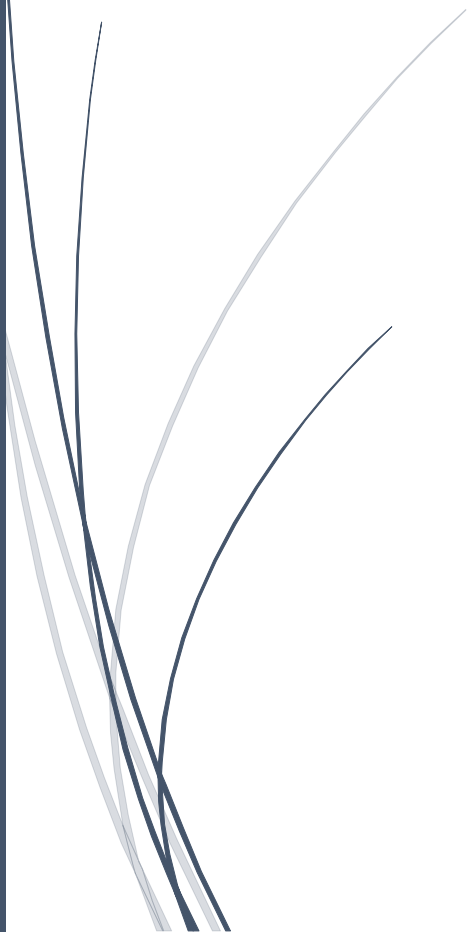
[Data]

O.D.D- Object Design Document

Team TopHw



Partecipanti

Nome	Cognome	Matricola
Alfonso	Rianna	0512104922
Davide	Zambelli	0512104688
Raffaele	Dragone	0512105016

Scritto da

Nome	Cognome	Matricola
Alfonso	Rianna	0512104922
Davide	Zambelli	0512104688
Raffaele	Dragone	0512105016

Sommario

1. INTRODUZIONE

1.1 Object Design

Durante l'ideazione e lo sviluppo del progetto, si è ritenuto idoneo fare alcune scelte progettuali, mirate soprattutto a ridurre la complessità nello sviluppo. Si è scelto di sviluppare la parte server del progetto usando il linguaggio di programmazione Java , sfruttando la programmazione orientata agli Oggetti permettendo la separazione tra dati e logica applicativa e utilizzando il Il Pattern Architetturale MVC(Model View Controller), criterio tramite il quale, si tende a separare dati persistenti dalla logica applicativa e di presentazione. Si è tenuto conto di alcune peculiarità della OOP(programmazione orientata agli oggetti), come l' architettura, la maggiore chiarezza e linearità del codice ad oggetti rispetto a quello procedurale permettendo di ottenere un sorgente modulare e facile da modificare. Inoltre la OOP consente di gestire meglio gli errori e semplifica il debugging. Per la parte client del progetto TopHW è stata, invece, effettuata la scelta di sviluppare in Javascript, Ajax e in particolar modo per la parte grafica HTML e CSS.
I linguaggi utilizzati garantiscono la caratteristica di portabilità, qualità di fondamentale importanza per un sistema.
Inoltre, il sistema è supportato dai browser più utilizzati (Chrome, Mozilla, Safari, Opera, Explorer).

Trade-offs

Comprensibilità vs Tempo: Il codice deve essere quanto più comprensibile possibile per facilitare la fase di testing ed eventuali future modifiche. Il codice sarà quindi accompagnato da commenti che ne semplifichino la comprensione. Ovviamente questa caratteristica aggiungerà un incremento di tempo allo sviluppo del nostro progetto.

Prestazioni vs Costi: Essendo il progetto sprovvisto di budget, al fine di mantenere prestazioni elevate, per alcune funzionalità verranno utilizzati dei template open source esterni in particolare Bootstrap.

Interfaccia vs Usabilità: L'interfaccia grafica è stata realizzata in modo da essere molto semplice, chiara e concisa, fa uso di form 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 del RAD, rappresenta uno degli aspetti importanti del sistema. Tuttavia, dati i tempi di sviluppo molto limitati, ci limiteremo ad implementare sistemi di sicurezza basati su username e password degli utenti.

1.2 Linee Guida per la Documentazione delle interfacce

Abbiamo ritenuto utile utilizzare alcune "regole" che saranno rispettate dagli sviluppatori del sistema durante la fase di implementazione. In particolar modo riguardano, i nomi di pacchetti, classi, metodi, campi e variabili che faranno parte del codice.

1.2.1 Classi

- I nomi delle classi esprimono la funzione degli oggetti istanziati di tale classe.
- Ogni classe deve avere un nome che sia diverso da quelli delle altre classi, dei pacchetti e dei metodi.

1.2.3 Metodi

- Ogni metodo deve avere un nome composto da uno o più termini singolari della lingua italiana o della lingua inglese.
- Il primo termine del nome di ogni metodo deve essere totalmente in minuscolo, mentre gli altri termini del nome devono avere la lettera iniziale in maiuscolo e le rimanenti lettere in minuscolo.
- Ogni metodo deve avere un nome diverso da quello degli altri metodi della sua stessa classe e del suo stesso package.
- Ogni nome di metodo deve essere significativo relativamente al comportamento e alloscopo dello stesso metodo all'interno della sua classe e dell'intera applicazione.
- Ogni classe deve avere uno o più costruttori, tra cui uno di default che non inizializzi alcun campo della classe a cui appartiene.

1.2.4. Campi, Variabili e Costanti

- Ogni campo ed ogni variabile deve avere un nome composto da uno o più termini che siano sostantivi della lingua italiana o inglese, il primo dei quali deve essere al singolare; tale nome può contenere solo caratteri alfanumerici.
- Ogni campo deve avere un nome diverso rispetto a quello di qualsiasi altro campo della stessa classe.
- Ogni variabile ed ogni costante deve avere un nome diverso da quello di qualsiasi altra variabile o costante presenti nel loro stesso blocco di codice della loro stessa classe.
- Ogni campo ed ogni variabile deve avere un nome il cui primo termine ha tutte le lettere minuscole, mentre gli altri termini devono avere la prima lettera in maiuscolo e le restanti lettere in minuscolo.

1.2.5. Commenti

- I commenti del codice devono essere commenti con la relativa sintassi di ogni linguaggio.
- Ogni commento deve spiegare una parte di codice in maniera molto chiara e facile da comprendere da sviluppatori affiancati all'implementazione del progetto in un secondo momento.
- Ogni commento deve essere breve, in modo da favorire la sua lettura.

