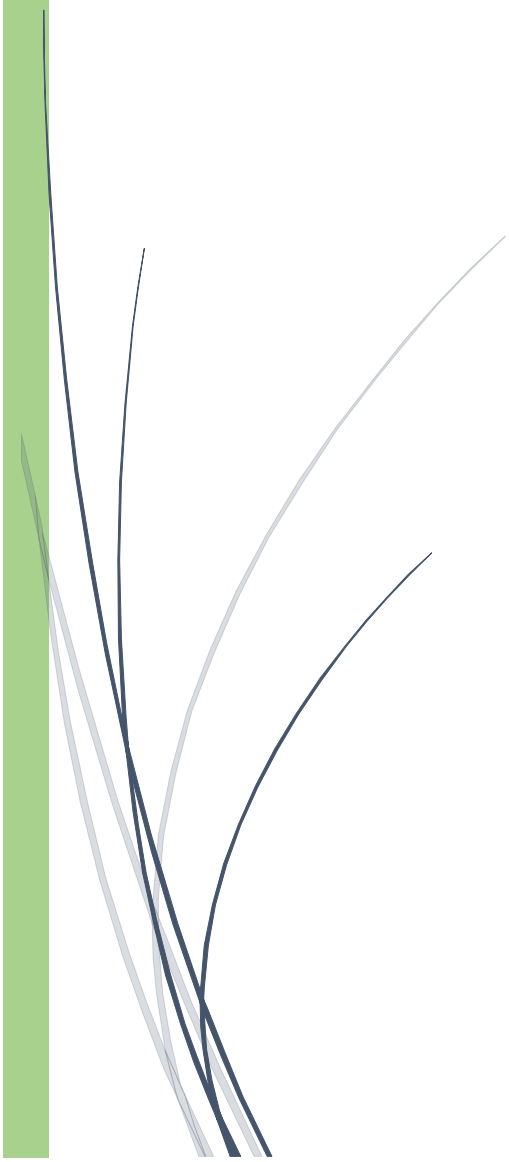
2021/2022

ODD

Object Design Document



System design

GERARDO LEONE, ANTONIO SANTOSUOSSO, MARIO LEZZI

Sommario

Revision History

Data	Versione	Descrizione	Autori
20/12/2021	0.1	Prima Stesura	LM
28/12/2021	0.2	Creazione dei Packages	LM, LG, SA
30/12/2021	0.3	Modifiche ai Packages e aggiunta delle class Interfaces	LG, SA
20/01/2021	0.4	Modifica dei Class Interface	LG, SA
14/02/2022	1.0	Revisione finalle	SA,LM

Team Members

Nome	Ruolo del progetto	Acronimo	Informazioni di contatto
Lezzi Mario	Team Member	LM	m.lezzi@studenti.unisa.it
Leone Gerardo	Team Member	LG	g.leone35@studenti.unisa.it
Santosuosso Antonio	Team Member	SA	a.santosuosso3@studenti.unisa.it

1. Introduzione

Dopo la realizzazione dei documenti RAD e SDD abbiamo descritto, in linea di massima, quello che sarà il nostro sistema e quindi i nostri obiettivi, tralasciando gli aspetti dell'implementazione. Il seguente documento ha lo scopo di produrre un modello capace di integrare in modo coerente e preciso tutte le diverse funzionalità individuate nelle fasi precedenti. In particolare, questo documento si vanno a descrivere i trade-offs generali realizzati dagli sviluppatori, le linee guida sulla documentazione delle interfacce e le convenzioni di codifica, le Interfacce delle classi, le

operazioni, i tipi, gli argomenti e la signature dei sottosistemi definiti nel System Design.

1.1 Object design goals

Riusabilità: Il sistema deve basarsi sulla riusabilità, attraverso l'utilizzo di ereditarietà e design patterns.

Robustezza: Il sistema deve risultare robusto, reagendo correttamente a situazioni impreviste attraverso il controllo degli errori e la gestione delle eccezioni.

Incapsulamento: Il sistema garantisce la segretezza sui dettagli implementativi delle classi grazie all'utilizzo delle interfacce, rendendo possibile l'utilizzo di funzionalità offerte da diversi componenti.

1.2 Object Design Trade-off

Prestazioni vs costi	Il team cercherà di ottenere le migliori prestazioni utilizzando il budget a disposizione.
Tempi di risposta vs sicurezza	Aumentare la sicurezza attraverso sistemi, comporta un aumento dei tempi di risposta delle operazioni.

1.2 Definizione, acronimi e abbreviazioni

- **RAD:** Requirements Analysis Document))
- **SDD:** System Design Document
- **ODD:** Object Design Document
- **Package:** raggruppamento di classi, interfacce o file correlati;
- **Design pattern:** template di soluzioni a problemi ricorrenti impiegati per ottenere riuso e flessibilità;
- **Interfaccia:** insieme di signature delle operazioni offerte dalla classe;
- **View:** nel pattern MVC rappresenta ciò che viene visualizzato a schermo da un utente e che gli permette di interagire con le funzionalità offerte dalla piattaforma;

1.3 Linee guida per la documentazione

Le linee guida includono una lista di regole che gli sviluppatori dovrebbero rispettare durante la progettazione delle interfacce.

Link a documentazione ufficiale sulle convenzioni

- Lo standard nella definizione delle classi e delle interfacce java è quello definito da Google(https://google.github.io/styleguide/javaguide.html).
- HTML: HTML Style Guide and Coding Conventions (w3schools.com)

1.4 Riferimenti

- Slides del corso;
- B.Bruegge, A. H. Dutoit, Object Oriented Software Engineering - Using UML, Pattern and Java, Prentice Hail, 3rd edition, 2009;
- RAD del progetto UniCinema;

Javadoc di UniCinema

Di seguito il link al sito contente il javadoc di UniCinema:

https://bloodmask.github.io/UniCinema.github.io/site/JavaDoc/index.html

2. Packages

In questa sezione viene mostrata la suddivisione del sistema in package, in base a quanto definito nel documento di System Design. Tale suddivisione è motivata dalle scelte architetturali prese e ricalca la struttura di directory standard definita da Maven.

