|  |
| --- |
| A logo with numbers and letters  Description automatically generated  **Università degli Studi di Salerno Corso di Ingegneria del Software** |

**Rated  
Object Design Document  
Versione 1.1**

**A black and white logo

Description automatically generated**

Data: 10/01/2025

**Coordinatore del progetto:**

|  |  |
| --- | --- |
| Nome | Matricola |
|  |  |
|  |  |

**Partecipanti:**

|  |  |
| --- | --- |
| Nome | Matricola |
| Francesco Rao | 0512116836 |
| Bruno Nesticò | 0512117268 |
|  |  |
|  |  |
|  |  |
|  |  |

|  |  |
| --- | --- |
| **Scritto da:** | Francesco Rao, Bruno Nesticò |

**Revision History**

|  |  |  |  |
| --- | --- | --- | --- |
| Data | Versione | Descrizione | Autore |
| 16/12/2024 | 1.0 | Prima stesura completa | Francesco Rao, Bruno Nesticò |
| 10/01/2025 | 1.1 |  | Francesco Rao, Bruno Nesticò |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

**Indice**

1. Introduzione 4

1.1 Scopo del Sistema 4

1.2 Obiettivi di progettazione 4

1.3 Linee guida per la documentazione dell’interfaccia 4

1.4 Ottimizzazione del modello a oggetti 4

2. Packages 5

2.1 Struttura del progetto 6

3. Interfaccia delle classi 7

4. Design Pattern 12

**1. Introduzione**

L’**Object Design Document** ha l'obiettivo di approfondire gli aspetti tecnici legati all'implementazione del sistema **Rated**, ampliando quanto definito nei documenti precedenti, focalizzati sull’architettura e sulla progettazione generale. Questo documento fornisce una descrizione dettagliata delle scelte progettuali effettuate durante le fasi di analisi e design, includendo i principali trade-off, linee guida per la documentazione delle interfacce e l’identificazione dei **Design Pattern** utilizzati.

Saranno definiti i packages, le interfacce delle classi e i relativi diagrammi, descrivendo nel dettaglio operazioni, parametri e firme, in coerenza con i sottosistemi individuati nel **System Design Document** e con i requisiti funzionali e non funzionali indicati nel **Requirements Analysis Document**.

Il documento si concentra su tutti i requisiti funzionali indicati nel **RAD**, che saranno implementati nella prima versione del sistema garantendo il rispetto delle scadenze.

* 1. **Object design trade-offs**

**Sicurezza vs. Tempi di Sviluppo:** Tenendo conto delle scadenze e dei tempi ristretti di sviluppo, saranno implementati sistemi di sicurezza essenziali per garantire un livello adeguato di protezione. Questi includono l'autenticazione tramite email e password crittografata, la sanificazione dei campi di input dei form e una gestione degli accessi alle pagine basata sui ruoli definiti nel RAD (Guest, Recensore, Gestore del Catalogo, Moderatore). Le pagine di errore, tuttavia, saranno progettate in modo minimale e orientate esclusivamente alla funzionalità essenziale. Nella prima versione del sistema, si presuppone che il Gestore del Catalogo, in qualità di operatore della piattaforma, abbia familiarità con le modalità di compilazione dei form relativi alle sue funzioni. Di conseguenza, i controlli su questi form saranno limitati a verifiche di base. Questo approccio rappresenta un compromesso mirato a garantire un adeguato livello di sicurezza per la protezione dei dati sensibili e il corretto funzionamento della piattaforma, mantenendo al contempo la rapidità e la semplicità di implementazione necessarie per rispettare le scadenze.

**Prestazioni vs. Supportabilità:** Nel contesto del progetto, è importante considerare il possibile trade-off tra prestazioni e supportabilità. Sebbene il requisito di prestazioni, definito nel RAD, richieda tempi di caricamento inferiori a 2 secondi per ogni pagina o funzione principale, si ritiene prioritario privilegiare la supportabilità del codice. Questa scelta si fonda sull'ipotesi che l'applicativo web non presenti una complessità elevata. Di conseguenza, è ragionevole supporre che una struttura del codice orientata alla manutenibilità e agli aggiornamenti futuri non comprometta in modo significativo le prestazioni del sistema. In altre parole, adottare pratiche di sviluppo che favoriscano la supportabilità, come una progettazione modulare, codice leggibile e testabile, dovrebbe consentire di mantenere un tempo di risposta inferiore ai 2 secondi, soddisfacendo così entrambi i requisiti. Tale approccio garantirebbe un equilibrio tra efficienza immediata e sostenibilità a lungo termine del sistema, riducendo il rischio di complicazioni durante l'evoluzione dell'applicativo.