- Tutti i metodi differenti da quelli di tipo “get” o “set” devono essere commentati poco prima della loro firma e dopo l’implementazione di possibili altri metodi che li precedono.
- Per ogni metodo diverso da quelli di tipo “get” o “set”, tutti i parametri espliciti ed i valori di ritorno devono essere commentati semplicemente e brevemente; tali commenti essere posti immediatamente prima la firma di un metodo.
- Ogni controllo fatto nel codice per garantire la corretta esecuzione di una o più istruzioni deve essere commentato brevemente e semplicemente.

1.4 Riferimenti

Libro di testo “Object-Oriented Software Engineering – Using UML, Patterns and Java” di BerndBruegge e Allen H. Dutoit, edito da Prentice Hall

Programmi utilizzati per realizzare la documentazione:
- Visual Paradigm, star UML.

2. Package

La gestione del nostro sistema è suddivisa in tre livelli (three-tier):

- Interface layer
- Application Logic layer
- Storage layer

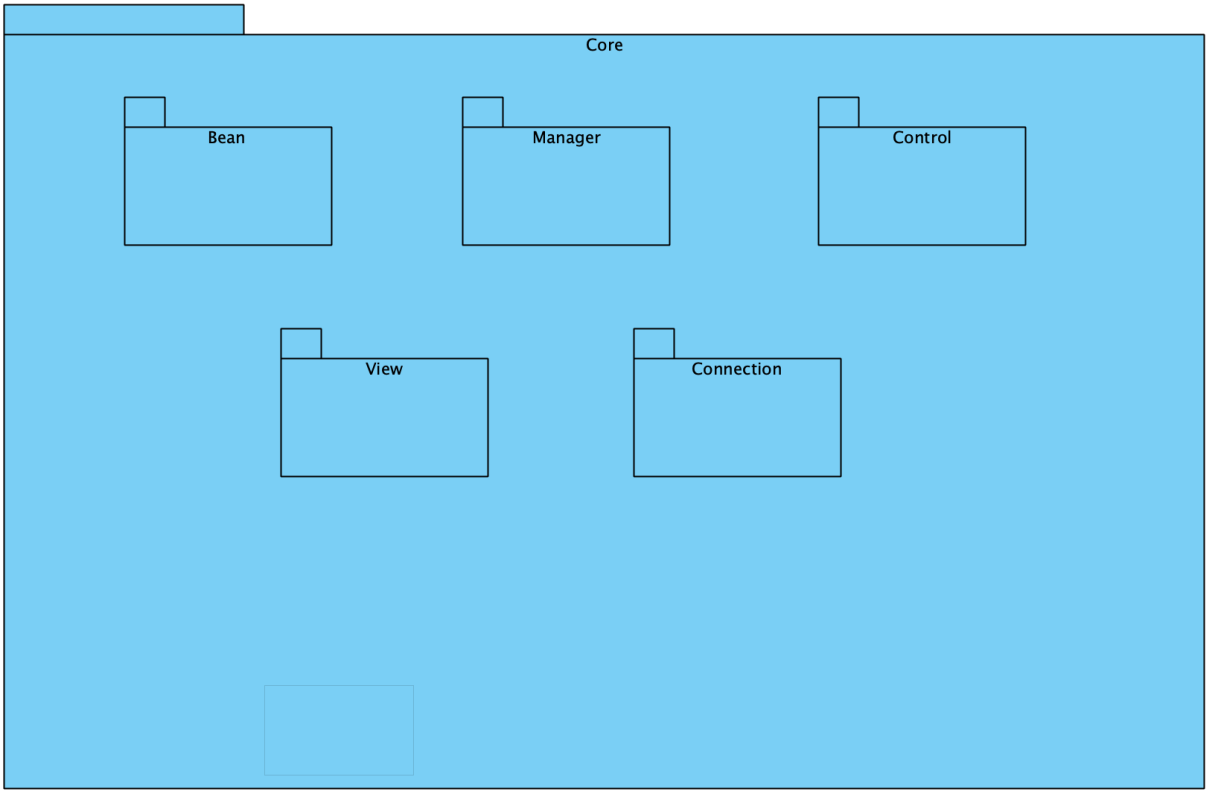
Il package “it” contiene i sottopackage che a loro volta inglobano classi atte alla gestione delle richieste utente. Le classi contenute nel package svolgono il ruolo di gestore logico del sistema.

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: 1. Gestione Utente 2. Gestione Ordine 3. Gestione Prodotto 4. Gestione Commenti 5. Gestione Pagamento 6. Gestione Mail
Storage layer	Ha il compito di memorizzare i dati sensibili del sistema, utilizzando un DBMS, inoltre riceve le varie richieste dall’ Application Logiclayer inoltrandole al DBMS e restituendo i dati richiesti.

2.1 Package Core

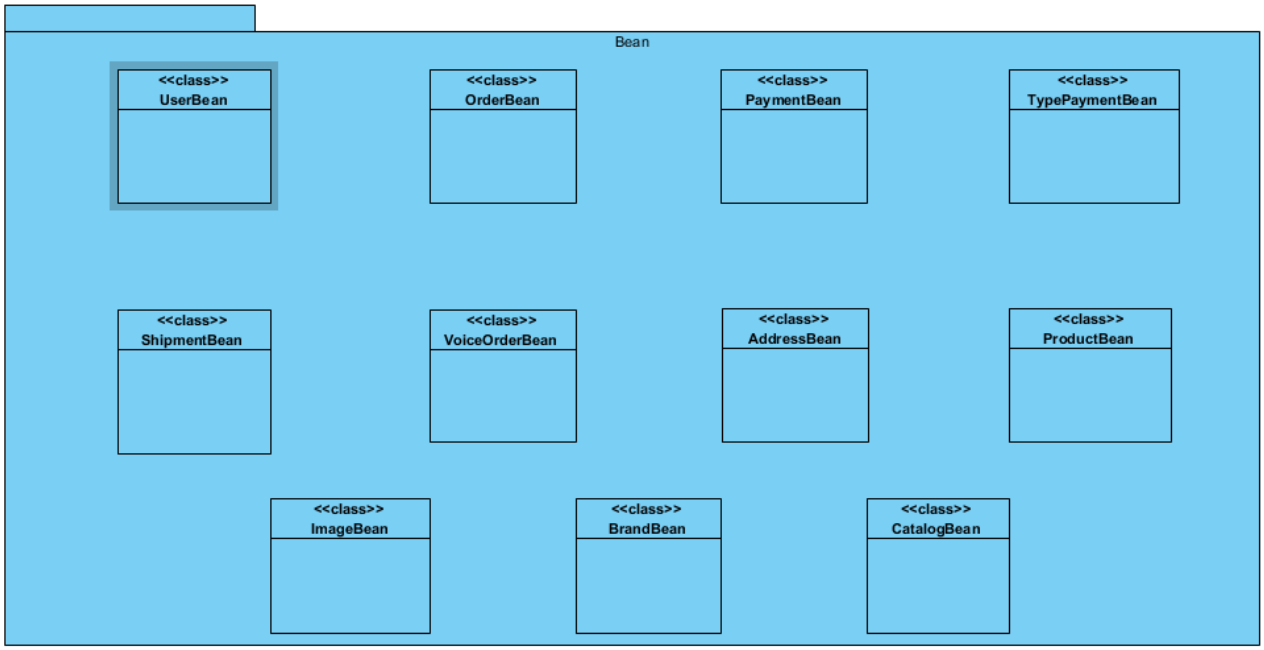
Di seguito è riportata la schematizzazione dei package del progetto.