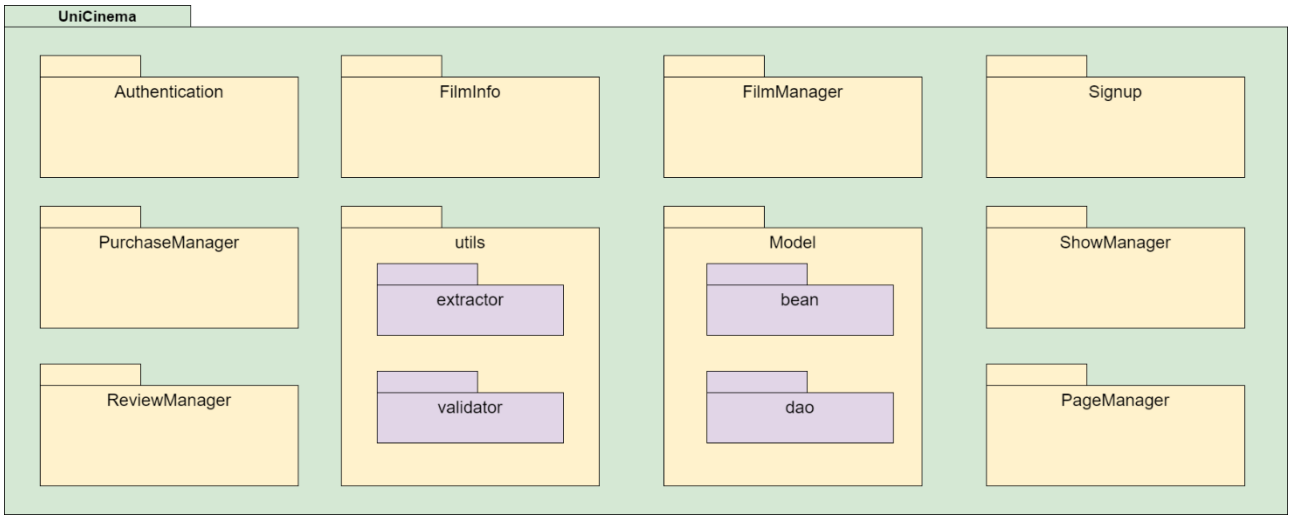
- **.idea**
- **src** contiene tutti i file sorgente
    - **main**
        - **Java,** contiene le classi Java relative alle componenti Control e Model

        - **webapp** contiene i file relativi alle componenti View
            - **static,** contiene i fogli di stile CSS e gli script JS
            - **views,** contiene i file JSP
            - **WEB-INF** contiene librerie
    - **test,** contiene tutto il necessario per il testing
        - **java,** contiene le classi Java per l'implementazione del testing.
- **target** contiene tutti i file prodotti dal sistema di build di Maven.

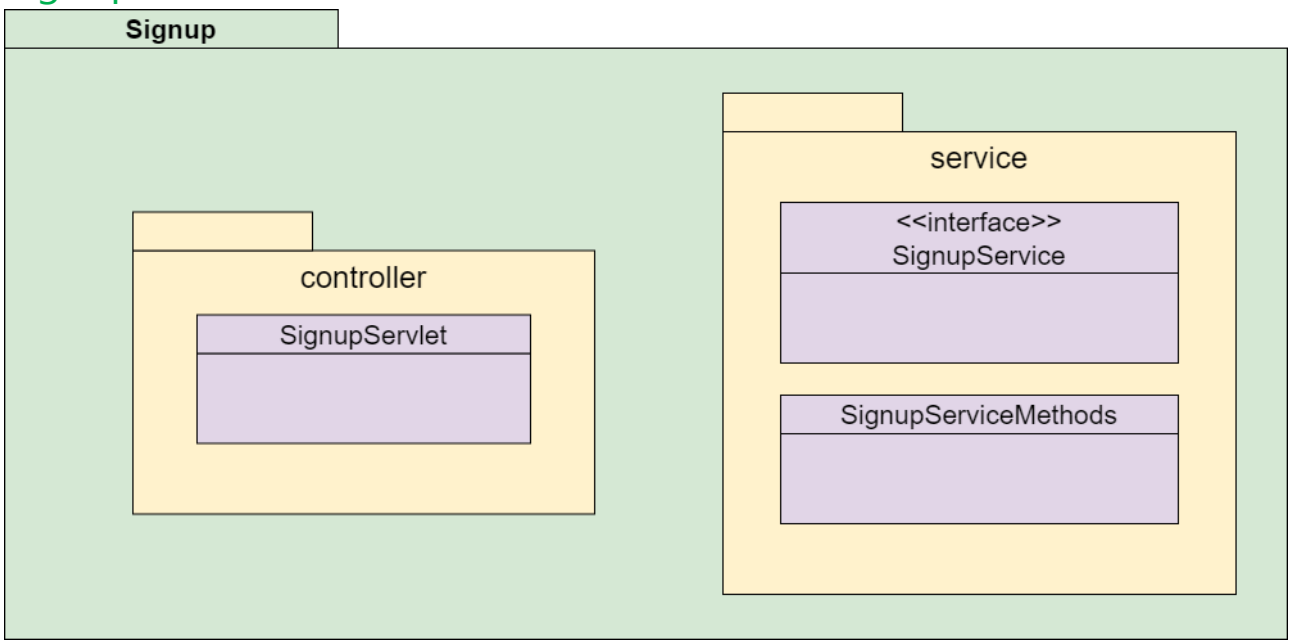
Package UniCinema

Nella presente sezione si mostra la struttura del package principale di UniCinema. La struttura generale è stata ottenuta da quattro scelte principali, cioè:

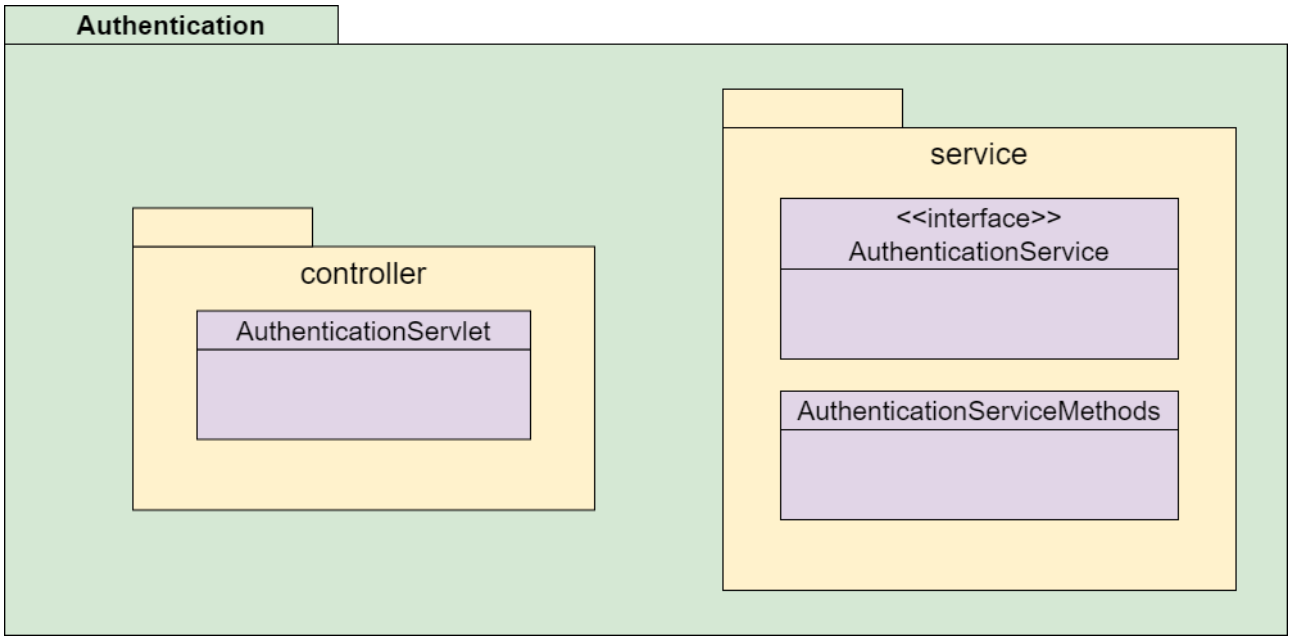
1. Creare un package per ogni sottosistema, ognuno contenente le classi service e il controller del sottosistema.
2. Creare un package a parte chiamato *model*, contenente i *bean* e i *DAO* per l'accesso al Database. È stata adottata questa disposizione per migliorare la leggibilità del codice, velocizzare il processo di ricerca dei vari moduli java, dal momento che il Database risulta essere molto complesso e pieno di operazioni.
3. Creare un package chiamato *utils* all'interno del quale sono state inserite tutte le classi di utilità generiche, dunque utilizzabili da qualsiasi sottosistema.
4. Creare un package per la servlet che gestisce la pagina principale, ovvero homepage.


UniCinema
Authentication
FilmInfo
FilmManager
Signup
PurchaseManager
utils
extractor
validator
Model
bean
dao
ShowManager
ReviewManager
PageManager

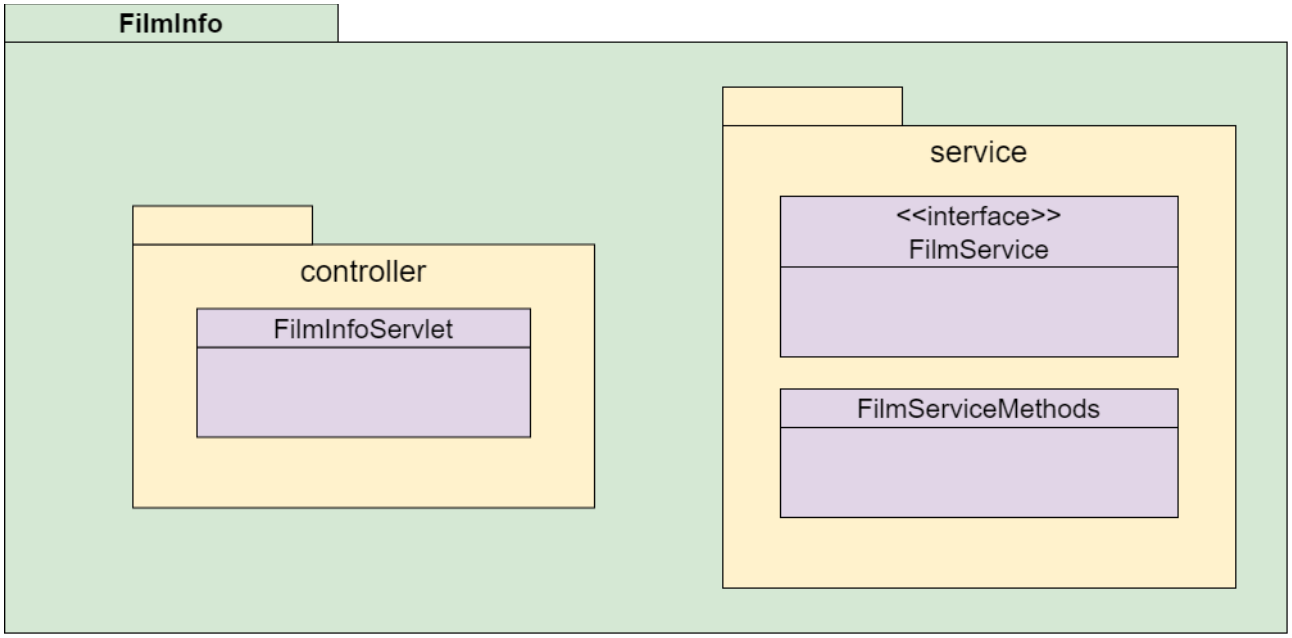
Signup


Signup
controller
SignupServlet
service
<<interface>>
SignupService
SignupServiceMethods

