ODD

OBJECT DESIGN DOCUMENT

Sommario

1 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:

static final int 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.

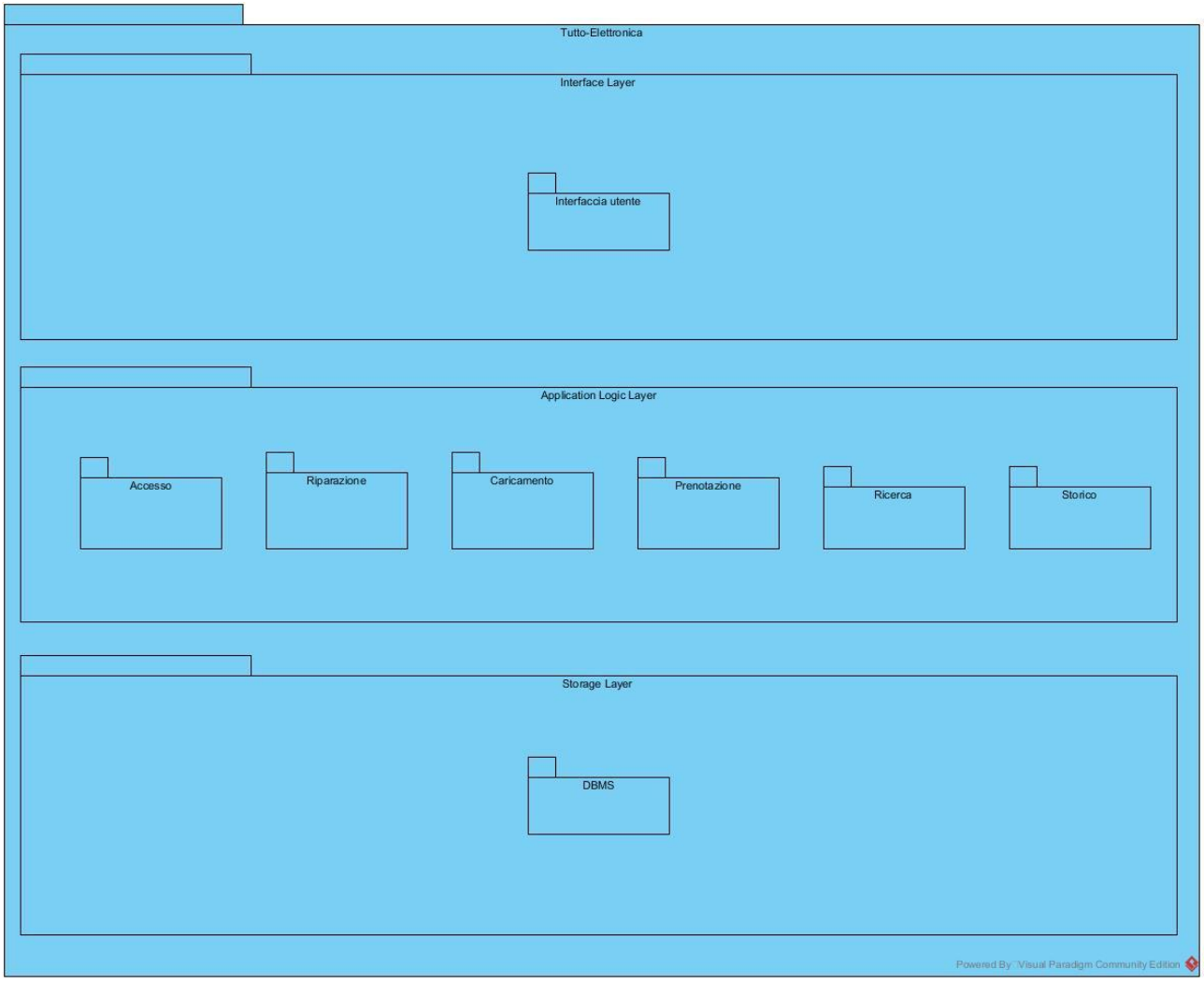
2. Packages

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
    - RecensioneManager
  - 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.

