**1. Introduzione**

**1.1 Object design trade-off**

**1.1.1 Modularità contro efficienza**

La modularità definita nel progetto Tutto-Elettronica si scontra con l’efficienza

nell’elaborazione in lato server. La modularità facilita la creazione e la

manutenzione del programma (simile al principio del divide et impera), inoltre ci

garantisce l’utilizzo del codice in altri progetto/applicazione. Allo stesso tempo

riduce l’efficienza dei tempi di risposta dei moduli che si occupano di determinati

servizi.

**1.1.2 Sicurezza contro efficienza**

Nel nostro sistema i clienti vengono gestiti attraverso le sessioni ed un controllo del

livello di utenza. Ciò comporta una diminuzione dell’efficienza ma tali controlli sono

necessari per rispettare i requisiti iniziali del sistema. All’interno di ogni pagina

utilizziamo delle precondizioni per gestire il controllo degli utenti, per evitare che i

rischi di sicurezza, qualora l’utente malizioso digiti dal proprio browser il percorso

esatto della chiamata al controller. Tali controlli sono un buon compromesso a

discapito della poca efficienza persa per ogni chiamata ed aggiungono robustezza al

sistema.

**1.1.3 Portabilità contro efficienza**

La portabilità del sistema Tutto-Elettronica è garantita dalla scelta del

linguaggio di programmazione Java. Lo svantaggio dato da questa scelta è nella

perdita di efficienza introdotta dal meccanismo della macchina virtuale Java. Tale

compromesso è accettabile per i numerosi supporti forniti dal linguaggio Java.

**1.2 Interface Documentation Guidelines**

Gli sviluppatori dovranno seguire alcune linee guida per la scrittura del codice.

**1.2.1 File Java**

Ogni file sorgente deve contenere una sola classe o interfaccia pubblica. Ogni file

deve contenere nel seguente ordine:

* Commenti per una migliore comprensione
* Dichiarazione del package
* Sezione import
* Dichiarazione di interfaccia o classe:
  + Attributi pubblici
  + Attributi privati
  + Attributi protetti
  + Costruttori
  + Altri metodi
* Classi interne

È previsto l’utilizzo di commenti JavaDoc.

**1.2.2 Naming**

L’utilizzo di convenzioni sui nomi rendono il programma più leggibile e

comprensibile da tutti i membri del team. In particolare secondo il modello del

codice programmato, è auspicabile che tutti siano in grado di intervenire su una

qualsiasi linea di codice.

Classi e interfacce

I nomi delle classi sono nomi (composti anche da più parole) la cui iniziale è in

maiuscolo. Ogni parola che compone un nome ha l’iniziale in maiuscolo.

I nomi delle classi devono essere semplici e descrittivi. Evitare l’uso di acronimi e

abbreviazioni per i nomi delle classi.

Nel caso una o più classi incarnino design patterns noti è consigliato l’utilizzo di

suffissi (inglesi) che richiamano lo specifico componente del design pattern

(esempio: DatabaseAdapter, GiocatoreFactory, …).

E’ consigliato l’uso della lingua italiana per i nomi, fatta eccezione per nomi inglesi

di uso comune (esempio: TestingClass, …).

Metodi

I metodi devono essere verbi (composti anche da più parole) con iniziale minuscola.

Costanti

In accordo con le convenzioni suggerite dalla Sun, i nomi di constanti vengono

indicati da nomi con tutte le parole in maiuscolo. Le parole vengono separate da

underscore “\_”.

Ad esempio:

staticfinalint MAX\_LENGHT = 24;

**1.2.3 Uso dei commenti**

E’ permesso l’utilizzo di due tipi di commenti:

Commenti Javadoc (aree di testo compresa tra il simbolo /\*\* e \*/ )

Commenti in stile C ( righe delimitate da // )

L’utilizzo dei commenti Javadoc è suggerito prima della dichiarazione di:

classi e interfacce

costruttori

metodi di almeno 3 righe di codice

variabili di classe

Ogni commento, compreso tra il simbolo /\*\* e \*/, deve specificare le funzionalità e

le specifiche del codice, senza esplicitare dettagli legati all’implementazione, in

maniera tale da rendere leggibile tale documentazione anche a sviluppatori che non

posseggono l’implementazione.

I commenti di Javadoc consentono la generazione automatica della

documentazione del codice, attraverso l’utilizzo di appositi tools.

Il commenti stile C, ovvero le linee di codice precedute da //, sono utilizzati

all’interno dei metodi, al fine di descrivere in maniera concisa e sintetica branch,

cicli, condizioni o altri passi del codice.

**1.2.4 Altre regole di stile**

E’ importante che vengano seguite anche ulteriori “regole di stile”, al fine di

produrre codice chiaro, leggibile e privo di errori.

