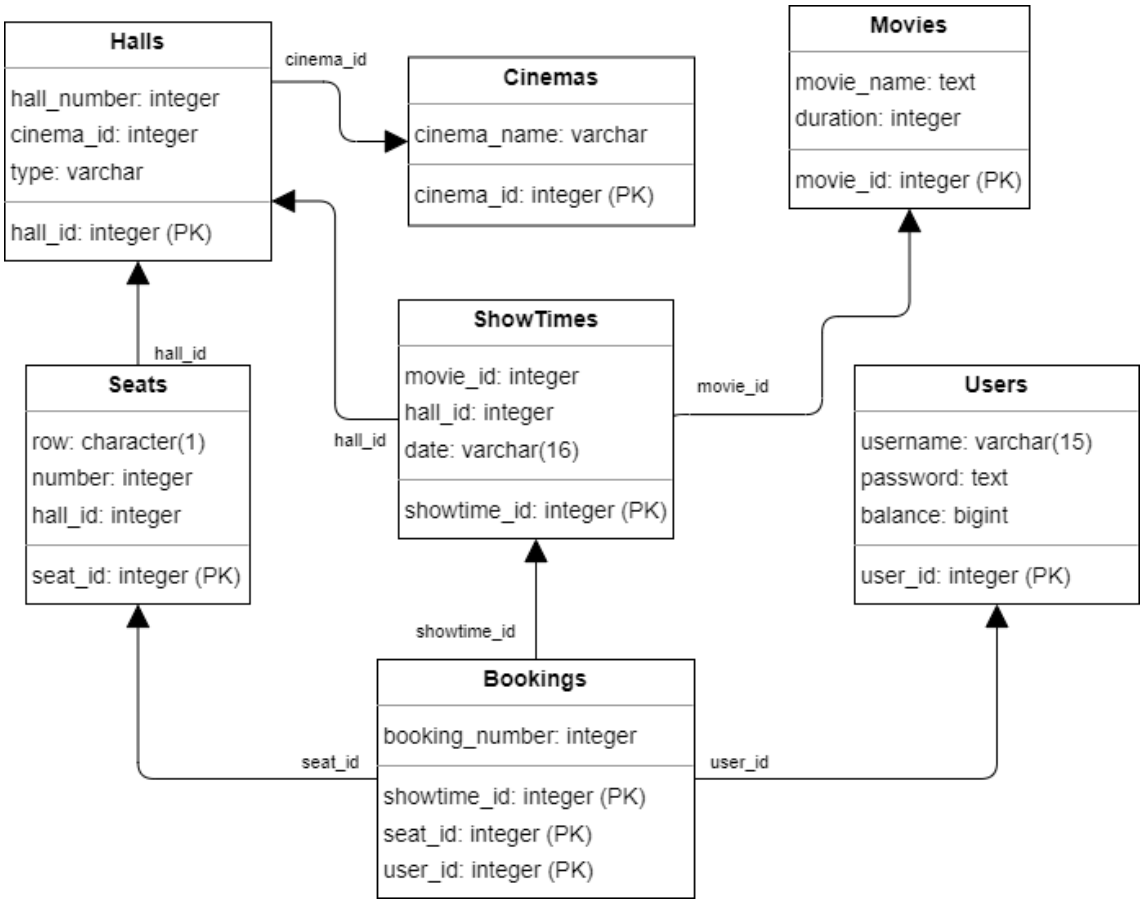
3.2.1 Interazione con il Database

L'applicazione utilizza un database relazionale per memorizzare i dati relativi ai cinema, ai posti e alle prenotazioni. Per garantire un accesso centralizzato e mantenere la consistenza delle operazioni, viene utilizzato il pattern Singleton implementato nella classe `CinemaDatabase`. Questa classe gestisce le connessioni al database e assicura che tutte le operazioni di accesso ai dati vengano eseguite attraverso una

singola istanza.
`CinemaDatabase` fornisce metodi per aprire e chiudere le connessioni al database, mantenendo il controllo sulle operazioni CRUD (Create, Read, Update, Delete) eseguite dai DAO.



Di seguito è mostrato lo schema relazionale:



3.3 Descrizione dei layers e delle classi

In questa sezione verranno descritte le classi principali che compongono il sistema di prenotazione. Queste classi sono organizzate nei seguenti pacchetti: `model`, `dao`, `repositories`, e `service`. Ogni pacchetto ha uno scopo ben definito nell'architettura del sistema, contribuendo alla separazione delle responsabilità.