UserManager	
Servizio	**Descrizione**
public void doSave(Utente utente)	Il sottosistema permette di registrare uno utente nel sistema attraverso la compilazione di un apposito form. L' utente passato come parametro verrà salvato nel database
Public Utente doRetrieveUtente(String username, String password)	Il sottosistema permette di ottenere i dati di un utente. L'username e la password servono per identificare l'utente dal database.
public boolean doDelete(String codice)	Il sottosistema permette di cancellare un account. Il codice passato come parametro identifica l'utente che verrà eliminato dal database
public Collection<Utente> doRetrieveAll()	Il sottosistema permette di visualizzare gli utenti del sito. La lista di utenti passati come parametro verrà prelevato dal database
public Utente doRetrieveByKey(String codiceFiscale)	Il sottosistema permette di visualizzare un utente del sito. Il codice fiscale passato come parametro serve per prelevare l' utente dal database
Public boolean doUpdatePassword(String codice, String pass)	Il sottosistema permette di aggiornare la password di un utente. Il codice passato come parametro serve ad identificare l'utente, la password corrisponde alla nuova password che deve essere salvata nel database
Public boolean doUpdateEmail(String codice, String email)	Il sottosistema permette di aggiornare l'e-mail di un utente. Il codice passato come parametro serve ad identificare l'utente, email corrisponde alla nuova email che deve essere salvata nel database
Public boolean doSaveClienteRegistrato(Utente utente)	Il sottosistema permette di registrare un cliente registrato nel sistema. L' utente passato come parametro verrà salvato nel database
Public boolean doUpdateDateGestore(String codice, Date nuovaData, String att)	Il sottosistema permette di aggiornare le date di un gestore delle riparazioni. Il

	codice identifica il gestore, nuovaData rappresenta la nuova data da salvare nel database, attributo indica quale attributo data deve essere aggiornato
Public Collection<Prodotto> doRetrieveByCodiceFiscale(String codice, String tipo)	Il sottosistema permette di ottenere una collezione di prodotti. Il codice identifica l'utente, il tipo indica se la collezione deve essere di prodotti prenotati oppure di prodotti in riparazione
Public boolean doUpdateCliente(User cliente, String ruolo)	Il sottosistema permette di aggiornare il ruolo di un utente registrato al sito. Il cliente rappresenta l'entità da modificare, il ruolo indica il nuovo ruolo che dovrà ricoprire

ProdottoManager	
Servizio	**Descrizione**
Public boolean doSave(Prodotto prodotto)	Il sottosistema permette di aggiungere un prodotto al catalogo. Il prodotto passato come parametro verrà salvato nel database
Public Collection<Prodotto> doRetrieveOnSale()	Il sottosistema permette di visualizzare tutti i prodotti in promozione.
Public boolean doDelete(int codice)	Il sottosistema permette di eliminare un prodotto dal catalogo. Il codice passato come parametro serve ad identificare il prodotto che verrà eliminato dal database
public Prodotto doRetrieveByKey(int id, String tipo)	Il sottosistema permette di visualizzare un prodotto. L' id preso come parametro serve per recuperare un prodotto dal database e il tipo identifica il tipo di prodotto
public Collection<Prodotto> doRetriveAll()	Il sottosistema permette di visualizzare i prodotti presenti nel catalogo. Serve per recuperare una collezione di oggetti dal database
Public boolean doUpdateQuantitaNelCarrello(int codice, int quantita)	Il sottosistema permette di aggiornare la quantità nel carrello. Il codice identifica il prodotto, quantità indica il nuovo valore che deve essere salvato
Public boolean doUpdateQuantitaInMagazzino(int codice, int quantita)	Il sottosistema permette di aggiornare la quantità in magazzino. Il codice identifica il prodotto, quantità indica il nuovo valore che deve essere salvato
Public ArrayList<Prodotto> doRetrieveCategoria(String categoria)	Il sottosistema permette di visualizzare tutti i prodotti che fanno parte di una determinata categoria. Recupera una collezione di oggetti

	dal database e la categoria serve ad identificare tutti i prodotti di quella determinata categoria.
Public boolean doSaveInMagazzino(Prodotto p, boolean promo, int quantita)	Il sottosistema permette di salvare un prodottoinmagazzino nel database.
Public boolean doUpdatePromo(int codice, String tipo)	Il sottosistema permette di aggiornare l'attributo promo di un prodotto. Il codice identifica il prodotto, tipo indica se il prodotto viene aggiunto in promozione oppure viene rimosso.
Public boolean doUpdatePrezzo(int codice, double nuovoPrezzo)	Il sottosistema permette di aggiornare il prezzo di un prodotto. Il codice identifica il prodotto, il nuovoPrezzo rappresenta il nuovo valore che deve essere salvato nel database

