Unisa Cinema

UNIVERSITA' DEGLI STUDI DI SALERNO
FACOLTA' DI SCIENZE MM. FF. NN.
CORSO DI LAUREA TRIENNALE IN INFORMATICA
CORSO DI INGEGNERIA DEL SOFTWARE

Object Design Document

Versione 3.0 Anno Accademico 2017/2018

Top Manager:

Professori
Prof. De Lucia Andrea
Prof. Francese Rita

Partecipanti:

Nome	**Matricola**
Amato Federica	0512103606
Cosenza Giuseppe	0512103486
De Sio Maria Grazia	0512103594
Pizzo Domenico	0512103652

Revision History:

Data	**Versione**	**Descrizione**	**Autore**
12/12/2017	1.0	Prima stesura del documento	Team members
01/02/2018	2.0	Stesura del documeto	Team members
17/02/2018	3.0	Revisione del documento	Pizzo Domenico De Sio Maria Grazia Amato Federica

Sommario

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 Trade-offs

- **Comprensibilità vs Tempo:**
  Il codice deve essere al quanto più comprensibile per poter facilitare la fase di testing ed eventuali future modifiche del codice.
  A tale scopo, il codice sarà quindi accompagnato da commenti che ne semplifichino la comprensione.
  Questa caratteristica incrementerà il tempo di sviluppo, ma allo stesso tempo lo renderà più comprensibile.

  Interfaccia vs Usabilità:
  Il sistema verrà sviluppato con un interfaccia grafica realizzata in modo da poter essere molto semplice, chiara ed intuitiva.
  Nell'interfaccia saranno presenti form, menu e pulsanti, disposti in maniera da
  Rendere semplice l'utilizzo del sistema da parte dell'utente finale.

- **Sicurezza vs Efficienza:**
  La sicurezza, come descritto nei requisiti non funzionali, rappresenta uno degli aspetti importanti del sistema.
  A causa dei tempi di sviluppo molto limitati, ci limiteremo ad implementare un sistema di sicurezza basato sull'utilizzo di username e password degli utenti.

1.2 Linee Guida per la Documentazione delle Interfacce

Gli sviluppatori seguiranno alcune linee guida per la scrittura del codice:

Naming convention

- E' buona norma utilizzare nomi:
    1. Descrittivi
    2. Pronunciabili
    3. Di uso comune
    4. Di lunghezza medio-corta
    5. Non abbreviati
    6. Evitando la notazione ungherese
    7. Utilizzando solo caratteri consentiti (a-z, A-Z, 0-9)

Variabili:

- I nomi delle variabili devono cominciare con una lettera minuscola, e le parole seguenti con la lettera maiuscola. Quest'ultime devono essere dichiarate ad inizio blocco, solamente una per riga e devono essere tutte allineate e facilitarne la leggibilità. Esse possono essere annotate con dei commenti.
  Esempio: nomeFilm

- E' inoltre possibile, in alcuni casi, utilizzare il carattere underscore ("_") per la definizione del nome.

Metodi:

- I nomi dei metodi devono cominciare con una lettera minuscola, e le parole seguenti con la lettera maiuscola. Il nome del metodo tipicamente consiste in un verbo che identifica una azione, seguito dal nome di un oggetto.
  I nomi dei metodi per l'accesso e la modifica delle variabili dovranno essere del tipo getNomeVariabile() e SetNomeVariabile().

- I commenti dei metodi devono essere raggruppati in base alla loro funzionalità, la descrizione dei metodi deve apparire prima di ogni dichiarazione di metodo, e deve descriverne lo scopo. Deve includere anche informazioni sugli argomenti, sul valore di ritorno, e se applicabile, sulle eccezioni.

Classi e pagine:

- I nomi delle classi e delle pagine devono cominciare con una lettera maiuscola, e anche le parole seguenti all'interno del nome devono cominciare con una lettera maiuscola. I nomi di quest'ultime devono fornire informazioni sul loro scopo.

- La dichiarazione di classe deve essere caratterizzata da:

    1. Dichiarazione della classe pubblica
    2. Dichiarazioni di costanti
    3. Dichiarazioni di variabili di classe
    4. Dichiarazione di variabili d'istanza
    5. Costruttore
    6. Commento e dichiarazione dei metodi

