Università degli studi di Salerno
Corso di Laurea in Informatica
Corso di Ingegneria del Software a.a. 2023/2024

Object Design Document

[Versione 1]

Sommario

Informazioni sul documento

Generalità

- **Progetto**: TechHeaven
- **Versione**: [Versione 1]
- **Documento**: Documento di object design
- **Data**: [26/02/2024]

Team Project

Nome Membro	Matricola	Ruolo	Contatti
Dorotea Serrelli	0512113740	Project manager	d.serrelli1@studenti.unisa.it
Raffaella Sabatino	0512115114	Team member	r.sabatino17@studenti.unisa.it

Revision History

Data	Versione	Descrizione	Autore

26/02/2024	0.1	Stesura dell'introduzione al documento ODD e definizione dei trade-off e dei object design goals	Dorotea Serrelli
27/02/2024	0.2	Specifica interfaccia del package Registrazione	Dorotea Serrelli
02/03/2024	0.3	Specifica interfacce dei package Autenticazione, Navigazione, GestioneCarrello, GestioneWishlist	Raffaella Sabatino
		Specifica interfacce GestioneOrdini, GestioneApprovvigionamenti, Pagamento, GestioneCatalogo	Dorotea Serrelli
		Stesura sezione Design Patterns	Tutto il team
01/08/2024	0.4	Aggiunta metodo recuperaWishlist all'interfaccia del package GestioneWishlist	Dorotea Serrelli
17/09/2024	0.5	Aggiornamento interfacce RegistrazioneService e AutenticazioneService	Tutto il team
18/09/2024	0.6	Aggiornamento interfacce GestioneWishlistService, GestioneCarrelloService e NavigazioneService.	Tutto il team
19/09/2024	0.7	Aggiornamento interfaccia GestioneOrdiniService.	Tutto il team
04/11/2024	0.8	Revisione object model	Tutto il team

		Rilascio ODD	DoroteaSerrelli

1. Introduzione

La società TechHeavenSrl è responsabile dall'anno 2000 della gestione del negozio "TechHeaven – Il paradiso digitale", specializzato nella vendita di prodotti elettronici, elettrodomestici, telefonia.
Tale negozio è, attualmente, un punto vendita di riferimento nella zona per lo smercio, la qualità e il prezzo dei prodotti, al punto che soddisfa un grande bacino di utenza, quasi esclusivamente residente nella provincia.
La società intende espandere i confini della propria attività ed ampliare la clientela, avvalendosi di un sistema software che consenta, sotto il profilo soggettivo, una maggiore conoscibilità della società e dell'affidabilità della stessa; sotto il profilo oggettivo, favorisca l'incremento della vendita dei prodotti.

Il sistema software verrà sviluppato per fornire alla clientela informazioni sulla società, sul punto vendita e sui prodotti trattati.
La piattaforma permetterà, infatti, al cliente di registrarsi, in modo da poter visionare i prodotti in vendita ed acquistarli, tenere traccia dello stato degli ordini effettuati presso il negozio online, creare una lista di prodotti desiderati (wishlist).
La piattaforma, inoltre, consentirà l'accesso ai seguenti dipendenti:

- Gestore degli ordini: responsabile del processo di acquisizione, registrazione ed evasione degli ordini dei clienti, nonché dell'elaborazione di richieste di approvvigionamento di prodotti da inoltrare, poi, all'ufficio acquisti.
- Gestore del catalogo: responsabile della presentazione, organizzazione e gestione del catalogo dei prodotti venduti dal negozio.

La piattaforma, quindi, consentirà al gestore degli ordini di visionare gli ordini commissionati dai clienti al negozio e gli ordini che sono stati spediti, preparare un ordine alla spedizione e fare richiesta di approvvigionamento di prodotti mancanti.
Essa, inoltre, permetterà al gestore del catalogo di visionare il catalogo e di poter inserire, cancellare e modificare un prodotto nel catalogo.

In questo documento verranno descritti i compromessi di progettazione degli oggetti effettuati, le linee guida seguite per le interfacce dei sottosistemi - riguardanti la nomenclatura, la documentazione e le convenzioni sui formati -, la decomposizione dei sottosistemi in packages e classi e le interfacce delle classi.

1.1 Object design trade-offs

Nella fase di progettazione degli oggetti del sistema si sono analizzati i seguenti trade-offs:

- **Leggibilità vs Costi**
  Un aspetto importante da prendere in considerazione è la leggibilità del codice: non necessariamente coloro che faranno manutenzione o monitoraggio del sistema saranno i creatori del sistema stesso.
  Pertanto, si vorrà garantire la leggibilità del codice utilizzando commenti e documentazione dei vincoli di implementazione delle interfacce e classi coinvolte nel sistema, anche se ciò comporterà un aumento dei costi e del tempo di sviluppo.

- **Spazio di memoria vs Tempo di risposta**
  Maggiore spazio di memoria significa archiviare più dati, come la cronologia degli ordini, i dettagli dei prodotti e le informazioni sui clienti.
  Privilegiare questo aspetto migliorerebbe l'esperienza utente e la funzionalità del software, ma può anche aumentare i costi e la complessità.
  Un tempo di risposta rapido è fondamentale per un'esperienza utente fluida e per l'efficienza dell'esecuzione delle funzionalità offerte dal sistema.
  L'ottimizzazione del tempo di risposta può richiedere la memorizzazione nella cache di dati e l'utilizzo di algoritmi efficienti.
  Visto che un tempo di risposta rapido è cruciale per la soddisfazione del cliente, la gestione fluida del processo di evasione degli ordini e la competitività del negozio online, si intende privilegiare il tempo di risposta.

- **Sicurezza vs Prestazioni**
  Visto che il cliente sottolinea la necessità dello sviluppo di meccanismi di protezione agli attacchi informatici SQLInjection e Cross-site scripting e di integrità e riservatezza dei dati scambiati tra client e server, si intende privilegiare il requisito di sicurezza, a discapito del livello di prestazioni elevate che potenzialmente raggiungerebbe il sistema.
  Tale decisione farà in modo che verrà garantito il livello di prestazioni nel tempo di risposta delineato nel primo trade-off discusso in questo paragrafo.

A seguire i trade-offs, si intende raggiungere i seguenti obiettivi:

- **Robustezza** : il sistema deve reagire correttamente a situazioni impreviste, attraverso il controllo degli errori e la gestione delle eccezioni.
- **Incapsulamento** : si vogliono nascondere i dettagli implementativi delle classi grazie all'utilizzo delle interfacce, rendendo possibile l'utilizzo di funzionalità offerte da diversi componenti o layer sottoforma di black-box.

1.2 Linee guida per la documentazione dell'interfaccia

Si riportano in questo paragrafo alcune definizioni presenti nel documento:

- **Package**: un meccanismo utilizzato per organizzare e raggruppare insieme moduli, classi, funzioni e altri elementi di codice correlati;
- **Design pattern**: template di soluzioni a problemi ricorrenti impiegati per ottenere riuso e flessibilità;
- **Interfaccia**: insieme di signature delle operazioni offerte dalla classe;
- **Presentation**: nell'architettra three-layer, lo strato che include tutti gli oggetti di confine che interagiscono con l'utente, tra cui finestre, moduli, pagine web, ecc…
- **Application**: nell'architettura three-layer, lo strato che comprende tutti gli oggetti di controllo e di entità, realizzando l'elaborazione, il controllo delle regole e le notifiche richieste dall'applicazione.
- **Storage**: nell'architettura three-layer, lo strato che realizza la memorizzazione, il recupero e le interrogazioni degli oggetti persistenti.
- **lowerCamelCase**: la pratica di scrivere frasi in modo tale che ogni parola o abbreviazione nel mezzo della frase inizi con una lettera maiuscola, senza spazi o punteggiatura intermedi;
- **UpperCamelCase**: la pratica di scrivere frasi in modo tale che ogni parola o abbreviazione inizi con una lettera maiuscola, senza spazi o punteggiatura intermedi;
- **Javadoc**: sistema di documentazione offerto da Java, che viene generato sottoforma di interfaccia in modo da rendere la documentazione accessibile e facilmente leggibile.