* 1. **Interface documentation guidelines**

**Nomi dei Package**

* I nomi dei package devono essere scritti in minuscolo, senza spazi o caratteri speciali.
* Per nomi composti da più parole, è necessario utilizzare il formato snake\_case.

**Nomi delle Classi**

* Le classi devono seguire il formato PascalCase, iniziando con una lettera maiuscola.
* I nomi devono essere descrittivi, rappresentando chiaramente l’entità o la funzionalità implementata.

**Classi DAO**

* Le classi DAO devono seguire il formato PascalCase e terminare con il suffisso DAO per indicarne il ruolo di accesso ai dati.

**Classi che Forniscono Servizi**

* Queste classi devono rispettare il formato PascalCase.

**Nomi delle Servlet**

* Le Servlet devono seguire il formato PascalCase e terminare con il suffisso Servlet.

**Nomi dei Metodi**

* I metodi devono avere nomi descrittivi, che riflettano chiaramente l'operazione eseguita.
* Devono seguire il formato camelCase.

**Nomi delle Variabili**

* I nomi delle variabili devono essere descrittivi.
* È possibile utilizzare sia il formato camelCase che il formato snake\_case, in base al contesto.

**Nomi dei File JSP**

* I file JSP devono seguire il formato camelCase, riflettendo chiaramente il contenuto della pagina.

**Nomi delle Classi che Implementano il Pattern Strategy**

* Queste classi devono seguire il formato PascalCase e terminare con la parola Validator, per identificare chiaramente il loro ruolo.

**Organizzazione delle Risorse Statiche**

* Fogli di stile, script e immagini devono essere organizzati nella directory webapp/static, suddivisa in sottocartelle per ogni tipo di file.
  1. **Design Pattern**

Per implementare le funzionalità del sistema CineNow, sono stati adottati due design pattern: Connection Pool Management Pattern e Strategy Pattern. Di seguito vengono riportate le motivazioni che hanno portato all’adozione dei suddetti pattern nel contesto dell’applicazione.

**Connection Pool Management Pattern**

L'applicazione **Rated** richiede un accesso ottimizzato e centralizzato alle connessioni al database, data la natura concorrente delle operazioni effettuate dagli utenti. L’implementazione di un **Connection Pool Management** si rivela essenziale per migliorare l’efficienza nella gestione delle connessioni al database MySQL, garantendo una condivisione efficace delle risorse.

L’utilizzo di questa soluzione consente di mantenere un pool di connessioni già aperte e riutilizzabili, evitando l’overhead causato dalla creazione e dalla chiusura continua di nuove connessioni. Ciò garantisce:

* La gestione di un numero limitato di connessioni attive, prevenendo così un utilizzo inefficiente delle risorse del database.
* Una maggiore scalabilità del sistema, grazie alla possibilità di servire più richieste concorrenti.
* Un ciclo di vita chiaro e centralizzato per tutte le connessioni.

Nel progetto **Rated**, il **DriverConnectionPool** è responsabile della gestione di queste connessioni. Attraverso un’allocazione intelligente e il rilascio delle connessioni utilizzate, il pool di connessioni garantisce la continuità operativa dell'applicazione senza sovraccaricare il DBMS.

L’accesso alle connessioni nel pool avviene tramite metodi che permettono di acquisire una connessione disponibile e di restituirla una volta terminato l’utilizzo. In questo modo si ottimizza l’utilizzo delle risorse e si previene il verificarsi di problemi di saturazione delle connessioni.

**Strategy Pattern**

Nel progetto, per la gestione della validazione dei campi di input, è stato adottato il Strategy Pattern. Questo design pattern consente di definire una famiglia di algoritmi di validazione (ad esempio, validazione di email, numeri, date, ecc.), incapsularli in metodi specifici all'interno di una classe e renderli intercambiabili. La classe di validazione creata funge da contenitore per tutte le funzioni di validazione necessarie, ognuna delle quali rappresenta una strategia separata per verificare uno specifico tipo di input. Questo approccio garantisce:

* Modularità: Ogni algoritmo di validazione è implementato come un metodo distinto, facilitando la leggibilità e la manutenzione del codice.
* Flessibilità: È possibile aggiungere facilmente nuove funzioni di validazione o aggiornare quelle esistenti senza modificare il codice già scritto.
* Riutilizzabilità: Le funzioni di validazione possono essere richiamate in modo indipendente o combinate, a seconda delle necessità del sistema. Chiarezza: Separando la logica di validazione dalla logica principale dell'applicazione, si ottiene un design più chiaro e mantenibile.