1.3 Definizioni, acronimi e abbreviazioni

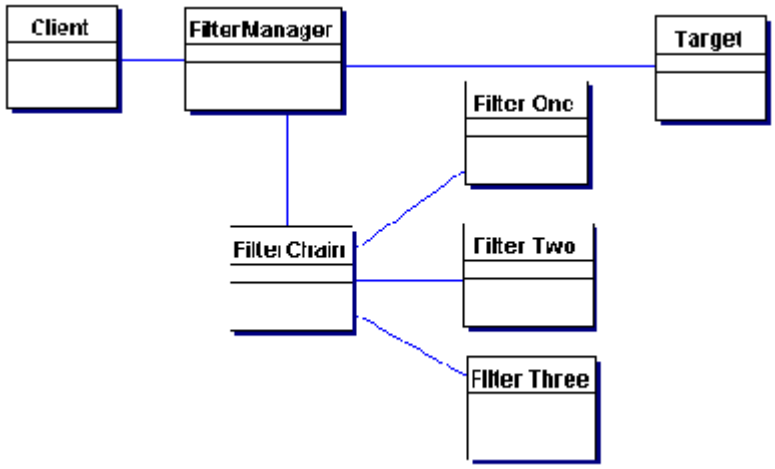
- **RAD :** Requirements Analysis Document
- **SDD :** System Design Document
- **ODD :** Object Design Document

1.4 Riferimenti

- **B.Bruegge, A. H. Dutoit, Object Oriented Software Engineering - Using UML, Pattern and Java, Prentice Hail, 3rd edition, 2009.**
- **Documento RAD del progetto Unisa Cinema.**
- **Docuemento Dati Persistenti del progetto Unisa Cinema.**

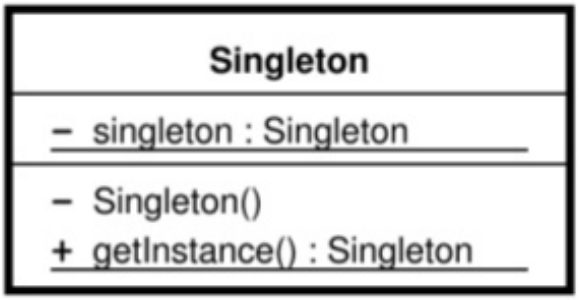
2. Design pattern

2.1 Protection Proxies



Unisa Cinema fa uso del Protection Proxies Pattern, quest'ultimo deriva dal Proxy Pattern. Nella sua forma più generale, un proxy è una classe che funziona come interfaccia per qualcos'altro. L'altro potrebbe essere qualunque cosa: una connessione di rete, un grosso oggetto in memoria, un file e altre risorse che sono costose o impossibili da duplicare. Nelle situazioni in cui molte copie di un oggetto complesso devono esistere, il proxy pattern può essere adottato per incorporare il Flyweight pattern per ridurre l'occupazione di memoria dell'oggetto. Tipicamente viene creata un'istanza di oggetto complesso, e molteplici oggetti proxy, ognuno dei quali contiene un riferimento al singolo oggetto complesso. Ogni operazione svolta sui proxy viene trasmessa all'oggetto originale. Una volta che tutte le istanze del proxy sono distrutte, l'oggetto in memoria può essere deallocato.
Le classi partecipanti nel proxy pattern sono:
Oggetto: interfaccia implementata da RealSubject e che rappresenta i suoi servizi. L'interfaccia deve essere implementata dal proxy, in modo che il proxy possa essere utilizzato in qualsiasi posizione in cui è possibile utilizzare RealSubject.
Proxy: Mantiene un riferimento che consente al proxy di accedere a RealSubject. Implementa la stessa interfaccia implementata da RealSubject in modo che il Proxy possa essere sostituito da RealSubject.
Controlla l'accesso a RealSubject e potrebbe essere responsabile della sua creazione e cancellazione.
Altre responsabilità dipendono dal tipo di proxy.
RealSubject: l'oggetto reale rappresentato dal proxy.
Ci sono varie situazioni in cui è applicabile il proxy pattern nel nostro caso come detto all'inizio utilizziamo il Protection Proxies dove un proxy controlla l'accesso alle risorse, dandone l'accesso ad alcuni utenti negando l'accesso ad altri. L'accesso alle risorse è gestito tramite i filtri applicati sulle classi:
Admin.java
Cliente.java
Generale.java
Operatore.java
Ospite.java.

2.2 Singleton Pattern



Il singleton è un design pattern creazionale che ha lo scopo di garantire che di una determinata classe venga creata una e una sola istanza, e di fornire un punto di accesso globale a tale istanza.
L'implementazione più semplice di questo pattern prevede che la classe singleton abbia un unico costruttore privato, in modo da impedire l'istanziazione diretta della classe. La classe fornisce inoltre un metodo "getter" statico che restituisce l'istanza della classe (sempre la stessa), creandola preventivamente o alla prima chiamata del metodo, e memorizzandone il riferimento in un attributo privato anch'esso statico. Il secondo approccio si può classificare come basato sul principio della lazy initialization (letteralmente "inizializzazione pigra") in quanto la creazione dell'istanza della classe viene rimandata nel tempo e messa in atto solo quando ciò diventa strettamente necessario (al primo tentativo di uso).
Utilizziamo il singleton pattern per collegare le classi al nostro DB tramite la classe: DriverManagerConnectionPool.java .

3. Packages

Il nostro sistema presenta una suddivisione basata su tre livelli (three-tier):

- Interface layer
- Application Logic layer
- Storage layer.