1.3 Definizioni, acronimi e abbreviazioni

Per la nomenclatura dei package del sistema, si è adottato il seguente meccanismo:

- **ClasseController**: contiene servlet e classi utili per gestire la logica di controllo del flusso del programma, seguendo i principi dell'architettura three-tier. Questo package funge da intermediario tra la presentazione e la gestione dei dati, assicurando che le richieste degli utenti vengano elaborate correttamente e che le risposte siano generate in modo appropriato.
- **ClasseService:** implementa la logica di business dell'architettura three-tier, contenendo classi dedicate alla realizzazione dei servizi del sistema. Queste classi gestiscono le operazioni e le regole di business, elaborando i dati ricevuti dal controller e fornendo le risposte necessarie per garantire il corretto funzionamento dell'applicazione.

Per la nomenclatura di alcune classi in un package tipo **ClasseService** si è adottato il seguente meccanismo:

- **FileClasseService**: l'interfaccia che dichiara un sottoinsieme di servizi del sistema che offre;
- **FileClasseServiceImpl**: l'implementazione che definisce il sottoinsieme di servizi dichiarati nell'omonimo file **ClasseService** che il sistema offre.

1.4 Riferimenti

Di seguito una lista di riferimenti ad altri documenti utili durante la lettura:

- Problem Statement;
- RAD;
- SDD;
- Database Design Document (DDD).

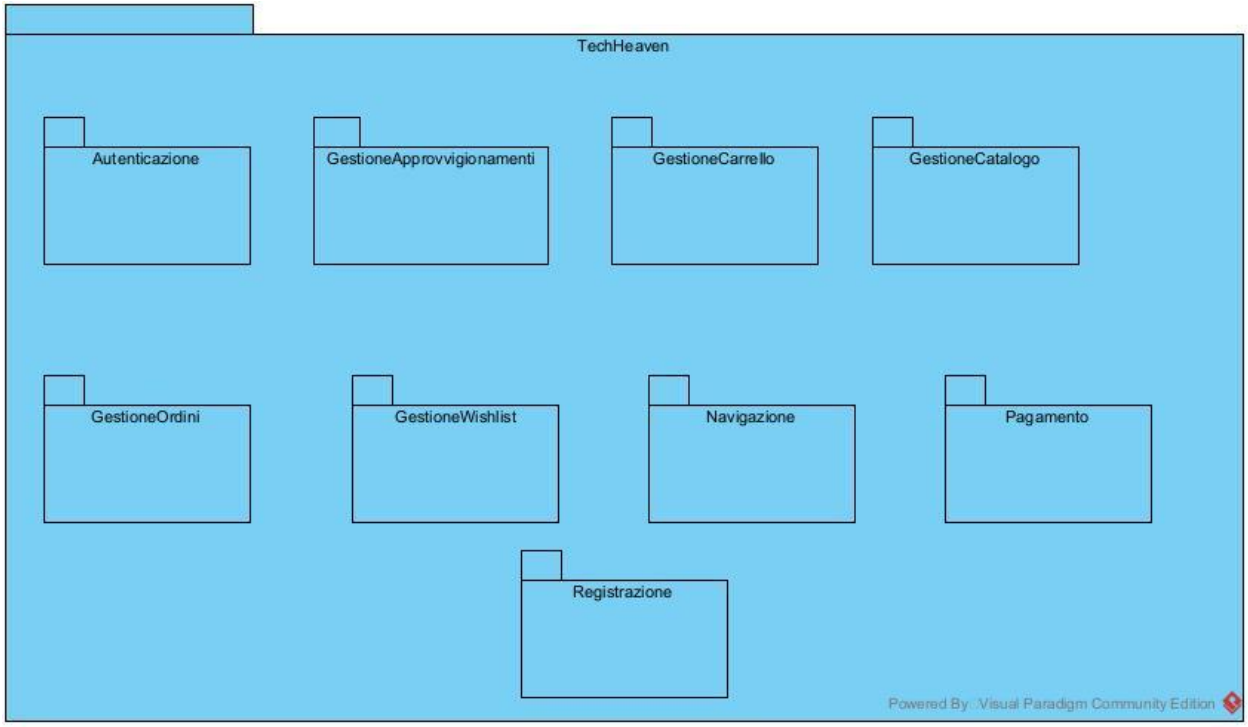
2. Packages

In questa sezione si illustra la suddivisione del sistema software in packages, in base a quanto definito nel documento di System Design.
Tale suddivisione è motivata dalle scelte architetturali definite nel SDD e ricalca la struttura di directory standard per un Web Dynamic Project.

- **.settings** : questa cartella contiene file di configurazione specifici per le preferenze e le impostazioni del progetto.
- **.classpath**: file per gestire le informazioni sulle librerie esterne utilizzate nel progetto.
- **.gitignore**

- **.project**: questo file contiene le informazioni di configurazione del progetto.
- **src :** contiene tutti i file sorgente
    - **main**
        - **java** : contiene le classi Java implementanti gli strati application e storage dell'architettura three-layer del sistema (suddivise nei package **application** e **storage**);
        - **webapp**: contiene i file relativi allo strato Presentation dell'architettura three-layer sistema.
            - **META-INF**: cartella contenente informazioni metadati essenziali per il deployment e la configurazione dell'applicazione web;
            - **WEB-INF**: cartella contenente il codice sorgente compilato, le librerie esterne utilizzate (posizionate in **lib**) e la configurazione principale (presente nel file **web.xml**);
            - **common**: contiene le pagine web del sito e-commerce visibili a tutte le tipologie di utenti (visitatore, cliente, gestore ordine e gestore catalogo);
            - **protected**: contiene le pagine web del sito e-commerce visibili solo agli utenti registrati nel sistema, suddivise (per tipologia di utente) in **cliente**, **gestoreOrdini**, **gestoreCatalogo**;
            - **images**: contiene le immagini impiegate per la realizzazione delle pagine web del sito e-commerce;
            - **scripts**: per migliorare l'interattività del sito e la robustezza, questa cartella contiene file in Javascript che gestiscono funzionalità relative alla corretta compilazione dei form, ricerca di prodotti, ecc…
            - **style**: contiene file CSS per la definizione della grafica delle pagine web, nonché delle pagine web graficamente intuitive e responsive.
    - **test / java** : contiene le classi Java per l'implementazione del testing.

Struttura Package Sistema

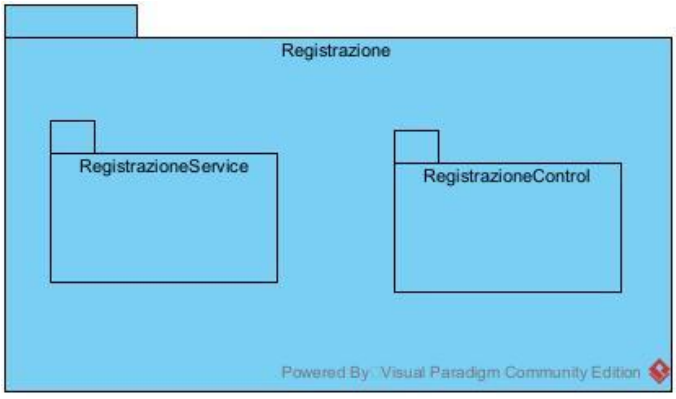


La struttura generale è stata ottenuta a partire dalle seguenti scelte:

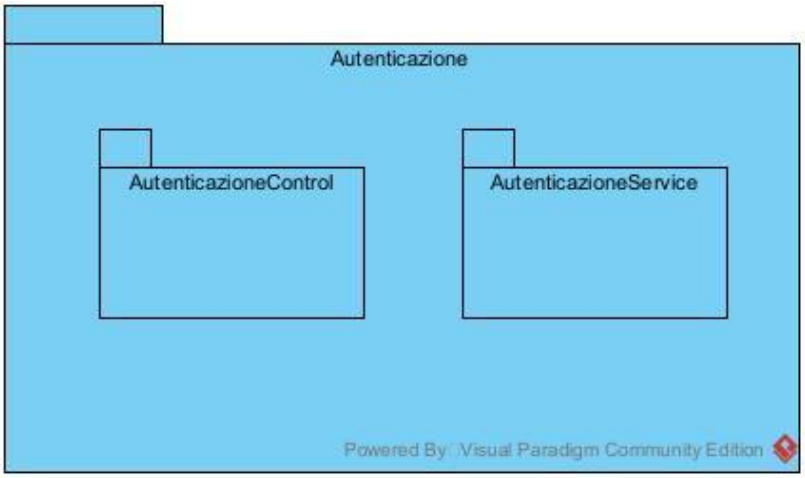
1. Creare un package separato per ogni sottosistema, contenente le classi service e controller del sottosistema, ed eventuali classi di utilità usate unicamente da esso.
2. Creare un package separato per le classi che fanno riferimento al presentation layer, per le classi appartenenti all'application layer e per le classi in storage layer, contenente le classi DAO per l'accesso al database. Tale scelta è stata presa vista l'elevata complessità del database di TechHeaven che prevede numerose relazioni tra le entità. Si è quindi preferito tenere tutto in un package separato e collegato a tutti gli altri package dei sottosistemi.

Per motivi di leggibilità a seguito del numero di classi presente per ogni package tipo ClasseControl e ClasseService, si è deciso di riportare solamente la struttura, in termini di packages, di ciascun sottosistema.

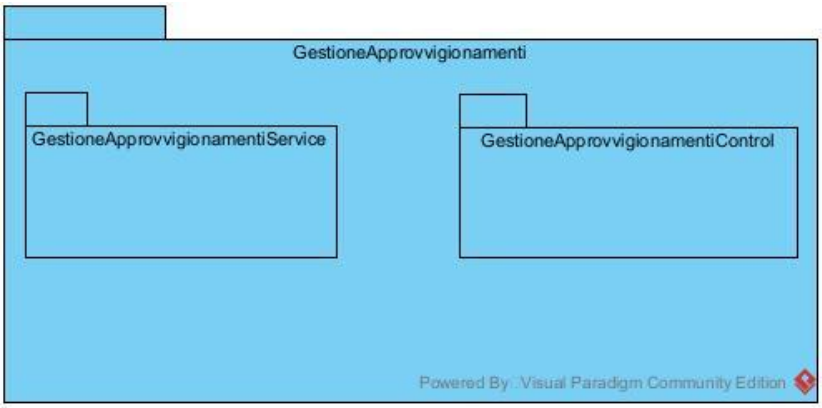
Package Registrazione



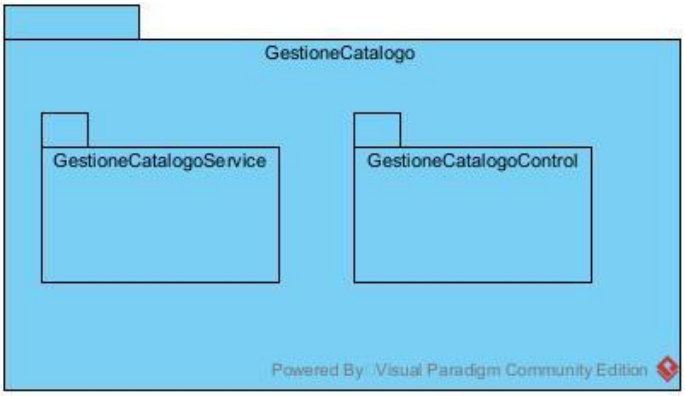
Package Autenticazione



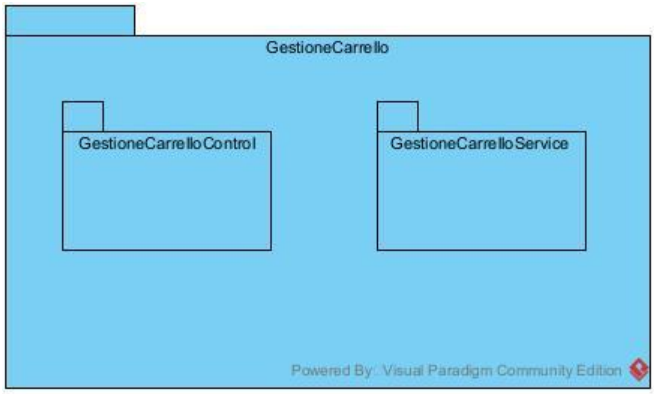
Package GestioneApprovvigionamenti



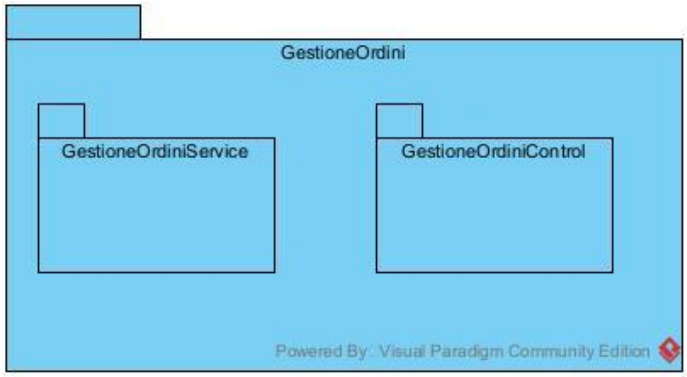
Package GestioneCatalogo



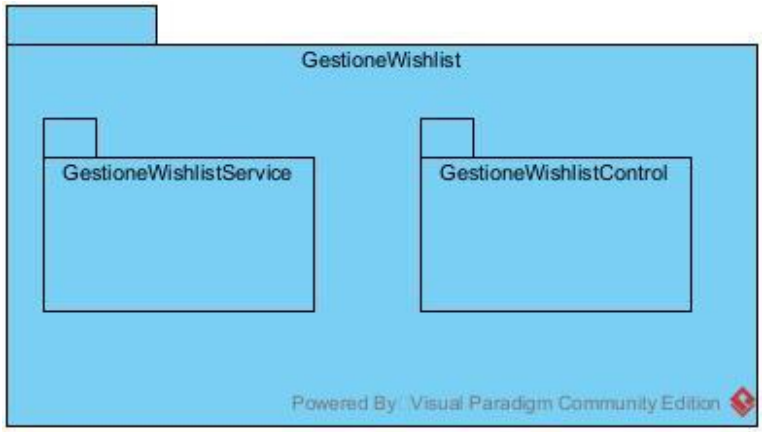
Package GestioneCarrello



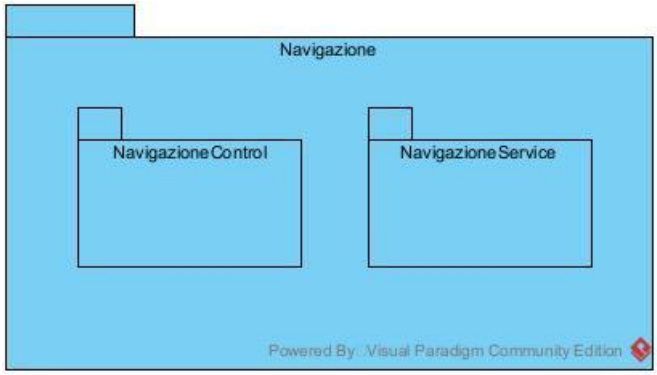
Package GestioneOrdini



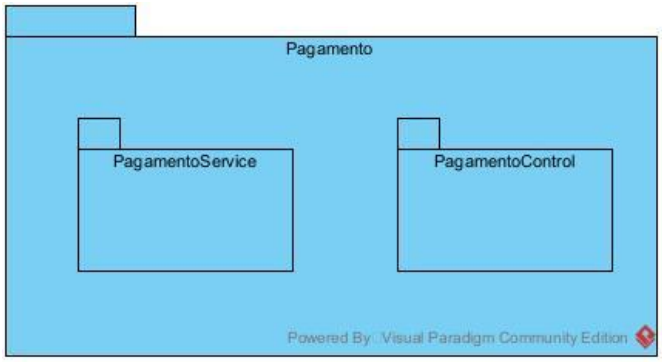
Package GestioneWishlist



Package Navigazione



Package Pagamento



3. Class interfaces

In questa sezione sono illustrate le interfacce di ciascun package, eccetto il package Storage e le classi control del package Application.
Per motivi di leggibilità si è scelto di creare un sito, hostato tramite GitHub pages, contenente la documentazione in JavaDoc del sistema TechHeaven.
In questo modo, chiunque può consultare la documentazione aggiornata dell'intero sistema.
Di seguito, il link al sito in questione:
https://doroteaserrelli.github.io/TechHeavenDocumentation/

Package RegistrazioneService (java.application.Registrazione.RegistrazioneService)

Nome classe	RegistrazioneService
Descrizione	Si occupa di gestire la registrazione di un nuovo utente nel sistema: cliente, gestore degli ordini, gestore del catalogo.