Tra queste “regole di stile” elenchiamo le seguenti:

* I nomi di package, classi e metodi devono essere nomi descrittivi, facilmente

pronunciabili e di uso comune

* Evitare l’utilizzo di abbreviazioni di parole
* Utilizzare, dove possibile, nomi largamente in uso nella comunità informatica

(ie: i nomi dei design patterns)

* Preferire nomi con senso positivo a quelli con senso negativo
* Omogeneità dei nomi all’interno dell’applicazione
* Ottimizzazioni del codice non devono comunque inficiare la leggibilità dello

stesso. Se si è costretti a sviluppare codice poco leggibile, perché le estreme

prestazioni sono indispensabili è necessario documentarlo adeguatamente.

* Evitare la scrittura di righe di codice più lunghe di 80 caratteri e di file con

più di 2000 righe

* È consigliato, per l’indentazione, l’utilizzo di spazi al posto dei “tab”. Questo

rende il codice ugualmente leggibile su tutti gli editor (alcuni editor convertono in

automatico le tabulazioni in 4/6 spazi)

* È consigliato l’utilizzo di nomi in italiano. Tuttavia è consigliato l’utilizzo di

termini inglesi laddove si tratta di uso comune o nel caso, molto comune, di termini

comunemente usati nella loro versione inglese. E’ di fondamentale importanza

l’utilizzo di un dizionario dei nomi unico per tutto il progetto, che tutti i

programmatori saranno tenuti a seguire.

* È consigliato l’utilizzo di nomi inglesi anche nel caso si adoperino termini

della libreria standard di Java (ie: OptimizedList anziché ListaOttimizzata)

* Si consiglia l’utilizzo di parti standard dei nomi in casi come:
* Classi astratte, suffisso Abstract- (ie: AbstractProdotto)
* Design patterns (ie: se si usa l’MVC utilizzare ListModel)
* Accezioni terminanti per Exception (ie: UtenteNonTrovatoException)
* Altre situazioni analoghe
* I nomi delle interfacce segue le regole standard dei nomi. E’ sconsigliato

usare il prefisso o suffisso “Interface”

* È consigliato l’utilizzo di suffissi “standard” come “get”, “set”, “is” o “has” in

inglese

* È possibile scrivere dichiarazioni di metodi e classi in due righe, se

eccessivamente lunghi

* Evitare la notazione ungherese. La notazione ungherese, che prevede

l’utilizzo di prefissi per descrivere il tipo di dato, non dovrebbe essere utilizzata. La

motivazione è semplice: la notazione ungherese va bene per linguaggi che hanno

tipi semplici, e dove è possibile creare un vocabolario di prefissi limitato. In

linguaggi OOP i tipi primitivi hanno un uso più limitato, mentre sono gli oggetti a

farla da padrone.

* Dichiarare le variabili ad inizio blocco, sia questo un metodo o una classe, in

modo da raccogliere in un unico punto tutte le dichiarazioni.

* Utilizzare la dichiarazione per definire una sola variabile – evitando più

dichiarazioni sulla stessa riga

* L’inizializzazione delle variabili deve essere eseguita in fase di dichiarazione,

impostando un valore di default o il risultato di un metodo. Se proprio ciò non è

possibile, in quanto il valore da impostare è il risultato di una elaborazione

compiuta nel metodo stesso, inizializzare la variabile appena prima del suo utilizzo

* Allineare la dichiarazione delle variabili per renderle più leggibili,

strutturandole in blocchi omogenei per contesto (e non per tipo di dato)

* Nel caso di algoritmi troppo complessi, eseguire un refactoring per separarlo

in diversi sotto-metodi più semplici.

* I cicli devono seguire le seguenti regole:
* Per le variabili, utilizzare l’area di visibilità più stretta possibile, dichiarando

le variabili appena prima del loro utilizzo.

* Per le chiamate a metodo non utilizzare spazi dopo il nome del metodo.

**1.3 Definizioni, acronimi e abbreviazioni**

|  |  |
| --- | --- |
| ACID | Atomicità, Consistenza, Isolamento e Durabilità |
| DBMS | Database Management System |
| HTML | HyperText Markup Language |
| HTTP | HyperText Transfer Protocol |
| JDBC | Java Database Connectivity |
| JSP | Java Server Page |
| MVC | Model View Controller |
| RAD | Requirements Analysis Document |
| SDD | System Design Document |
| SQL | Structured Query Language |

**1.4 Riferimenti**

- RAD Tutto-Elettronica documento analisi dei requisiti

- SDD Tutto-Elettronica documento di system design

- dispense dei corsi seguiti in precedenza

**1.5 Overview**

Nelle sezioni successive sarà descritta l’architettura del sistema e le sue componenti

principali. Saranno esposte le tipologie di utenza ed i comportamenti del sistema

previsti per ogni tipologia, nonché le funzionalità delle componenti invocate.