RiparazioneManager	
Servizio	**Descrizione**
public boolean doSaveRiparazione(Prodotto prodotto, Date data, String codiceCliente)	Il sottosistema permette di aggiungere un prodottoInRiparazione. La data passata come parametro serve per salvare una data nel database, il prodotto indica i dati del prodotto, il codiceCliente indica il cliente che commissiona la riparazione
public boolean doUpdateData(Date data, int idRiparazione)	Il sottosistema permette di modificare la data di fine lavoro. La data passata come parametro serve per modificare una data nel database, l'idRiparazione indica il prodottoInRiparazione da modificare
public boolean doUpdateStato(String stato, int idRiparazione)	Il sottosistema permette di aggiornare lo stato della riparazione di un prodotto. L'idRiparazione serve ad identificare il prodottoInRiparazione da modificare, stato indica il nuovo stato del prodotto
public Date[ ] doRetrieveAll()	Il sottosistema permette di visualizzare le date stabilite dal gestore delle riparazioni. Serve per recuperare le date presenti nel database
Public Collection<Prodotto> doRetrieveAllRiparazioni()	Il sottosistema permette di ottenere tutti i prodotti in riparazione presenti nel database

Carrellomanager	
Servizio	**Descrizione**
public boolean doInsertProdotti(Carrello carrello)	Il sottosistema permette di aggiungere i prodotti nel carrello Il carrello passato come parametro verrà salvato nel database
public boolean doDeleteProdotti(Carrello carrello)	Il sottosistema permette di eliminare i prodotti dal carrello Il carrello passato come parametro verrà eliminato dal database
public Carrello<Prodotto> doRetrieveByKey(String codiceCliente)	Il sottosistema permette di visualizzare i prodotti presenti nel carrello. Serve per recuperare tutti gli oggetti presenti nella tabella carrello del database, il codiceCliente serve ad identificare il carrello di un determinato cliente
Public boolean doPrenota(String codiceCliente, ProdottoInMagazzino prodotto)	Il sottosistema permette di salvare una prenotazione nel database. CodiceCliente indica il cliente che effettua la prenotazione, il prodotto indica il prodotto che viene prenotato

RecensioneManager	
Servizio	**Descrizione**
public boolean doSave(Recensione recensione)	Il sottosistema permette di aggiungere una recensione a un prodotto prenotato. La recensione passata come parametro verrà salvata nel database

Nome classe	UserManager
Pre-condizione	**context** UserManager::public doSave(Utente utenteBean); **pre** String utenteBean.nome!= null && String utenteBean.cognome!=null && String utenteBean.codiceFiscale!=null && String utenteBean.username!=null && String utenteBean.password!=null && String utenteBean.indirizzo!=null && String utenteBean.ruolo!=null **context** UserManager::public doRetrieveUtente(String username, String password); **pre** String username!= null && String password!=null **context** UserManager::public doDelete(String codiceFiscale); **pre** String codiceFiscale!=null **context** UserManager::public doRetrieveByKey(String codiceFiscale); **pre** String codiceFiscale!=null **context** UserManager::public doUpdatePassword(String codiceFiscale, String pass);

	pre String codiceFiscale!=null && String pass!=null **context** UserManager::public doUpdateEmail(String codiceFiscale, String email); **pre** String codiceFiscale!=null && String email!=null **context** UserManager::public doSaveClienteRegistrato(Utente utente); **pre** String utenteBean.nome!= null && String utenteBean.cognome!=null && String utenteBean.codiceFiscale!=null && String utenteBean.username!=null && String utenteBean.password!=null && String utenteBean.indirizzo!=null && String utenteBean.ruolo!=null **context** UserManager::public doUpdateDateGestore(String codice, Date nuovaData, String att); **pre** String codiceFiscale!=null && String nuovaData!=null && String att!=null **context** UserManager::public doRetrieveByCodiceFiscale(String codice, String tipo); **pre** String codiceFiscale!=null && String tipo!=null **context** UserManager::public doUpdateCliente(User cliente, String ruolo); **pre** String cliente.codiceFiscale!=null && String ruolo!=null
Post-condizione	**context** UserManager::public doSave(Utente utente); **post** doDelete = true **context** UserManager::public doUpdateDateGestore(String codice, Date nuovaData, String att); **post** doUpdateDateGestore = true **context** UserManager::public doRetrieveCliente(String user, String password); **post** doUpdateCliente != null **context** UserManager::public doDelete(String codice); **post** doDelete = true **context** UserManager::public doRetrieveAll(); **post** doRetrieveAll != null **context** UserManager::public doRetrieveByKey(String codiceFiscale); **post** doRetrieveByKey != null **context** UserManager::public doUpdatePassword(String codice, String password); **post** doUpdatePassword = true **context** UserManager::public doUpdateEmail(String codice, String email); **post** doUpdateEmail = true **context** UserManager::public doSaveClienteRegistrato(Utente utente) **post** doSaveClienteRegistrato= true **context** UserManager::public doRetrieveByCodiceFiscale(String codice, String tipo); **post** doRetrieveByCodiceFiscale != null

	context UserManager::public doUpdateCliente(User cliente, String ruolo); **post** doUpdateCliente = true
Invarianti	