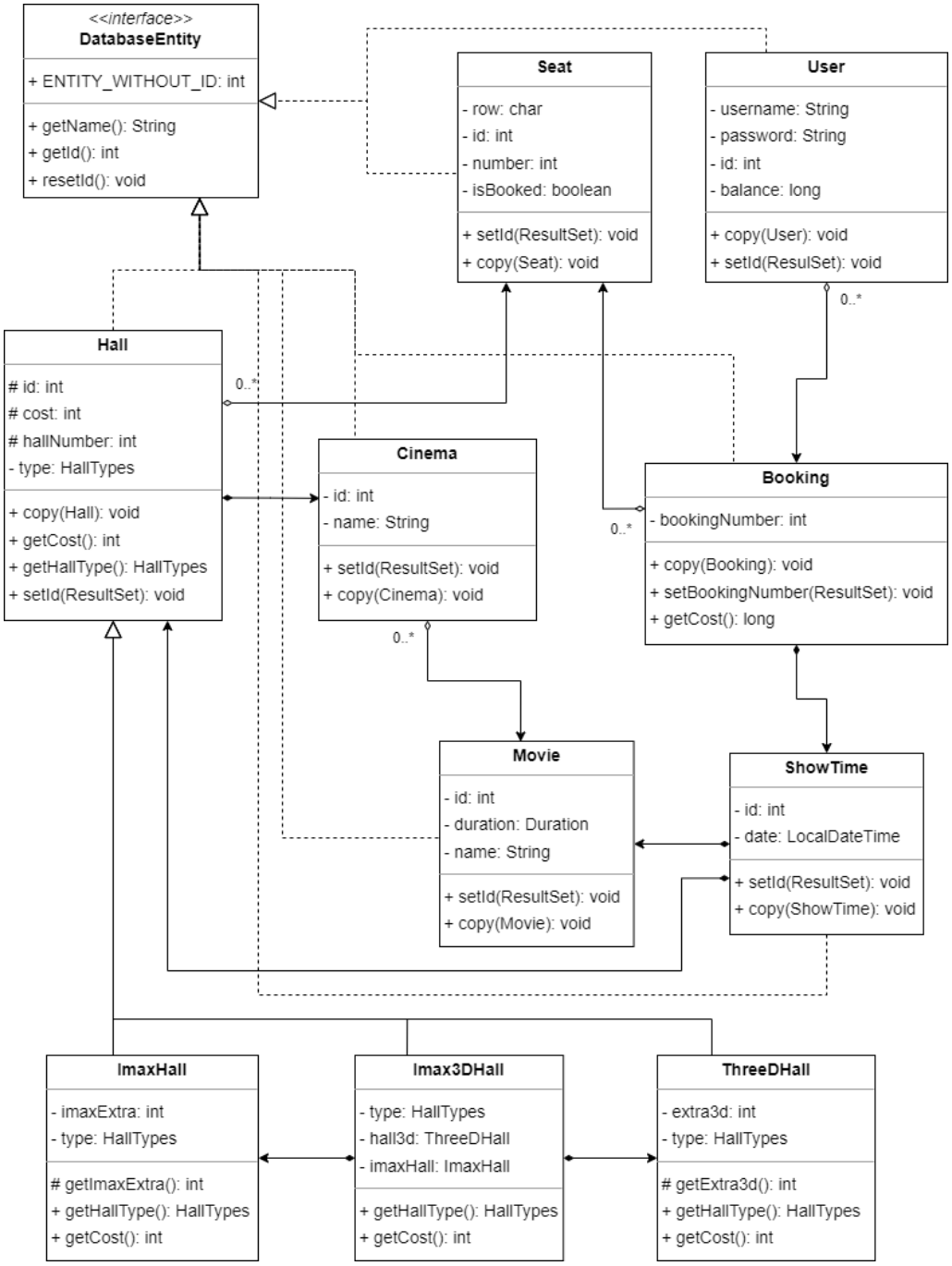
3.3.1 Model

Le classi nel pacchetto `model` rappresentano le entità del database e i loro dati. Tutte le classi implementano l'interfaccia `DatabaseEntity`, hanno un ID che può essere modificato soltanto attraverso un oggetto `ResultSet`, ottenibile solamente eseguendo una query, risulta quindi impossibile manipolare gli ID al di fuori del **Data Access Layer**. Tutte le classi presentano inoltre un costruttore di copia, che copia tutti i dati ad eccezione dell'ID, un metodo `copy`, che copia tutti i dati sul posto, sempre ad eccezione dell'ID e un metodo `resetId` che serve per ripristinare l'ID pari a `ENTITY_WITHOUT_ID`, valore che indica che l'oggetto non rappresenta nessuna entità nel database.

- **Cinema.java**: Questa classe rappresenta un cinema. Contiene informazioni come il nome del cinema e una lista di sale (`Hall`). La classe gestisce anche operazioni per aggiungere e rimuovere sale.
- **Hall.java**: Questa classe rappresenta una sala all'interno del cinema. Ogni sala ha un nome, una capienza (numero di posti) e una mappa dei posti disponibili. La classe `Hall` gestisce la disposizione dei posti, permette di verificare la disponibilità e supporta la prenotazione dei posti.
- **Seat.java**: Questa classe rappresenta un singolo posto in una sala cinematografica. Ogni posto ha uno stato (`disponibile`, `occupato`) e può cambiare stato in base alle azioni dell'utente. La classe `Seat` interagisce con la classe `Hall` per gestire le operazioni relative ai posti.
- **Booking.java**: Questa classe gestisce le prenotazioni. Ogni prenotazione è associata a un utente e a uno o più posti. La classe `Booking` contiene metodi per la creazione, cancellazione e modifica delle prenotazioni. Essa interagisce con le classi `Seat` e `Hall` per verificare la disponibilità dei posti prima di confermare una prenotazione.
- **ShowTime.java**: Questa classe rappresenta le proiezioni cinematografiche all'interno del sistema. Ogni proiezione è associata a una sala (`Hall`) e a un film (`Movie`), e viene identificata da una data e un orario specifici. Questa classe svolge un ruolo fondamentale nella gestione della programmazione dei film, permettendo agli utenti di selezionare le proiezioni disponibili durante il processo di prenotazione. La sua interazione con le classi `Cinema`, `Hall` e `Movie` assicura che le informazioni relative alla proiezione siano sempre aggiornate e coerenti.
- **User.java**: Rappresenta un utente del sistema, che può effettuare prenotazioni. Questa classe include informazioni relative all'utente come nome, crediti

disponibili e la cronologia delle prenotazioni. La classe `User` è utilizzata principalmente durante il processo di prenotazione, dove l'utente seleziona i posti e completa l'acquisto.

Di seguito il diagramma UML che rappresenta le relazioni tra le entità del pacchetto `model`.



3.3.2 DAL (Data Access Layer)

Le classi DAO si occupano dell'accesso e della gestione dei dati nel database. Il loro obiettivo è mantenere la logica di persistenza separata dalla business logic. Ogni DAO implementa il pattern **Singleton** che permette di limitare il numero di

classi istanziabili al numero di database presenti nel sistema. Di seguito descriviamo alcune di queste classi e mostriamo le query più complesse che vengono eseguite.

- **BookingDao.java**: è la classe che deve gestire tutte le operazioni nel database riguardanti le prenotazioni. Inoltre deve anche aggiornare il credito dell'utente, poiché queste query devono avvenire all'interno di una transazione, in modo da fare rollback in caso qualcuna dovesse fallire.

  Mostriamo di seguito il codice dell'inserimento che comprende 3 query, dove la più interessante è la `SELECT`, operazione che serve per ottenere il numero più piccolo che non compare nella colonna `booking_number` che a sua volta è composta da altre 3 `SELECT`, dove: la seconda ritorna semplicemente 1 se non è presente nella colonna; la terza, dato un ID, ritorna ID + 1 se ID + 1 non è presente; infine la prima estrae il numero minimo tra i 2 valori sopra.