Metodi	+**registraCliente**(username: String, password: String, nome: String, cognome: String, sesso: Sex, email: String, telefono: String, indirizzo: Indirizzo) : ProxyUtente +**registraGestoreOrdini**(username: String, password: String, nome: String, cognome: String, sesso: Sex, email: String, telefono: String, indirizzo: Indirizzo) : ProxyUtente +**registraGestoreCatalogo**(username: String, password: String, nome: String, cognome: String, sesso: Sex, email: String, telefono: String, indirizzo: Indirizzo) : ProxyUtente
Invariante di classe	Nessuno
Nome metodo	+registraCliente(username: String, password: String, email: String, nome: String, cognome: String, sesso: Sex, telefono: String, indirizzo: Indirizzo) : ProxyUtente
Descrizione	Il metodo permette di registrare un nuovo utente nel sistema (con ruolo *Cliente*) con le seguenti informazioni : username, password, nome, cognome, sesso, email, numero di telefono ed indirizzo di spedizione.
Precondizione	**context** RegistrazioneService:: registraCliente(username: String, password: String, nome: String, cognome: String, sesso: Sex, email: String, telefono: String, indirizzo: Indirizzo) **pre**: ObjectUtente.checkValidate(username: String, password: String) AND Cliente.checkValidate(nome: String, cognome: String, sesso: Sex, email: String, telefono: String, indirizzo: Indirizzo)
Postcondizione	**context** RegistrazioneService::registraCliente(username: String, password: String, nome: String, cognome: String, sesso: Sex, email: String, telefono: String, indirizzo: Indirizzo) **post**: Utente.ifExists(username)
Nome metodo	+registraGestoreOrdini(username: String, password: String, nome: String, cognome: String, sesso: Sex, email: String, telefono: String, indirizzo: Indirizzo): ProxyUtente
Descrizione	Il metodo permette di registrare un nuovo utente nel sistema (con ruoli *Gestore Ordini* e *Cliente*) con le seguenti informazioni : username, password, nome, cognome, sesso, email, numero di telefono ed indirizzo di spedizione.

Precondizione	**context** RegistrazioneService:: registraGestoreOrdini(username: String, password: String, nome: String, cognome: String, sesso: Sex, email: String, telefono: String, indirizzo: Indirizzo) **pre**: ObjectUtente.checkValidate(username: String, password: String) AND Cliente.checkValidate(nome: String, cognome: String, sesso: Sex, email: String, telefono: String, indirizzo: Indirizzo)
Postcondizione	**context** RegistrazioneService::registraGestoreOrdini(username: String, password: String, nome: String, cognome: String, sesso: Sex, email: String, telefono: String, indirizzo: Indirizzo) **post**: Utente.ifExists(username)
Nome metodo	+registraGestoreCatalogo(username: String, password: String, nome: String, cognome: String, sesso: Sex, email: String, telefono: String, indirizzo: Indirizzo) : ProxyUtente
Descrizione	Il metodo permette di registrare un nuovo utente nel sistema (con ruoli *Gestore Catalogo* e *Cliente*) con le seguenti informazioni : username, password, nome, cognome, sesso, email, numero di telefono ed indirizzo di spedizione.
Precondizione	**context** RegistrazioneService:: registraGestoreCatalogo(username: String, password: String, nome: String, cognome: String, sesso: Sex, email: String, telefono: String, indirizzo: Indirizzo) **pre**: ObjectUtente.checkValidate(username: String, password: String) AND Cliente.checkValidate(nome: String, cognome: String, sesso: Sex, email: String, telefono: String, indirizzo: Indirizzo)
Postcondizione	**context** RegistrazioneService::registraGestoreCatalogo(username: String, password: String, nome: String, cognome: String, sesso: Sex, email: String, telefono: String, indirizzo: Indirizzo) **post**: Utente.ifExists(username)

Package AutenticazioneService
(java.application.Autenticazione.AutenticazioneService)

Nome classe	**AutenticazioneService**

Descrizione	Offre servizi relativi ad un utente autenticato con il ruolo di cliente: autenticazione al sistema, reimpostazione della password e modifica delle informazioni presenti nel suo profilo personale (telefono, email e rubrica indirizzi)
Metodi	+**login**(username: String, password: String) : ProxyUtente +**resetPassword**(username: String, email: String, newPassword: String) : void +**aggiornaProfilo**(user: ProxyUtente, information : String, updatedData: String): ProxyUtente +**aggiornaRubricaIndirizzi**(user: ProxyUtente, information: String, updatedData: Indirizzo): ProxyUtente
Invariante di classe	Nessuno
Nome metodo	+login(username: String, password: String):ProxyUtente
Descrizione	Il metodo permette di autenticare un utente nel sistema.
Precondizione	Nessuna
Postcondizione	**context** AutenticazioneService::login(username: String, password : String) **post**: isLogged(username)
Nome metodo	+aggiornaProfilo(user: ProxyUtente, information: String, updatedData: String): ProxyUtente
Descrizione	Il metodo permette di modificare il numero di telefono o l'indirizzo e-mail dell'utente. Il parametro *information* rappresenta l'informazione da modificare mentre il parametro *updatedData* indica la nuova informazione da memorizzare.
Precondizione	**context** AutenticazioneService:: aggiornaProfilo(user : ProxyUtente, information : String, updatedData : String) **pre**: isLogged(user.getUsername()) AND (information == 'TELEFONO' OR information == 'EMAIL') AND updatedData != NULL
Postcondizione	**context** AutenticazioneService:: aggiornaProfilo(user: ProxyUtente, information: String, updatedData: String) **post**: self.getEmail() == updatedData OR self.getTelefono() == updatedData
Nome metodo	+aggiornaRubricaIndirizzi(user : ProxyUtente, information: String, updatedData: Indirizzo): ProxyUtente
Descrizione	Il metodo effettua l'inserimento/ la rimozione/l'aggiornamento di un indirizzo di spedizione nella rubrica degli indirizzi personali dell'utente.