Saranno inoltre descritti i requisiti minimi per la macchina che ospiterà il sistema e

le politiche di sicurezza adottate dal sistema.

**2.Packages**

Come possiamo notare dal documento SDD Tutto-Elettronica le componenti

base che costituiscono il sistema sono raccolte in moduli a loro volta raccolti in

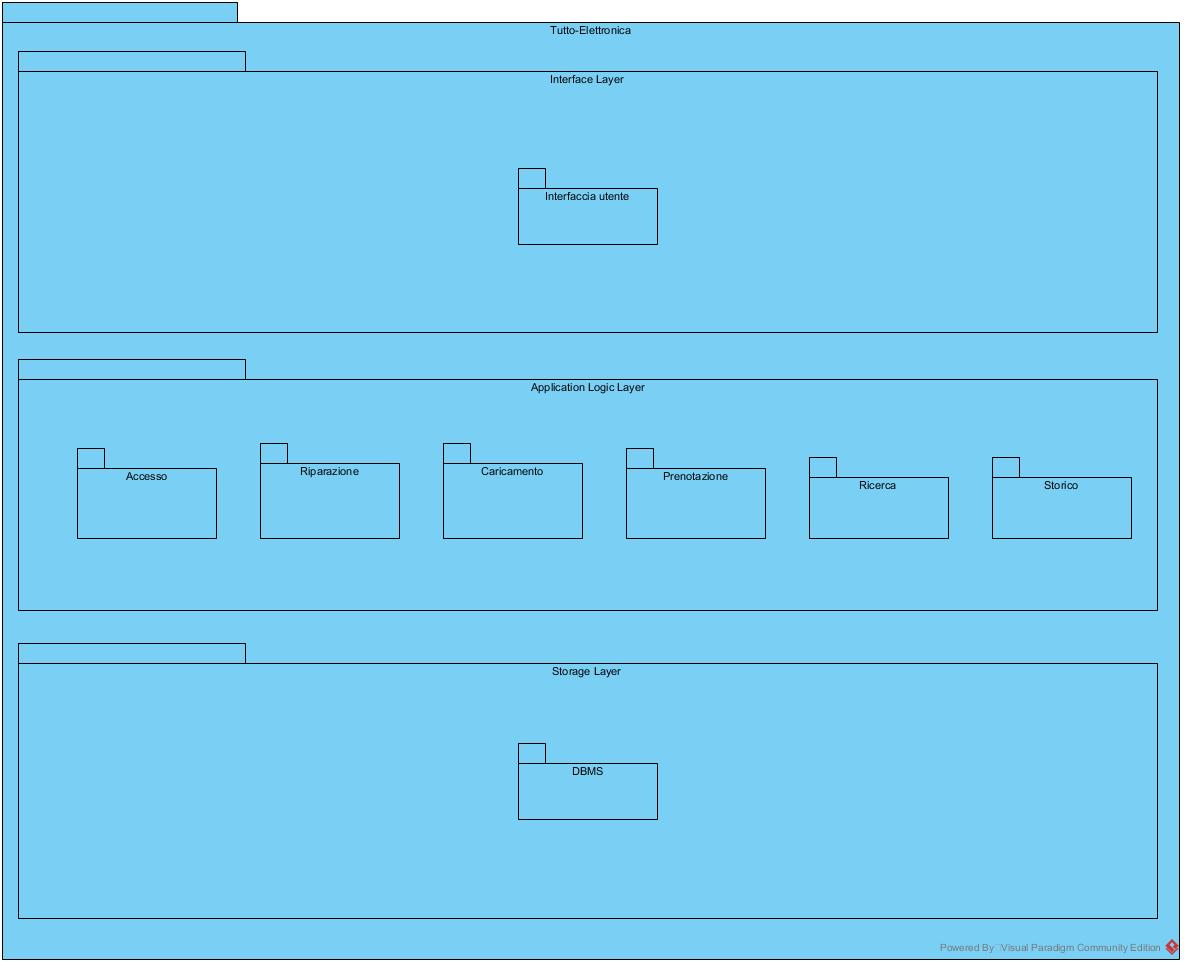
livelli. I tre livelli rappresentano la suddivisione dettata dal modello di

architettura preso in considerazione per il sistema Tutto-Elettronica “MVC”

(Model View Controller). Ciascun livello rappresenta un package contenente le

componenti relative alle funzioni associate al livello.

