|  |
| --- |
| UNISA | Home  **Università degli Studi di Salerno Corso di Ingegneria del Software** |

**Mockbuster  
Object Design Document**

**Versione 1.0**

****

Data: 17/01/2025

**Coordinatore del progetto:**

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

**Partecipanti:**

|  |  |
| --- | --- |
| Nome | Matricola |
| Roberto Ambrosino | 0512117886 |

|  |  |
| --- | --- |
| **Scritto da:** | Roberto Ambrosino |

**Revision History**

|  |  |  |  |
| --- | --- | --- | --- |
| Data | Versione | Descrizione | Autore |
| 16/12/2024 | 0.1 | Prima stesura del documento | Roberto Ambrosino |
| 17/01/2025 | 1.0 | Aggiunti introduzione e packages, aggiornate specifiche delle interfacce dei sottosistemi | Roberto Ambrosino |

|  |  |
| --- | --- |
| **1 - Introduzione** | 4 |
| **2 – Packages** | 5 |
| **3 - Specifica delle interfacce dei sotto-sistemi** | 7 |

**1 - Introduzione**

**1.1 - Object design trade-offs**Per lo sviluppo di Mockbuster, sono stati valutati i seguenti compromessi:

* **Buy vs. Build**: Si è scelto di costruire internamente la logica di business (classi service e DAO) per garantire maggiore controllo sul sistema e flessibilità nelle future modifiche. Per la gestione della persistenza, invece, è stato utilizzato JPA con EclipseLink.
* **Tempo di risposta vs. utilizzo della memoria:** Si è optato per strategie che minimizzano i tempi di risposta delle operazioni principali, anche a costo di un leggero aumento del consumo di memoria (es. caching delle entità e delle query tramite JPA).
* **Eccezioni vs. valori di ritorno:** Gli errori vengono gestiti tramite l'uso di eccezioni personalizzate (es. DAOException), garantendo maggiore chiarezza nella segnalazione dei problemi rispetto ai valori di ritorno.

**1.2 - Interface documentation guidelines**

Per garantire una progettazione coerente e facilitare la comunicazione tra sviluppatori, sono state definite le seguenti linee guida:

* Adozione di CamelCase come convenzione per il nome di classi e attributi.
* Adozione di SCREAMING\_SNAKE\_CASE, separando le parole con un underscore (es. RETRIEVE\_BY\_ID)
* Le classi sono denominate con nomi sostantivi singolari (es. MovieDAO, OrderService).
* I metodi sono denominati con frasi verbali che riconducano allo scopo del metodo (es. retrieveByID, save).
* I campi e i parametri sono denominati con sostantivi (es. movie, orderID).
* Gli errori vengono segnalati tramite eccezioni personalizzate (es. DAOException) invece di valori di ritorno.

**1.3 - Definizioni, acronimi, e abbreviazioni**

* DAO: Data Access Object, classi responsabili della gestione della persistenza dei dati.
* JPA: Java Persistence API, specifica per la gestione della persistenza.
* EclipseLink: Implementazione di JPA utilizzata nel progetto.
* GlassFish: Application server utilizzato per eseguire l'applicazione.
* EntityManager: Componente di JPA utilizzato per gestire le entità e le operazioni sul database.
* SDD: System Design Document, documento che descrive il design generale del sistema.

**1.4 - Riferimenti**

* Problem Statement (PS\_Mockbuster): Descrive il dominio del problema e i requisiti iniziali del sistema
* Requirements Analysis Document (RAD\_Mockbuster): Definisce i requisiti funzionali (RF) e non funzionali (RNF), oltre ai casi d'uso principali
* System Design Document (SDD\_Mockbuster): Fornisce una descrizione dettagliata dell'architettura del sistema e dei suoi sottosistemi
* Test Plan Document (TP\_Mockbuster): descrive il piano di testing del sistema

**2 - Packages**

La struttura dei pacchetti è suddivisa in base alle responsabilità principali del sistema, per garantire una chiara separazione dei sottosistemi e facilitare la manutenzione del codice. Di seguito viene fornita una descrizione dettagliata dei pacchetti, delle loro dipendenze e del loro utilizzo previsto.

**2.1 Package control**

Il package control contiene le servlet che gestiscono il flusso delle operazioni principali tra l'interfaccia utente e i servizi. È suddiviso in subpackage per una gestione modulare delle diverse funzionalità dell'applicazione.

**control.admin**

Contiene le servlet responsabili della gestione delle operazioni amministrative, come la modifica del catalogo dei film o la visualizzazione di tutti gli ordini

Dipendenze: persistence.service

Utilizzo previsto: Funzioni accessibili esclusivamente agli utenti con ruolo di amministratore.

**control.browse**

Contiene le servlet per la navigazione da parte dei clienti.

Dipendenze: persistence.service

Utilizzo previsto: Supporto alle operazioni di esplorazione e ricerca dei contenuti da parte di clienti.

