Università degli Studi di Salerno
Corso di Ingegneria del Software

PHarmaLife
Object Design Document
Version 1.0

PHARMA LIFE

(Ultima modifica in data: 29/01/2022)

Progetto: PHarmaLife	Versione: 1.0
Documento: Object Design Document	Data: 29/01/22

Coordinatore del progetto:

Nome	**Matricola**
Carmine Gravino	

Partecipanti:

Nome	**Matricola**
Alfredo Cannavaro [AC]	0512108651
Eduardo Scarpa [ES]	0512109503
Carmine Fierro [CF]	0512106203
Catello Staiano [CS]	0512106875

Scritto da:	Gruppo *G21*

Revision History

Data	Versione	Descrizione	Autore
29/12/2021	1.0	Introduzione, Object design goals, Object trade-off	Eduardo Scarpa [ES]
30/12/2021	1.0	Linee guida per la documentiazione dell'interfaccia	Eduardo Scarpa [ES]
30/12/2021	1.0	Definizioni, acronimi e abbreviazioni, Riferimenti	Gruppo G21
30/12/2021	1.0	Packages	Carmine Fierro [CF] e Alfredo Cannavaro [AC]
02/01/2022	1.0	Class Interfaces	Catello Staiano [CS] e Carmine Fierro [CF]
02/01/2022	1.0	Class Diagram	Alfredo Cannavaro [AC]
03/01/2022	1.0	Design Pattern	Gruppo G21
29/01/2022	1.0	Ultima revisione generale	Gruppo G21

Sommario

1. Introduzione

1.1. Object design goals

Robustezza:
Il sistema deve risultare robusto, reagendo correttamente a situazioni impreviste attraverso il controllo degli errori e la gestione delle eccezioni.

Incapsulamento:
Il sistema garantisce la segretezza sui 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. Object trade-off

Tempo di rilascio vs Funzionalità

Nonostante i tempi di sviluppo ridotti, il team si impegna nel consegnare il sistema nella data di consegna prevista , prevedendo di implementare dapprima le funzionalità necessarie per il corretto acquisto e gestione dei prodotti, ponendo successivamente , tramite opportuni interventi di manutenzione ed aggiornamenti, l'aggiunta delle funzionalità secondarie.

Prestazioni vs Costi

Per rientrare nel budget a disposizione, il team cercherà di ottenere le migliori prestazioni ma nelle ore-lavoro garantite dal budget.

Criteri di manutenzione vs Criteri di performance

Il sistema sarà implementato preferendo la manutenibilità alla performance in modo da facilitare gli sviluppatori nel processo di aggiornamento del software a discapito delle performance del sistema.

Di seguito è riportata una tabella che mostra i design goal preferiti nei trade off. Il **grassetto** indica la preferenza.

TRADE-OFF	
Tempo di rilascio	Funzionalità
Prestazioni	Costi
Criteri di manutenzione	Criteri di performance

1.3. Linee guide per la documentazione

Al fine di offrire una facile leggibilità del codice è stata utilizzata la notazione CamelCase, attribuendo alle variabili e alle classi nomi evocativi rispetto al ruolo che svolgono. Le classi che si occupano della gestione dei dati del sistema devono avere come suffisso "DAO".

In questo paragrafo verranno, quindi, indicate le linee guida a cui un programmatore deve attenersi nell'implementazione di questo sistema.

1.3.1 Classi e Interfacce Java

Lo standard nella definizione delle classi e delle interfacce Java è quello definito da Google (https://google.github.io/styleguide/javaguide.html). Ciascuno di questi deve essere documentato con JavaDoc.

1.3.2 Nomenclatura delle componenti

Nomi delle classi

- Ogni classe deve avere il nome in *CamelCase*.
- Ogni classe deve avere nome singolare.
- Ogni classe che modella un'entità deve avere per nome un sostantivo che possa associarla alla corrispondente entità di dominio.

Nomi delle variabili

- I nomi delle variabili devono cominciare con una lettera minuscola (es: prodotto). Se il nome della variabile è costituito da più parole, solo l'iniziale delle altre parole sarà maiuscola (es: codiceFiscale), inoltre è possibile abbreviare il nome della variabile solo se non peggiora la leggibilità e comprensibilità (es: numProdotti invece di numeroProdotti).
- Le costanti dovranno essere scritte interamente in maiuscolo e se il nome è costituito da più parole vengono separate con l'underscore (es: PI_GRECO).

Nomi dei metodi

- Ogni metodo deve avere nome in *lowerCamelCase*.

Nomi delle eccezioni

- Ogni eccezione deve avere nome esplicativo del problema segnalato.