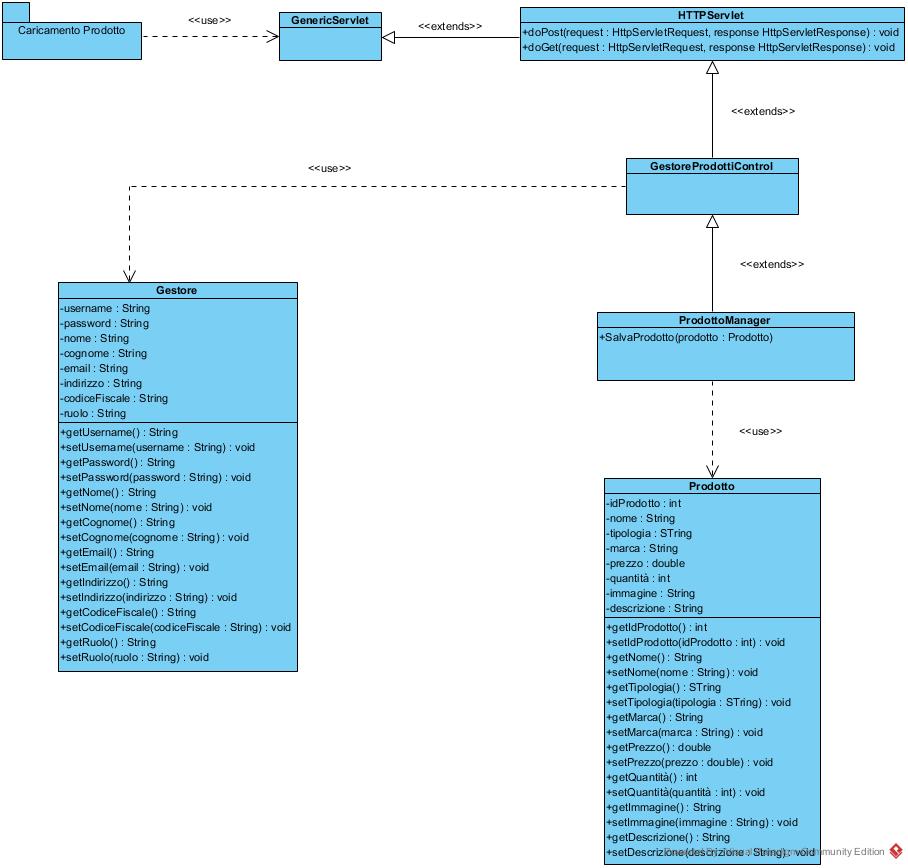
* PACKAGE SOURCE
  + MODEL
    - UserManager
    - ProdottoManager
    - CarrelloManager
    - RiparazioneManager
  + VIEW
    - ModificaRuoloBoundary
    - VisualizzaUtentiBoundary
    - VisualizzaBoundary
    - RimuoviUtenteBoundary
    - DettaglioProdottoBoundary
    - LoginBoundary
    - UserMenuBoundary
    - LogoutBoundary
    - MainPageBoundary
    - ModificaAccountBoundary
    - PrenotazioniProdottiBoundary
    - ProdottiPrenotatiBoundary
    - RiparazioniBoundary
    - ProdottiRiparatiBoundary
    - RegistrazioneAccountBoundary
    - VisualizzaAccountBoundary
    - VisualizzaDatiPersonaliBoundary
    - PrenotazioneRiparazioneBoundary
    - ProdottiRicercaBoundary
    - CarrelloBoundary
    - StatoRiparazioneBoundary
    - RecensioneBoundary
    - AddProductBoundary
    - ModificaQuantitàBoundary
    - GestisciPromozioneBoundary
    - ModificaStatoBoundary
    - AggiuntaDisponibilitàBoundary
    - ModificaDisponibilitàBoundary
  + CONTROLLER
    - VisualizzaUtentiControl
    - ModificaRuoloControl
    - RimozioneUtenteControl
    - VisualizzaDettaglioProdottoControl
    - CarrelloControl
    - AccountControl
    - LoginControl
    - RegistrazioneControl
    - VisualizzaAccountControl
    - PrenotazioneRiparazioneControl
    - RicercaProdottoControl
    - RiparazioneControl
    - InserisciRecensioneControl
    - GestisciProdottoControl
    - AddProductControl
    - ModificaStatoControl
    - AggiuntaDisponibilitàControl
    - ModificaDisponibilitàControl
    - FineRiparazioneControl