Nome classe	ProdottoManager
Pre-condizione	**context** ProdottoManager::public doSave(Prodotto prodottoBean); **pre** prodottoBean.idProdotto > 0 && prodottoBean.nome != null && prodottoBean.tipologia != null && prodottoBean.marca != null && prodottoBean.prezzo > 0 && prodottoBean.quantità > 0 && prodottoBean.immagine != null **context** ProdottoManager::public doDelete(int codice); **pre** codice > 0 **context** ProdottoManager::public doRetrieveByKey(int id, String tipo); **pre** id > 0 && String tipo!=null **context** ProdottoManager::public doUpdateQuantitaNelCarrello(int codice, int quantita); **pre** codice > 0 && quantita > 0 **context** ProdottoManager::public doUpdateQuantitaInMagazzino(int codice, int quantita); **pre** codice > 0 && quantita > 0 **context** ProdottoManager::public doRetrieveCategoria(String categoria); **pre** String categoria!=null **context** ProdottoManager::public doSaveInMagazzino(Prodotto p, boolean promo, int quantita); **pre** p.idProdotto > 0 && quantita > 0 && promo!=null **context** ProdottoManager::public doUpdatePromo(int codice, String tipo); **pre** codice > 0 && tipo!=null **context** ProdottoManager::public doUpdatePrezzo(int codice, double nuovoPrezzo); **pre** codice > 0 && nuovoPrezzo > 0
Post-condizione	**context** ProdottoManager::public doSave(Utente utenteBean); **post** doSave = true **context** ProdottoManager::public doRetrieveOnSale(); **post** doRetrieveOnSale = true **context** ProdottoManager::public doDelete(int codice); **post** doDelete = true **context** ProdottoManager::public doRetrieveByKey(int id, String tipo); **post** doRetrieveByKey != null **context** ProdottoManager::public doRetrieveAll(); **post** doRetrieveAll!= null

	context ProdottoManager::public doUpdateQuantitaNelCarrello(int codice, int quantita); **post** doUpdateQuantitaNelCarrello = true **context** ProdottoManager::public doUpdateQuantitaInMagazzino(int codice, int quantita); **post** doUpdateQuantitaNelCarrello = true **context** ProdottoManager::public doSaveInMagazzino(Prodotto p, boolean promo, int quantita); **post** doSaveInMagazzino = true **context** ProdottoManager::public doUpdatePromo(int codice, String tipo); **post** doUpdatePromo = true **context** ProdottoManager::public doUpdatePrezzo(int codice, double prezzo); **post** doUpdatePrezzo = true
Invarianti	