BookingDaoImpl.java - insert

```
public void insert(@NotNull Booking booking, @NotNull User user, @NotNull User
    copy){
    Connection conn = CinemaDatabase.getConnection(dbUrl);
    boolean oldAutoCommit = conn.getAutoCommit();
    conn.setAutoCommit(false);
    try(PreparedStatement s = conn.prepareStatement(
            "UPDATE Users SET balance = ? WHERE user_id = ?"
    )) {
        [...]
        try (PreparedStatement ps = conn.prepareStatement(
                "SELECT MIN(t) AS booking_number FROM (SELECT DISTINCT 1 AS t FROM
                    Bookings WHERE (SELECT MIN(booking_number) FROM Bookings) > 1
                    UNION SELECT Bookings.booking_number + 1 FROM Bookings WHERE
                    booking_number + 1 NOT IN (SELECT booking_number FROM
                    Bookings))"
        )) {
            try (ResultSet res = ps.executeQuery()) {
                [...]
                StringBuilder sql = new StringBuilder("INSERT INTO
                    Bookings(showtime_id, seat_id, user_id, booking_number) VALUES
                    ");
                for (Seat seat : seats) {
                    sql.append("(%d, %d, %d, %d), ".formatted(showTime.getId(),
                        seat.getId(), user.getId(), bookingNumber));
                }
                sql.replace(sql.length() - 2, sql.length(), "");
                try (PreparedStatement s1 = conn.prepareStatement(sql.toString()))
                    {
                    [...]
                }
                booking.setBookingNumber(res);
            }
        }
        conn.commit();
    } catch (SQLiteException | NullPointerException e) {
        conn.rollback();
        [...]
    } finally {
        conn.setAutoCommit(oldAutoCommit);
        if (conn.getAutoCommit())
            conn.close();
    }
}
```

Per aggiornare una prenotazione invece la cancelliamo, inseriamo una nuova prenotazione e aggiorniamo i crediti dell'utente, sempre all'interno di una transazione così da poter fare un rollback in caso di errore.

BookingDaoImpl.java - update

```
public void update(@NotNull Booking oldBooking, @NotNull Booking newBooking,
    @NotNull User user, @NotNull User copy) throws DatabaseFailedException {
    [...]
        try(PreparedStatement s1 = conn.prepareStatement(
                "DELETE FROM Bookings WHERE booking_number = ?"
        )){
            [...]
            for(Seat seat : newBooking.getSeats()) {
                try (PreparedStatement s2 = conn.prepareStatement(
                        "INSERT OR ROLLBACK INTO Bookings(showtime_id, seat_id,
                            user_id, booking_number) VALUES (?, ?, ?, ?)"
                )){
                    [...]
                    try(PreparedStatement s3 = conn.prepareStatement(
                            "UPDATE Users SET balance = ? WHERE user_id = ?"
                    )){
                        [...]
                    }
                }
            }
            conn.commit();
        } catch (SQLiteException | NullPointerException e){
            conn.rollback();
            [...]
        }
        [...]
}
```

- **SeatDao.java**: Si occupa di ritornare la lista dei posti di una sala dato uno spettacolo, impostando correttamente se il posto sia occupato o meno. Per fare ciò si usa un `LEFT JOIN` tra la tabella delle prenotazioni e la tabella dei posti combinata con quella delle proiezioni. Se il valore in `booking_number` è nullo allora il posto è libero.

SeatDaoImpl.java - get

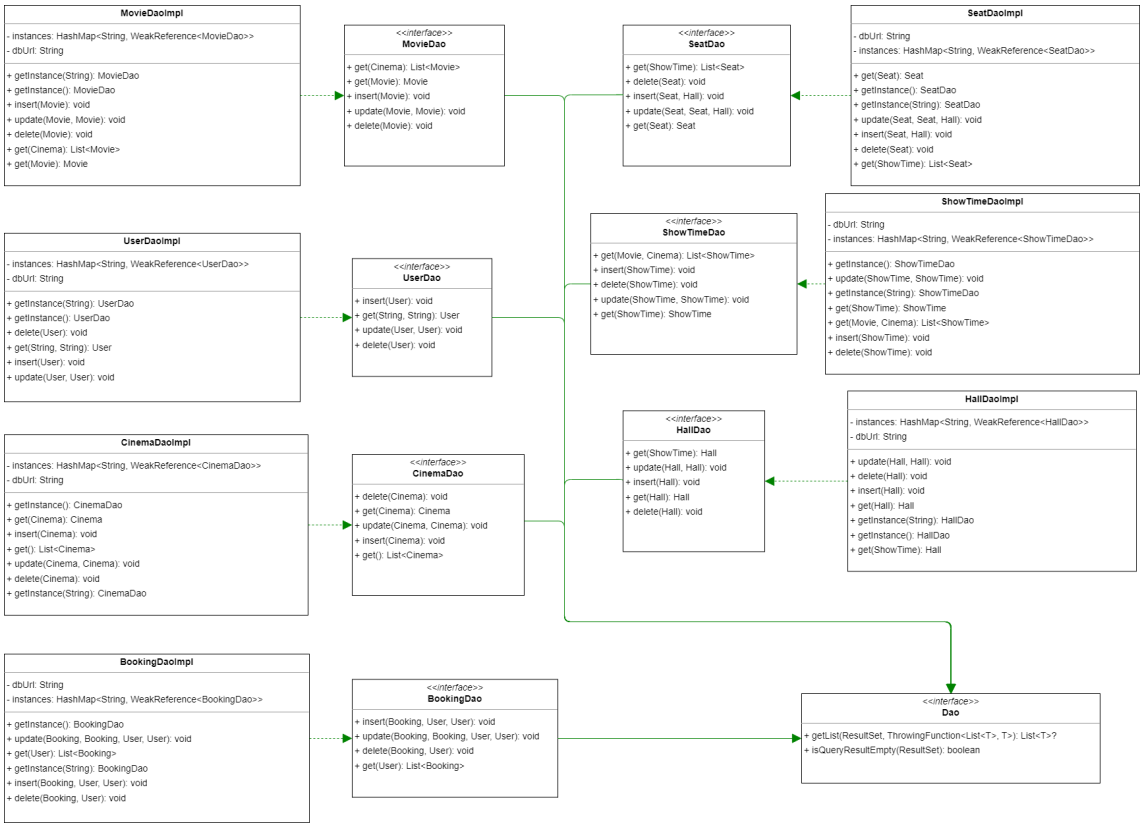
```
public List<Seat> get(@NotNull ShowTime showTime) {
    [...]
        try(PreparedStatement s = conn.prepareStatement(
                "SELECT DISTINCT Seats.seat_id, row, number, booking_number FROM
                    (ShowTimes JOIN Seats ON Seats.hall_id = ShowTimes.hall_id)
                    LEFT JOIN Bookings ON (ShowTimes.showtime_id =
                    Bookings.showtime_id AND Seats.seat_id = Bookings.seat_id)
                    WHERE ShowTimes.showtime_id = ?"
        )) {
            [...]
        }
    [...]
}
```

- **CinemaDao.java**: Questa classe fornisce i metodi per accedere alle informazioni sui cinema, tra cui la creazione, l'aggiornamento e la cancellazione di cinema dal database. Viene utilizzata dai repository per interagire con il livello di persistenza e garantire che le informazioni sui cinema siano sempre aggiornate.