Precondizione (aggiunta indirizzo)	**context** AutenticazioneService:: aggiornaRubricaIndirizzi(user: ProxyUtente, information: String, updatedData: Indirizzo) **pre**: isLogged(user.getUsername()) AND !(user.getIndirizzi() -> includes(updatedData)) AND information == 'AGGIUNGERE-INDIRIZZO'
Postcondizione (aggiunta indirizzo)	**context** AutenticazioneService:: aggiornaRubricaIndirizzi(user: ProxyUtente, information: String, updatedData: Indirizzo) **post**: user.getIndirizzi() -> includes(updatedData)
Precondizione (eliminazione indirizzo)	**context** AutenticazioneService:: aggiornaRubricaIndirizzi(user: ProxyUtente, information: String, updatedData: Indirizzo) **pre**: isLogged(user.getUsername()) AND user.getIndirizzi() -> includes(updatedData) AND information == 'RIMUOVERE-INDIRIZZO'
Postcondizione (eliminazione indirizzo)	**context** AutenticazioneService:: aggiornaRubricaIndirizzi(user: ProxyUtente, information: String, updatedData: Indirizzo) **post**: !(user.getIndirizzi() -> includes(updatedData))
Precondizione (aggiornamento indirizzo)	**context** AutenticazioneService:: aggiornaRubricaIndirizzi(user: ProxyUtente, information: String, updatedData: Indirizzo) **pre**: isLogged(user.getUsername()) AND user.getIndirizzi() -> includes(updatedData) AND information == 'AGGIORNARE-INDIRIZZO'
Postcondizione (aggiornamento indirizzo)	**context** AutenticazioneService:: aggiornaRubricaIndirizzi(user: ProxyUtente, information: String, updatedData: Indirizzo) **post**: user.getIndirizzi() -> includes(updatedData) AND user.getIndirizzi()->size = user@pre.getIndirizzi()->size AND user.getIndirizzi() != user@pre.getIndirizzi()
Nome metodo	+resetPassword(username: String, email: String, newPassword: String) : void
Descrizione	Il metodo permette di reimpostare la password di un utente richiedendo le credenziali *username* e *email*. La nuova password è rappresentata da *newPassword*.
Precondizione	**context** AutenticazioneService:: resetPassword(username: String, email: String, newPassword: String) **pre**: Utente.checkPassword(newPassword)

Postcondizione	**context** AutenticazioneService:: resetPassword(username: String, email: String, newPassword: String) **post**: (UtenteDAO. retrieveUserByUsername(username)).getPassword() == newPassword

Package NavigazioneService (java.application.Navigazione.NavigazioneService)

Nome classe	**NavigazioneService**
Descrizione	Si occupa di gestire le operazioni relative alla navigazione del negozio online: visualizzazione delle specifiche di un prodotto, ricerca dei prodotti mediante menu di navigazione e barra di ricerca, nonché della pagina dei risultati ottenuti dalla ricerca.
Metodi	+**visualizzaProdotto**(ProxyProdotto prod) : Prodotto +**ricercaProdottoMenu**(String category) : Collection(ProxyProdotto) +**ricercaProdottoBar**(String keyword) : Collection(ProxyProdotto)
Invariante di classe	Nessuno
Nome metodo	+visualizzaProdotto(ProxyProdotto prod) : Prodotto
Descrizione	Il metodo permette di visualizzare le specifiche di un prodotto *prod* selezionato dal cliente. Si suppone che *prod* sia un prodotto presente nel catalogo del negozio.
Precondizione	Nessuna
Postcondizione	Nessuna
Nome metodo	+ricercaProdottoMenu(String category) : Collection(ProxyProdotto)
Descrizione	Il metodo restituisce l'insieme dei prodotti nel catalogo che appartengono alla categoria *category* selezionata.
Precondizione	Nessuna
Postcondizione	**context** NavigazioneService:: ricercaProdottoMenu(category: String) **post**: result->forAll(element | element.getCategoria() == category)
Nome metodo	+ricercaProdottoBar(String keyword) : Collection(ProxyProdotto)
Descrizione	Il metodo restituisce l'insieme dei prodotti nel catalogo che hanno nel proprio nome, nella propria descrizione

	(dettagliata o di presentazione), nel proprio modello oppure nella propria marca la parola *keyword*.
Precondizione	Nessuna
Postcondizione	**context** NavigazioneService:: ricercaProdottoBar(keyword: String) **post**: result->((exists(element | element.getNome() == keyword)) OR (exists(element | element.getTopDescrizione() == keyword)) OR (exists(element | element.getDettagli() == keyword))) OR (exists(element | element.getModello() == keyword)) OR (exists(element | element.getMarca() == keyword))

Package GestioneCarrelloService (java.application.GestioneCarrello.GestioneCarrelloService)

Nome classe	**GestioneCarrelloService**
Descrizione	Si occupa di gestire le operazioni relative al carrello virtuale: visualizzazione del contenuto del carrello, inserimento prodotti nel carrello, eliminazione prodotti dal carrello, aumento delle quantità di un prodotto del carrello, decremento delle quantità di un prodotto del carrello; inoltre, consente di svuotare il carrello.
Metodi	+**visualizzaCarrello**(cart: Carrello) : Collection(ItemCarrello) +**aggiungiAlCarrello**(cart: Carrello, item: ItemCarrello) : Carrello +**rimuoviDalCarrello**(cart: Carrello, item: ItemCarrello) : Carrello +**aumentaQuantitaNelCarrello**(cart: Carrello, item:ItemCarrello, quantity:int) : Carrello +**decrementaQuantitaNelCarrello**(cart: Carrello, item:ItemCarrello, quantity: int) : Carrello +**svuotaCarrello**(cart:Carrello) : Carrello
Invariante di classe	Nessuna
Nome metodo	+visualizzaCarrello(cart: Carrello) : Collection(ItemCarrello)
Descrizione	Il metodo fornisce i prodotti presenti nel carrello virtuale.
Precondizione	Nessuna

Postcondizione	Nessuna
Nome metodo	+aggiungiAlCarrello(cart: Carrello, item: ItemCarrello) : Carrello
Descrizione	Il metodo aggiunge un prodotto *item* (di quantità 1) al carrello *cart*.
Precondizione	**context** Carrello:: aggiungiAlCarrello(cart: Carrello, item: ItemCarrello) **pre**: !(cart.prodotti -> includes(item))
Postcondizione	**context** Carrello:: aggiungiAlCarrello(cart: Carrello, item: ItemCarrello) **post**: cart.prodotti -> includes(item)
Nome metodo	+rimuoviDalCarrello(cart: Carrello, item: ItemCarrello) : Carrello
Descrizione	Il metodo rimuove il prodotto *item* dal carrello *cart*.
Precondizione	**context** Carrello:: rimuoviDalCarrello(cart: Carrello, item: ItemCarrello) **pre**: cart.prodotti -> includes(item)
Postcondizione	**context** Carrello:: rimuoviDalCarrello(cart: Carrello, item: ItemCarrello) **post**: !(cart.prodotti -> includes(item))
Nome metodo	+aumentaQuantitaNelCarrello(cart: Carrello, item:ItemCarrello, quantity:int) : Carrello
Descrizione	Il metodo aumenta la quantità di un prodotto *item* del carrello *cart*, impostandola a *quantity*. Si assume che *stock_quantity* è la quantità del prodotto in magazzino.
Precondizione	**context** Carrello:: aumentaQuantitaNelCarrello(cart: Carrello, item:ItemCarrello, quantity:int) **pre**: cart.prodotti -> includes(item) AND (cart.prodotti->select(p | p = item).getQuantità() < quantity) AND quantity > 0 AND quantity <= stock_quantity
Postcondizione	**context** Carrello:: aumentaQuantitaNelCarrello(cart: Carrello, item:ItemCarrello, quantity:int) **post**: (cart.prodotti->select(p | p = item).getQuantità() == quantity)
Nome metodo	+decrementaQuantitaNelCarrello(cart: Carrello, item:ItemCarrello, quantity: int) : Carrello
Descrizione	Il metodo diminuisce la quantità di un prodotto *item* del carrello *cart*, impostandola a *quantity*. Si assume

	che *stock_quantity* è la quantità del prodotto in magazzino.
Precondizione	**context** Carrello:: decrementaQuantitaNelCarrello(cart: Carrello, item:ItemCarrello, quantity: int) **pre**: cart.prodotti -> includes(item) AND (cart.prodotti->select(p | p = item).getQuantità() > quantity) AND quantity > 0 AND quantity <= stock_quantity
Postcondizione	**context** Carrello:: decrementaQuantitaNelCarrello(cart: Carrello, item:ItemCarrello, quantity: int) **post**: (cart.prodotti->select(p | p = item).getQuantità() == quantity)
Nome metodo	+svuotaCarrello(cart:Carrello) : Carrello
Descrizione	Il metodo elimina tutti gli articoli presenti nel carrello *cart*.
Precondizione	**context** Carrello:: svuotaCarrello(cart:Carrello) **pre**: (cart.prodotti->size) > 0
Postcondizione	**context** Carrello:: svuotaCarrello(cart:Carrello) **post**: (cart.prodotti->size) == 0