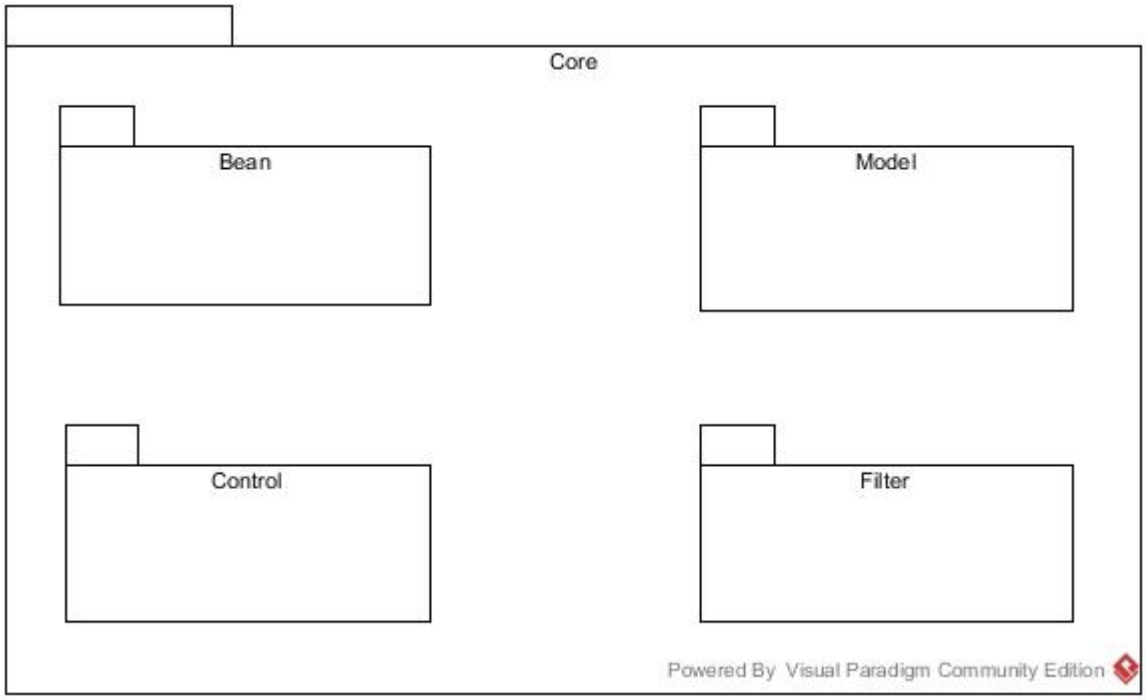
Il package Unisa Cinema contiene sottopackage che a loro volta inglobano classi atte alla gestione delle richieste utente. Le classi contenute nel package svolgono il ruolo di gestore logico del sistema.

Interface layer	Rappresenta l'interfaccia del sistema, ed offre la possibilità all'utente di interagire con quest'ultimo, offrendo sia la possibilità di inviare, in input, che di visualizzare, in output, dati.

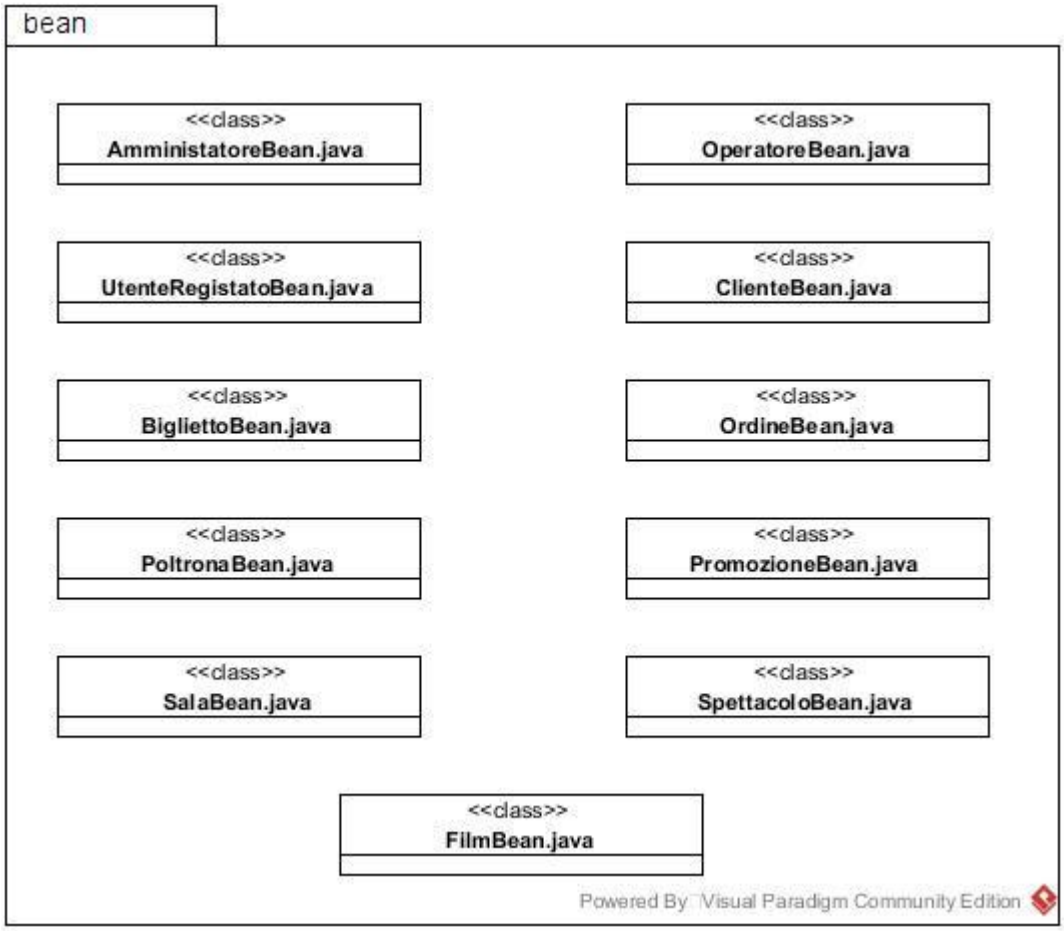
Application Logic layer	Ha il compito di elaborare i dati da inviare al client, e spesso grazie a delle richieste fatte al database, tramite lo Storage Layer, accede ai dati persistenti. Si occupa di varie gestioni quali: • Gestione Account • Gestione Film • Gestione Spettacoli • Gestione Acquisti • Gestione Promozioni