Nomi degli altri sorgenti

- Ogni documento HTML deve avere nome che possa ricondurre al contenuto da essa mostrato.

1.3.3 Java Servlet Pages (JSP)

Le JSP costruiscono pagine HTML in maniera dinamica che devono rispettare il formato definito nel sotto paragrafo sottostante. Devono anche attenersi alle linee guida per le classi Java definito nel sotto paragrafo soprastante.

1.3.4 Pagine HTML

Le pagine HTML devono essere conformi allo standard *HTML5*. Inoltre, il codice HTML deve utilizzare l'indentazione per facilitare la lettura e di conseguenza la manutenzione. Le regole di indentazione sono le seguenti:

1. Ogni tag deve avere un'indentazione maggiore del tag che lo contiene;
2. Ogni tag di chiusura deve avere lo stesso livello di indentazione di quello di apertura;

1.3.5 File JavaScript

Gli script JavaScript seguono anche essi le linee guida definite dallo standard Google.
Sia i file che i metodi JavaScript devono essere documentati seguendo lo stile di JavaDoc.

1.3.6 Fogli di stile CSS

Ogni foglio di stile CSS deve avere ad inizio nome la parola "style" e in ognuno di questi deve avere le modifiche per *PC, TABLET E CELLULARE.*

1.3.7 Script SQL

Le istruzioni e le clausole SQL devono essere costituite da sole lettere minuscole.
I nomi delle tabelle hanno la prima lettera maiuscola e le successive minuscole e il loro nome deve essere un sostantivo singolare.

I nomi degli attributi deve rispettare il *CamelCase.*

1.4. Definizioni, acronomi e abbreviazioni

Andremo ad utilizzare due componenti Off-the-Shelf:

Servlet: Oggetti scritti in linguaggio Java che operano all'interno di un server web (Apachi Tomcat) che permettono la creazione di applicazioni web. Le jsp ci permettono di rendere più dinamico il sito.

JDBC: Un connettore per il database che connette l'accesso e la gestione dei dati persistenti da qualsiasi programma scritto in linguaggio Java.

Package: insieme di classi, interfacce o file correlati;
Design pattern: uno schema di soluzioni a problemi ricorrenti impiegati per ottenere flessibilità;
Interfaccia: insieme di signature delle operazioni offerte dalla classe;
View: visualizza i dati contenuti nel model e si occupa dell'interazione con utenti e agenti;
Javadoc: è un applicativo utilizzato per la generazione automatica della documentazione del codice sorgente scritto in linguaggio java.
DOM = Modella la struttura di pagine web.

Acronimi

ODD = Object Design Document
CD = Class Diagram
GUI = Graphical User Interface (Interfaccia grafica utente)
HTML = HyperText Markup Language
CSS = Cascading Style Sheet
DOM = Document Object Model
AJAX = Asynchronous JavaScript And XML
JavaEE = Java Enterprise Edition
DBMS: Database Management System
API: Application Programming Interface
JDBC: Java Database Connectivity

1.5. Riferimenti

➔ Libro: -- Object-Oriented Software Engineering (Using UML, Patterns and Java) Third Edition

◆ *Autori*: -- Bernd Bruegge & Allen H. Dutoit

➔ Documenti:
-- RAD_PHarmaLife – Requirements Analysis Document
-- SDD_PHarmaLife – System Design Document

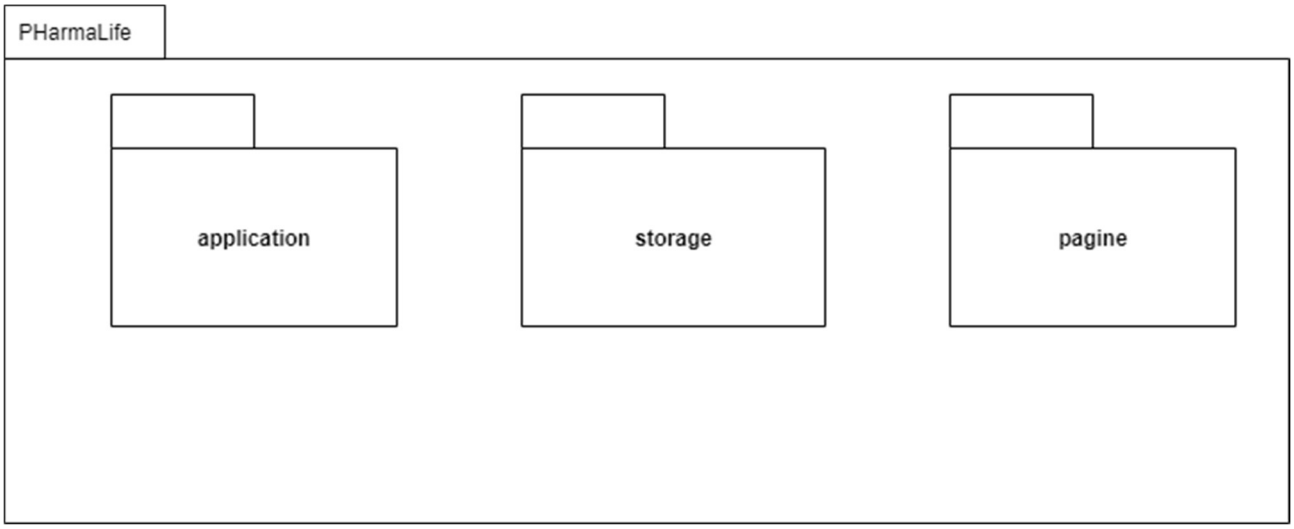
✚ Slide lezioni del Prof. Carmine Gravino