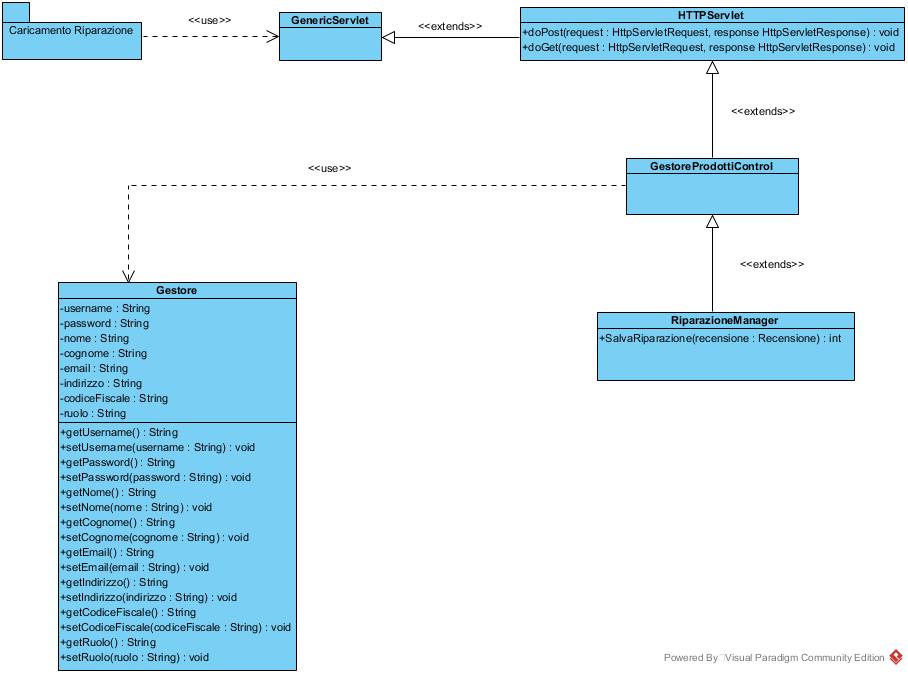


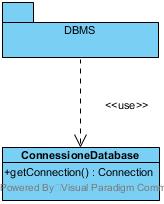
**3 Interfacce delle classi**

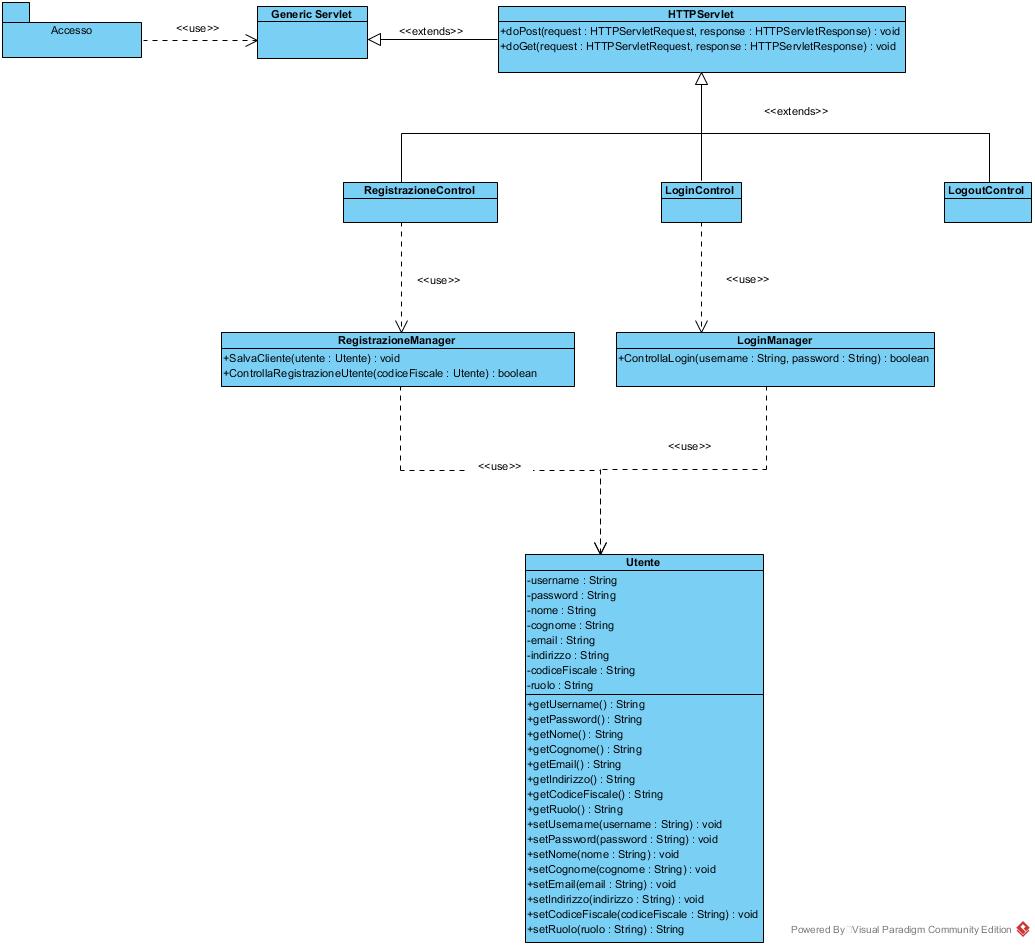
Si procede all’analisi dettagliata delle piccole classi implementate nel sistema. L’analisi serve ad evidenziare le interfacce di interazione utilizzate nella progettazione del software.