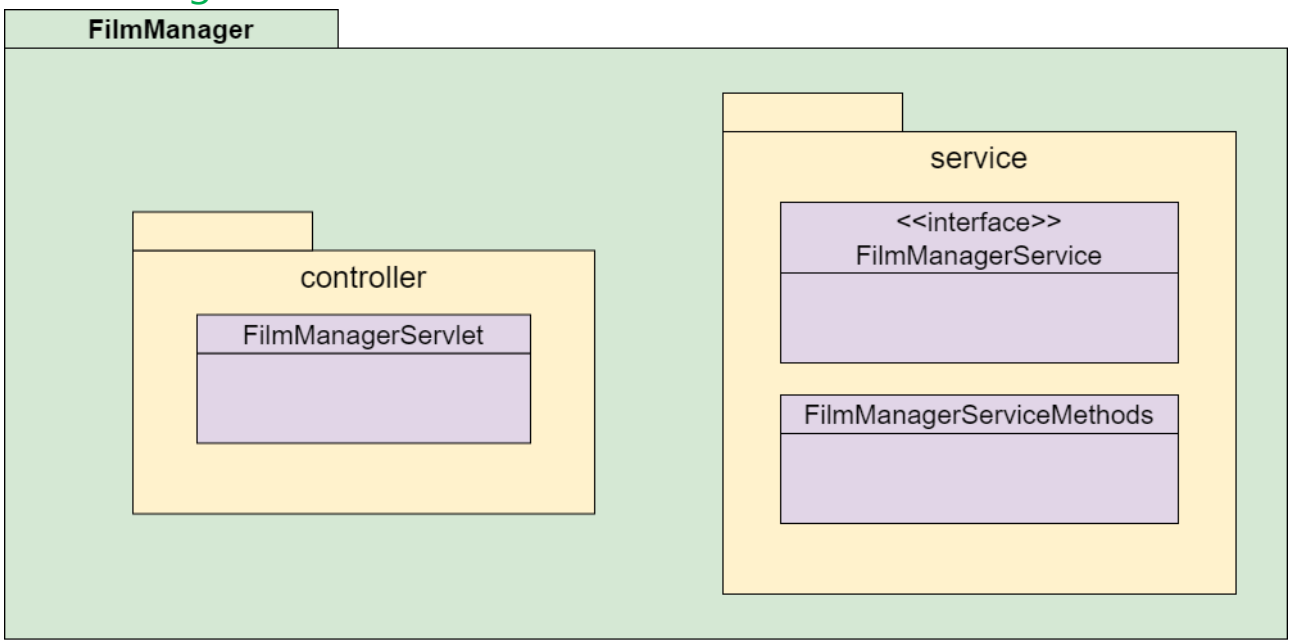
Authentication



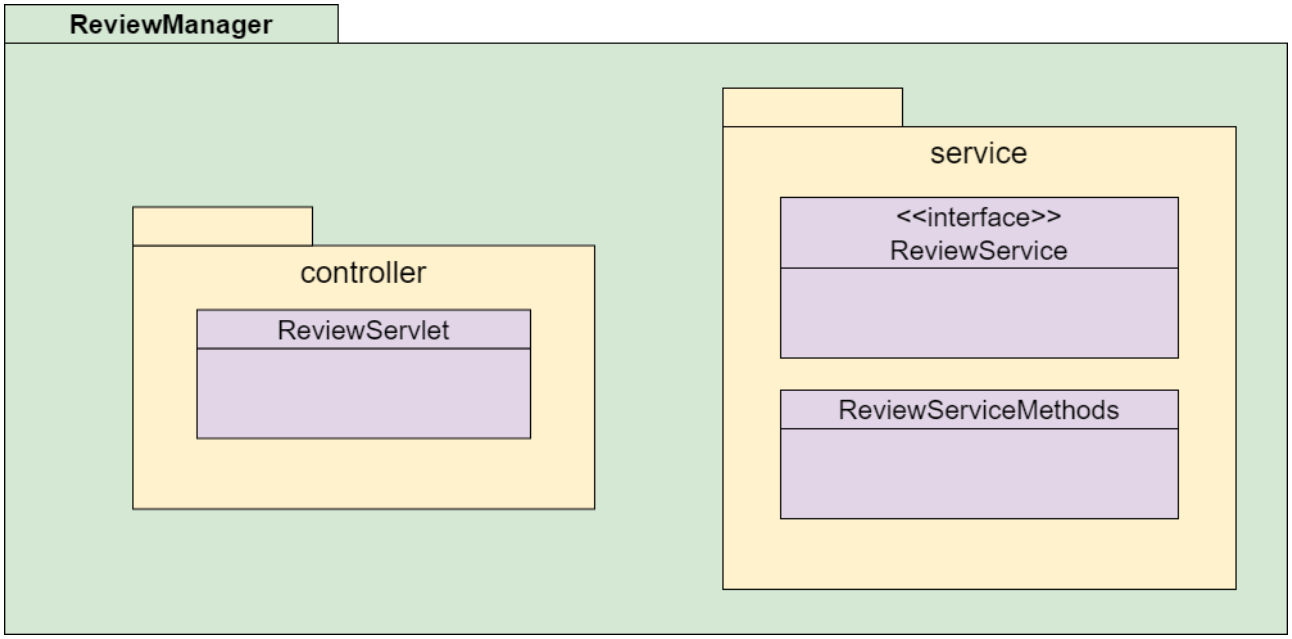
FilmInfo



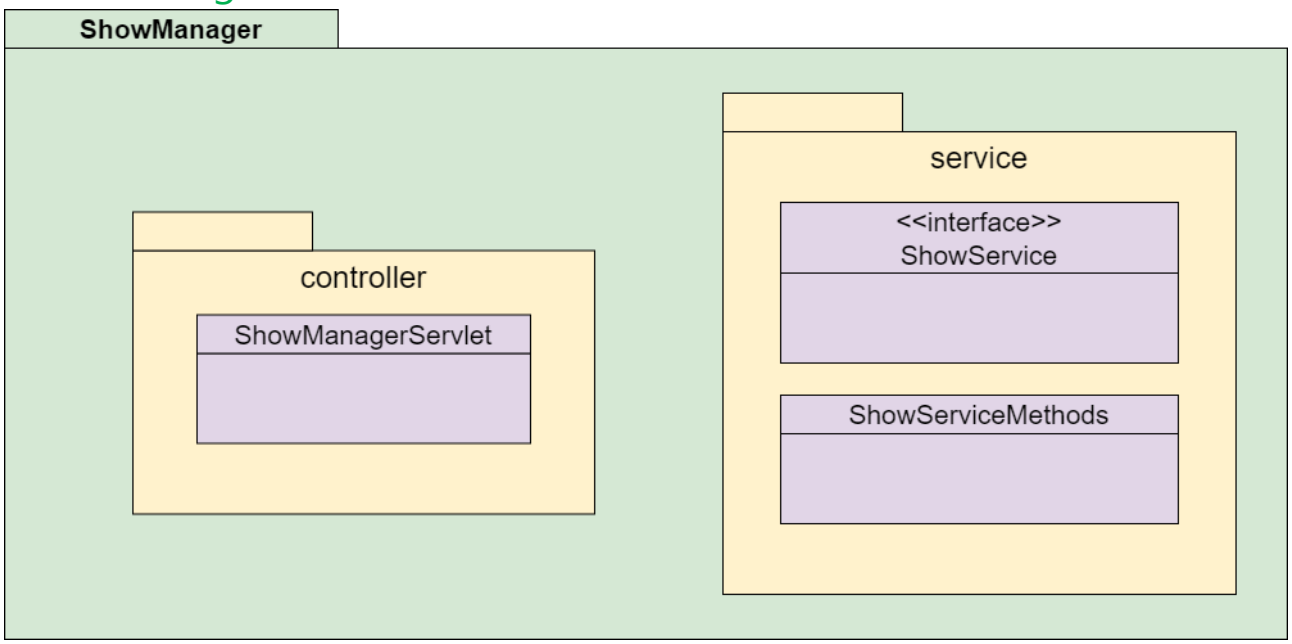
FilmManager



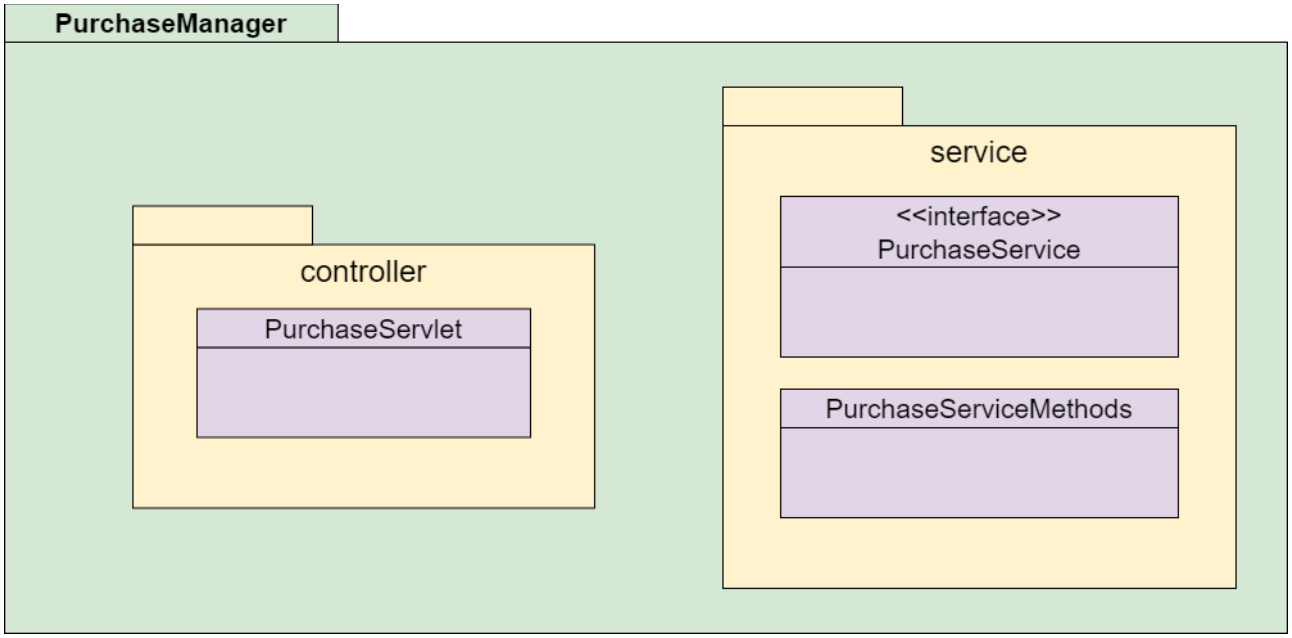
ReviewManager



ShowManager



PurchaseManager



3. Class Interface

In questa sezione verranno presentate le interfacce di ogni package.

3.1 Package Signup

Nome classe	**SignupService**
Descrizione	Questa classe permette di gestire le operazioni relative alla registrazione di un utenze.
Metodi	+signup(Account account): Boolean
Invariante di classe	

Nome Metodo	Signup(Account account)
Descrizione	Questo metodo permette di registrare un account.
Pre-condizione	/
Post-condizione	**context:** SignupService(Account account) **post:** accountDAO.fetch(account.email).equals(account)

3.2 Package Authentication

Nome classe	**AuthenticationService**
Descrizione	Questa classe permette di gestire le operazioni relative all'autenticazione, modifica e visualizzazione di un Account.
Metodi	+signin(Account account): Account +fetch(String email): Account +edit(Account account): boolean
Invariante di classe	/

Nome Metodo	+signin(Account account)
Descrizione	Questo metodo consente di autenticare un nuovo account.
Pre-condizione	/
Post-condizione	**Context**: AccountService:: signin(Account account) **Post:** accountDAO.find(account.email, account.pswrd, false) != null

Nome metodo	+fetch(String email)
Descrizione	Questo metodo consente di restituire un account.
Pre-condizione	/
Post-condizione	/
Nome metodo	+edit(Account account)
Descrizione	Questo metodo consente di modificare un account
Pre-condizione	**context:** AccountService:: edit(Account account) **pre:** accountDAO.fetch(account.email).equals(account)
Post-condizione	**context:** AccountService:: edit(Account account) **post:** not accountDAO.fetch(account.email).equals(@pre.accountDAO.fetch(account.email))

3.3 Package FilmInfo

Nome classe	**FilmService**
Descrizione	Questa classe permette la visualizzazione e la ricerca dei film.