2. Packages

Il nostro sistema è suddiviso in package basandoci sulle scelte architetturali prese durante la fase di progettazione (System Design).

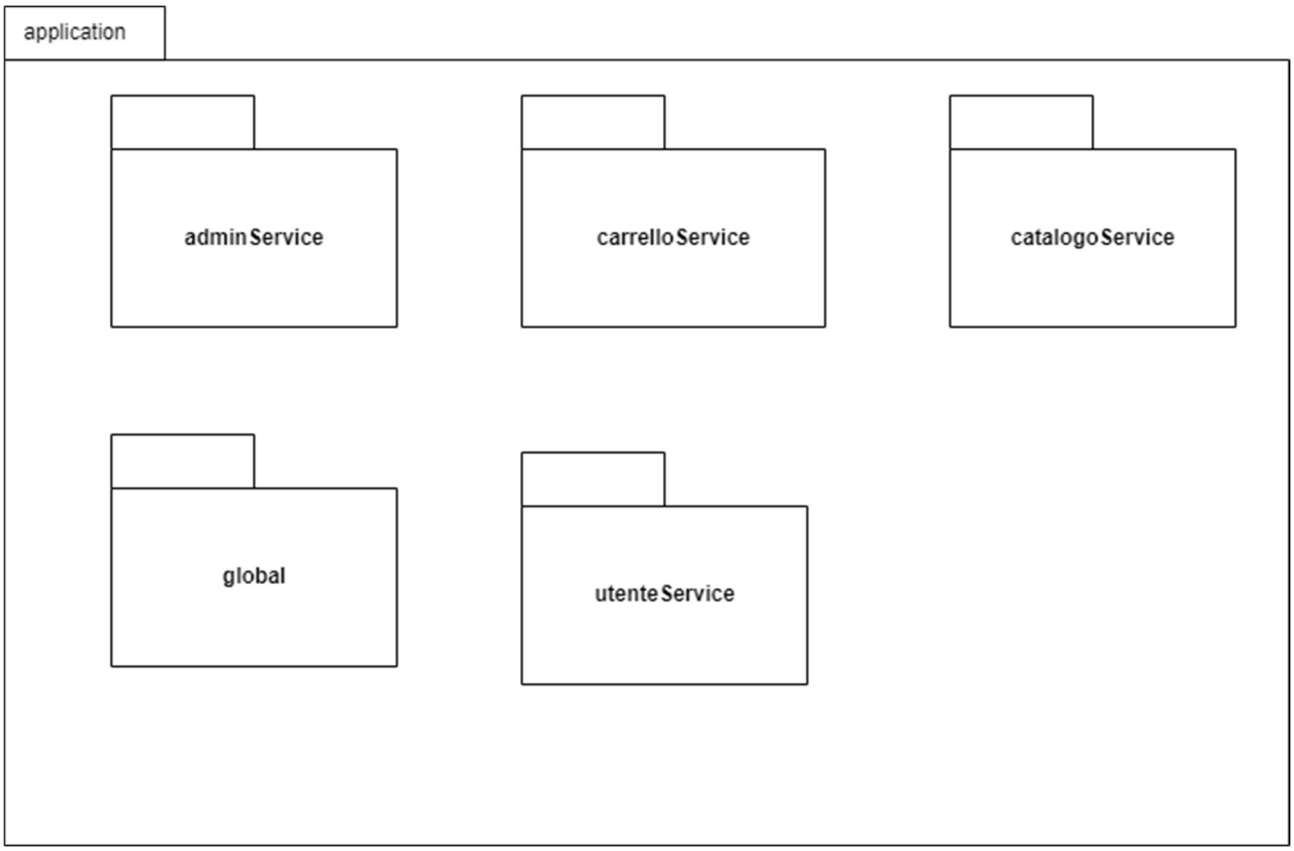
- docs
  - jacoco
  - javadoc
- src: contiene il codice sorgente e file come immagini e librerie.
  - main
    - java
      - application
        - utenteService
        - carrelloService
        - catalogoService
        - adminService
        - global
      - storage
        - utente
        - carrello
        - categoria
        - marchio
        - messaggio
        - ordine
        - prodotto
      - utils
        - ConPool