I package sono stati suddivisi a seconda delle operazioni che eseguono nel sistema. Il Package Core raggruppa tutti i sotto package funzionali.



2.1.1 Package Bean

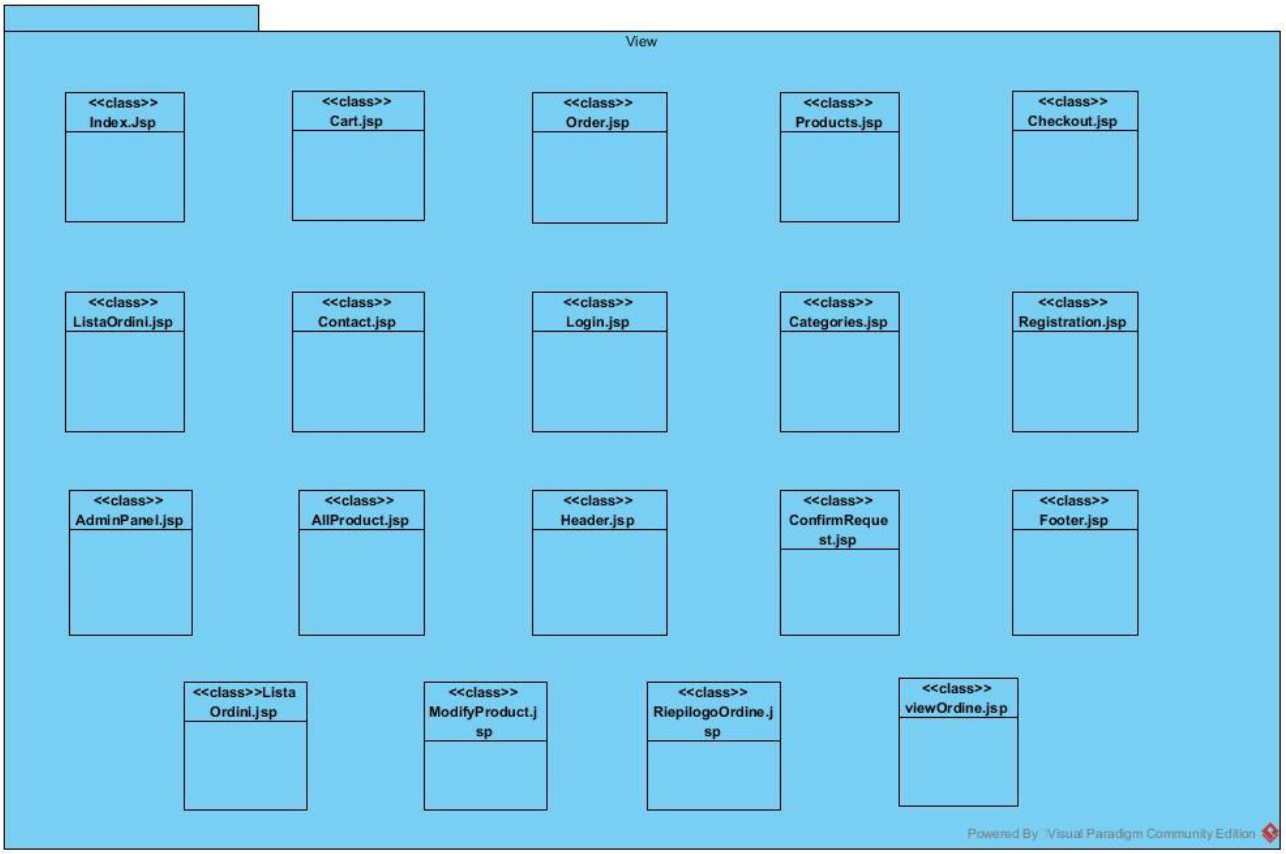
All'interno del package bean sono state riportate le classi java bean



Nome	Descrizione
UserBean.java	Descrive un account del sistema
OrderBean.java	Descrive un ordine del sistema
PaymentBean.java	Descrive uno specifico pagamento di un ordine del sistema
VoiceOrder.java	Descrive uno specifico ordine effettuato
AddressBean.java	Descrive un indirizzo per la spedizione
ProductBean.java	Rappresenta un prodotto in vendita
BrandBean.java	Descrive una marca generica
CatalogBean.java	Descrive un catalogo di prodotti