Package GestioneWishlistService (java.application.GestioneWishlist.GestioneWishlistService)

Nome classe	**GestioneWishlistService**
Descrizione	Offre le operazioni relative alla gestione della wishlist: visualizzazione, inserimento prodotti nella wishlist, eliminazione prodotti dalla wishlist.
Metodi	+**recuperaWishlist**(user: ProxyUtente, id: int): Wishlist +**visualizzaWishlist**(wishes: Wishlist, user: ProxyUtente) : Collection(ProxyProdotto) +**aggiungiProdottoInWishlist**(wishes: Wishlist, prod:ProxyProdotto, user: ProxyUtente) : Wishlist +**rimuoviProdottoDaWishlist**(wishes: Wishlist, user: ProxyUtente, prod: ProxyProdotto) : Wishlist
Invariante di classe	Nessuna
Nome metodo	+recuperaWishlist(user: ProxyUtente, id : int): Wishlist

Descrizione	Il metodo fornisce la wishlist dell'utente con identificativo *id*.
Precondizione	**context** Wishlist:: recuperaWishlist(user: ProxyUtente, id : int) **pre**: isLogged(user.getUsername())
Postcondizione	**context** Wishlist:: recuperaWishlist(user: ProxyUtente, id:int) **post**: wishlist.getId() == id
Nome metodo	+visualizzaWishlist(wishes: Wishlist, user: ProxyUtente) : Collection(ProxyProdotto)
Descrizione	Il metodo fornisce i prodotti presenti nella wishlist dell'utente.
Precondizione	**context** Wishlist:: visualizzaWishlist(wishes: Wishlist, user:Utente) **pre**: isLogged(user.getUsername())
Postcondizione	Nessuna
Nome metodo	+aggiungiProdottoInWishlist(wishes: Wishlist, prod:ProxyProdotto, user: ProxyUtente) : Wishlist
Descrizione	Il metodo aggiunge un prodotto *prod* nella wishlist *wishes*.
Precondizione	**context** Wishlist:: aggiungiProdottoInWishlist(wishes: Wishlist, prod:ProxyProdotto, user: ProxyUtente) **pre**: !(wishes.prodotti -> includes(prod)) AND isLogged(user.getUsername())
Postcondizione	**context** Wishlist:: aggiungiProdottoInWishlist(wishes: Wishlist, prod:ProxyProdotto, user: ProxyUtente) **post**: wishes.prodotti -> includes(prod)
Nome metodo	+rimuoviProdottoDaWishlist(wishes: Wishlist, user: ProxyUtente, prod: ProxyProdotto) : Wishlist
Descrizione	Il metodo rimuove il prodotto *prod* dalla wishlist *wishes*.
Precondizione	**context** Wishlist:: rimuoviProdottoDaWishlist(wishes: Wishlist, user: ProxyUtente, prod: ProxyProdotto) **pre**: wishes.prodotti -> includes(prod) AND isLogged(user.getUsername())
Postcondizione	**context** Wishlist:: rimuoviProdottoDaWishlist(wishes: Wishlist, user: ProxyUtente, prod: ProxyProdotto) **post**: !(wishes.prodotti -> includes(prod))

Package GestioneOrdiniService (java.application.GestioneOrdini.GestioneOrdiniService)

Nome classe	GestioneOrdiniService
Descrizione	Si occupa di effettuare le operazioni relative alla gestione degli ordini: creazione di un ordine, visualizzazione ordini evasi, visualizzazione ordini commissionati da evadere, preparazione di un ordine alla spedizione.
Metodi	+**visualizzaOrdiniEvasi**(user: Utente) : Collection(ProxyOrdine) +**visualizzaOrdiniDaEvadere**(user: Utente): Collection(ProxyOrdine) + **commissionaOrdine**(cart: Carrello, order : Ordine, payment : Pagamento, user : ProxyUtente) : Carrello +**preparazioneSpedizioneOrdine**(order: Ordine, user: Utente, report : ReportSpedizione) : Ordine
Invariante di classe	Nessuna
Nome metodo	+visualizzaOrdiniEvasi(user: Utente) : Collection(ProxyOrdine)
Descrizione	Il metodo fornisce gli ordini che sono stati commissionati al negozio online e che sono stati evasi correttamente.
Precondizione	**context** GestioneOrdiniService:: visualizzaOrdiniEvasi(user: Utente) **pre**: isLoggedAsOrderManager(user)
Postcondizione	**context** GestioneOrdiniService :: visualizzaOrdiniEvasi(user:Utente) **post**: result -> forAll(o : Ordine | o.getStato() == 'Spedito')
Nome metodo	+visualizzaOrdiniDaEvadere(user: Utente): Collection(ProxyOrdine)
Descrizione	Il metodo fornisce gli ordini che sono stati commissionati al negozio online e che devono essere ancora spediti.
Precondizione	**context** GestioneOrdiniService:: visualizzaOrdiniDaEvadere(user:Utente) **pre**: isLoggedAsOrderManager(user)
Postcondizione	**context** GestioneOrdiniService :: visualizzaOrdiniDaEvadere(user:Utente) **post**:

	result -> forAll(o : Ordine | o.getStato() == 'Richiesta effettuata') OR result->forAll(o : Ordine | o.getStato() == 'Preparazione incompleta')
Nome metodo	+ commissionaOrdine(cart: Carrello, order : Ordine, payment : Pagamento, user : ProxyUtente) : Carrello
Descrizione	Il metodo permette l'acquisto dei prodotti nel carrello *cart* da parte dell'utente *user*, memorizzati nell'ordine *order* , a seguito della procedura di pagamento *payment.*
Precondizione	**context** GestioneOrdiniService:: commissionaOrdine(cart: Carrello, order : Ordine, payment : Pagamento, user : ProxyUtente) **pre**: isLogged(user.getUsername()) AND cart.prodotti -> size() > 0 AND payment.ordine == order AND order.prodotti == cart.prodotti
Postcondizione	**context** GestioneOrdiniService:: commissionaOrdine(cart: Carrello, order : Ordine, payment : Pagamento, user : ProxyUtente) **post**: (cart.prodotti -> size() == 0) AND order.getStato() == "Richiesta_effettuata" AND isSaved(order)
Nome metodo	+preparazioneSpedizioneOrdine(order: Ordine, user: Utente, report : ReportSpedizione) : Ordine
Descrizione	Il metodo permette di preparare un ordine *order* alla spedizione, tenedo traccia del suo report *report*.
Precondizione	**context** GestioneOrdiniService:: preparazioneSpedizioneOrdine(order: Ordine, user: Utente, report : ReportSpedizione) **pre**: (order.getStato() == 'Richiesta effettuata' OR order.getStato() == 'Preparazione incompleta') AND report.ordine == order AND isLoggedAsOrderManager(user)
Postcondizione	**context** GestioneOrdiniService:: preparazioneSpedizioneOrdine(order: Ordine, user: Utente, report : ReportSpedizione) **post**: order.getStato() == 'Spedito'

Package GestioneApprovvigionamenti
(java.application.GestioneApprovvigionamenti.GestioneApprovvigionamenti)