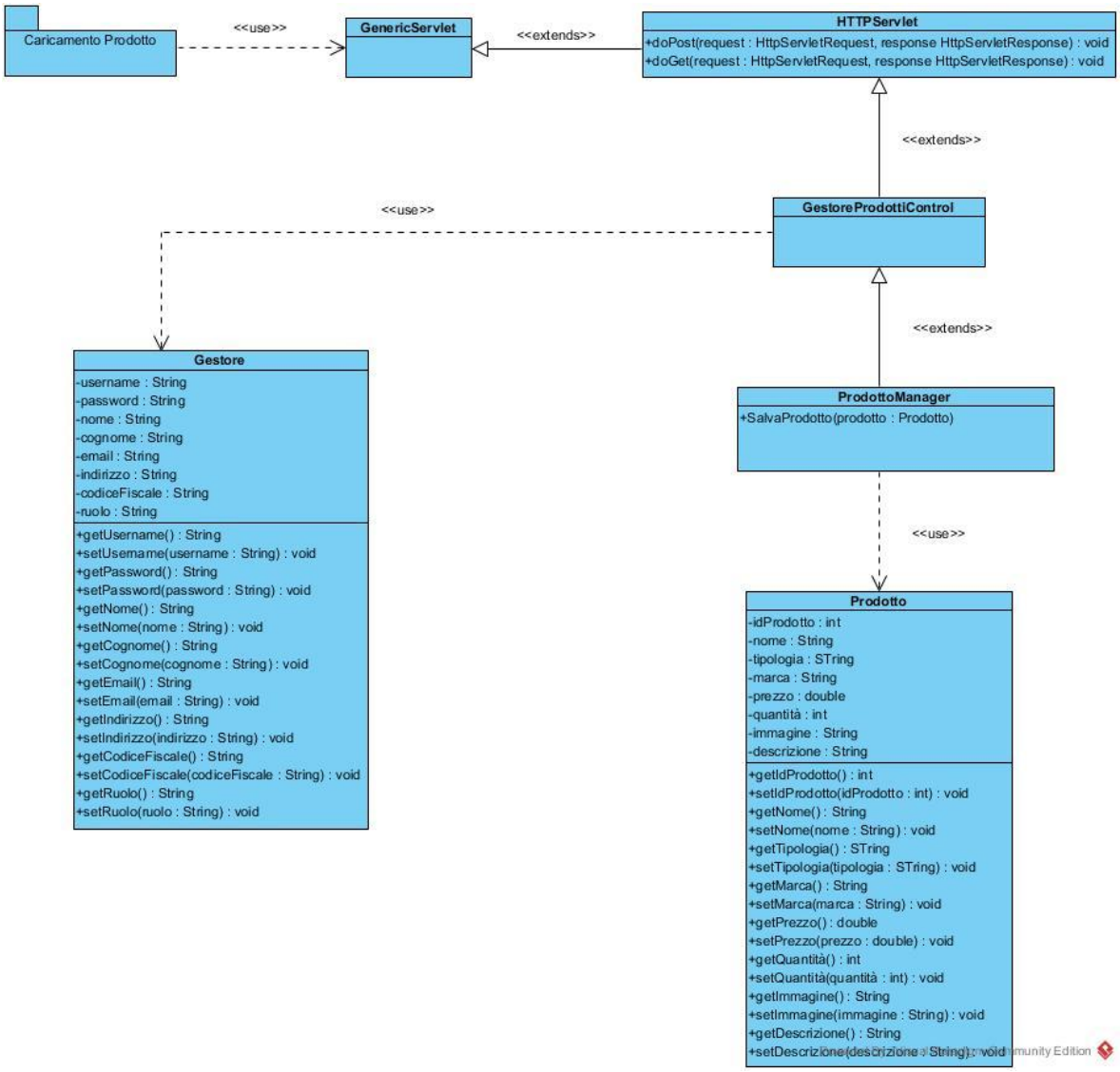
Nome classe	RiparazioneManager
Pre-condizione	**context** RiparazioneManager::public doSaveRiparazione(Prodotto p,Date data, String codiceCliente); **pre** p.idProdotto > 0 && data!=null, String codiceCliente!=null **context** RiparazioneManager::public doUpdateData(Date data, int idRiparazione); **pre** data != null && idRiparazione > 0 **context** RiparazioneManager::public doUpdateStato(String stato, int idRiparazione); **pre** data != null && idRiparazione > 0
Post-condizione	**context** RiparazioneManager::public doSaveRiparazione(Prodotto p, Date data, String codiceCliente); **post** doSaveRiparazione = true **context** RiparazioneManager::public doUpdateData(Date data, int idRiparazione); **post** doUpdateData = true **context** RiparazioneManager::public doUpdateStato(String stato, int idRiparazione); **post** doUpdateStato = true **context** RiparazioneManager::public doRetrieveAll(); **post** doRetrieveAll != null **context** RiparazioneManager::public doRetrieveAllRiparazioni(); **post** doRetrieveAllRiparazioni != null
Invarianti	

Nome classe	CarrelloManager
Pre-condizione	**context** CarrelloManager::public doInsertProdotti(Carrello carrello); **pre** carrello.id > 0 && carrello.lista != null && carrello.codiceFiscaleCliente != null **context** CarrelloManager::public doDeleteProdotti(Carrello carrello); **pre** carrello.id > 0 **context** CarrelloManager::public doRetrieveByKey(String codiceCliente); **pre** String codiceCliente!=null **context** CarrelloManager::public doPrenota (String codiceCliente, Prodotto prodotto); **pre** codiceCliene != null && prodotto.id > 0
Post-condizione	**context** CarrelloManager::public doInsertProdotti(Carrello carrello); **post** doInsertProdotti = true **context** CarrelloManager::public doDeleteProdotti(Carrello carrello); **post** doDeleteProdotti = true **context** CarrelloManager::public doRetrieveByKey(String codiceCliente); **post** doRetrieveByKey != null **context** CarrelloManager::public doPrenota(String codiceCliente, ProdottoInMagazzino prodotto); **post** doPrenota = true **context** CarrelloManager::public doRetrieveAllCode(String codiceCliente); **post** doRetrieveAllCode = true
Invarianti	