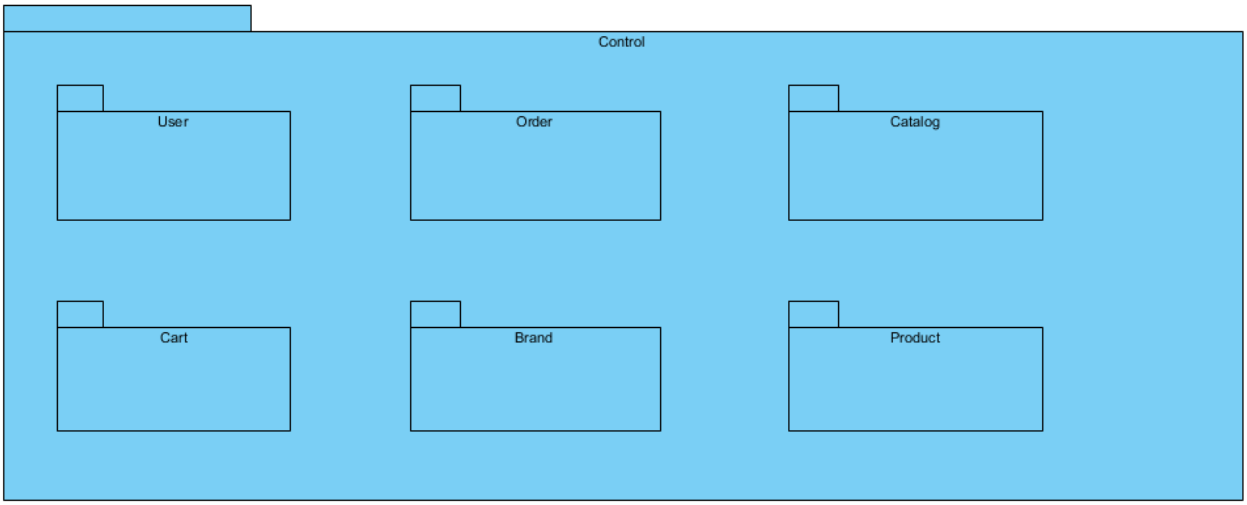
2.1.2 Package View

All'interno del package view sono presenti le pagine che gestiscono la visualizzazione dei contenuti, cioè le pagine jsp.



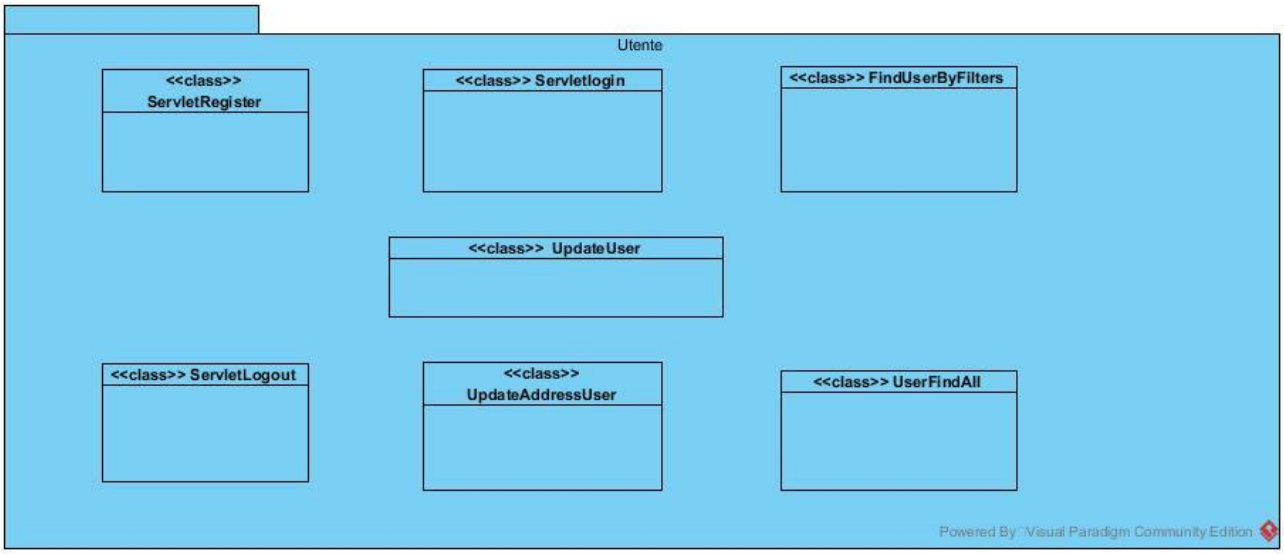
2.1.3 Package Control

All'interno del package control sono presenti tutte le classi riguardanti la logica applicativa del sistema.



2.1.3.1 Package User

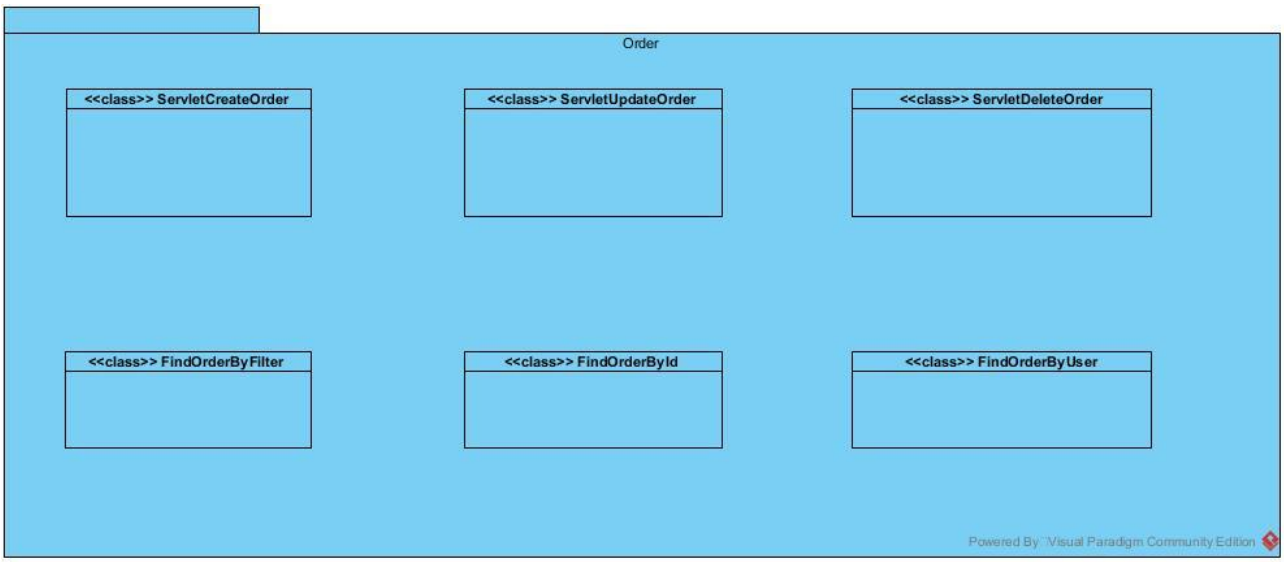
Il package user include le classi servlet che svolgono una funzionalità del sottosistema di gestione dell'utente.