**control.common**

Contiene le servlet che forniscono funzionalità necessarie a tutti gli utilizzatori della piattaforma, come operazioni di login, logout o signup.

Dipendenze: persistence.service

Utilizzo previsto: fornire operazioni comuni a tutti gli utilizzatori della piattaforma.

**control.exceptions**

Contiene le definizioni delle eccezioni personalizzate utilizzate nel sistema per gestire errori specifici della logica applicativa.

Dipendenze: Nessuna dipendenza diretta.

Utilizzo previsto: Gestione personalizzata degli errori e propagazione delle eccezioni.

**control.filters**

Contiene i filtri utilizzati per la gestione della sicurezza e del controllo degli accessi, come il controllo del ruolo dell'utente o la validazione dei parametri inviati.

Dipendenze: security

Utilizzo previsto: Controllo delle richieste, controllo dei ruoli, validazione dei parametri.

**2.2 Package persistence**

Il package persistence gestisce l'interazione con il database e l'accesso ai dati persistenti. È suddiviso in tre subpackage per una separazione chiara delle responsabilità.

**persistence.dao**

Contiene le classi DAO (Data Access Object) responsabili delle operazioni CRUD sul database.

Dipendenze: Nessuna dipendenza diretta.

Utilizzo previsto: Incapsulamento della logica per l’accesso al database per ridurre il coupling con altri livelli.

**persistence.model**

Contiene le classi che rappresentano gli oggetti del dominio del sistema, come Movie, Order e Customer. Questi oggetti sono mappati alle tabelle del database tramite JPA.

Dipendenze: Nessuna dipendenza diretta.

Utilizzo previsto: Rappresentazione degli oggetti persistenti.

**persistence.service**

Contiene le classi di servizio che implementano la logica di business e interagiscono con i DAO. Queste classi fungono da intermediari tra il package di controllo e il package di persistenza.

Dipendenze: persistence.dao, persistence.model

Utilizzo previsto: Centralizzazione della logica applicativa legata ai dati.

**2.3 Package security**

Il package security è responsabile della gestione degli aspetti di sicurezza del sistema, come l'autorizzazione.

Dipendenze: Nessuna dipendenza diretta.

Utilizzo previsto: Implementazione delle funzionalità per la gestione delle autorizzazioni.

**3 - Specifica delle interfacce dei sotto-sistemi**

|  |  |
| --- | --- |
| **UserService** | |
| **Descrizione** | Questa classe fornisce i servizi legati agli utenti, come la registrazione, il login, la gestione del credito. Utilizza il DAO UserDAO per interfacciarsi con il database. |
| **Package** | persistence.service |
| **Attributi** | -userDAO: UserDAO |
| **Metodi** | |
| + signup(user: Customer): void | * context UserService::signup(user: Customer): void * pre: true * post: User.allInstances()->exists(u | u.email = user.email and u.password = user.password and u.name = user.name and u.surname = user.surname and u.billingAddress = user.billingAddress) |
| + customerLogin(email: String, password: String): Customer | * context UserService::customerLogin(email: String, password: String): Customer * pre: true * post: result = null or (Customer.allInstances()->exists(u | u.email = email and u.password = password)) |
| + catalogManagerLogin(email: String, password: String): CatalogManager | * context UserService:: catalogManagerLogin(email: String, password: String): CatalogManager * pre: true * post: result = null or (CatalogManager.allInstances()->exists(u | u.email = email and u.password = password)) |
| + orderManagerLogin(email: String, password: String): OrderManager | * context UserService:: orderManagerLogin(email: String, password: String): CatalogManager * pre: true * post: result = null or (OrderManager.allInstances()->exists(u | u.email = email and u.password = password)) |
| - toHash(password: String): String | * context UserService::toHash(password: String): String * pre: true * post: result.size() > 0 |
| + retrieveAllCustomers(): Collection<Customer> | * context UserService::retrieveAllCustomers(): Collection<Customer> * pre: true * post: result->forAll(c | c instanceof Customer) |
| + deductCredit(user: Customer, amount: Float): void | * context UserService::deductCredit(user: Customer, amount: Float) * pre: true * post: user.credit = user.credit - amount |
| + update(user: Customer): void | * context UserService::update(user: Customer) * pre: User.allInstances()->exists(u | u = user) * post: User.allInstances()->exists(u | u.id = user.id and u = user) |
| + checkEmailAvailability(email: String): Boolean | * context UserService::checkEmailAvailability(email: String): Boolean * post: result = (User.allInstances()->exists(u | u.email = email)) |