Metodi	+fetch(int filmId): Film +search(String title): ArrayList<Film> +fetchLastReleases(int total): ArrayList<Film> +fetchComingSoon(int total): ArrayList<Film>
Invariante di classe	/

Nome Metodo	+fetch(int filmId)
Descrizione	Questo metodo restituisce un film a partire dal suo id.
Pre-condizione	/
Post-condizione	/
Nome metodo	+search(String title)
Descrizione	Questo metodo restituisce la lista dei film i cui titoli contengono in parte la stringa passata.
Pre-condizione	/

Post-condizione	/
Nome metodo	+fetchLastReleases(int total)
Descrizione	Questo metodo restituisce la lista degli ultimi film usciti. Il numero di questi è stabilito dal parametro *total*.
Pre-condizione	/
Post-condizione	/
Nome metodo	+fetchComingSoon(int total)
Descrizione	Questo metodo restituisce la lista dei film in uscita. Il numero di questi è stabilito dal parametro *total*.
Pre-condizione	/
Post-condizione	/

3.4 Package FilmManager

Nome classe	**FilmManagerService**
Descrizione	Questa classe consente la gestione dei Film.
Metodi	+removeFilm(int filmId): boolean +insert(Film film): boolean +update(Film film): boolean
Invariante di classe	/

Nome Metodo	+removeFilm(int filmId)
Descrizione	Questo metodo rimuove un film a partire dal suo id.
Pre-condizione	**context:** FilmManagerService:: removeFilm(int id) **pre:** filmDAO.fetch(film.id) != null
Post-condizione	**context:** FilmManagerService:: removeFilm(int id) **post:** filmDAO.fetch(filmId) == null
Nome metodo	+insert(Film film)
Descrizione	Questo metodo aggiunge un film.
Pre-condizione	**context:** FilmManagerService:: insert(Film film) **pre:** filmDAO.fetch(film.id) == null
Post-condizione	**context:** FilmManagerService:: insert (Film film) **post:** filmDAO.fetch(film.id).equals(film)
Nome metodo	+update(Film film)
Descrizione	Questo metodo aggiorna le informazioni di un film.
Pre-condizione	**context:** FilmManagerService:: update(Film film) **pre:** filmDAO.fetch(film.id).equals(film)