Nome classe	**GestioneApprovvigionamentiService**

Descrizione	Offre operazioni relative alla gestione delle richieste di approvvigionamento di prodotti: creazione di una richiesta di approvvigionamento e visualizzazione delle richieste di rifornimento effettuate.
Metodi	**+visualizzaRichiesteFornitura** (user: Utente) : Collection(RichiestaApprovvigionamento) + **effettuaRichiestaApprovvigionamento**(request : RichiestaApprovvigionamento, user: Utente) : RichiestaApprovvigionamento
Invariante di classe	Nessuna
Nome metodo	+visualizzaRichiesteFornitura (user: Utente) : Collection(RichiestaApprovvigionamento)
Descrizione	Il metodo fornisce le richieste di rifornimento effettuate dal negozio.
Precondizione	**context** GestioneApprovvigionamentiService:: visualizzaRichiesteFornitura(user:Utente) **pre**: isLoggedAsOrderManager(user)
Postcondizione	Nessuna
Nome metodo	+ effettuaRichiestaApprovvigionamento(request : RichiestaApprovvigionamento, user: Utente) : RichiestaApprovvigionamento
Descrizione	Il metodo permette di creare la richiesta di approvvigionamento *request*.
Precondizione	**context** GestioneApprovvigionamentiService:: effettuaRichiestaApprovvigionamento(prod:Prodotto, user:Utente) **pre**: isLoggedAsOrderManager(user) AND (request.getProdotto()).getQuantitàDisponibile() == 0
Postcondizione	**context** GestioneApprovvigionamentiService:: effettuaRichiestaApprovvigionamento(prod:Prodotto, user:Utente) **post**: isSaved(request)

Package PagamentoService (java.application.Pagamento.PagamentoService)

Nome classe	**PagamentoService**
Descrizione	Consente di effettuare il pagamento di un ordine del cliente.
Metodi	**+effettuaPagamento**(user: Utente, payment : Pagamento) : Pagamento
Invariante di classe	Nessuna

Nome metodo	+effettuaPagamento(user: Utente, payment : Pagamento) : Pagamento
Descrizione	Il metodo tiene traccia del pagamento *payment* fatto dall'utente *user*.
Precondizione	**context** PagamentoService:: effettuaPagamento(user: Utente, payment : Pagamento) **pre**: isLogged(user.getUsername())
Postcondizione	**context** PagamentoService:: effettuaPagamento(user: Utente, payment : Pagamento) **post**: (payment.getOrdine()).getStato == 'Richiesta effettuata') AND isSaved(payment)

Package GestioneCatalogoService (java.application.GestioneCatalogo.GestioneCatalogoService)

Nome classe	GestioneCatalogoService
Descrizione	Fornisce le operazioni relative alla gestione del catalogo dei prodotti del negozio: visualizzazione del catalogo, inserimento di prodotti nel catalogo, eliminazione di prodotti dal catalogo ed aggiornamento delle specifiche di un prodotto del catalogo.
Metodi	+**visualizzaCatalogo**(user: Utente) : Collection(ProxyProdotto) +**aggiuntaProdottoInCatalogo**(user: Utente, codice: String, nome: String, marca: String, modello: String, topDescrizione: String, dettagli: String, prezzo: float, quantita: int, categoria: String, sottocategoria: String, inCatalogo: boolean, inVetrina: boolean, ProdottoDAODataSource productDAO) : Collection(ProxyProdotto) +**rimozioneProdottoDaCatalogo**(user: Utente, product: ProxyProdotto) : Collection(ProxyProdotto) + **aggiornamentoSpecificheProdotto**(user: Utente, product: Prodotto, infoSelected: String, updatedData: String) : Collection(ProxyProdotto) + **aggiornamentoProdottoInVetrina**(user:Utente, product: Prodotto, information: String , updatedData: int): Collection(ProxyProdotto) +**aggiornamentoDisponibilitàProdotto**(user: Utente, product: Prodotto, information: String , quantity: int) : Collection(ProxyProdotto)

	+**aggiornamentoPrezzoProdotto**(user: Utente, product: Prodotto, information: String, price: float) : Collection(ProxyProdotto) +**inserimentoTopImmagine**(user: Utente, product : Prodotto, information: String , image: Immagine) : Collection(ProxyProdotto) + **inserimentoImmagineInGalleriaImmagini** (user: Utente, product: Prodotto, information: String , image: Immagine) : Collection(ProxyProdotto) +**cancellazioneImmagineInGalleria**(user: Utente, product: Prodotto, information: String, image: Immagine) : Collection(ProxyProdotto)
Invariante di classe	Nessuna
Nome metodo	+visualizzaCatalogo(user: Utente) : Collection(ProxyProdotto)
Descrizione	Il metodo fornisce il catalogo dei prodotti venduti dal negozio.
Precondizione	**context** GestioneCatalogoService:: visualizzaCatalogo(user: Utente) **pre**: isLoggedAsCatalogueManager(user)
Postcondizione	**context** GestioneCatalogoService:: visualizzaCatalogo(user: Utente) **post**: result -> forAll(o : ProxyProdotto | o.isInCatalogo() == True)
Nome metodo	+aggiuntaProdottoInCatalogo(user: Utente, codice: String, nome: String, marca: String, modello: String, topDescrizione: String, dettagli: String, prezzo: float, quantita: int, categoria: String, sottocategoria: String, inCatalogo: boolean, inVetrina: boolean, ProdottoDAODataSource productDAO) : Collection(ProxyProdotto)
Descrizione	Il metodo permette di aggiungere nel catalogo il prodotto *product* avente come specifiche codice, nome, marca, modello, topDescrizione, dettagli, prezzo, quantita, categoria, sottocategoria, inCatalogo, inVetrina.
Precondizione	**context** GestioneCatalogoService:: aggiuntaProdottoInCatalogo(user: Utente, product: Prodotto) **pre**:

	isLoggedAsCatalogueManager(user) AND isInCatalogo(product) == FALSE
Postcondizione	**context** GestioneCatalogoService:: aggiuntaProdottoInCatalogo(user: Utente, product: Prodotto) **post**: result -> includes(product)
Nome metodo	+rimozioneProdottoDaCatalogo(user: Utente, product: ProxyProdotto) : Collection(ProxyProdotto)
Descrizione	Il metodo permette di cancellare il prodotto *product* dal catalogo del negozio.
Precondizione	**context** GestioneCatalogoService:: rimozioneProdottoDaCatalogo(user: Utente, product: ProxyProdotto) **pre**: isLoggedAsCatalogueManager(user) AND isInCatalogo(product) == True
Postcondizione	**context** GestioneCatalogoService:: rimozioneProdottoDaCatalogo(Utente user, Prodotto prod) **post**: !(result -> includes(prod))
Nome metodo	+ aggiornamentoSpecificheProdotto(user: Utente, product: Prodotto, infoSelected: String, updatedData: String) : Collection(ProxyProdotto)
Descrizione	Il metodo permette di aggiornare le seguenti specifiche del prodotto *product*: modello, marca, descrizione in evidenza, descrizione dettagliata, categoria, sottocategoria. Il parametro *infoSelected* è l'informazione che si vorrebbe modificare mentre *updatedData* è la nuova informazione da memorizzare.
Precondizione	**context** GestioneCatalogoService:: aggiornamentoSpecificheProdotto(user: Utente, product: Prodotto, infoSelected: String, updatedData: String) **pre**: isLoggedAsCatalogueManager(user) AND (isInCatalogo(product) == True) AND ((infoSelected == 'DESCRIZIONE_EVIDENZA') OR (infoSelected == 'DESCRIZIONE_DETTAGLIATA') OR (infoSelected == 'MODELLO') OR (infoSelected == 'MARCA') OR (infoSelected == 'CATEGORIA') OR (infoSelected == 'SOTTOCATEGORIA'))

Postcondizione	**context** GestioneCatalogoService:: aggiornamentoSpecificheProdotto(user: Utente, product: Prodotto, infoSelected: String, updatedData: String) **post**: (product.getModello() == updatedData) OR (product.getMarca() == updatedData) OR (product.getTopDescrizione() == updatedData) OR (product.getDettagli() == updatedData) OR (product.getCategoria() == updatedData) OR (product.getSottocategoria() == updatedData)
Nome metodo	+ aggiornamentoProdottoInVetrina(user:Utente, product: Prodotto, information: String , updatedData: int): Collection(ProxyProdotto)
Descrizione	Il metodo permette di aggiornare la messa in evidenza del prodotto *product*, ossia aggiungerlo o rimuoverlo nella vetrina virtuale del negozio in base al valore *updatedData*.
Precondizione	**context** GestioneCatalogoService:: aggiornamentoProdottoInVetrina(user:Utente, product: Prodotto, updatedData: int) **pre**: isLoggedAsCatalogueManager(user) AND (isInCatalogo(product) == True) AND (information == 'VETRINA')
Postcondizione	**context** GestioneCatalogoService:: aggiornamentoProdottoInVetrina(user:Utente, product: Prodotto, updatedData: int) **post**: product.getInEvidenza() == updatedData
Nome metodo	+aggiornamentoDisponibilitàProdotto(user: Utente, product: Prodotto, information: String, quantity: int) : Collection(ProxyProdotto)
Descrizione	Il metodo permette di aggiornare la quantità di scorte in magazzino del prodotto *product* a *quantity*.
Precondizione	**context** GestioneCatalogoService:: aggiornamentoDisponibilitàProdotto(user: Utente, product: Prodotto, quantity: int) **pre**: isLoggedAsCatalogueManager(user) AND (quantity > product.getQuantità() AND quantity > 0) AND (information == "QUANTITA")