Storage layer	Ha il compito di memorizzare i dati sensibili del sistema, utilizzando un DBMS, inoltre riceve le varie richieste dall' Application Logic layer inoltrandole al DBMS e restituendo i dati richiesti.

3.1 Package core

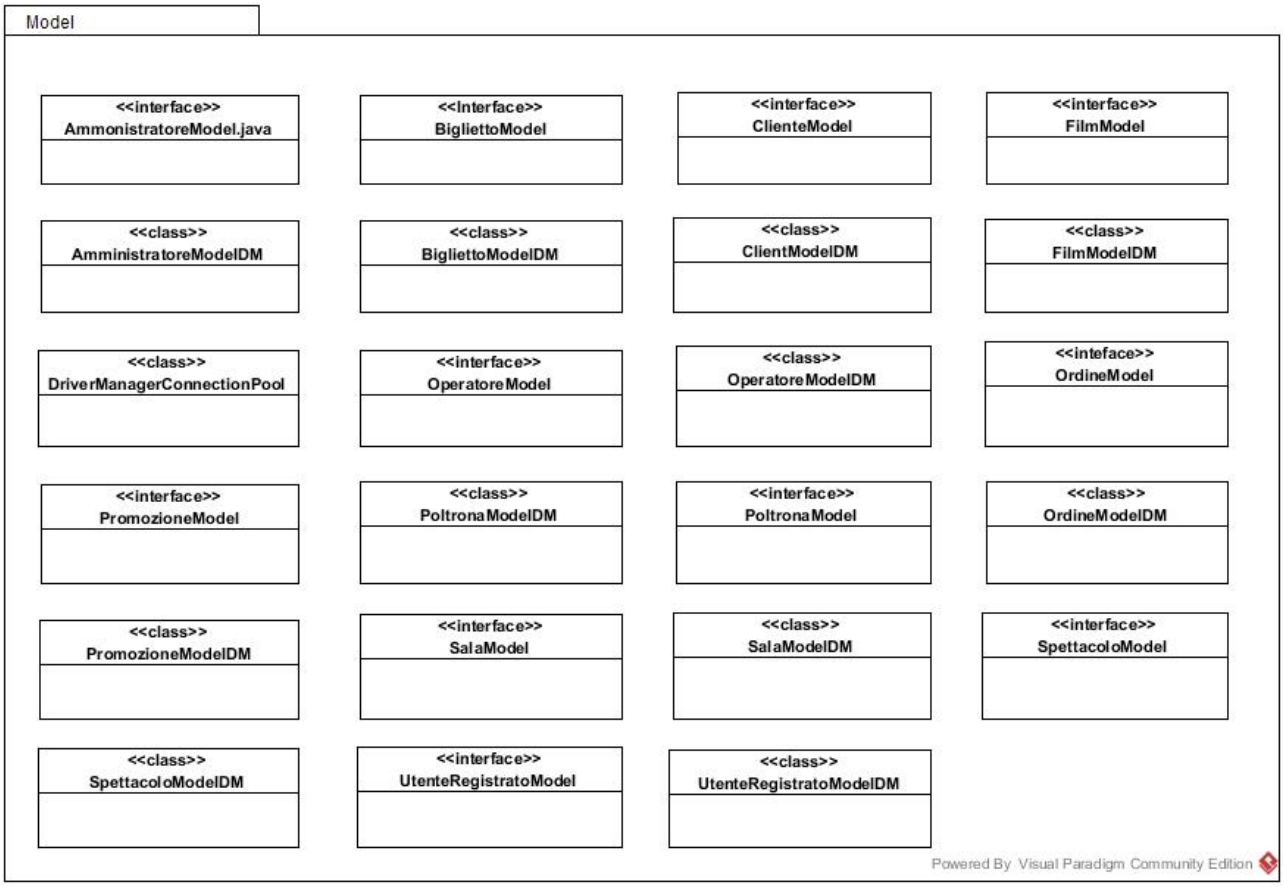


3.2 Package bean



Classe:	Descrizione:
AmministratoreBean.java	Questa classe rappresenta l'amministratore.
OperatoreBean.java	Questa classe rappresenta l'operatore.
UtenteRegistrato.java	Questa classe rappresenta l'utente registrato.
ClienteBean.java	Questa classe rappresenta il cliente.
BigliettoBean.java	Questa classe rappresenta l'amministratore.
OrdineBean.java	Questa classe rappresenta l'ordine dell'utente.
PoltronaBean.java	Questa classe rappresenta la poltrona.
PromozioneBean.java	Questa classe rappresenta la promozione.
SalaBean.java	Questa classe rappresenta la sala.
SpettacoloBean.java	Questa classe rappresenta lo spettacolo.
FilmBean.java	Questa classe rappresenta il film.

3.3 Package model

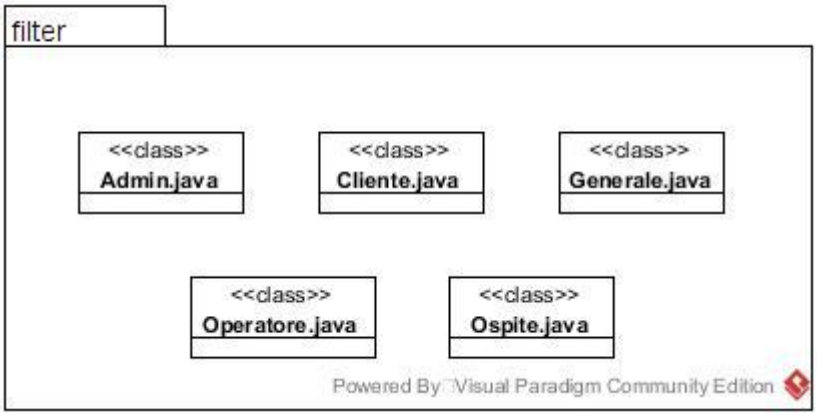


Classe:	Descrizione:
AmministratoreModel.java	Questa classe contiene i metodi che permettono di effettuare inserimento, ricerca e cancellazione di un amministratore.
BigliettoModel.java	Questa classe contiene i metodi che permettono di effettuare inserimento, ricerca e cancellazione di un biglietto.