Post-condizione	**context:** FilmManagerService:: update(Film film) **post** not filmDAO.fetch(film.id).equals(@pre.filmDAO.fetch(film.id))

3.5 Package ReviewManager

Nome classe	**ReviewService**
Descrizione	Questa classe permette di gestire le operazioni relative alle recensioni.
Metodi	+countAll (Film film): int +fetchAll (Film film, Paginator paginator): ArrayList<Review> +fetchAll (int filmId): ArrayList<Review> +averageStars (ArrayList<Review> reviewList, int stars): int +percentageStars (ArrayList<Review> reviewList, int stars): double +insert (Review review): boolean +fetch (int accountId, int filmId): <Review> +delete(int accountId): boolean +countByAccountId(int id): int
Invariante di classe	

Nome Metodo	+countAll (Film film)
Descrizione	Questo metodo restituisce il numero di recensioni a partire dal film a cui si riferiscono.
Pre-condizione	/
Post-condizione	/
Nome metodo	+fetchAll(Film film, Paginator paginator)
Descrizione	Questo metodo restituisce la lista delle recensioni a partire dal film a cui si riferiscono.
Pre-condizione	/
Post-condizione	/
Nome Metodo	+fetchAll (int filmId)
Descrizione	Questo metodo restituisce tutte le recensioni a partire dall'identificato del film a cui si riferiscono.
Pre-condizione	/
Post-condizione	/
Nome Metodo	+averageStars (ArrayList<Review> reviewList, int stars)
Descrizione	Questo metodo restituisce la media delle valutazioni in base al numero di stelle passato.
Pre-condizione	/
Post-condizione	/

Nome metodo	+percentageStars (ArrayList<Review> reviewList, int stars)
Descrizione	Questo metodo restituisce la percentuale delle valutazioni in base al numero di stelle passato.
Pre-condizione	/
Post-condizione	/
Nome metodo	+insert (Review review): boolean
Descrizione	Questo metodo registra una recensione.
Pre-condizione	**context:** ReviewService:: insert(Review review) **pre:** not reviewDAO.fetchAll(filmId).includes(review)
Post-condizione	**context:** ReviewService:: insert(Review review) **post:** reviewDAO.fetchAll(filmId).includes(review) and reviewDAO.fetchAll(filmId).size() == @pre. reviewDAO.fetchAll(filmId).size()+1
Nome metodo	+fetch (int accountId, int filmId)
Descrizione	Questo metodo restituisce la recensione a partire dall'account che l'ha scritta e dal film a cui si riferisce.
Pre-condizione	/
Post-condizione	/
Nome metodo	+delete(int accountId)
Descrizione	Questo metodo permette di cancellare una recensione pubblicata da un account.
Pre-condizione	**context:** ReviewService:: delete(int accountId) **pre:** reviewDAO.fetchAll(filmId).includes(review)
Post-condizione	**context:** ReviewService:: delete(int accountId) **post:** not reviewDAO.fetchAll(filmId).includes(review) and reviewDAO.fetchAll(filmId).size() == @pre. reviewDAO.fetchAll(filmId).size()-1
Nome metodo	+countByAccountId(int id)
Descrizione	Questo metodo restituisce il numero di recensioni scritte da un account.
Pre-condizione	/
Post-condizione	/

3.6 Package ShowManager

Nome classe	**ShowService**

Descrizione	Questa classe permette di gestire le operazioni relative agli spettacoli e alle sale.
Metodi	+fetchAll (Film film): ArrayList<Show> +fetchAll(): ArrayList<Show> +fetchDaily (int roomId, LocalDate date): ArrayList<Show> +fetchDaily (int roomdId, LocalDate date, Show show): ArrayList<Show> +fetch (int id): Show +fetchRoom (int showId): Room +remove (int showId): boolean +insert (Show show): boolean +update (Show show): boolean
Invariante di classe	

Nome Metodo	+fetchAll (Film film)
Descrizione	Questo metodo restituisce tutti gli spettacoli che proiettano un dato film.
Pre-condizione	/
Post-condizione	/
Nome metodo	+fetchAll()
Descrizione	Questo metodo restituisce tutti gli spettacoli registrati nel database.
Pre-condizione	/
Post-condizione	/
Nome Metodo	+fetchDaily (int roomId, LocalDate date)
Descrizione	Questo metodo restituisce tutti gli spettacoli in una data specifica.
Pre-condizione	/
Post-condizione	/
Nome Metodo	+fetchDaily (int roomdId, LocalDate date, Show show)
Descrizione	Questo metodo restituisce tutti gli spettacoli in una data escludendo lo spettacolo specificato. Tale esclusione avviene perché quando bisogna modificare l'orario di uno spettacolo, non si deve tener conto degli orari occupati in precedenza dallo stesso.
Pre-condizione	/
Post-condizione	/
Nome metodo	+fetch (int id)
Descrizione	Questo metodo restituisce uno spettacolo a partire dal suo identificativo.
Pre-condizione	/

Post-condizione	/
Nome metodo	+fetchRoom (int showId)
Descrizione	Questo metodo restituisce una sala a partire dall'identificativo dello spettacolo.
Pre-condizione	/
Post-condizione	/
Nome metodo	+remove (int showId)
Descrizione	Questo metodo rimuove uno spettacolo a partire dal suo identificativo.
Pre-condizione	**context:** ShowService:: remove(int id) **pre:** showDAO.fetch(showId) != null
Post-condizione	**context:** ShowService:: remove(int id) **post:** showDAO.fetch(showId) == null and showDAO.fetchAll().size == @pre.showDAO.fetchAll().size()-1
Nome metodo	+insert (Show show)
Descrizione	Questo metodo registra uno spettacolo.
Pre-condizione	**context:** ShowService:: insert(Show show) **pre:** not showDAO.fetchAll().includes(show)
Post-condizione	**context:** ShowService:: insert(Show show) **post:** showDAO.fetchAll().includes(show) and showDAO.fetchAll().size == @pre.showDAO.fetchAll().size()+1
Nome metodo	+update (Show show)
Descrizione	Questo metodo aggiorna le informazioni dello spettacolo.
Pre-condizione	**context:** ShowService:: update(Show show) **pre:** showDAO.fetch(show.id).equals(account)
Post-condizione	**context:** ShowService:: update(Show show) **post: post:** not showDAO.fetch(show.id).equals(@pre.showDAO.fetch(show.id))