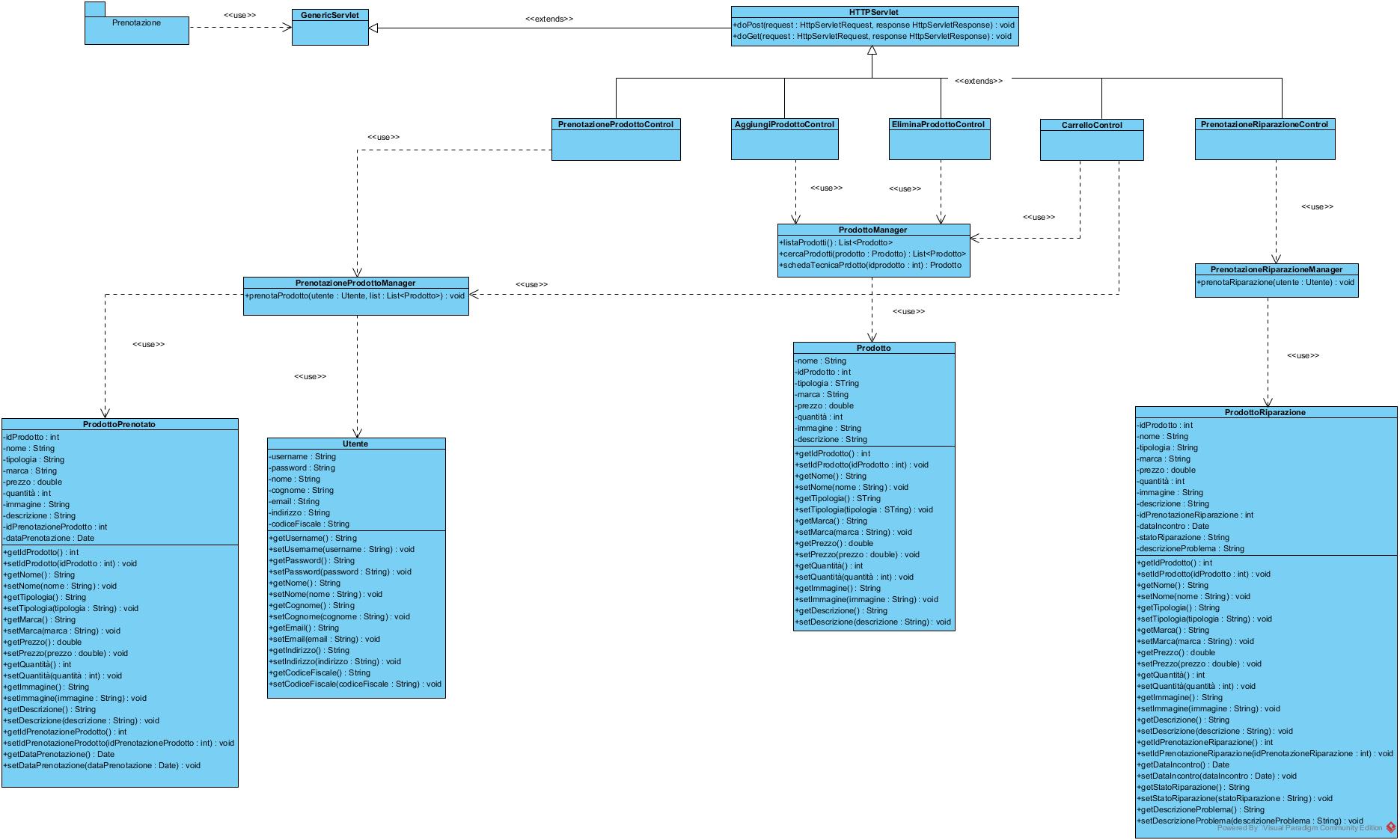
**3.1 Class Diagram**

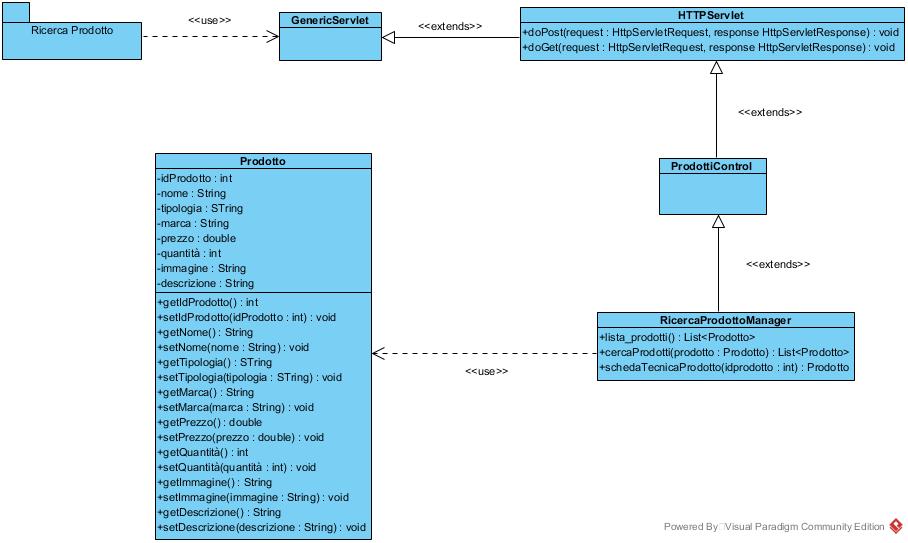


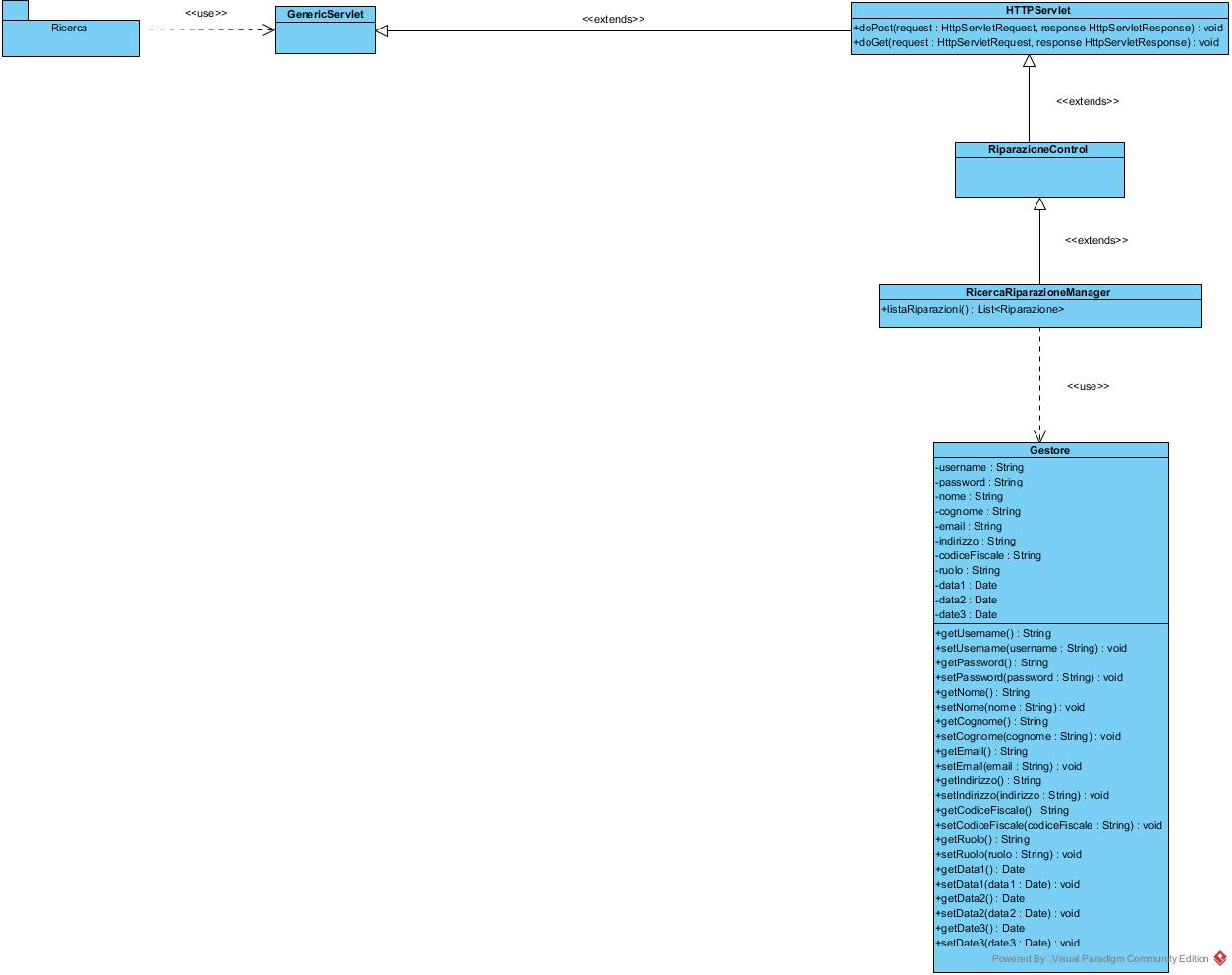












**3.2 Descrizione delle classi**

**3.2.1 Utente**

La classe contiene le informazioni relative ad un generico utente  


* Private username as string

Username dell’utente

* Private password as string

Password dell’utente

* Private nome as string

Nome dell’utente

* Private cognome as string

Cognome dell’utente

* Private email as string

Email dell’utente

* Private indirizzo as string

Indirizzo dell’utente

* Private codiceFiscale as string

Codice fiscale dell’utente

Sono inoltre presenti tutti i metodi di lettura e scrittura (set e get) per gli attributi private della classe.

**3.2.2 Prodotto**

La classe contiene le informazioni relative ad un generico prodotto



* Private idProdotto as int

Indica l’Id del prodotto

* Private nome as string

Nome del prodotto

* Private tipologia as string

Tipologia del prodotto

* Private marca as string

Marca del prodotto

* Private prezzo as double

Indica il prezzo del prodotto

* Private quantità as int

Indica la quantità disponibile del prodotto

* Private immagine as string

Indica il path dell’immagine del prodotto

* Private descrizione as string

Descrizione del prodotto

Sono inoltre presenti tutti i metodi di lettura e scrittura (set e get) per gli attributi private della classe.

**3.2.3 Recensione**

La classe contiene le informazioni relative ad una generica recensione



* Private idRecensione as int

Indica l’id della recensione

* Private voto as string