Nome	Descrizione
ServletRegister.java	Controller che permette di effettuare una registrazione
ServletLogin.java	Controller che permette di effettuare un'operazione di login
ServletLogout.java	Controller che permette di effettuare un'operazione di logout
ServletFindUserByFilters.java	Controller che permette di visualizzare gli utenti tramite filtri
ServletUserFindAll.java	Controller che restituisce tutti gli utenti
ServletUpdateAddressUser.java	Controller che permette ad un utente di modificare l'indirizzo inserito
ServletUpdateUser.java	Controller che permette di modificare i dati dell'utente

2.1.3.2 Package Order

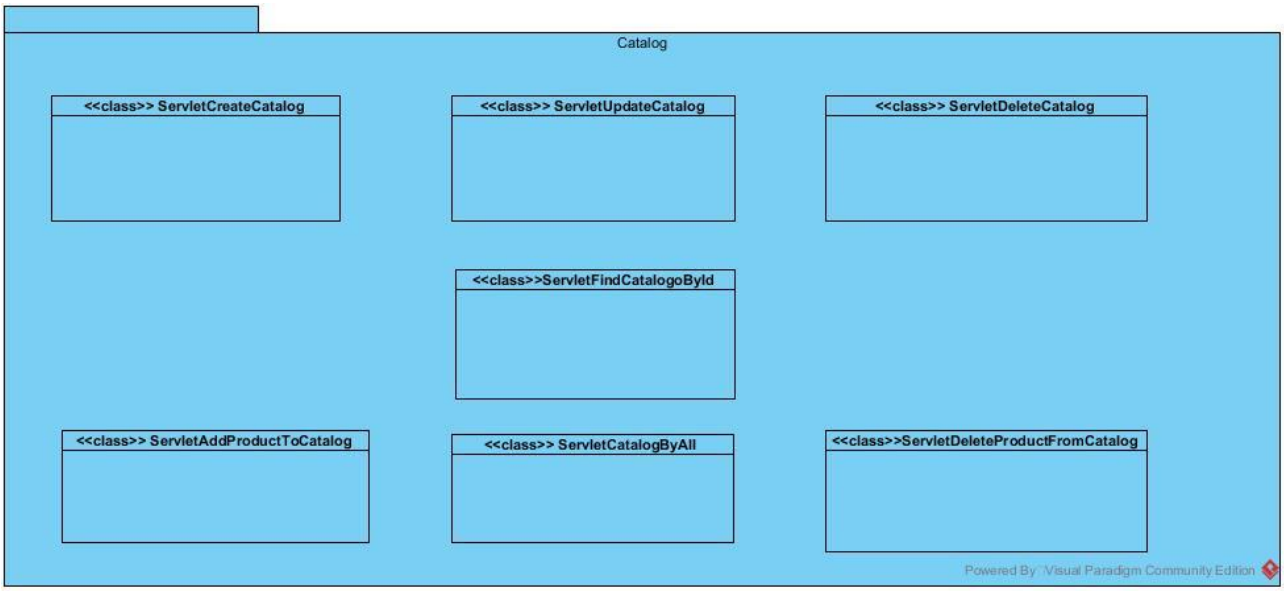
Il package order include le classi servlet che svolgono una funzionalità del sottosistema di gestione di un ordine.



Nome	Descrizione
ServletCreateOrder.java	Controller che permette di creare un ordine di acquisto
ServletUpdateOrder.java	Controller che permette di modificare in alcune specifiche circostanze l'ordine effettuato
ServletDeleteOrder.java	Controller che permette di eliminare un ordine
ServletFindOrderByFilter.java	Controller che permette di filtrare un ordine
ServletFindOrderById.java	Controller che restituisce gli ordini con un determinato Id
ServleFindOrderByUser.java	Controller che restituisce gli ordini di un determinato utente

2.1.3.3 Package Catalog

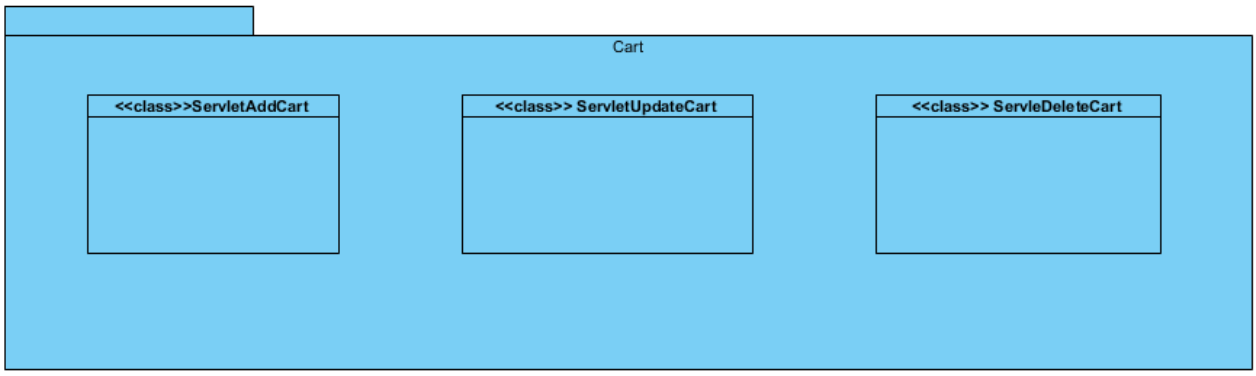
Il package Catalog include le classi servlet che svolgono una funzionalità del sottosistema di gestione di un catalogo.



Nome	Descrizione
ServletCreateCatalog.java	Controller che permette di creare un catalogo di prodotti
ServletUpdateCatalog.java	Controller che permette di modificare un catalogo prodotti
ServletDeleteCatalog.java	Controller che permette di eliminare un catalogo prodotti
ServletAddProductToCatalog.java	Controller che permette di aggiungere prodotti ad un catalogo
ServletDeleteOrderFromCatalog.java	Controller che permette di eliminare un prodotto da un catalogo
ServletFindCatalogById.java	Controller che permette di visualizzare un catalogo in base all'Id
ServletCatalogByAll.java	Controller che permette di visualizzare tutti i cataloghi