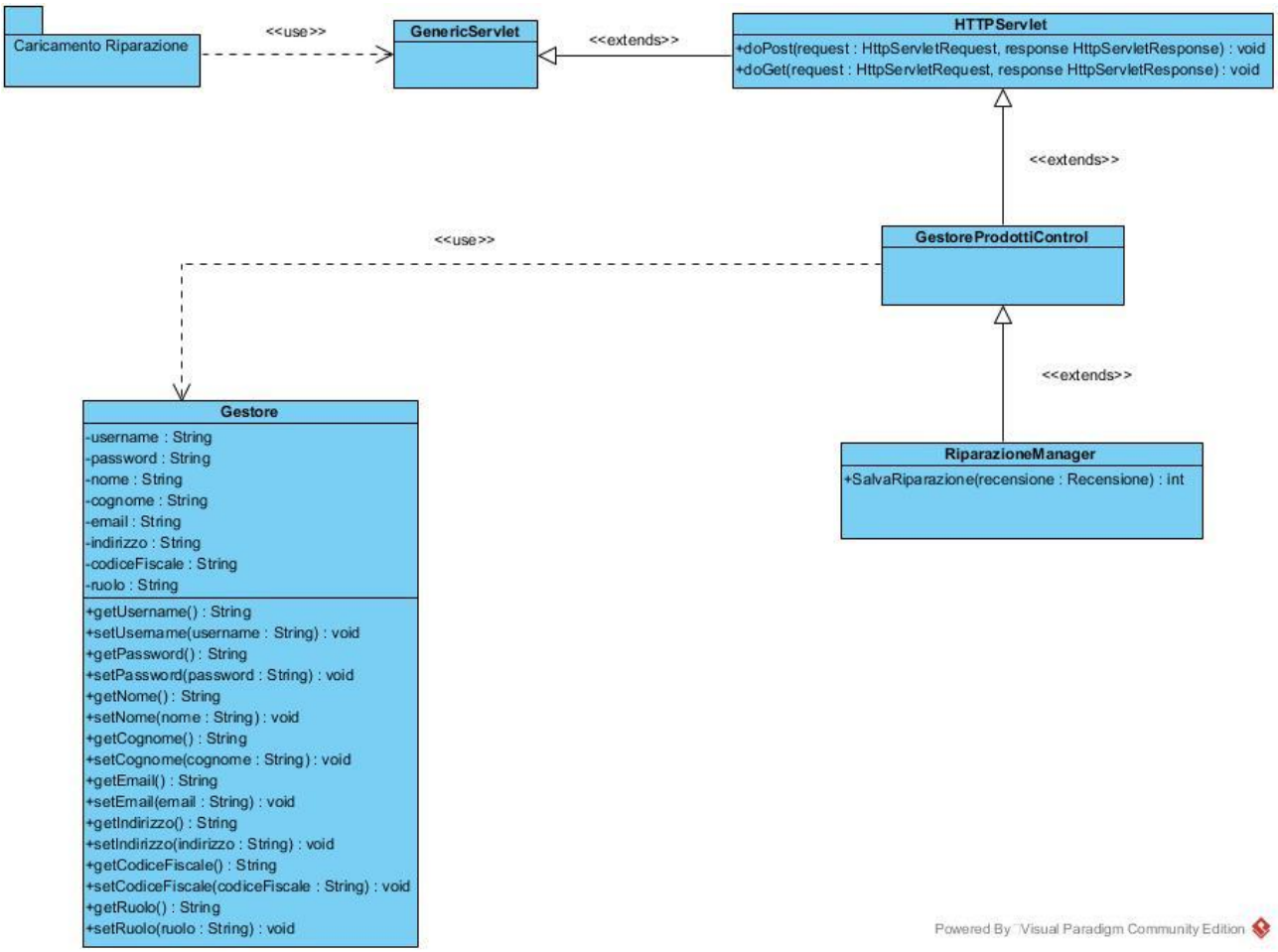
Nome classe	RecensioneManager
Pre-condizione	**context** RecensioneManager::public doSave(Recensione recensione); **pre** recensione.id > 0 && recensione.codiceCliente!=null && recensione.codiceProdotto > 0 && recensione.testo != null && recensione.voto > 0
Post-condizione	**context** RecensioneManager::public doSave(Recensione recensione); **post** doSave = true
Invarianti	