ClienteModel.java	Questa classe contiene i metodi che permettono di effettuare

	inserimento, ricerca e cancellazione di un cliente.
FilmModel.java	Questa classe contiene i metodi che permettono di effettuare inserimento, ricerca e cancellazione di un film.
DriverManagerConnectionPool.java	Questa classe si occupa di gestire un pool di connessioni al database.
OperatoreModel.java	Questa classe contiene i metodi che permettono di effettuare inserimento, ricerca e cancellazione di un operatore.
OrdineModel.java	Questa classe contiene i metodi che permettono di effettuare inserimento, ricerca e cancellazione di un ordine.
PromozioneModel.java	Questa classe contiene i metodi che permettono di effettuare inserimento, ricerca e cancellazione di una promozione.
SalaModel.java	Questa classe contiene i metodi che permettono di effettuare inserimento, ricerca e cancellazione di una sala.
SpettacoloModel.java	Questa classe contiene i metodi che permettono di effettuare inserimento, ricerca e cancellazione di uno spettacolo.
UtenteRegistratoModel.java	Questa classe contiene i metodi che permettono di effettuare inserimento, ricerca e cancellazione di un utente registrato.
PoltronaModel.java	Questa classe contiene i metodi che permettono di effettuare

	inserimento, ricerca e cancellazione di una poltrona.