- webapp
  - css
  - font
  - imgErrore
  - immagini
  - immaginiProddotti
  - immaginiSlider
  - js
  - WEB-INF
    - lib
    - pagine
      - admin
      - default
      - utente
      - catalogo
      - ordine
      - messaggio
      - carrello
  - index.jsp



2.1 Application

Il package Application contiene le classi che si occupano di modellare la logica di business del sistema.

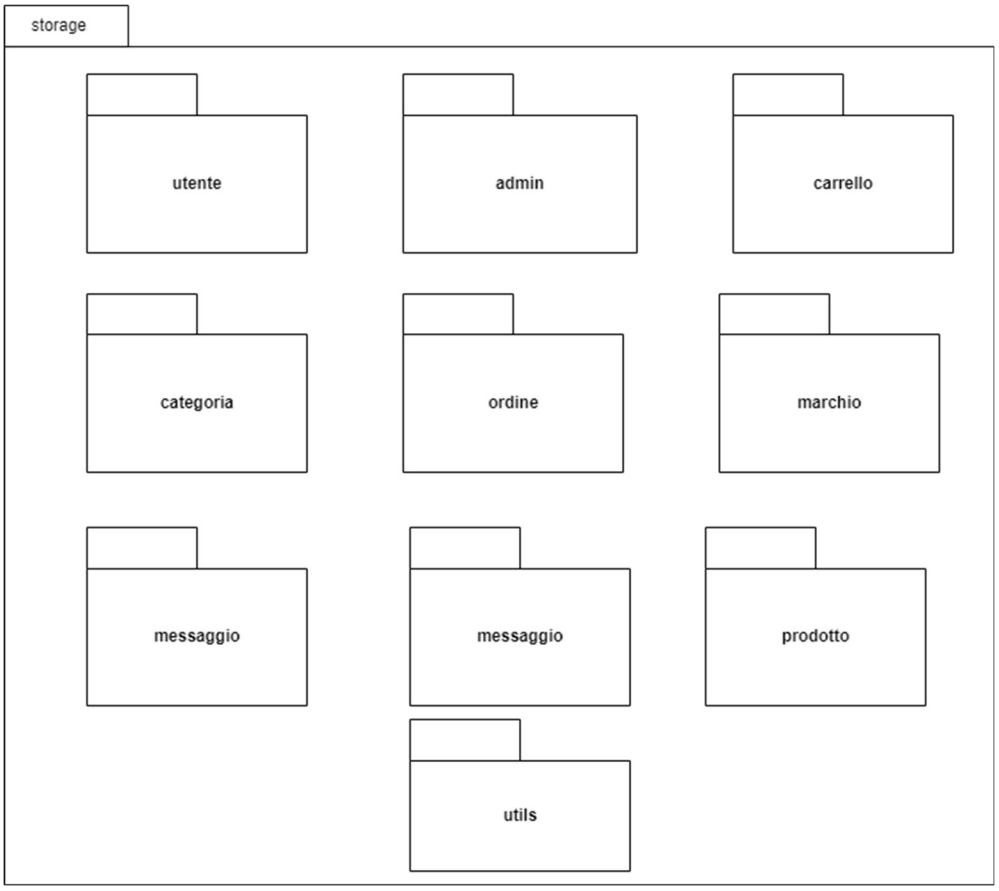


2.2 Storage

Il package Storage contiene le classi che gestiscono le richieste del client. Tutti i sottopackage contengono classi che estendono la classe *HttpServlet.*

Sottopackage:

- **admin:** contiene le servlet per gestire le richieste dell'admin.
- **carrello:** contiene le servlet per gestire le operazioni sul carrello.
- **catalogo:** contiene le servlet per gestire le operazioni sul catalogo.
- **global:** contiene la ServletStart che sarebbe la servlet che viene chiamata ancora prima della visualizzazione del homepage che ha il compito di reperire la lista dei nomi dei marchi e dei prodotti.
- **utente:** contiene le servlet per gestire le richieste degli utenti e consente anche di effettuare operazioni di autenticazione e di log-out.
- **utils:** contiene ConPool**.**