3.7 Package PurchaseManager

Nome classe	**PurchaseService**
Descrizione	Questa classe permette di gestire gli acquisti e i biglietti.
Metodi	+fetchTickets(int showId): ArrayList<Ticket> +findTicket(int showId, char row, int seat): Boolean +insert(ArrayList<Ticket> ticketList): boolean +insert(Purchase purchase): int +fetchAll(int accountId, Paginator paginator): ArrayList<Purchase>

	+countAll(int accountId): int
Invariante di classe	/

Nome Metodo	+fetchTickets(int showId)
Descrizione	Questo metodo restituisce la lista di tutti i biglietti acquistati per un dato spettacolo, identificato da showId.
Pre-condizione	/
Post-condizione	/
Nome metodo	+findTicket(int showId, char row, int seat)
Descrizione	Questo metodo verifica che un biglietto è stato aquistato per un dato spettacolo, in una data poltrona. Viene utilizzato per verificare che il posto selezionato non sia occupato da altre persone.
Pre-condizione	/
Post-condizione	/
Nome Metodo	+insert(ArrayList<Ticket> ticketList)
Descrizione	Questo metodo permette di registrare un insieme di biglietti acquistati.
Pre-condizione	**context:** PurchaseService:: insert(ArrayList<Ticket>ticketList) **pre:** ticketList.size() <= 4 and ticketList.size() >= 1
Post-condizione	**context:** PurchaseService:: insert(ArrayList<Ticket>ticketList) **post:** ticketDAO.fetchTickets(showId).size() == @pre.ticketDAO.fetchTickets(showId).size()+ ticketList.size()
Nome Metodo	+insert(Purchase purchase)
Descrizione	Questo metodo permette di registrare l'acquisto effettuato.
Pre-condizione	**context:** PurchaseService:: +insert(Purchase purchase) **pre:** not purchaseDAO.fetchAll(accountId, paginator).includes(purchase)
Post-condizione	**context:** PurchaseService:: +insert(Purchase purchase) **post:** purchaseDAO.fetchAll(accountId, paginator).includes(purchase) and purchaseDAO.fetchAll(accountId, paginator).size() == @pre. purchaseDAO.fetchAll(accountId, paginator).size()+1
Nome metodo	+fetchAll(int accountId, Paginator paginator)
Descrizione	Questo metodo restituisce una lista di acquisti effettuati da un Utente Registrato, identificato dall'accountId.
Pre-condizione	/
Post-condizione	/

Nome metodo	+countAll(int accountId)
Descrizione	Questo metodo restituisce il numero di acquisti effettuati da un utente.
Pre-condizione	/
Post-condizione	/

4. Class Diagram

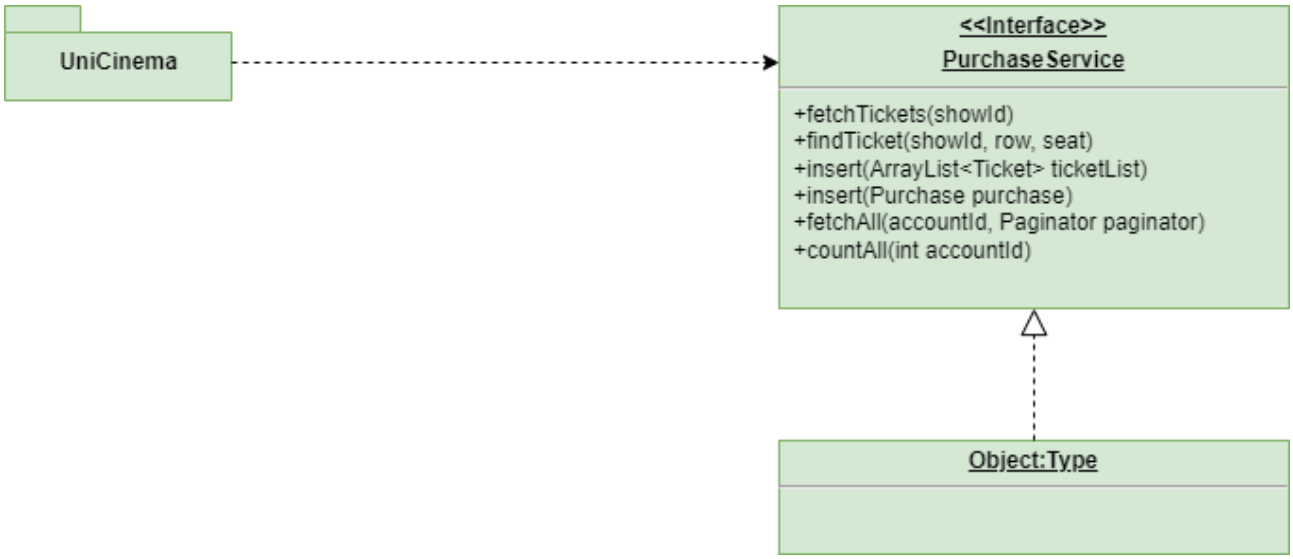
Per motivi di leggibilità, data la grandezza del class diagram, si riporta il link per visionarlo: https://ibb.co/g99wbsY

5. Design Pattern

Nella presente sezione verranno presentati i design pattern utilizzati per lo sviluppo del sistema.