**1.3. Definizioni, acronimi e abbreviazioni**

**2. Packages**

**2.1. Struttura del progetto**

Di seguito viene indicata la struttura organizzativa di file e cartelle che compongono la parte implementativa del sistema.

**.mvn**, contenente i file di configurazione di Maven

**src**, contente i file sorgente del progetto

**main**

**java,** contentente I package files di Java

**resources,** contenente le risorse relative all’interfaccia utente

**static,** contenente le risorse statiche come CSS e JS files

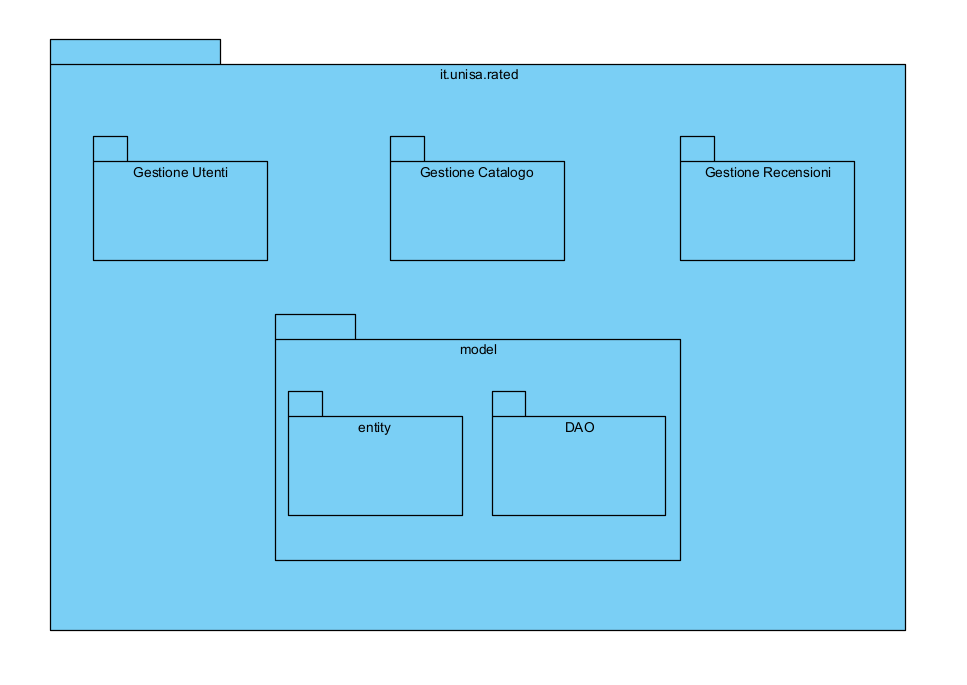
**templates,** contenente file HTML dinamici

**test**

**java,** contenente le classi di testing

**target,** contenente i file build i Maven

Il sistema è formato da un package generale chiamato *it.unisa.rated* all’interno del quale sarà presente un package per ogni sottosistema individuato. Oltre a questi sarà presente anche un package model , contenente le class entity e i DAO.



**3. Interfaccia delle classi**

**Package Gestione Utenti**

|  |  |
| --- | --- |
| Interfaccia | GestioneUtentiService |
| Descrizione | Gestione Utenti fornisce il servizio relativo all’autenticazione e all’aggiornamento degli account |
| Metodi | + LogIn(String email, String password) : Utente  + SingUp(String username, Blob icon, String email, String password) : Utente  + getUtente(String email) : Utente  + getUtenti() : Collection<Utente> |
| Invariante classe | // |

|  |  |
| --- | --- |
| Nome Metodo | + LogIn(String email, String password) : Utente |
| Descrizione | Il metodo LogIn consente di verificare se un utente esiste nel sistema ed è autorizzato ad accedere. |
| Pre-condizione | context GestioneUtentiService::autentica(String email, String password): Utente  pre: email != null and password != null |
| Post-condizione | context GestioneUtentiService::autentica(String email, String password): Utente  post: result != null and result.email = email |