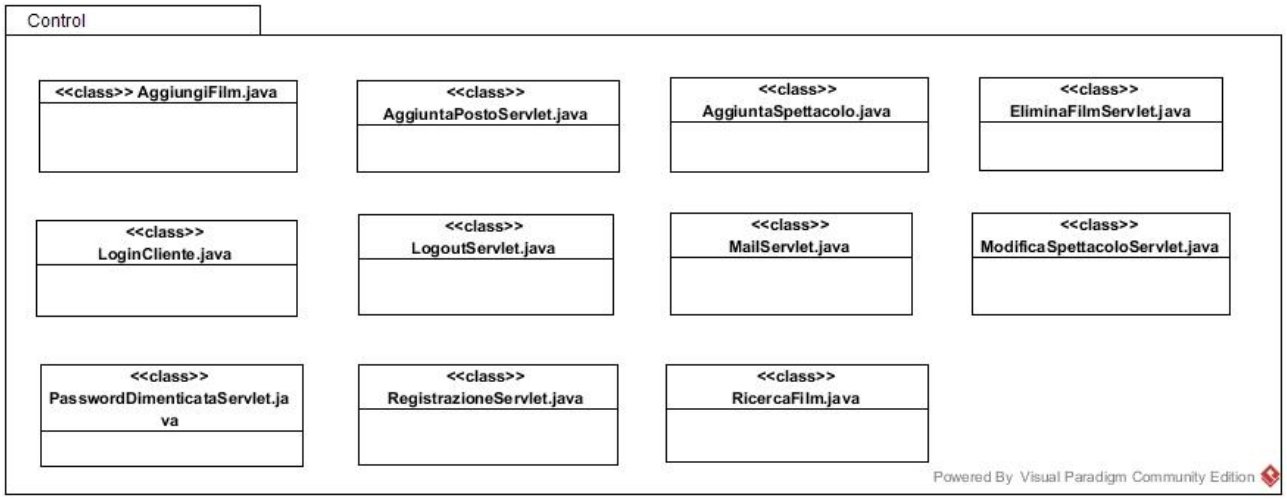
3.3 Package filter



Classe:	Descrizione:
Admin.java	Questo filtro si occupa di assicurare che solo gli utenti non amministratori accedano alle funzionalità loro dedicate.
Cliente.java	Questo filtro si occupa di assicurare che solo gli utenti amministratori accedano alle funzionalità loro dedicate e quindi blocca l'accesso ai clienti.
Generale.java	Questo filtro si occupa di gestire l'accesso alle restanti pagine del sistema da parte degli utenti di quest'ultimo.
Operatore.java	Questo filtro si occupa di assicurare che solo gli utenti non operatore accedano alle funzionalità loro dedicate.

Ospite.java	Questo filtro si occupa di assicurare che solo gli utenti non ospiti accedano alle funzionalità loro dedicate.

3.4 Package control



Classe:	Descrizione:
AggiuntaFilmServlet.java	Questa servlet si occupa di aggiungere film all'interno del DB.
LoginClienteServlet.java	Questa servlet si occupa di ricevere i dati di login, elaborarli e decidere se consentire o meno l'accesso all'area personale.

AggiuntaPostoServlet.java	Questa servlet si occupa di aggiungere un posto nel DB.
AggiuntaSpettacoloServlet.java	Questa servlet si occupa di aggiungere uno spettacolo nel DB.
EliminaFilmServlet.java	Questa servlet si occupa di eliminare film all'interno del DB.
LogoutServlet.java	Questa servlet si occupa di invalidare la sessione per consentire il logout.
ModificaSpettacoloServlet.java	Questa servlet si occupa di modificare dettagli relativi ad uno spettacolo nel DB.
RicercaFilmServlet.java	Questa servlet si occupa di ricercare un film all'interno del DB.
RegistrazioneServlet.java	Questa servlet si occupa di ricevere i dati relativi alla registrazione e di elaborarli. Se i dati sono corretti usa i servizi di UtenteRegistratoModel e di ClienteModel per completare la registrazione.

4. Class Interfaces

4.1 Client Model

ClienteModel	
doSave	**Context:** ClienteModel: doSave(cliente) **Pre:** cliente!=null && cliente.email !=null && cliente.nome !=null && cliente.cognome != null && cliente.indirizzo !=null && cliente.dataDiNascita !=null && clienti->forAll(c |c.email != cliente.email); **Post:** clienti ->include(cliente)
doDelete	**Context:** ClienteModel: doDelete(email) **Pre:** Clienti-> exists(c|c.email==email); **Post:** !Clienti-> exists(c|c.email==email);
doRetrieveByKey	**Context:** ClienteModel: doRetrieveByKey(email) **Pre:** Clienti-> exists(c|c.email==email); **Post:** Clienti->select (c|c.email == email);

4.2 Amministratore Model

Amministratore Model	
doSave	**Context:** AmministratoreModel: doSave(amministratore) **Pre:** amministratore !=null && amministratore.email !=null && amministratore.nome !=null && amministratore.cognome != null && amministratore.indirizzo !=null && amministratore.dataDiNascita !=null && amministratore ->forAll(a|a.email !=amministratore.email)

	Post: amministratore-> include(amministratore);
doDelete	**Context:** AmministratoreModel: doDelete(email) **Pre:** email !=null && amministratore-> exists(a|a.email==email); **Post:** !Amministratore-> exists(a|a.email==email);
doRetrieveByKey	**Context:** AmministratoreModel: doRetrieveByKey(email) **Pre:** email !=null && Amministratore-> exists(a|a.email==email); **Post:** amministratore->select(amministratore.email==email);