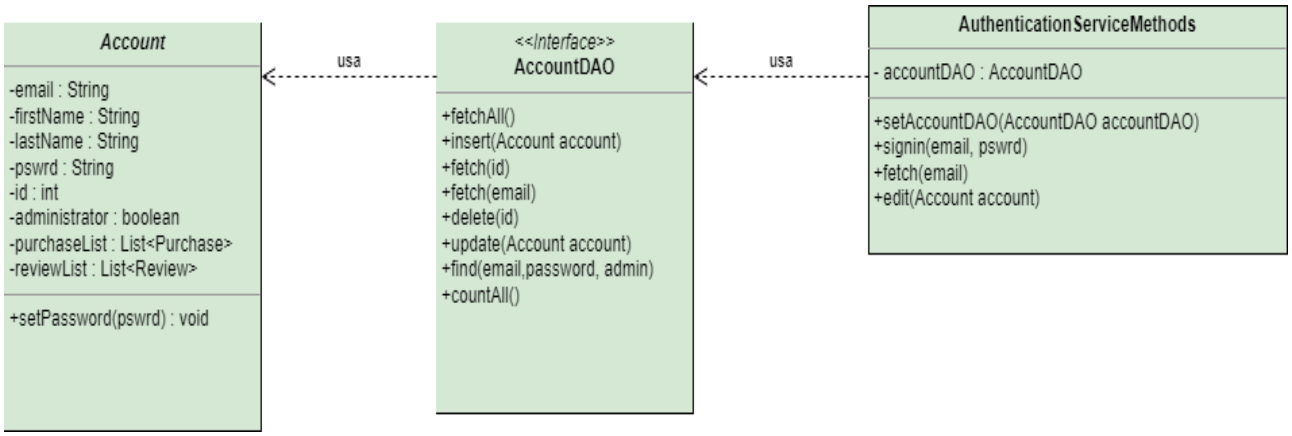
Facade

Il Facade è un design pattern che permette, implementando una interfaccia semplificata di accedere a sottosistemi più complessi. In questo modo si può nascondere al sistema la complessità delle librerie, dei framework o dei set di classi che si stanno usando. Si garantisce così un alto disaccoppiamento e si rende la piattaforma più manutenibile e più aggiornabile, poiché basterà cambiare l'implementazione dei metodi dell'interfaccia per implementare le modifiche. UniCinema, essendo un sistema molto complesso, sfrutta il design pattern Facade per implementare tutta la sua logica di business e rendere più facile l'interfacciarsi con essa. Nello specifico UniCinema utilizza il Facade per ogni suo sottosistema, implementandolo attraverso delle interfacce che sono usate per accedere ai metodi interni. Di seguito un esempio di Facade nel sistema UniCinema:



DAO

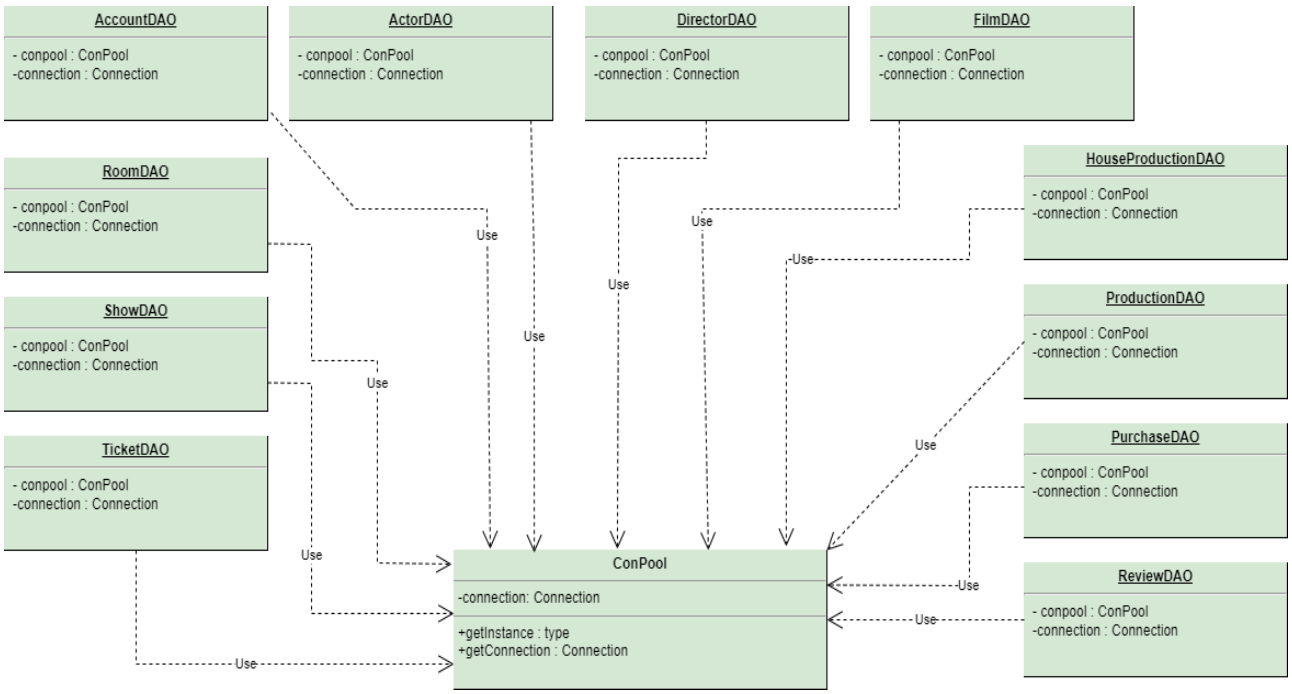
Un DAO (Data Access Object) è un pattern che offre un'interfaccia astratta per alcuni tipi di database. Mappando le chiamate dell'applicazione allo stato persistente, il DAO fornisce alcune operazioni specifiche sui dati senza esporre i dettagli del database. I DAO sono utilizzabili nella maggior parte dei linguaggi e la maggior parte dei software con bisogni di persistenza, principalmente viene associato con applicazioni JavaEE che utilizzano database relazionali. UniCinema è una web application che presenta un database molto vasto; quindi, ha bisogno di poter interagire con il database in modo rapido e sicuro con numerose query per quella che è la moltitudine di dati da gestire. Il DAO, inoltre, ci consente di avere una separazione tra logica di buisiness e persistenza.



Singleton

Il Singleton è un pattern creazionale, ha lo scopo di instanziare oggetti garantendo la creazione di una sola instanza di quella classe, il Singleton pattern risolve la problematica relativa alla creazione di una sola connessione per ogni evocazione di un metodo DAO al fine di migliorar la performance.

La classe ConPool si occupa di creare e mantenere una connessione al database, è una classe singleton, così può essere acceduta in maniera atomica,senza creare molteplici connessioni.



6. Glossario

- **Package**: Raggruppamento di classi ed interfacce.
- **DAO**: Data Access Object, implementazione dell'omonimo pattern architetturale che si occupa di fornire un accesso in modo astratto ai dati persistenti.
- **Controller**: Classe che si occupa di gestire le richieste effettuate dai client.
- **Service**: Classe che implementa la logica di business viene utilizzata dal controller o da un altro sottosistema.
- **Model**: Parte del design architetturale MVC che fornisce al sistema i metodi per accedere ai dati utili al sistema.
- **Facade**: Un oggetto che permette, attraverso un'interfaccia più semplice, l'accesso a sottosistemi che espongono interfacce complesse e molto diverse tra loro.
- **Singleton:** E' un design pattern creazionale che ha lo scopo di garantire che di una determinata classe venga struttura una sola istanza e di fornire un punto di accesso globale a tale istanza.