- **HallDao.java**: Gestisce le operazioni di accesso ai dati relativi alle sale cinematografiche. Questa classe consente di creare, leggere, aggiornare e cancellare le informazioni relative alle sale, come la disposizione dei posti e le informazioni sulla capienza.

- **BookingDao.java**: Gestisce tutte le operazioni di persistenza legate alle prenotazioni. Offre metodi per salvare una nuova prenotazione, aggiornare una prenotazione esistente, cancellare prenotazioni e recuperare la cronologia delle prenotazioni di un utente.

Di seguito il diagramma delle classi.



3.3.3 Repositories Layer

Il pacchetto `repositories` funge da interfaccia tra i DAO e i service. Ogni `repository` implementa il pattern **Singleton**, che garantisce che ad ogni classe DAO sia associata una sola classe `repository` corrispondente. Inoltre tra queste classi viene implementato anche il pattern **Observer** che garantisce la consistenza dei dati tra il livello database e il livello applicazione quando avviene l'eliminazione di una entità, parte che verrà approfondita in 3.4.

Ogni `repository` ha inoltre una HashMap in cui vengono salvate tutte le referenze agli oggetti istanziati, così da non avere oggetti diversi con stesso ID, anche questa parte verrà approfondita in 3.5.

Le operazioni di modifica, quali l'inserimento e l'aggiornamento, avvengono con l'uso di funzioni lambda, che prendono in ingresso la copia dell'entità da modificare, eseguono le operazioni definite e inviano i dati al Data Access Layer. Se le operazioni

nel DAL non falliscono, le modifiche vengono applicate all'entità originale. Di seguito si mostra il metodo **update** della classe `UserRepositoryImpl`, dove si eseguono le operazioni appena descritte.

UserRepositoryImpl.java - update

```
public void update(@NotNull User user, ThrowingConsumer<User> edits) {
    [...]
    User copy = new User(user);
    try {
        edits.accept(copy);
    } catch (Exception e){
        if(e instanceof NotEnoughFundsException)
            throw (NotEnoughFundsException) e;
        else throw new RuntimeException(e);
    }
    userDao.update(user, copy);
    user.copy(copy);
}
```

- **BookingRepository.java**: Questa classe gestisce l'accesso ai dati relativi alle prenotazioni tramite il DAO `BookingDao`. Fornisce metodi per recuperare le prenotazioni, creare nuove prenotazioni e aggiornare lo stato delle prenotazioni esistenti.

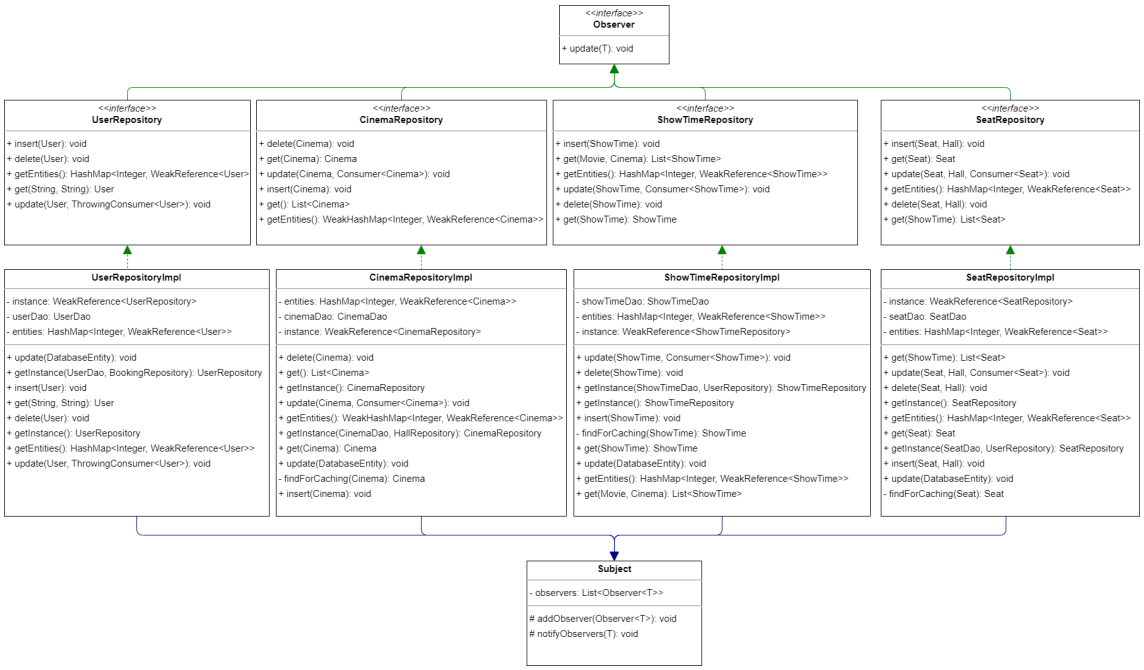
- **CinemaRepository.java**: Interagisce con `CinemaDao` per gestire le operazioni relative ai cinema, come la creazione di nuovi cinema, l'aggiornamento delle sale cinematografiche e la gestione delle informazioni legate ai cinema.