4.3 Biglietto Model

Biglietto Model	
doSave	**Context:** BigliettoModel: doSave(biglietto) **Pre:** biglietto.codiceBiglietto != null&& biglietto.prezzo != null && biglietto.numero != null && !(biglietti->exist(b|b.numero == biglietto.numero); **Post:** biglietti->exist(b|b.numero == biglietto.numero);

doDelete	**Context:** BigliettoModel: doDelete(numero) **Pre:** biglietti->exist(b|b.numero == biglietto.numero); **Post:** !biglietti->exist(b|b.numero == biglietto.numero);
doRetrieveByKey	**Context:** BigliettoModel: doRetrieveByKey(numero) **Pre:** numero !=null && biglietti->exists(b|b.numero == numero); **Post:** biglietti->select(biglietti.numero == numero);

4.4 Film Model

Film Model	
doSave	**Context:** FilmModel: doSave(film) **Pre:** film.idFilm != null && film.nome != null && film.durata!= null && film.genere != null && film.trama != null && film.immagine != null && !(film->exist(f| f.idFilm == film.idFilm); **Post:** film->exist(f.idFilm == film.idFilm);
doDelete	**Context:** FilmModel: doDelete(idFilm) **Pre:** idFilm != null && film->exist(f|f.idFilm == idFilm); **Post:** !film->exist(f|f.idFilm == idFilm);
doRetrieveByKey	**Context:** FilmModel: doRetrieveByKey(idFilm) **Pre:** idFilm !=null && film->exists(f|f.idFilm==idFilm); **Post:** film->select(film.IdFilm == idFilm);

4.5 Operatore Model

Amministratore	**Model**
doSave	**Context:** OperatoreModel: doSave(operatore) **Pre:** operatore!=null && operatore.email !=null && operatore.nome !=null && operatore.cognome != null && operatore.indirizzo !=null && operatore.dataDiNascita !=null && operatori->forAll(o |o.email != operatore.email); **Post:** operatori ->include(operatore)
doDelete	**Context:** OperatoreModel: doDelete(email) **Pre:** email!= null && operatori->exist(o|o.email == email); **Post:** !operatori->exist(o|o.email == email);
doRetrieveByKey	**Context:** OperatoreModel: doRetrieveByKey(email) **Pre:** email != null && operatori->exist(o|o.email == email); **Post:** operatori->select(o|o.email == email);

4.6 Ordine Model

Ordine Model	
doSave	**Context:** OrdineModel: doSave(ordine) **Pre:** ordine!=null && ordine.numeroOrdine =! Null && ordine.numeroPromozione !=null && ordine.numeroBigliettiAcquistati !=null && ordine.sceltaPagamento != null && ordine.SceltaRitiro !=null && ordine.sceltaPoltrona !=null && ordini->forAll(o |o.email != operatore.email);

	Post: ordini ->include(ordine)
doDelete	**Context:** OrdineModel: doDelete(numeroOrdine) **Pre:** numeroOrdine!= null && ordine->exist(o|o.numeroOrdine == numeroOrdine); **Post:** !ordine->exist(o|o.numeroOrdine == numeroOrdine);
doRetrieveByKey	**Context:** OrdineModel: doRetrieveByKey(numeroOrdine) **Pre:** numeroOrdine != null && ordini->exist(o|o.numeroOrdine == numeroOrdine); **Post:** ordini->select(o|o.numeroOrdine == numeroOrdine);

4.7 Promozione Model

Promozione Model	
doSave	**Context:** PromozioneModel: doSave(promozione) **Pre:** promozione !=null && promozione.idPromozione !=null && promozione.nome Promozione!=null && promozione.durata promozione!=null && promozione ->forAll(p |p.idPromozione != promozione.idPromozione); **Post:** promozioni ->include(promozione)
doDelete	**Context:** Promozione: doDelete(idPromozione) **Pre:** promozioni -> exist(p|p.idPromozione == idPromozione); **Post:** !promozioni->exist(p|p.idPromozione == idPromozione);
doRetrieveByKey	**Context:** PromozioneModel: doRetrieveByKey(idPromozione) **Pre:** idPromozione !=null &&

	promozioni->exist(p|p.idPromozione ==idPromozione); **Post:** promozioni→ select(p|p.idPromozione==idPromozione);

4.8 Spettacolo Model

Spettacolo Model	
doSave	**Context:** SpettacoloModel: doSave(spettacolo) **Pre:** spettacolo.idSpettacolo !=null && spettacolo.dataSpettacolo!=null && spettacolo.treD!=null && spettacolo.numeriPostiLiberi!=null && spettacoli ->forAll(s|s.idSpettacolo != spettacolo.idSpettacolo); **Post:** spettacoli ->include(spettacolo)
doDelete	**Context:** SalaModel: doDelete(idSpettacolo) **Pre:** spettacoli -> exist(s|s.idSpettacolo ==Spettacolo); **Post:** !spettacoli ->exist(s|s.idSpettacolo == Spettacolo);
doRetrieveByKey	**Context:** SpettacoloModel: doRetrieveByKey(idSpettacolo) **Pre:** idSpettacolo !=null && spettacoli->exist(s|s.idSpettacolo==idSpettacolo); **Post:** spettacoli→ select (s| s. idSpettacolo==idSpettacolo);