3.1 Class Diagram

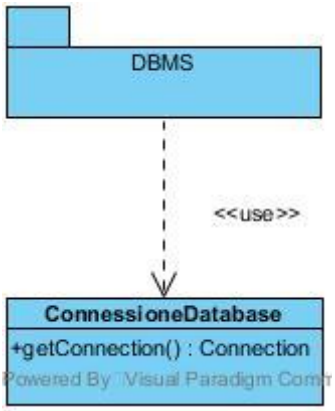
3.1.1 Caricamento prodotto



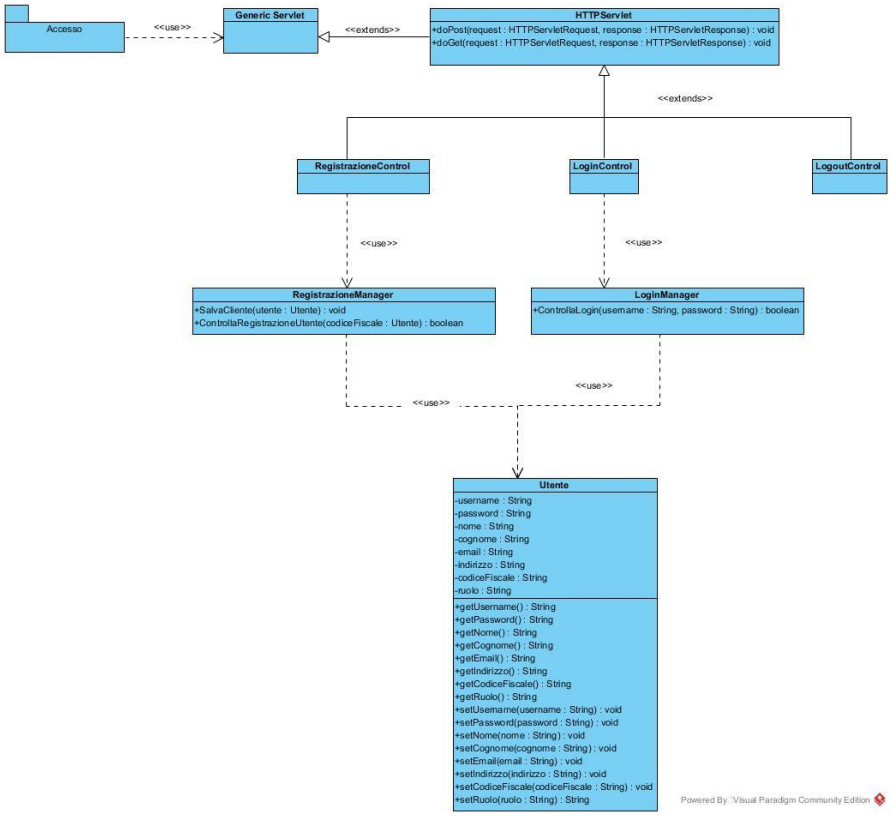