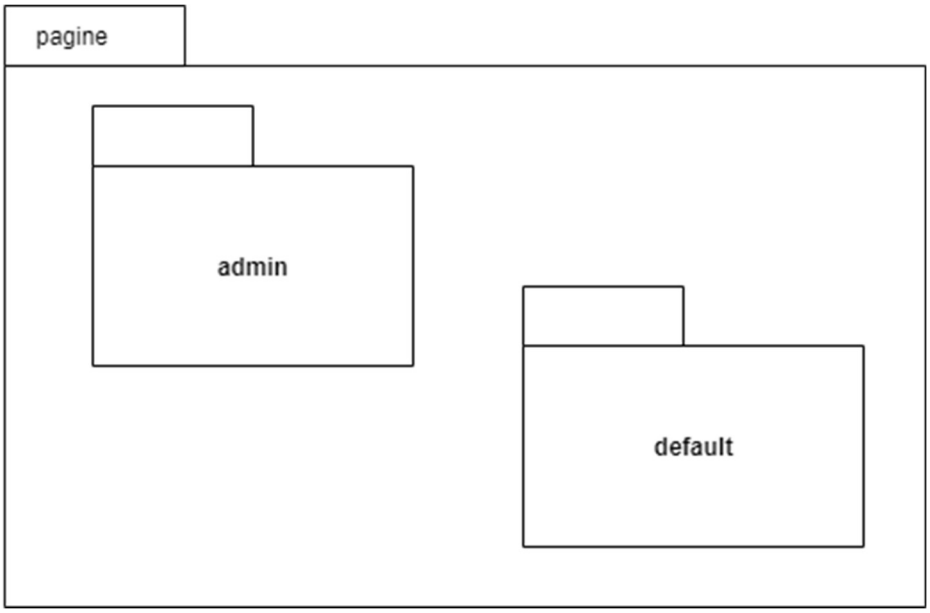


2.3 Pagine

Il package pagine contiene tutte le pagine JSP tramite le quali gli utenti interagiscono con il sistema.

Il package si divide in diversi sottopackage:

- **admin:** contiene le pagine JSP attraverso le quali l'admin comunica con il sistema.
- **carrello:** contiene le pagine JSP riguardanti le operazioni del carrello.
- **catalogo:** contiene le pagine JSP riguardanti le operazioni del catalogo.
- **messaggio:** contiene le pagine JSP riguardanti le operazioni del messaggio.
- **default:** contiene le pagine JSP di default, ovvero: head, header e footer.
- **utente:** contiene le pagine JSP attraverso le quali l'utente comunica con il sistema.



3. Class Interfaces

JavaDoc di *PHarmaLife*

Per motivi di leggibilità si è scelto di creare un sito, hostato tramite GitHub pages, contenente la ***JavaDoc di PHarmaLife***. In tale maniera, chiunque può consultare la documentazione aggiornata dell'intero sistema. Di seguito, il link al sito in questione:
https://edd0y-dev.github.io/pharmalife-2022/

- **Package utente**

Nome classe	ServletIscrizione
Descrizione	Questa classe permette di gestire le operazioni legate alla registrazione di un nuovo utente alla piattaforma.
Metodi	+registraUtente(String nome, String cognome, String codiceFiscale, String email, String password, String ripeti_password, String via, int numeroCivico, String cap, String telefono) +isNotPresentCf (String codiceFiscale) +saveParameter()
Invariante di classe	

Nome metodo	+saveParameter()
Descrizione	Questo metodo ha la funzione di recuperare tutti i parametri necessari al fine di effettuare l'iscrizione.
Pre-condizione	
Post-condizione	

Nome metodo	+isNotPresentCf(String cf) : Boolean
Descrizione	Questo metodo ha la funzione di verificare che il codice fiscale di utente che vuole registrarsi non sia già presente nel database, nel caso in cui il codice fiscale è presente il metodo restituisce false, altrimenti true.
Pre-condizione	
Post-condizione	

Nome metodo	+registraUtente (String nome, String cognome, String codiceFiscale, String email, String password, String ripeti_passwrd, String via, int numeroCivico, String cap, String telefono)
Descrizione	Questo metodo ha la funzione di registrare un nuovo utente alla piattaforma di controllare che tutti i parametri da lui inseriti rispettino un determinato formato.
Pre-condizione	**Context:** ServletIscrizione::registraUtente(nome, cognome, codiceFiscale, email, password, ripeti_password, via, numeroCivico, cap, telefono) **pre:** isNotPresentCf(cf)
Post-condizione	**Context:** ServletIscrizione::registraUtente(nome, cognome, codiceFiscale, email, password, ripeti_password, via, numeroCivico, cap, telefono) **post:** !isNotPresentCf(cf)