2.1.3.4 Package Cart

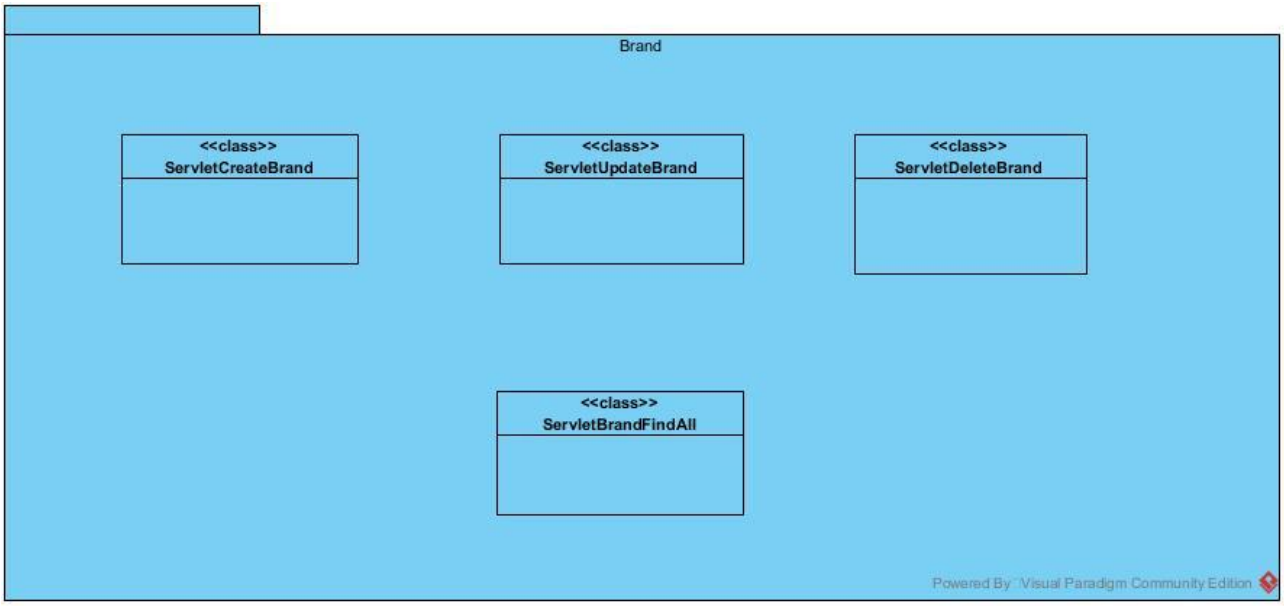
Il package Cart include le classi servlet che svolgono una funzionalità del sottosistema di gestione di una carta di credito.



Nome	Descrizione
ServletAddCart.java	Controller che permette di aggiungere prodotti al carrello
ServletUpdateCart.java	Controller che permette di modificare i prodotti all'interno del carrello
ServletDeleteToCart.java	Controller che permette di eliminare prodotti presenti nel carrello

2.1.3.5 Package Brand

Il package Brand include le classi servlet che svolgono una funzionalità del sottosistema di gestione di un brand, cioè di una marca.

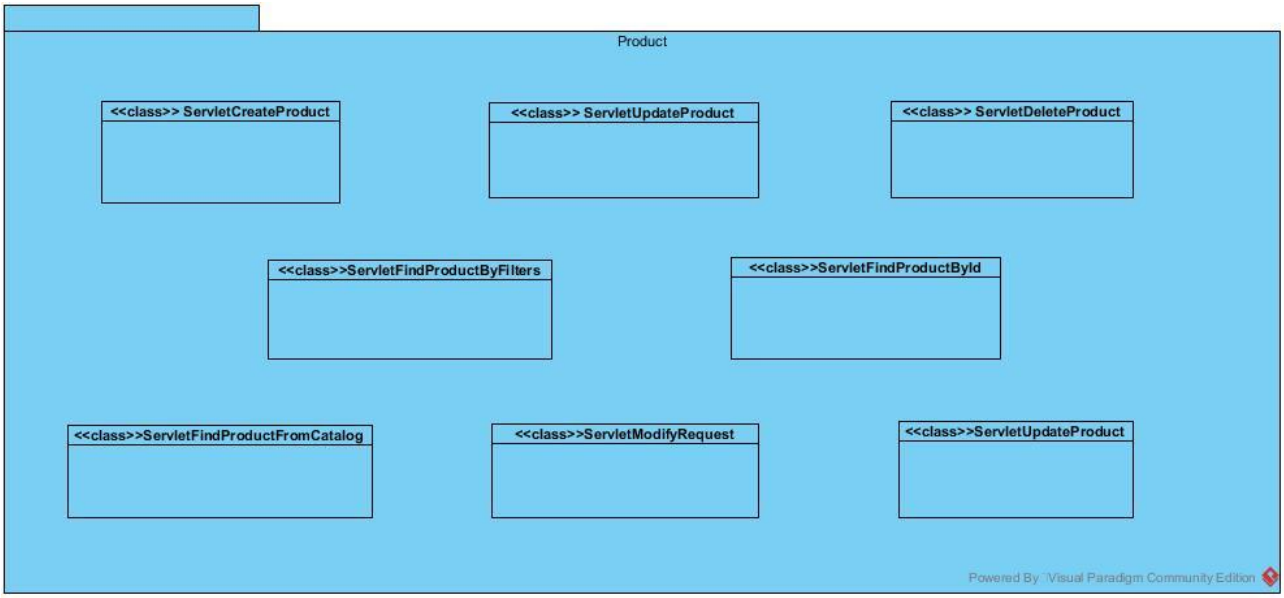


Nome	Descrizione
ServletCreateBrand.java	Controller che permette di aggiungere una marca
ServletUpdateBrand.java	Controller che permette di modificare gli attributi relativi ad una marca
ServletDeleteBrand.java	Controller che permette di eliminare una marca dal sistema
ServletBrandFindAll.java	Controller che permette di visualizzare tutti i

	brand

2.1.3.6 Package Product

Il package Product include le classi servlet che svolgono una funzionalità del sottosistema di gestione di un prodotto.