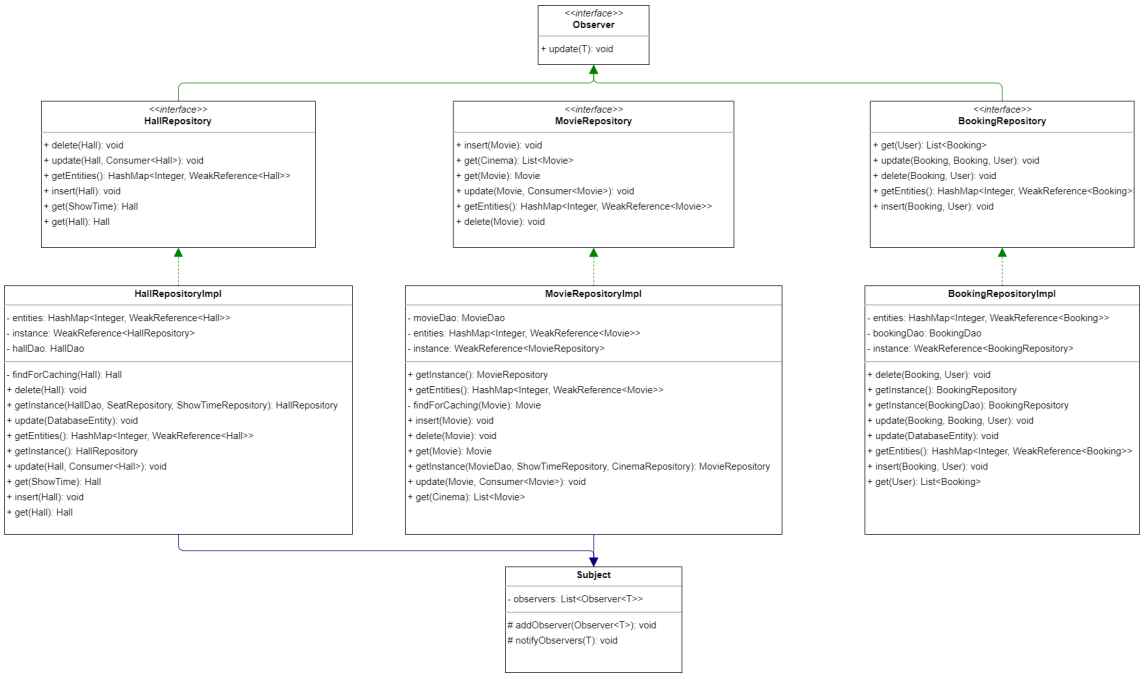
  La classe `CinemaRepository` implementa il pattern **Observer**, estendendo la classe `Subject`. Questo consente di notificare automaticamente gli altri oggetti correlati, come le sale (`Hall`) e i posti (`Seat`), quando un'entità `Cinema` viene modificata o eliminata. L'utilizzo di questo pattern garantisce che le entità correlate vengano aggiornate o rimosse in modo coerente, evitando che oggetti obsoleti restino associati ad altre entità.

- **HallRepository.java**: Si interfaccia con `HallDao` per gestire la persistenza delle sale cinematografiche. Fornisce metodi per gestire la disposizione dei posti, verificare la disponibilità delle sale e recuperare informazioni dettagliate sulle sale.

- **SeatRepository.java**: Utilizza `SeatDao` per gestire la disponibilità e lo stato dei posti in una sala. Questo repository fornisce metodi per aggiornare lo stato di un posto (prenotato o disponibile) e per recuperare informazioni sui posti di una determinata sala.

Di seguito il diagramma delle classi.


<<interface>>
Observer
+ update(T): void
<<interface>>
UserRepository
+ insert(User): void
+ delete(User): void
+ getEntities(): HashMap<Integer, WeakReference<User>
+ get(String, String): User
+ update(User, ThrowingConsumer<User>): void
<<interface>>
CinemaRepository
+ delete(Cinema): void
+ get(Cinema): Cinema
+ update(Cinema, Consumer<Cinema>): void
+ insert(Cinema): void
+ get(): List<Cinema>
+ getEntities(): WeakHashMap<Integer, WeakReference<Cinema>>
<<interface>>
ShowTimeRepository
+ insert(ShowTime): void
+ get(Movie, Cinema): List<ShowTime>
+ getEntities(): HashMap<Integer, WeakReference<ShowTime>>
+ update(ShowTime, Consumer<ShowTime>): void
+ delete(ShowTime): void
+ get(ShowTime): ShowTime
<<interface>>
SeatRepository
+ insert(Seat, Hall): void
+ get(Seat): Seat
+ update(Seat, Hall, Consumer<Seat>): void
+ getEntities(): HashMap<Integer, WeakReference<Seat>>
+ delete(Seat, Hall): void
+ get(ShowTime): List<Seat>
UserRepositoryImpl
- instance: WeakReference<UserRepository>
- userDao: UserDao
- entities: HashMap<Integer, WeakReference<User>>
+ update(DatabaseEntity): void
+ getInstance(UserDao, BookingRepository): UserRepository
+ insert(User): void
+ get(String, String): User
+ delete(User): void
+ getInstance(): UserRepository
+ getEntities(): HashMap<Integer, WeakReference<User>>
+ update(User, ThrowingConsumer<User>): void
CinemaRepositoryImpl
- entities: HashMap<Integer, WeakReference<Cinema>>
- cinemaDao: CinemaDao
- instance: WeakReference<CinemaRepository>
+ delete(Cinema): void
+ get(): List<Cinema>
+ getInstance(): CinemaRepository
+ update(Cinema, Consumer<Cinema>): void
+ getEntities(): WeakHashMap<Integer, WeakReference<Cinema>>
+ getInstance(CinemaDao, HallRepository): CinemaRepository
+ get(Cinema): Cinema
+ update(DatabaseEntity): void
- findForCaching(Cinema): Cinema
+ insert(Cinema): void
ShowTimeRepositoryImpl
- showTimeDao: ShowTimeDao
- entities: HashMap<Integer, WeakReference<ShowTime>>
- instance: WeakReference<ShowTimeRepository>
+ update(ShowTime, Consumer<ShowTime>): void
+ delete(ShowTime): void
+ getInstance(ShowTimeDao, UserRepository): ShowTimeRepository
+ getInstance(): ShowTimeRepository
+ insert(ShowTime): void
- findForCaching(ShowTime): ShowTime
+ get(ShowTime): ShowTime
+ update(DatabaseEntity): void
+ getEntities(): HashMap<Integer, WeakReference<ShowTime>>
+ get(Movie, Cinema): List<ShowTime>
SeatRepositoryImpl
- instance: WeakReference<SeatRepository>
- seatDao: SeatDao
- entities: HashMap<Integer, WeakReference<Seat>>
+ get(ShowTime): List<Seat>
+ update(Seat, Hall, Consumer<Seat>): void
+ delete(Seat, Hall): void
+ getInstance(): SeatRepository
+ getEntities(): HashMap<Integer, WeakReference<Seat>>
+ get(Seat): Seat
+ getInstance(SeatDao, UserRepository): SeatRepository
+ insert(Seat, Hall): void
+ update(DatabaseEntity): void
- findForCaching(Seat): Seat
Subject
- observers: List<Observer<T>>
addObserver(Observer<T>): void
notifyObservers(T): void