Nome classe	ServletAccessoUtente
Descrizione	Si occupa di gestire li login e logout di un utente dalla piattaforma.
Metodi	+logoutUtente(HttpServetRequest request, HttpServetRequestresponse) +loginUtente(HttpServetRequest request, HttpServetRequestresponse)
Invariante di classe	

Nome metodo	+logoutUtente(HttpServletRequest request, HttpServletResponse response)
Descrizione	La funzione di questo metodo è quella di rimuovere un utente dalla sessione corrente.
Pre-condizione	**Context:** ServletIAccessoUtente::logoutUtente (HttpServletRequest request, HttpServletResponse response) **pre:** session.contains(utente)
Post-condizione	**Context :** ServletIAccessoUtente::logoutUtente (HttpServletRequest request, HttpServletResponse response) **pre:** !session.contains(utente)

Nome metodo	+loginUtente(HttpServletRequest request, HttpServletResponse response)
Descrizione	La funzione di questo metodo è quella di permette ad un utente che ha effettuato ala registrazione di accedere alla piattaforma con le proprie credenziali.
Pre-condizione	
Post-condizione	**Context:** ServletIAccessoUtente::logoutUtente (HttpServletRequest request, HttpServletResponse response) **pre:** session.contains(utente)

- **Package carrello**

Nome classe	ServletAggiungiAlCarrello
Descrizione	Questa classe permette di gestire l'operazione relativa all'aggiunta di un prodotto al carrello.
Metodi	+aggiuntaAlCarrello(int idProdotto, HttpServletRequest request, HttpServletResponse response)
Invariante di classe	

Nome metodo	+aggiuntaAlCarrello(int idProdotto, HttpServletRequest request, HttpServletResponse response)
Descrizione	Questo metodo consente di aggiungere un nuovo prodotto al carrello.
Pre-condizione	//
Post-condizione	**Context:** ServletAggiungiAlCarrello:: aggiuntaAlCarrello(idProdotto, request, response) **post:** getNumProdotti = @pre getNumProdotti+1

Nome classe	ServletRimuoviDalCarrello
Descrizione	Questa classe permette di gestire l'operazione relativa alla rimozione di un prodotto dal carrello.
Metodi	+rimozioneDalCarrello(int codiceProdotto, HttpServletRequest request)
Invariante di classe	

Nome metodo	+rimozioneDalCarrello(int codiceProdotto, HttpServletRequest request)
Descrizione	Questo metodo consente di rimuovere un prodotto dal carrello.
Pre-condizione	//
Post-condizione	**Context:** ServletRimuoviDalCarrello:: rimozioneDalCarrello(codiceProdotto, request) **post:** getNumProdotti = @pre getNumProdotti-1

- **Package admin**

Nome Classe	ServletInsertProdotto
Descrizione	Questa classe permette all'amministratore di aggiungere un nuovo prodotto al catalogo.
Metodi	+aggiungiProdottoAlCatalogo(String nomeProdotto,double prezzoProdotto,String marchioProdotto,int quantita,String categoria, String descrizione, String pathImmagine)
Invariante di classe	

Nome metodo	+aggiungiProdottoAlCatalogo (String nomeProdotto, double prezzoProdotto, String marchioProdotto, int quantita, String categoria, String descrizione, String pathImmagine)
Descrizione	Questo metodo ha la funzione di aggiungere un nuovo prodotto al catalogo.
Pre-condizione	
Post-condizione	**Context:** ServletInsertProdotto::aggiungiProdottoAlCatalogo (String nomeProdotto,double prezzoProdotto,String marchioProdotto,int quantita,String categoria, String descrizione, String pathImmagine) **post:** *prodottoDAO.doRetrieveByAllProdotti.size=@pre prodottoDAO.doRetrieveByAllProdotti.size+1*

2Nome classe	ServletDeleteProdotto
Descrizione	Questa classe permette all'amministratore di rimuovere un prodotto dal catalogo.
Metodi	+eliminaProdottoDalCatalogo (idProdotto)
Invariante di classe	

Nome metodo	+eliminaProdottoDalCatalogo (idProdotto)
Descrizione	Questo metodo ha la funzione di rimuovere un prodotto dal catalogo.
Pre-condizione	
Post-condizione	**Context:** ServletInsertProdotto::eliminaProdottoAlCatalogo (int idProdotto) **post:** *prodottoDAO.doRetrieveByAllProdotti.size=@pre prodottoDAO.doRetrieveByAllProdotti.size-1*

4. Class Diagram

Il class diagram è suddiviso secondo lo schema Three Tier.

Lascaremo qui sottostante il *LINK* per visionare meglio il Class Diagram siccome le nuove notevoli dimensioni.