4.9 Sala Model

Sala Model

doSave	**Context:** SalaModel: doSave(sala) **Pre:** sala.idSala !=null && sala.numeroPostiDisponibili!=null && sale->forAll(s|s.idSala != sala.idSala); **Post:** sale ->include(sala)
doDelete	**Context:** SalaModel: doDelete(idSala) **Pre:** sale->exists(s| s.idSala = idSala) **Post:** !sale->exists(s| s.idSala = idSala)
doRetrieveByKey	**Context:** SalaModel: doRetrieveByKey(idSala) **Pre:** idSala !=null && sala ->exists(s|s.idSala==idSala); **Post:** sala→ select (s|s.idSala==idSala);

4.10 Utente Registrato Model

Utente Registrato Model	
doSave	**Context:** UtenteRegistratoModel:doSave(utenteRegistrato) **Pre:** utenteRegistrato !=null && utenteRegistrato.email !=null && utenteRegistrato.nome !=null && utenteRegistrato.cognome != null && utenteRegistrato.indirizzo !=null && utenteRegistrato.dataDiNascita !=null && utentiRegistrati->forAll(ur |ur.email != UtenteRegistrato.email); **Post:** UtenteRegistrato ->include(utenteRegistrato)

doDelete	**Context:** UtenteRegistratoModel: doDelete(email) **Pre:** utentiRegistrati -> exists(ur|ur.email=email); **Post:** !utentiRegistrati -> exists(ur|ur.email=email);
doRetrieveByKey	**Context:** UtenteRegistratoModel: doRetrieveByKey(email) **Pre:** email !=null && utentiRegistrati ->exists(ur|ur.email==email); **Post:** utentiRegistrati→ select (ur|ur.email ==email);

4.11 Poltrona Model

Poltrona Model	
doDelete	**Context:** PoltronaModel: doDelete(idPoltrona) **Pre:** poltrona->exists(p| p.idPoltrona == idPoltrona) **Post:** !poltrona ->exists(p| p. idPoltrona == idPoltrona)
doRetrieveByKey	**Context:** PoltronaModel: doRetrieveByKey(idPoltrona) **Pre:** idPoltrona !=null && Poltro na ->exists(p|p.idPoltrona==idPoltrona); **Post:** poltrona→ select (p|p.idPoltrona == idPoltrona);

4.12 Servlet Film

Servlet Film	
aggiungiFilm	**Pre:** film.idFilm != null && film.nome != null && film.durata!= null && film.genere != null && film.trama != null && film.immagine != null && !(film->exist(f| f.idFilm == film.idFilm); **Post:** Film->exists(f| f.idFilm== idFilm);
ricercaFilm	**Pre:** idFilm !=null && film-> exists(f|f.idFilm==idFilm); **Post:** film->select(film.IdFilm == idFilm);
eliminaFilm	**Pre:** idFilm!=null && Film-> exists(f|f.idFilm==idFilm); **Post:** !Film-> exists(f| f.idFilm==idFilm);

4.13 Servlet Spettacolo

Servlet Spettacolo	
aggiungiSpettacolo	**Pre:** spettacolo.idSpettacolo !=null && spettacolo.dataSpettacolo!=null && spettacolo.treD!=null && spettacolo.numeriPostiLiberi!=null && spettacoli ->forAll(s|s.idSpettacolo != spettacolo.idSpettacolo); **Post:** spettacoli ->include(spettacolo)
modificaSpettacolo	**Pre: :** idSpettacolo !=null && spettacolo-> exists(s|s.idSpettacolo==idSpettacolo) **Post:** spettacoli->update(Spettacolo)

4.14 Servlet Account

Servlet Account	
LoginCliente	**Pre:** UserName =! Null && Password != null; **Post:** clienti->exist(c | c.username == username && c.password == password)
RegistrazioneCliente	**Pre:** cliente !=null && cliente.email !=null && cliente.nome !=null && cliente.cognome != null && cliente.indirizzo !=null && cliente.dataDiNascita !=null && clienti->forAll(c |c.email != cliente.email);

	Post: clienti ->include(cliente)

5. Glossario

RAD: Documento di Analisi dei Requisiti.
DBMS: Sistema di gestione di basi di dati.
SDD: Documento di System Design.
ODD: Documento di Object Design.
Database: Insieme organizzato di dati persistenti.
Query: Termine utilizzato per indicare l'interrogazione da parte di un utente di un database
Utente Registrato: Il termine identifica l'utente che ha effettuato la registrazione sul sistema.