|  |  |
| --- | --- |
| Nome Metodo | + SingUp(String username, Blob icon, String email, String password) : Utente |
| Descrizione | Il metodo SingUp consente di reggistrare un nuovo utente nel sistema |
| Pre-condizione | context GestioneUtentiService:: SingUp(String username, Blob icon, String email, String password) : Utente  pre: username != null and nome.size() > 0  pre: email != null and email.size() > 0  pre: password != null and password.size() >= 8  pre: self.getUtente(email) = null |
| Post-condizione | context GestioneUtentiService:: SingUp(String username, Blob icon, String email, String password) : Utente  post: result != null and self.getUtente(email) = result |

|  |  |
| --- | --- |
| Nome Metodo | + getUtente(String email) : Utente |
| Descrizione | Il mettodo getUtente() permette di trovare l’utente associato ad una mail |
| Pre-condizione | Context GestioneUtentiService::getUtente(String email) : Utente  pre email != null and email.size() > 0 |
| Post-condizione | Context GestioneUtentiService::getUtente(String email) : Utente  post result != null and self.getUtenti().exist(u | u.email = email)) |

|  |  |
| --- | --- |
| Nome Metodo | + getUtenti() : Collection<Utente> |
| Descrizione | Il metodo getUtenti() restituisce la lista di tutti gli utenti del sistema |
| Pre-condizione | // |
| Post-condizione | // |

**Package Gestione Catalogo**

|  |  |
| --- | --- |
| Interfaccia | GestioneCatalogoService |
| Descrizione | Gestione Catalogo fornisce il servizio relativo alla aggiunta, modifica e rimozione dei film |
| Metodi | + getFilm(String name) : Collection<Film> +addFilm(String name, int year, int length, String Attori):Film +updateFilm(Film m): Film +deleteFilm(Film m):boolean |
| Invariante classe | // |

|  |  |
| --- | --- |
| Nome Metodo | + getFilm(String name) : Collection<Camera> |
| Descrizione | Il metodo getFim(String name) permette di cercare un film tramite il suo nome |
| Pre-condizione | // |
| Post-condizione | // |

|  |  |
| --- | --- |
| Nome Metodo | +addFilm(String name, int year, int length, String Attori):Film |
| Descrizione | Il metodo addFilm permette di aggiungere/creare nuovi film nel catalogo |
| Pre-condizione | context GestioneCatalogoService:: addFilm(String name, int year, int length, String Attori):Film  pre: name != null and nome.size() > 0  pre: year != null  pre: length >0 |
| Post-condizione | context GestioneCatalogoService:: addFilm(String name, int year, int length, String Attori):Film  post: result != null and self.getFilm(name) = result |

|  |  |
| --- | --- |
| Nome Metodo | +updateFilm(Film f): Film |
| Descrizione | Il metodo updateFilm permette di salvare le modifiche effettuate su di un film |
| Pre-condizione | context GestioneCatalogoService:: updateFilm(Film f): Film  pre: name != null and nome.size() > 0  pre: year != null  pre: length >0 |
| Post-condizione | context GestioneCatalogoService:: addFilm(String name, int year, int length, String Attori):Film  post: result != null and self.getFilm(name) = result |

|  |  |
| --- | --- |
| Nome Metodo | +deleteFilm(Film m):Film |
| Descrizione | Il metodo permette di eliminare un determinato film dal Catalogo |
| Pre-condizione | context GestioneCatalogoService:: deleteFilm(Film m):Film: boolean  pre: m != null  pre: self.getFilm (“”) 🡪 include(m) |
| Post-condizione | context GestioneServiziService:: deleteFilm(Film m):Film: boolean  post: result = true and (not self.getFilm () 🡪 include(m)) |