Postcondizione	**context** GestioneCatalogoService:: aggiornamentoDisponibilitàProdotto(user: Utente, product: Prodotto, quantity: int) **post**: product.getQuantitàDisponibile() == quantity
Nome metodo	+aggiornamentoPrezzoProdotto(user: Utente, product: Prodotto, information: String , price: float) : Collection(ProxyProdotto)
Descrizione	Il metodo permette di aggiornare il prezzo del prodotto *product* a *price*.
Precondizione	**context** GestioneCatalogoService:: aggiornamentoPrezzoProdotto(user: Utente, product: Prodotto, price: float) **pre**: isLoggedAsCatalogueManager(user) AND price > 0.0 AND (information == "PREZZO")
Postcondizione	**context** GestioneCatalogoService:: aggiornamentoPrezzoProdotto(user: Utente, product: Prodotto, price: float) **post**: product.getPrezzo() == price
Nome metodo	+inserimentoTopImmagine(user: Utente, product : Prodotto, information: String , image: Immagine) : Collection(ProxyProdotto)
Descrizione	Il metodo permette di aggiungere l'immagine di presentazione *image* al prodotto *product*.
Precondizione	**context** GestioneCatalogoService:: inserimentoTopImmagine(user: Utente, product : Prodotto, image: Immagine) **pre**: isLoggedAsCatalogueManager(user) AND (information == "TOP_IMMAGINE")
Postcondizione	**context** GestioneCatalogoService:: inserimentoTopImmagine(user: Utente, product : Prodotto, image: Immagine) **post**: product.getTopImmagine() == image
Nome metodo	+ inserimentoImmagineInGalleriaImmagini (user: Utente, product: Prodotto, information: String, image: Immagine) : Collection(ProxyProdotto)
Descrizione	Il metodo permette di aggiungere l'immagine di dettaglio *image* all'insieme delle immagini di dettaglio del prodotto *product*.

Precondizione	**context** GestioneCatalogoService:: inserimentoImmagineInGalleriaImmagini (user: Utente, product: Prodotto, image: Immagine) **pre**: isLoggedAsCatalogueManager(user) AND !(product.getGalleriaImmagini() -> includes(image)) AND (information == "AGGIUNTA_DETT_IMMAGINE")
Postcondizione	**context** GestioneCatalogoService:: inserimentoImmagineInGalleriaImmagini (user: Utente, product: Prodotto, image: Immagine) **post**: product.getGalleriaImmagini() -> includes(image)
Nome metodo	+cancellazioneImmagineInGalleria(user: Utente, product: Prodotto, information: String , image: Immagine) : Collection(ProxyProdotto)
Descrizione	Il metodo permette di cancellare l'immagine di dettaglio *image* all'insieme delle immagini di dettaglio del prodotto *product*.
Precondizione	**context** GestioneCatalogoService:: cancellazioneImmagineInGalleria(user: Utente, product: Prodotto, image: Immagine) **pre**: isLoggedAsCatalogueManager(user) AND product.getGalleriaImmagini() -> includes(image) AND (information == "RIMOZIONE_DETT_IMMAGINE")
Postcondizione	**context** GestioneCatalogoService:: cancellazioneImmagineInGalleria(user: Utente, product: Prodotto, image: Immagine) **post**: product.getGalleriaImmagini() -> includes(image)

4. Design patterns

In questo paragrafo si intende trattare dei design patterns scelti per garantire alcuni system design goals e design objects trade-offs individuati.
Per ogni pattern utilizzato nello sviluppo dell'applicativo TechHeaven si darà:

- Una brevissima introduzione teorica.
- Il problema che doveva risolvere all'interno di TechHeaven.
- Una brevissima spiegazione della soluzione al problema individuato.
- Un grafico della struttura delle classi che implementano il pattern.

Proxy design pattern

Il Proxy design pattern pattern permette di controllare l'accesso ad un oggetto, fornendo un surrogato o un'interfaccia di sostituzione a questo oggetto.
Il ProxyObject agisce per conto di una classe RealObject: esso memorizza un sottoinsieme degli attributi del RealObject e gestisce completamente determinate richieste (ad esempio, determinare la dimensione di un'immagine), mentre altre richieste vengono delegate al RealObject.
Dopo la delega, viene creato il RealObject e caricato in memoria.

Questo design pattern è stato scelto tenendo conto del trade-off *Spazio di memoria vs Tempo di risposta* definito in questo documento: per migliorare le performance del sistema, si rimandano le computazioni più dispendiose (es. creazione di un oggetto on-demand, gestione di risorse remote, gestione degli accessi) ad un momento successivo (quando si ha bisogno effettivamente di quell'oggetto).

Visto che la visualizzazione delle immagini in evidenza e della galleria di immagini dettagliate dei prodotti ottenuti a seguito di una ricerca (per menù di navigazione o per barra di ricerca) è costosa, si intende caricare solo l'immagine in evidenza e le caratteristiche più rilevanti di un prodotto (nome, prezzo, brand, categoria).
Nel caso in cui l'utente volese approfondire le specifiche di un prodotto allora verrà caricato in memoria il prodotto reale, con la sua galleria di immagini e tutte le sue specifiche.
Tale design pattern viene adottato anche per il caricamento delle informazioni di un utente a seguito dell'autenticazione.
Analogamente, si intende tenere traccia delle informazioni anagrafiche dell'utente.
Nel caso in cui l'utente volesse visionare lo storico degli ordini effettuato in negozio, accedendo all'area riservata, allora verranno caricati in memoria tutti gli ordini effettuati dall'utente.

Facade design pattern

Il Facade Design Pattern fornisce un'interfaccia unificata semplificata per un insieme di interfacce in un sottosistema più complesso, facilitando l'utilizzo di tali interfacce senza dover conoscere i dettagli interni.
Il design pattern è stato scelto perché risulta essere una soluzione efficace per garantire l'incapsulamento e gestire il trade-off tra leggibilità del codice e costi nel contesto descritto.

Esso rappresenta un modo per mantenere il codice pulito, semplice e facilmente comprensibile.
I vantaggi raggiunti da questo design pattern sono:

- nascondere al client i componenti di sistema, garantendo maggiore manutenibilità (viene invocato un solo oggetto);
- ridurre le dipendenze tra un sottoinsieme di sottosistemi ed il resto del sistema.

Essendo il nostro sistema molto complesso, si sfrutta il design pattern Facade per implementare tutta la sua logica di business e rendere più facile l'interfacciarsi con essa. Nello specifico, "TechHeaven" utilizza tale design pattern per ogni suo sottosistema, implementandolo attraverso delle interfacce che sono usate per accedere ai metodi interni.

DAO (Data Access Object)
Il pattern DAO 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.
Essendo TechHeaven una web application che deve soddisfare il design goal di performance sulla gestione di grandi quantità di dati (DG_1 Quantità dei dati), nonché la separazione tra la logica di business e la logica di acceso ai dati come vuole l'architettura three-tier, vi è la necessità di interagire con il database in modo rapido e sicuro.
Per questo motivo abbiamo definito varie interfacce DAO all'interno del nostro sistema, raccolte nel package **java.storage** per accedere al database nel resto dell'applicazione.