LINK*:* - Click me

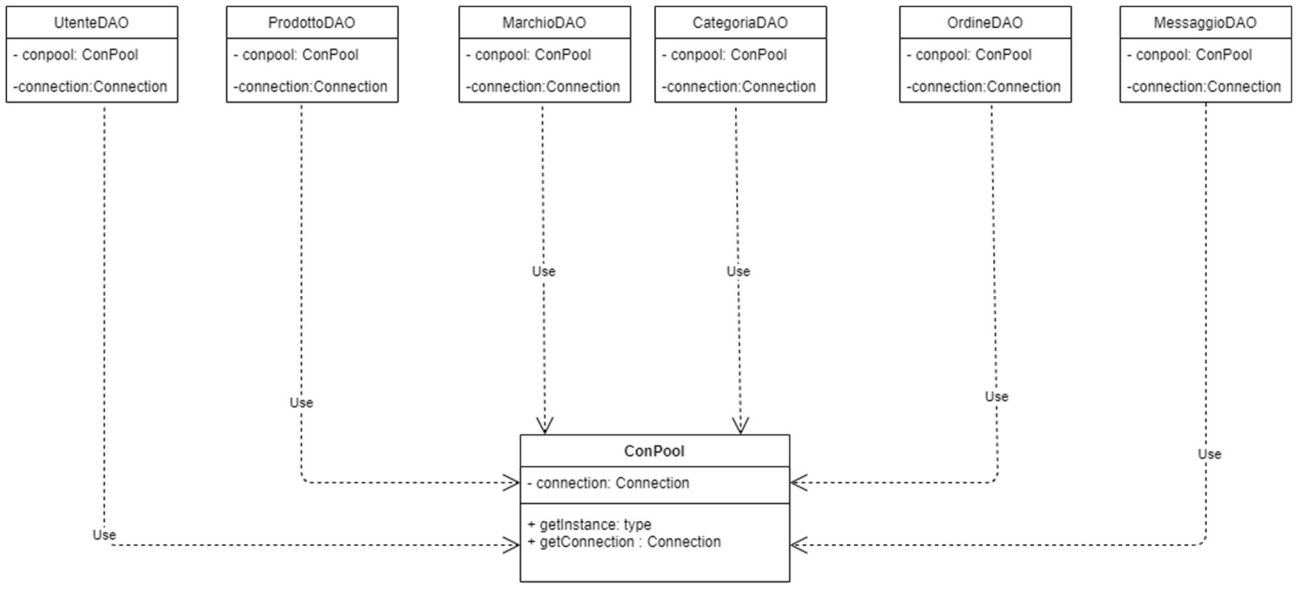
5. Design Patterns

Il nostro sistema utilizzerà diversi Design Pattern:

- **Singleton**

  Il Singleton è classificato come un design pattern creazionale, ovvero il design pattern che ha lo scopo di instanziare oggetti garantendo la creazione di una sola instanza di quella classe. La problematica che il Singleton si pone di risolvere è quello di creare una sola connessione per ogni evocazione di un metodo DAO migliorando così le performance.

  E' la classe *ConPool*, che si occupa di creare e mantenere una connessione al database, è una classe singleton, così che possa essere acceduta in maniera atomica, senza creare molteplici connessioni.