**Package Gestione Recensioni**

|  |  |
| --- | --- |
| Interfaccia | GestioneRecensioniService |
| Descrizione | Gestione Recensioni fornisce il servizio relativo alla aggiunta, modifica e rimozione delle recensioni e della vautazione delle stesse |
| Metodi | +getRecensioni(Film f): Collection<Recensioni> +addRecensione(Film f, Utente u, String titolo, String Contenuto, int valutazione): Recensione +addValutazione(Film f, Utente u, Vote v, Recensione r) : boolean  +removeRecensione(Recensione r):boolean  +getValutazione (Film f, Utente u):Valutazione |
| Invariante classe | // |

|  |  |
| --- | --- |
| Nome Metodo | +getRecensioni(Film f): Collection<Recensioni> |
| Descrizione | Il metodo getRecensioni permette di ottenere tutte le recensioni di un determinato film |
| Pre-condizione | // |
| Post-condizione | // |

|  |  |
| --- | --- |
| Nome Metodo | +addRecensione(Film f, Utente u, String titolo, String Contenuto, int valutazione): Recensione |
| Descrizione | Il metodo addRecensione permette di aggiungere una recensione scritta da una persona ad un film |
| Pre-condizione | context GestioneRecensioniService:: addRecensione(Film f, Utente u, String titolo, String Contenuto, int valutazione): Recensione  pre: f != null pre: u != null  pre: titolo != null and titolo.length()>0 pre: Contenuto != null and Contenuto.legth()>0 pre: valutazione>0 and valutazione<=5 |
| Post-condizione | context GestioneRecensioniService:: addRecensione(Film f, Utente u, String titolo, String Contenuto, int valutazione): Recensione   post: result != null and self.getRecensione (titolo) = result |

|  |  |
| --- | --- |
| Nome Metodo | +addValutazione(Film f, Utente u, Vote v, Recensione r) : boolean |
| Descrizione | Il metodo addValutazione permette di aggiungere la valutazione di un utente rispetto ad una recensione di un film |
| Pre-condizione | context GestioneRecensioniService:: addValutazione(Film f, Utente u) : boolen pre: f != null pre: u != null  pre: v != null pre r != null |
| Post-condizione | context GestioneRecensioniService:: addValutazione(Film f, Utente u) : Valutazione  post: result = self.getValutazione (Film f, Utente u,) !=null |

|  |  |
| --- | --- |
| Nome Metodo | +removeRecensione(Recensione r):boolean |
| Descrizione | Il metodo removeRecensione permette di rimuovere una recensione |
| Pre-condizione | Context GestioneRecensioniService::removeRecensione(Recensione r) : boolean  pre: r!=null |
| Post-condizione | post: result = !self.getRecensioni(r.getFilm()).iclude(r) |

|  |  |
| --- | --- |
| Nome Metodo | +getValutazione (Film f, Utente u):Valutazione |
| Descrizione | Il metodo getValutazione permette di ottenere la valutazione di un utente su riguardante un film specificato |
| Pre-condizione | // |
| Post-condizione | Context GestioneRecensioniService:: getValutazione (Film f, Utente u):Valutazione post: result =recensione and recensione.getFilm()=f and recensione.getUtente()=u |

**4. Design Pattern**

**Design Pattern: Facade**

Il Facade Pattern è un design pattern strutturale utilizzato per semplificare l'interazione con sistemi complessi composti da più sottosistemi. Questo approccio fornisce un'interfaccia chiara e accessibile, nascondendo i dettagli di implementazione e offrendo agli utenti una visione unificata e di alto livello.

L'applicazione di questo pattern è particolarmente utile quando è necessario semplificare un sistema complesso, rendendolo più intuitivo e facile da utilizzare. Inoltre, il Facade favorisce un elevato livello di disaccoppiamento, migliorando la manutenibilità e l'estensibilità del sistema: eventuali modifiche possono essere apportate intervenendo esclusivamente sui metodi esposti dall’interfaccia, senza impattare direttamente i singoli sottosistemi.

In questo contesto specifico, ogni sottosistema dispone della propria interfaccia dedicata, ad esempio:

* GestioneUtentiService
* GestioneCatalogoService
* GestioneRecensioniService

**Data Access Object (DAO)**

Il Data Access Object (DAO) è un design pattern che astrae e centralizza l'accesso ai dati, fornendo un'interfaccia uniforme per l'interazione con il database. Grazie a questa astrazione, il codice applicativo può manipolare i dati senza doversi preoccupare dei dettagli specifici del database o delle query SQL.

Il DAO agisce come un ponte tra l'applicazione e il sistema di persistenza, garantendo un codice più pulito, organizzato e facilmente manutenibile. Questo pattern facilita inoltre l'integrazione di eventuali cambiamenti nella logica di accesso ai dati, riducendo al minimo l'impatto sulle altre componenti del sistema.