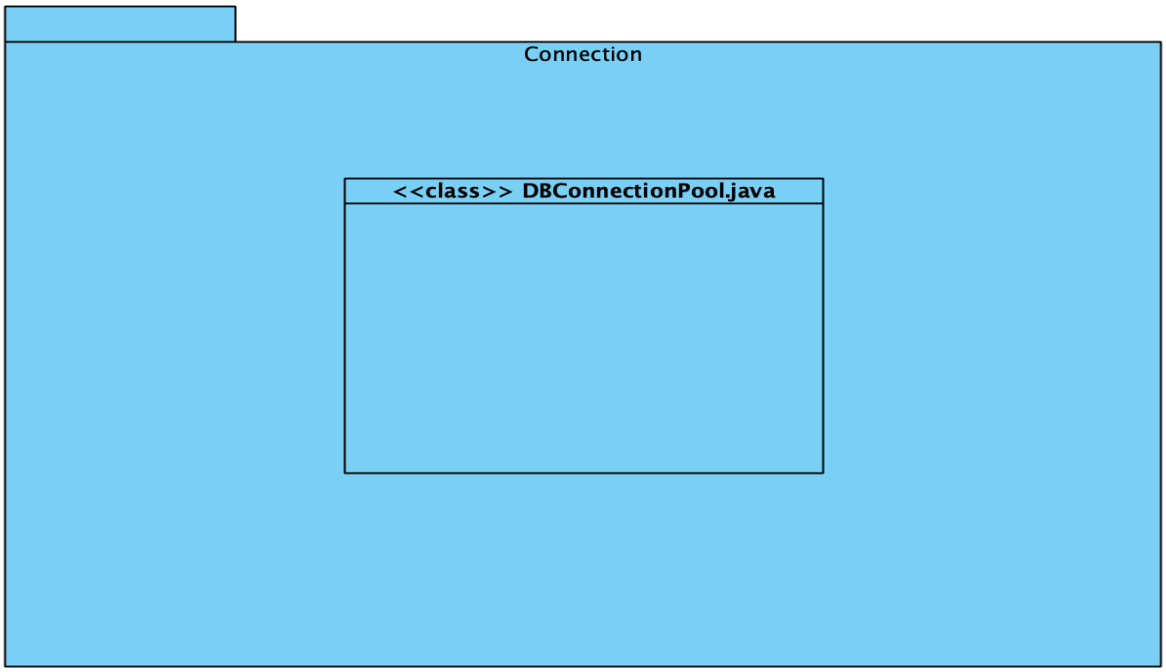


Nome	Descrizione
ServletCreateProduct.java	Controller che permette di creare un prodotto
ServletUpdateProduct.java	Controller che permette di modificare gli attributi relativi ad un prodotto
ServletDeleteProduct.java	Controller che permette di eliminare un prodotto dal sistema
ServletFindProductByFilters.java	Controller che permette di filtrare i prodotti
ServletFindProductById.java	Controller che visualizza i prodotti in base all'Id

ServletFindProductFromCatalog.java	Controller che permette di visualizzare prodotti in base al catalogo
ServletModifyRequest.java	Controller che gestisce la richiesta di modifica di un prodotto
ServletUpdateProduct.java	Controller che permette di modificare un prodotto

2.1.4 Package Connection

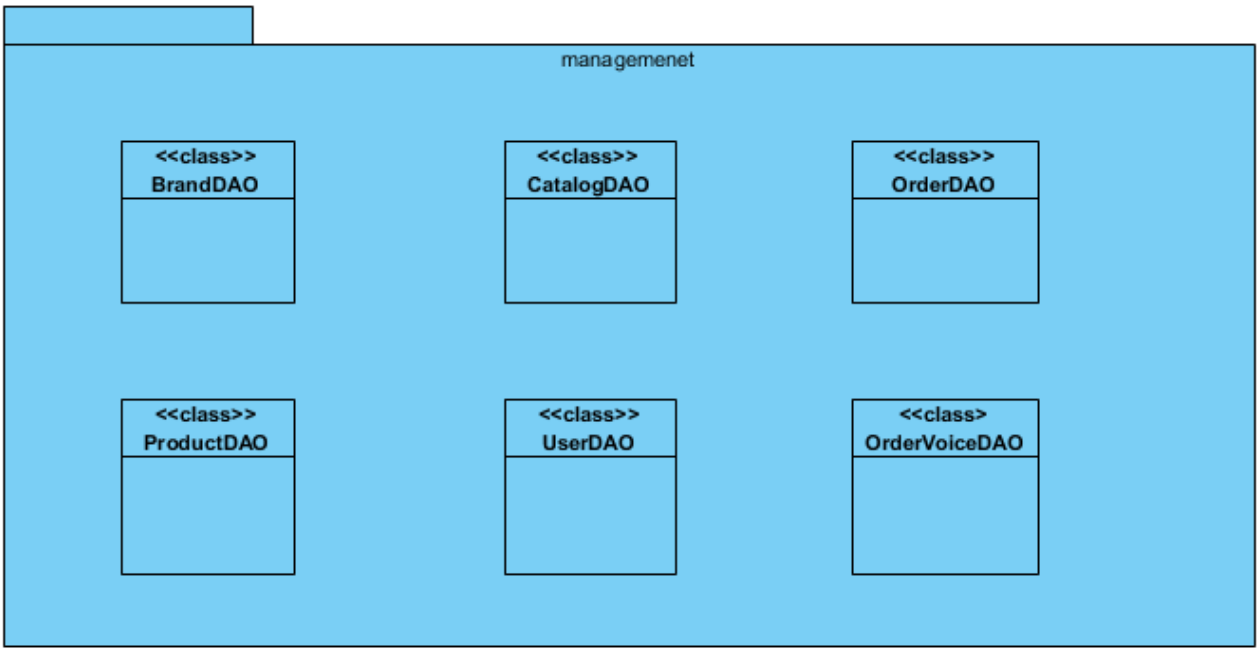
Il package connection include tutte le classi Java adibite alla gestione della connessione con il database del sistema.



Nome	Descrizione
DBConnectionpool.java	Controller che permette di gestire tutti gli aspetti riguardanti la connessione con il database

2.1.5 Package Management

Il package Manager include tutte le classi Java adibite all'accesso dei dati presenti nel sistema.



3. Class Interface

I contratti rappresentano le pre-condizioni, post-condizioni e le invarianti delle classi Manager progettate per compiere funzionalità atomiche legate ai servizi dei sottosistemi specificati nel documento di System Design.

3.1 User Manager

Nome Classe	UserManager
Descrizione	Classe che gestisce alcune funzionalità legate ai servizi del sottosistema di Gestione Utente in maniera persistente.
Pre-condizioni	**context** UserManager::doRetrieveById(id): **pre: id**!=null **context** UserManager::doRetrieveByEmail(email): **pre: email**!=null **context** UserManager::doSave (utente): **pre: utente**!=null **context** UserManager::doDelete (id_utente): **pre: id_utente**!=null **context** UserManager::doUpdate(utente): **pre: utente**!=null

Post-condizioni	**context** UserManager::doRetrieveById(id): **post:** doRetrieveById(id).getId() == id **context** UserManager::doRetrieveByEmail(email): **post:** doRetrieveByEmail(email).getEmail() == email
Invarianti:	