<<interface>>
Observer
+ update(T): void
<<interface>>
HallRepository
+ delete(Hall): void
+ update(Hall, Consumer<Hall>): void
+ getEntities(): HashMap<Integer, WeakReference<Hall>>
+ insert(Hall): void
+ get(ShowTime): Hall
+ get(Hall): Hall
<<interface>>
MovieRepository
+ insert(Movie): void
+ get(Cinema): List<Movie>
+ get(Movie): Movie
+ update(Movie, Consumer<Movie>): void
+ getEntities(): HashMap<Integer, WeakReference<Movie>>
+ delete(Movie): void
<<interface>>
BookingRepository
+ get(User): List<Booking>
+ update(Booking, Booking, User): void
+ delete(Booking, User): void
+ getEntities(): HashMap<Integer, WeakReference<Booking>
+ insert(Booking, User): void
HallRepositoryImpl
- entities: HashMap<Integer, WeakReference<Hall>>
- instance: WeakReference<HallRepository>
- hallDao: HallDao
- findForCaching(Hall): Hall
+ delete(Hall): void
+ getInstance(HallDao, SeatRepository, ShowTimeRepository): HallRepository
+ update(DatabaseEntity): void
+ getEntities(): HashMap<Integer, WeakReference<Hall>>
+ getInstance(): HallRepository
+ update(Hall, Consumer<Hall>): void
+ get(ShowTime): Hall
+ insert(Hall): void
+ get(Hall): Hall
MovieRepositoryImpl
- movieDao: MovieDao
- entities: HashMap<Integer, WeakReference<Movie>>
- instance: WeakReference<MovieRepository>
+ getInstance(): MovieRepository
+ getEntities(): HashMap<Integer, WeakReference<Movie>>
- findForCaching(Movie): Movie
+ insert(Movie): void
+ delete(Movie): void
+ get(Movie): Movie
+ getInstance(MovieDao, ShowTimeRepository, CinemaRepository): MovieRepository
+ update(Movie, Consumer<Movie>): void
+ get(Cinema): List<Movie>
BookingRepositoryImpl
- entities: HashMap<Integer, WeakReference<Booking>>
- bookingDao: BookingDao
- instance: WeakReference<BookingRepository>
+ delete(Booking, User): void
+ getInstance(): BookingRepository
+ getInstance(BookingDao): BookingRepository
+ update(Booking, Booking, User): void
+ update(DatabaseEntity): void
+ getEntities(): HashMap<Integer, WeakReference<Booking>>
+ insert(Booking, User): void
+ get(User): List<Booking>
Subject
- observers: List<Observer<T>>
addObserver(Observer<T>): void
notifyObservers(T): void

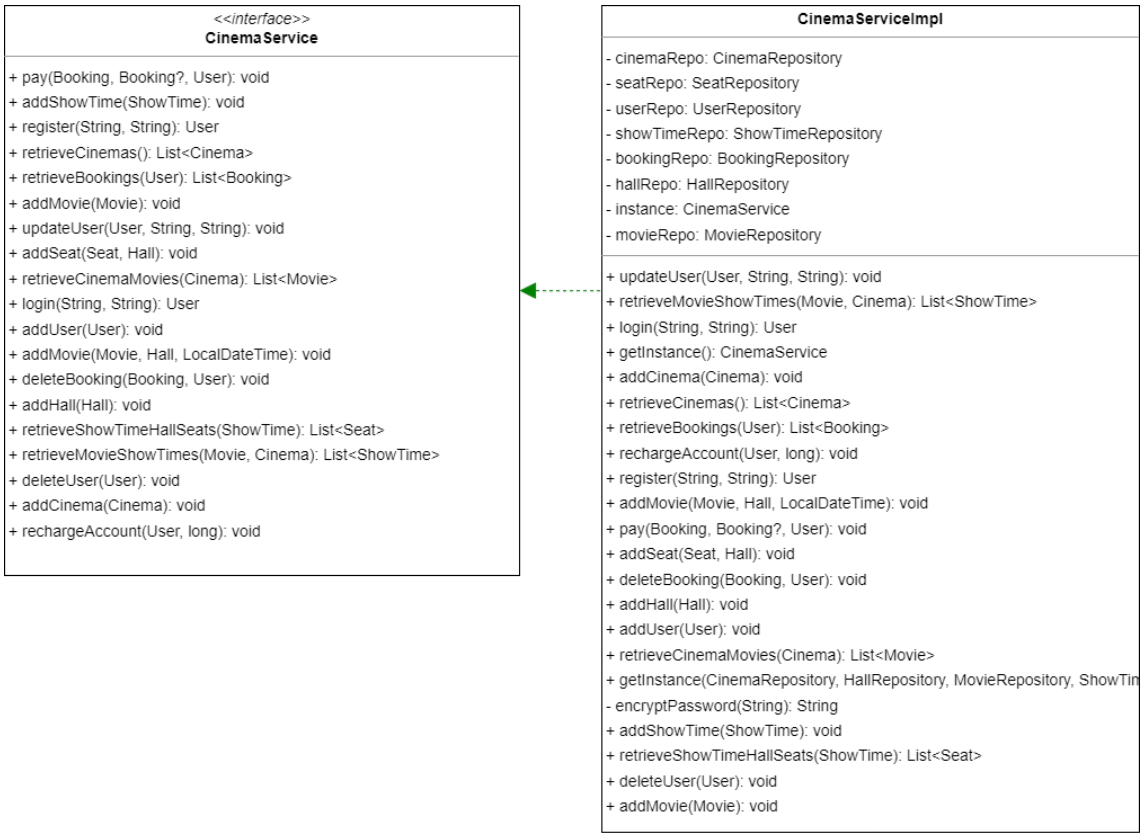
3.3.4 Service Layer

A livello `service` abbiamo la classe `CinemaService`, che centralizza le operazioni di gestione delle sale e dei cinema. Questa classe interagisce con i repository e i DAO per delegare le operazioni di accesso ai dati, garantendo una separazione tra la business logic e la persistenza dei dati. Sebbene la gestione delle prenotazioni sia delegata principalmente ai repository, la classe `CinemaService` si occupa anche di gestire operazioni come la conferma delle prenotazioni e i pagamenti, assicurando il corretto funzionamento di queste funzionalità cruciali.