Voto della recensione

* Private commento as string

Commento per uno specifico prodotto

Sono inoltre presenti tutti i metodi di lettura e scrittura (set e get) per gli attributi private della classe.

**3.2.4 Prodotto in riparazione**

La classe contiene le informazioni relative ad un generico prodotto in riparazione



* Private idProdotto as int

Indica l’Id del prodotto

* Private nome as string

Nome del prodotto

* Private tipologia as string

Tipologia del prodotto

* Private marca as string

Marca del prodotto

* Private prezzo as double

Indica il prezzo del prodotto

* Private quantità as int

Indica la quantità disponibile del prodotto

* Private immagine as string

Indica il path dell’immagine del prodotto

* Private descrizione as string

Descrizione del prodotto

* Private idPrenotazioneRiparazione as int

Indica l’identificativo della riparazione di uno specifico prodotto in riparazione

* Private dataIncontro as date

Indica la data in cui un utente si incontra con il gestore delle riparazioni

* Private statoRiparazione as string

Indica lo stato della riparazione di uno specifico prodotto in riparazione

* Private descrizioneProblema as string

Descrizione del difetto di un prodotto da riparare consegnato

Sono inoltre presenti tutti i metodi di lettura e scrittura (set e get) per gli attributi private della classe.

**3.2.5 Prodotto prenotato**

La classe contiene le informazioni relative ad un generico prodotto prenotato

* Private idProdotto as int

Indica l’Id del prodotto

* Private nome as string

Nome del prodotto

* Private tipologia as string