3.2 Brand Manager

Nome Classe	Brand Manager
Descrizione	Classe che gestisce alcune funzionalità legate ai servizi del sottosistema di Gestione del brand in maniera persistente.
Pre-condizioni	**context** BrandManager::doRetrieveByKey(id): **pre: id**!=null **context** BrandManager::doRetrieveByNomel(nome): **pre: nome**!=null **context** BrandManager::doSave (marca): **pre: marca**!=null

	context BrandManager::doDelete (id_marca): **pre: id_marca**!=null **context** BrandManager::doUpdate(marca): **pre: marca**!=null
Post-condizioni	**context** BrandManager::doRetrieveByKey(id): **post:** doRetrieveByKey(id).getId() == id **context** BrandManager::doRetrieveByNome (nome): **post:** doRetrieveByNome(nome).getNome() == nome
Invarianti:	

3.3 Product Manager

Nome Classe	Product Manager
Descrizione	Classe che gestisce alcune funzionalità legate ai servizi del sottosistema di Gestione dei prodotti in maniera persistente.
Pre-condizioni	**context** ProductManager::doRetrieveById(id): **pre: id**!=null

	context ProductManager::doRetrieveByNomel(nome): **pre: nome**!=null **context** ProductManager::doRetrieveByBrand(brand): **pre: brand**!=null **context** ProductManager::doSave (prodotto): **pre: prodotto**!=null **context** ProductManager::doDelete (id_prodotto): **pre: id_prodotto**!=null **context** ProductManager::doUpdate(prodotto): **pre: prodotto**!=null
Post-condizioni	**context** ProductManager::doRetrieveById(id): **post:** doRetrieveById(id).getId() == id **context** ProductManager::doRetrieveByNome(nome): **post:** doRetrieveByNome(nome).getNome() == nome **context** ProductManager::doRetrieveByBrand(brand): **post:** doRetrieveByBrand(brand).getBrand() == brand

Invarianti:	

3.4 Order Manager

Nome Classe	Order Manager
Descrizione	Classe che gestisce alcune funzionalità legate ai servizi del sottosistema di Gestione degli ordini in maniera persistente.
Pre-condizioni	**context** OrderManager::doRetrieveById(id): **pre: id**!=null **context** OrderManager::doRetrieveByStato(stato): **pre: stato**!=null **context** OrderManager::doRetrieveByData_creazione(Data_creazione): **pre: Data_creazione**!=null **context** OrderManager::doRetrieveByProdotto(prodotto): **pre: prodotto**!=null **context** OrderManager::doRetrieveByUtente(utente): **pre: utente**!=null **context** OrderManager::doSave (ordine):

	pre: ordine!=null **context** OrderManager::doDelete (id_ordine): **pre: id_ordine**!=null **context** OrderManager::doUpdate(ordine): **pre: ordine**!=null
Post-condizioni	**context** OrderManager::doRetrieveById(id): **post:** doRetrieveById(id).getId() == id **context** OrderManager::doRetrieveByStato(stato): **post:** doRetrieveByStato(stato).getStato() == stato **context** OrderManager::doRetrieveByData_creazione(Data_creazione): **post:** doRetrieveByData_creazione (Data_creazione).getData_creazione() == Data_creazione **context** OrderManager::doRetrieveByProdotto(prodotto): **post:** doRetrieveByProdotto(prodotto).getProdotto() == prodotto **context** OrderManager::doRetrieveByUtente(utente): **post:** doRetrieveByUtente(utente).getUtente() == utente

Invarianti:	

3.5 Order Voice Manager

Nome Classe	Order voice Manager
Descrizione	Classe che gestisce alcune funzionalità legate ai servizi del sottosistema di Gestione dei prodotti che sono presenti all'interno degli ordini in maniera persistente.
Pre-condizioni	**context** OrderVoiceManager::doRetrieveById(id): **pre: id**!=null **context** OrderVoiceManager::doRetrieveByOrdine(ordine): **pre: ordine**!=null **context** OrderVoiceManager::doSave (voceOrdine): **pre: voceOrdine**!=null **context** OrderVoiceManager::doDelete (id_voce_ordine): **pre: id_voce_ordine**!=null **context** OrderVoiceManager::doUpdate(voceOrdine): **pre: voceOrdine**!=null

Post-condizioni	**context** OrderVoiceManager::doRetrieveById(id): **post:** doRetrieveById(id).getId() == id **context** OrderVoiceManager::doRetrieveByOrdine(ordine): **post:** doRetrieveByOrdine(ordine).getOrdine() == ordine
Invarianti:	

3.6 Catalog Manager

Nome Classe	Catalog Manager
Descrizione	Classe che gestisce alcune funzionalità legate ai servizi del sottosistema di Gestione del catalogo in maniera persistente.
Pre-condizioni	

	context CatalogManager::doRetrieveById(id): **pre: id**!=null **context** CatalogManager::doRetrieveByNome(nome): **pre: nome**!=null **context** CatalogManager::doSave (catalogo): **pre: voceOrdine**!=null **context** CatalogManager::doDelete (id_catalogo): **pre: id_voce_ordine**!=null **context** CatalogManager::doUpdate(catalogo): **pre: voceOrdine**!=null
Post-condizioni	**context** CatalogManager::doRetrieveById(id): **post:** doRetrieveById(id).getId() == id **context** CatalogManager::doRetrieveByNome(nome): **post:** doRetrieveByNome(nome).getNome() == nome
Invarianti:	

3.7 Payment Manager

Nome Classe	Payment Manager
Descrizione	Classe che gestisce alcune funzionalità legate ai servizi del sottosistema di Gestione del pagamento in maniera persistente.
Pre-condizioni	**context** PaymentManager::doRetrieveByOrder(order): **pre: order**!=null **context** PaymentManager::doSave (pagamento): **pre: pagamento**!=null
Post-condizioni	**context** PaymentManager::doRetrieveByOrder(order): **post:** doRetrieveByOrder(order).getOrder() == order
Invarianti:	

4. Glossario

Astrazioni: Concetto che, in informatica, astrae l'implementazione per concentrarsi sulla progettazione strutturale di un oggetto

Modularità: Progettazione di un sistema basato sullo sviluppo delle sue attività in componenti indipendenti collegate tra di loro.

Java Bean: Classi Java che hanno il compito di incapsulare altri oggetti in modo da facilitare la gestione del trasferimento dei dati del sistema.

Java Control: Classi Java che hanno il compito di gestire la logica applicativa del sistema

Java Manager: Classi Java che hanno il compito di gestire l'accesso e le modifiche ai dati persistenti del sistema.

JSP: Java Servlet Page, hanno il compito di gestire l'interfacciamento dell'utente con il sistema

Package: Insieme di classi, pagine o altri tipi di elementi che vengono raggruppati secondo una particolare logica.

Filter: Componenti in grado di filtrare i dati e gestirli secondo alcune condizioni specifiche implementate.