- **CinemaService.java**: Questa classe si occupa principalmente di gestire le informazioni che arrivano dall'UI, che devono essere elaborate e inviate al layer `repository`. Fornisce metodi per aggiungere nuove sale, verificare la disponibilità delle sale, e gestire le operazioni relative alla loro manutenzione.
  Questa classe svolge un ruolo anche nella gestione delle prenotazioni. I metodi come `pay` e `addShowTime` per esempio organizzano le operazioni necessarie per completare una prenotazione, garantendo che le transazioni siano sicure e che i posti siano riservati correttamente. Si mostra il metodo `retrieveBookings`, che sebbene potesse essere implementato come una semplice chiamata al metodo get di BookingRepository, ciò non avrebbe garantito l'unicità degli oggetti con stesso ID che compongono una prenotazione, come `ShowTime`, `Seat` oppure `Hall`.

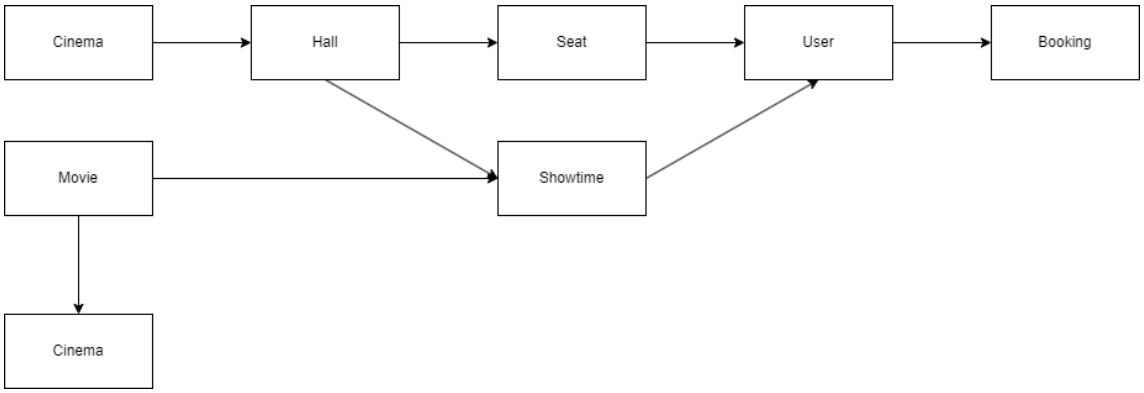
CinemaServiceImpl.java - retrieveBookings

```
    public List<Booking> retrieveBookings(User user) {
    List<Booking> bookings = bookingRepo.get(user);
    for (Booking b : bookings){
        ShowTime sht = showTimeRepo.get(b.getShowTime());
        b.setShowTime(sht);
        sht.setHall(hallRepo.get(sht.getHall()));
        sht.setMovie(movieRepo.get(sht.getMovie()));
        sht.getHall().setCinema(cinemaRepo.get(sht.getHall().getCinema()));
        ArrayList<Seat> seats = new ArrayList<>();
        for(Seat s : b.getSeats()){
            seats.add(seatRepo.get(s));
        }
        b.setSeats(seats);
    }
    return bookings;
}
```



3.4 Propagazione delle eliminazioni tramite Observer

Il sistema utilizza il *Pattern Observer* per implementare un meccanismo di propagazione delle eliminazioni tra le entità correlate. In questo contesto l'Observer viene impiegato per garantire che le relazioni tra oggetti siano mantenute coerenti quando un'entità viene rimossa dal database. Di seguito il diagramma di propagazione:



Ad esempio, supponiamo di rimuovere attraverso `ShowTimeRepository` uno `ShowTime` dal database. Di seguito dobbiamo anche rimuovere tutte le prenotazioni (`Booking`) per quel dato spettacolo. Questa operazione viene eseguita automaticamente nel database dalla eliminazione in cascata, ma una prenotazione già istanziata continuerebbe ad avere un ID valido e potrebbe anche essere presente nella lista di qualche utente.

Per risolvere questo problema, `ShowTimeRepository` notifica l'avvenuta can-

cellazione di tale spettacolo all' `UserRepository` che provvederà a cercare nella sua mappa gli utenti che hanno delle prenotazioni per quello spettacolo, rimuoverà queste prenotazioni dalla loro lista e rimborserà il costo, a sua volta poi notificherà `BookingRepository` che setterà l'ID delle prenotazioni coinvolte uguale a `ENTITY_WITHOUT_ID` e le rimuoverà dalla sua mappa. In questo modo si assicura che gli oggetti istanziati rappresentino sempre le informazioni presenti nel database.

UserRepositoryImpl.java - delete

```
public void delete(@NotNull ShowTime showTime) {
    [...]
    try {
        CinemaDatabase.withTransaction(() -> {
            showTimeDao.delete(showTime);
            notifyObservers(showTime); // Notifica UserRepository
        });
    } catch (Exception e) {
        [...]
    }
    entities.remove(showTime.getId());
    showTime.resetId();
}
```

Lo schema di propagazione è applicato in modo simile a tutte le altre entità che nello schema del database presentano delle *foreign keys*, poiché la loro esistenza dipende da altre entità. Fanno eccezione la classe `CinemaRepository` e `UserRepository` le quali vegnono notificate soltanto per rimuovere elementi dalle loro liste e non per resettare l'ID delle loro entità.

3.5 Weak Reference e Gestione della Memoria

All'interno del progetto, ogni classe `Repository` utilizza una `HashMap` per salvare le entità istanziate. Questa scelta progettuale è fondamentale per evitare che 2 oggetti con stesso ID abbiano riferimenti diversi, minando quindi la consistenza dell'informazione, in quanto modifiche su un oggetto non sarebbero riflesse sull'altro. Salvare ogni oggetto istanziato può causare problemi di memoria a lungo termine, se entità non più referenziate altrove non vengono collezionate dal garbage collector: è quindi necessario che elementi che non hanno altre referenze oltre a quella nella hash map vengano disposte. Per risolvere questo problema, gli oggetti vengono salvati nella mappa attraverso riferimenti deboli (`WeakReference`), così che vengano eliminati nel momento in cui le uniche referenze rimaste siano di tipo `WeakReference`.