Tipologia del prodotto

* Private marca as string

Marca del prodotto

* Private prezzo as double

Indica il prezzo del prodotto

* Private quantità as int

Indica la quantità disponibile del prodotto

* Private immagine as string

Indica il path dell’immagine del prodotto

* Private descrizione as string

Descrizione del prodotto

* Private idPrenotazioneProdotto as int

Indica l’identificativo di una prenotazione di uno specifico prodotto

* Private dataPrenotazione as date

Indica la data in cui uno specifico prodotto viene prenotato da un utente

Sono inoltre presenti tutti i metodi di lettura e scrittura (set e get) per gli attributi private della classe.

**3.2.6 Gestore**

La classe contiene le informazioni relative ad un gestore



* Private username as string

Username dell’utente

* Private password as string

Password dell’utente

* Private nome as string

Nome dell’utente

* Private cognome as string

Cognome dell’utente

* Private email as string

Email dell’utente

* Private indirizzo as string

Indirizzo dell’utente

* Private codiceFiscale as string

Codice fiscale dell’utente

* Private ruolo as string

Indica il ruolo che riveste il gestore

Sono inoltre presenti tutti i metodi di lettura e scrittura (set e get) per gli attributi private della classe.

**3.2.7 Database**

Viene utilizzato per la connessione al database



* Public getConnection() as Connection

Serve per accedere al database

**4. Glossario**

|  |  |
| --- | --- |
| Termini | Descrizione |
| ODD | Object Design Document |
| SDD | System Design Document |
| DBMS | Database management system |