|  |  |
| --- | --- |
| **OrderService** | |
| **Descrizione** | Questa classe gestisce le operazioni relative agli ordini come creazione e recupero di dettagli. Utilizza i servizi di UserService e MovieService per l'acquisto e il noleggio di film. |
| **Package** | persistence.service |
| **Attributi** | -orderDAO: OrderDAO  -userService: UserService  -movieService: MovieService |
| **Metodi** | |
| + retrieveByUser(user: Customer): Collection<Order> | * context OrderService::retrieveByUser(user: Customer): Collection<Order> * pre: true * post: result = Order.allInstances()->select(o | o.user = user) |
| + retrieveOrderDetails(orderID: Integer, userID: Integer): Order | * context OrderService::retrieveOrderDetails(orderID: Integer, userID: Integer): Order * pre: true * post: result = Order.allInstances()->exists(o | o.id = orderID and o.user.id = userID) |
| + retrieveOrderDetails(orderID: Integer): Order | * context OrderService::retrieveOrderDetails(orderID: Integer): Order * pre: true * post: result = Order.allInstances()->exists(o | o.id = orderID) |
| + placeOrder(user: Customer, cart: Cart): Order | * context OrderService::placeOrder(user: Customer, cart: Cart): Order * pre: true * post: user.credit = user.credit - cart.total |
| - retrieveAll(): Collection<Order> | * context OrderService::retrieveAll(): Collection<Order> * pre: true * post: result = Order.allInstances() |
| + retrieveAllBetween(from: LocalDate, to: LocalDate, userID: Integer): Collection<Order> | * context OrderService::retrieveAllBetween(from: LocalDate, to: LocalDate, userID: Integer): Collection<Order> * pre: true * post: result = Order.allInstances()->select(o | o.date >= from and o.date <= to and (userID = null or o.user.id = userID)) |

|  |  |
| --- | --- |
| **MovieService** | |
| **Descrizione** | Questa classe gestisce le operazioni relative ai film, come il recupero, il caricamento di nuovi film, l'aggiornamento di film esistenti, le operazioni di noleggio o acquisto. Utilizza il MovieDAO per interagire con il database. |
| **Package** | persistence.service |
| **Attributi** | -movieService: MovieService |
| **Metodi** | |
| + retrieveAll(): Collection<Movie> | * context MovieService::retrieveAll(): Collection<Movie> * pre: true * post: result = Movie.allInstances() |
| + retrieveByID(id: Integer): Movie | * context MovieService::retrieveByID(id: Integer): Movie * pre: true * post: result = Movie.allInstances()->exists(m | m.id = id) |
| + retrieveByTitle(title: String): Collection<Movie> | * context MovieService::retrieveByTitle(title: String): Collection<Movie> * pre: true * post: result = Movie.allInstances()->select(m | m.title = title) |
| + upload(movie: Movie): void | * context MovieService::upload(movie: Movie): void * pre: true * post: Movie.allInstances()->exists(m | m = movie) |
| - update(movie: Movie): void | * context MovieService::update(movie: Movie): void * pre: true * post: Movie.allInstances()->exists(m | m = movie) |
| + rentMovie(rm: RentedMovie): void | * context MovieService::rentMovie(rm: RentedMovie): void * pre: true * post: rm.getMovie().availableLicences = rm.getMovie().availableLicences – rm.getDays() |
| + purchaseMovie(pm: PurchasedMovie): void | * context MovieService::purchaseMovie(pm: PurchasedMovie): void * pre: true * post: pm.getMovie().availableLicences = pm.getMovie().availableLicences – 1 |

|  |  |
| --- | --- |
| **CartService** | |
| **Descrizione** | Questa classe gestisce le operazioni relative al carrello, come l’aggiornamento del carrello, l'aggiunta di film da noleggiare o acquistare. Usa MovieService per ottenere informazioni sui film e verificare la disponibilità delle licenze. |
| **Package** | persistence.service |
| **Attributi** | -movieService: MovieService |
| **Metodi** | |
| + refreshCart(cart: Cart): void | * context CartService::refreshCart(cart: Cart): void * pre: true * post: cart.rentedMovies = cart.rentedMovies->select(rm | rm.movie.isVisible and rm.days <= rm.movie.availableLicenses) and cart.purchasedMovies = cart.purchasedMovies->select(pm | pm.movie.isVisible and pm.movie.availableLicenses >= 1) |
| + addRent(cart: Cart, movieID: Integer, days: Integer): void | * context CartService::addRent(cart: Cart, movieID: Integer, days: Integer): void * pre: not cart.purchasesContains(movieID) and not cart.rentsContains(movieID)   and days <= Movie.allInstances()->select(m | m.id = movieID.availableLicenses   * post: cart.rentedMovies->exists(rm | rm.movie.id = movieID and rm.days = days) |
| + addPurchase(cart: Cart, movieID: Integer): void | * context CartService::addPurchase(cart: Cart, movieID: Integer): void * pre: not cart.purchasesContains(movieID) and not cart.rentsContains(movieID) and Movie.allInstances()->select(m | m.id = movieID.availableLicenses >= 1 * post: cart.purchasedMovies->exists(pm | pm.movie.id = movieID) |