4 Testing

Il testing svolge un ruolo fondamentale nel garantire la correttezza e l'affidabilità dell'applicativo. Nel progetto, i test sono stati implementati utilizzando il framework `JUnit`, suddivisi in test sui DAO, repository e service. Ogni tipologia di test è stata progettata per verificare specifiche funzionalità del sistema.

4.1 Test sui DAO

I DAO gestiscono l'accesso ai dati, consentendo operazioni CRUD (Create, Read, Update, Delete) sui vari componenti del sistema, come cinema, sale, posti e prenotazioni. I test sui DAO verificano che tali operazioni siano eseguite correttamente e che i dati siano coerenti nel database.

- **BookingDaoTest.java**: Verifica che le prenotazioni siano gestite correttamente. I test includono operazioni come la creazione di nuove prenotazioni, l'aggiornamento dello stato di prenotazioni esistenti e la cancellazione di prenotazioni dal database.
- **CinemaDaoTest.java**: Questo test verifica che i cinema siano creati, letti, aggiornati e cancellati correttamente nel database. Si testano anche query per recuperare informazioni dettagliate sui cinema.
- **HallDaoTest.java**: Controlla che le sale cinematografiche siano gestite correttamente, comprese operazioni come l'aggiunta di nuove sale, l'aggiornamento delle informazioni e la cancellazione delle sale.
- **SeatDaoTest.java**: Testa la gestione dei posti all'interno delle sale cinematografiche, assicurandosi che lo stato dei posti (prenotato, disponibile) sia correttamente aggiornato e memorizzato nel database.
- **ShowTimeDaoTest.java**: Verifica la gestione degli orari di proiezione, compresa la creazione e l'aggiornamento di orari specifici per le proiezioni dei film nelle sale.
- **UserDaoTest.java**: Testa la gestione degli utenti nel sistema, verificando operazioni come la registrazione, l'aggiornamento delle informazioni personali e la gestione delle prenotazioni associate a ciascun utente.

4.2 Test sui Repository

I repository sono responsabili di interfacciare i DAO con il livello di business logic. I test sui repository verificano che le operazioni complesse che coinvolgono più DAO siano eseguite correttamente.

- **BookingRepositoryTest.java**: Verifica la gestione delle prenotazioni, testando operazioni come la prenotazione di posti, la cancellazione delle prenotazioni e la verifica della disponibilità dei posti tramite il DAO.

- **CinemaRepositoryTest.java**: Controlla che la gestione delle informazioni sui cinema avvenga correttamente. Il test include operazioni complesse, come la gestione di più sale associate a un cinema e la verifica della coerenza dei dati.

- **HallRepositoryTest.java**: Verifica la gestione delle sale cinematografiche, controllando che i posti all'interno delle sale siano correttamente associati e gestiti in base alle operazioni del DAO.

- **MovieRepositoryTest.java**: Controlla la gestione dei film associati a un cinema, verificando che le operazioni relative alla programmazione dei film e alla gestione degli orari siano eseguite correttamente.

- **SeatRepositoryTest.java**: Testa la gestione dei posti, assicurandosi che lo stato di ogni posto (prenotato, disponibile) venga aggiornato correttamente nel database.

- **ShowTimeRepositoryTest.java**: Verifica la gestione degli orari di proiezione, assicurandosi che i film vengano programmati correttamente nelle sale e che gli utenti possano prenotare posti per gli spettacoli programmati.

- **UserRepositoryTest.java**: Verifica la gestione degli utenti, controllando che gli utenti possano registrarsi, aggiornare le proprie informazioni e gestire le prenotazioni tramite il repository.

4.3 Test sul Service

Il `CinemaServiceTest.java` verifica la business logic dell'applicativo, assicurandosi che le operazioni di alto livello siano eseguite correttamente e che i repository e i DAO siano utilizzati in modo appropriato per accedere ai dati.

- **CinemaServiceTest.java**: Testa la logica di gestione dei cinema e delle sale, verificando operazioni complesse come la creazione di un nuovo cinema con più sale, la gestione della disponibilità dei posti e la programmazione dei film. Il test verifica che la business logic segua correttamente le regole definite e che le interazioni con i repository siano corrette.

4.4 Test del Database

I test sul database sono stati progettati per verificare l'integrità delle operazioni di accesso ai dati, in particolare per quanto riguarda la connessione e le operazioni CRUD (Create, Read, Update, Delete) eseguite sulle tabelle del database. Il file `CinemaDatabaseTest.java` implementa questi test utilizzando un database SQLite, con dati di esempio rappresentanti le entità principali del sistema.

- **Verifica delle Connessioni**: Il metodo `runQuery()` viene utilizzato per eseguire query SQL sul database e verificare che le connessioni al database siano stabilite correttamente.

- **Setup del Database**: Il metodo `setUp()` predispone il database di test, inserendo dati di esempio per le entità chiave come `Cinema`, `Hall`, `Booking`, `Seat`, `Movie`, e `User`. Questo assicura che le tabelle del database siano popolate prima dell'esecuzione dei test.

- **Operazioni CRUD**: Sono stati eseguiti test per verificare la correttezza delle operazioni di inserimento, lettura e cancellazione nel database. Ad esempio, vengono inseriti cinema, sale e film nel database e successivamente vengono recuperati per garantire che le operazioni di lettura siano eseguite correttamente. I test includono anche la cancellazione delle tabelle per verificare che il database sia in grado di gestire correttamente la rimozione dei dati.

- **Dati di Esempio**: Per ogni entità del sistema (`Cinema`, `Hall`, `Seat`, `Movie`, `Booking`, `User`, ecc.), sono stati generati dati di esempio. Questi dati vengono utilizzati per simulare scenari reali e verificare che le entità siano gestite correttamente all'interno del database.

I test garantiscono che il database sia configurato e gestito correttamente durante le operazioni quotidiane del sistema, assicurando che la persistenza dei dati sia coerente e affidabile.

4.5 Risultati dei Test

Tutti i test sono stati eseguiti con successo utilizzando `JUnit`. I risultati indicano che il sistema gestisce correttamente le operazioni sui dati, rispettando la coerenza e l'integrità del database.