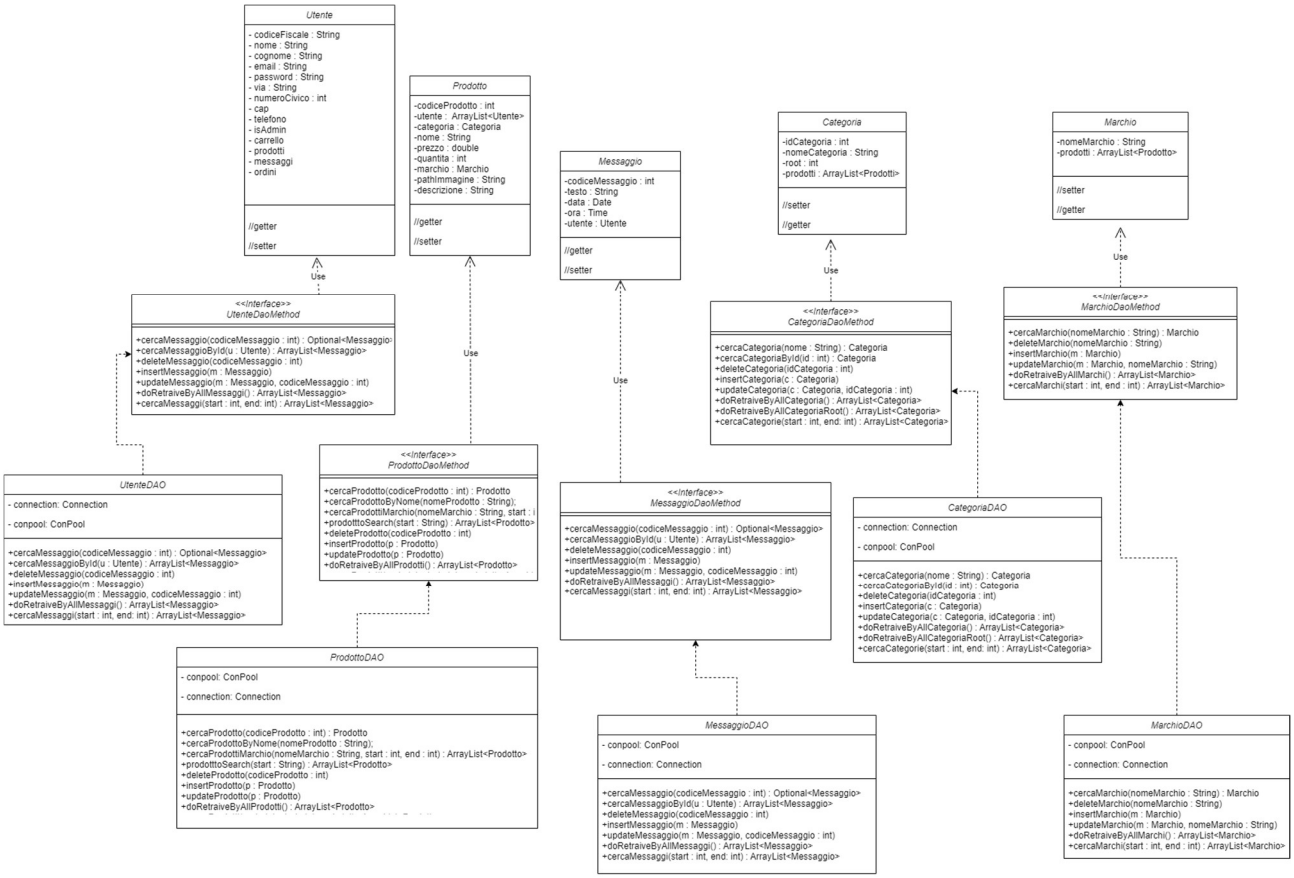


- **Data Access Object (DAO)**

  Il *DAO Pattern* si occupa della gestione dei dati persistenti in un sistema software che utilizza l'architettura di tipo Three Tier, separando il logic tier dal data tier.
  Si tratta fondamentalmente di una classe con i relativi metodi che rappresenta un'entità tabellare di un DBMS, generalmente utilizzato in un'applicazione web.

  Per garantire un accesso alle informazioni persistenti senza prescindere dal sistema utilizzato per la persistenza, utilizzeremo un'interfaccia DAO che ci permetterà di disaccoppiare la logica del sistema dal gestore della persistenza (che potrà cambiare nel tempo in maniera completamente isolata).

  Permette l'aumento della comprensione e della manutenibilità del codice.



- **Observer**
  L'Observer Pattern è di tipo comportamentale e definisce una relazione uno-a-molti tra oggetti in modo tale che quando un oggetto cambia stato tutti gli oggetti che dipendono da esso ne ricevano notifica.

  Trova applicazione nei casi in cui diversi oggetti detti "observers" devono conoscere lo stato di un oggetto detto "subject". In poche parole abbiamo un oggetto che viene "osservato" (il subject) e tanti oggetti che "osservano" i cambiamenti di quest'ultimo (gli observers).

  Per rendere noto lo stato del subject all'observer implementiamo il **modello pull** dove gli observer richiedono al subject informazioni in caso di necessità. Lo scopo del pattern all'interno del nostro sistema è quello di mantenere un basso accoppiamento tra oggetti rendendo Subject e Observer indipendenti; questo comporta che una modifica di uno di questi componenti non richiede la modifica dell'altro componente.

- **Front controller**
  Fornisce un meccanismo centralizzato di gestione delle richieste. Consente di ottenere maggiori performance, rispetto a implementazioni ad hoc, ma questo si riflette negativamente sulla flessibilità del codice. Le richieste vengono gestite da un singolo gestore che può eseguire l'autenticazione o l'autorizzazione o la registrazione o il rilevamento della richiesta e quindi passare le richieste ai gestori corrispondenti.

  Le componenti sono:

  - **Controller**: Il controller è il punto di contatto iniziale per la gestione di tutte le richieste nel sistema.

  - **View**: Una vista rappresenta e visualizza le informazioni al client. La vista recupera le informazioni da un modello.

  - **Dispatcher**: Un dispatcher è responsabile della gestione della visualizzazione e della navigazione, gestendo la scelta della vista successiva da presentare all'utente.