3.1.2 Caricamento riparazione



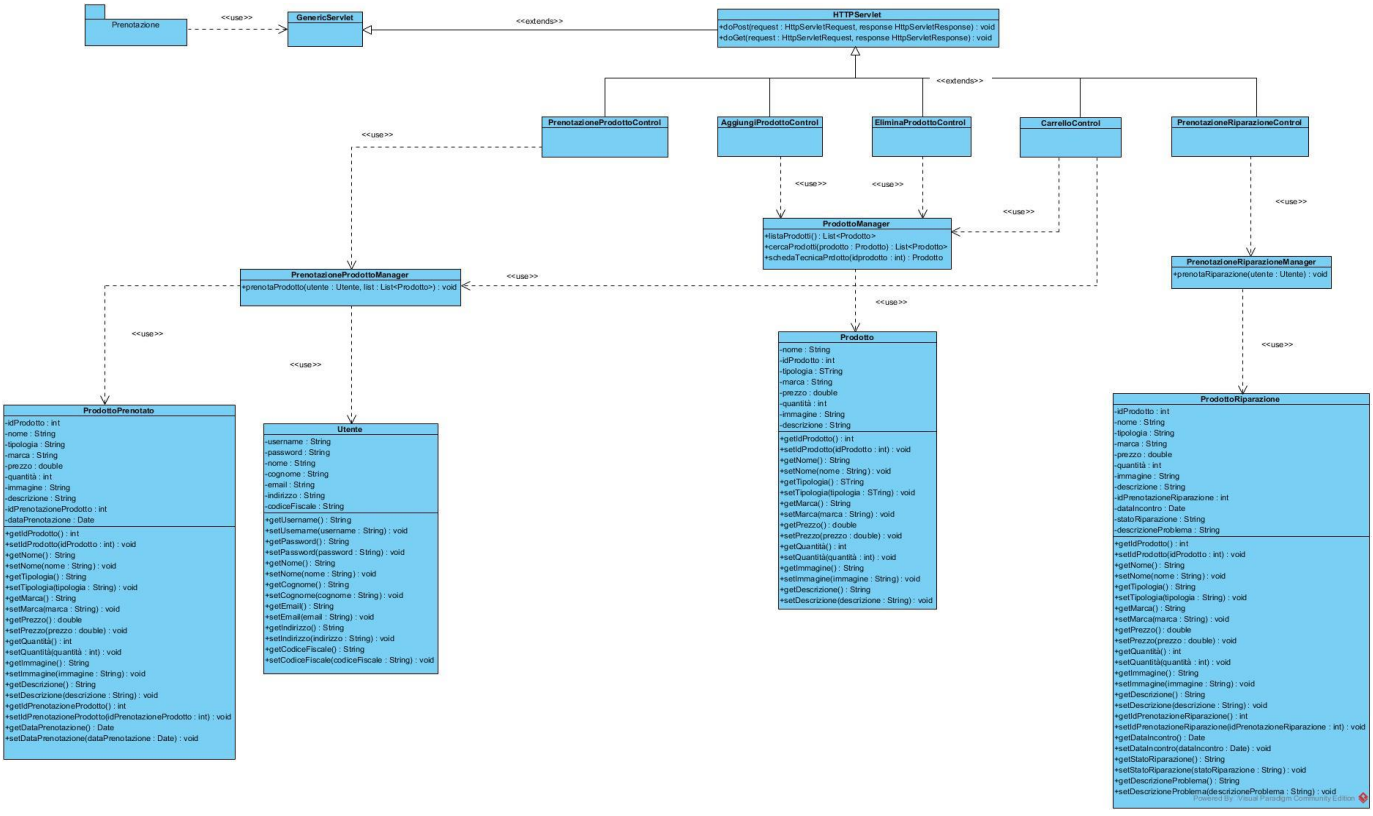
3.1.3 Connessione database



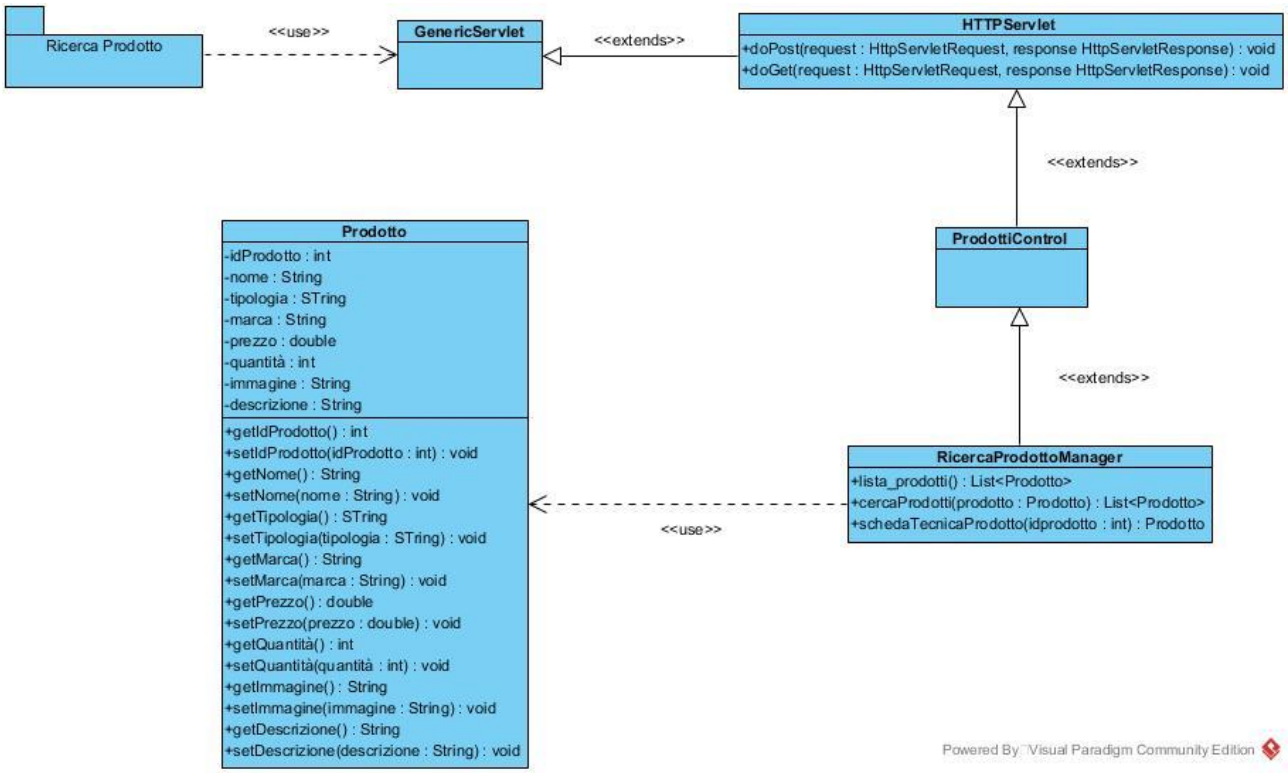
3.1.4 Login



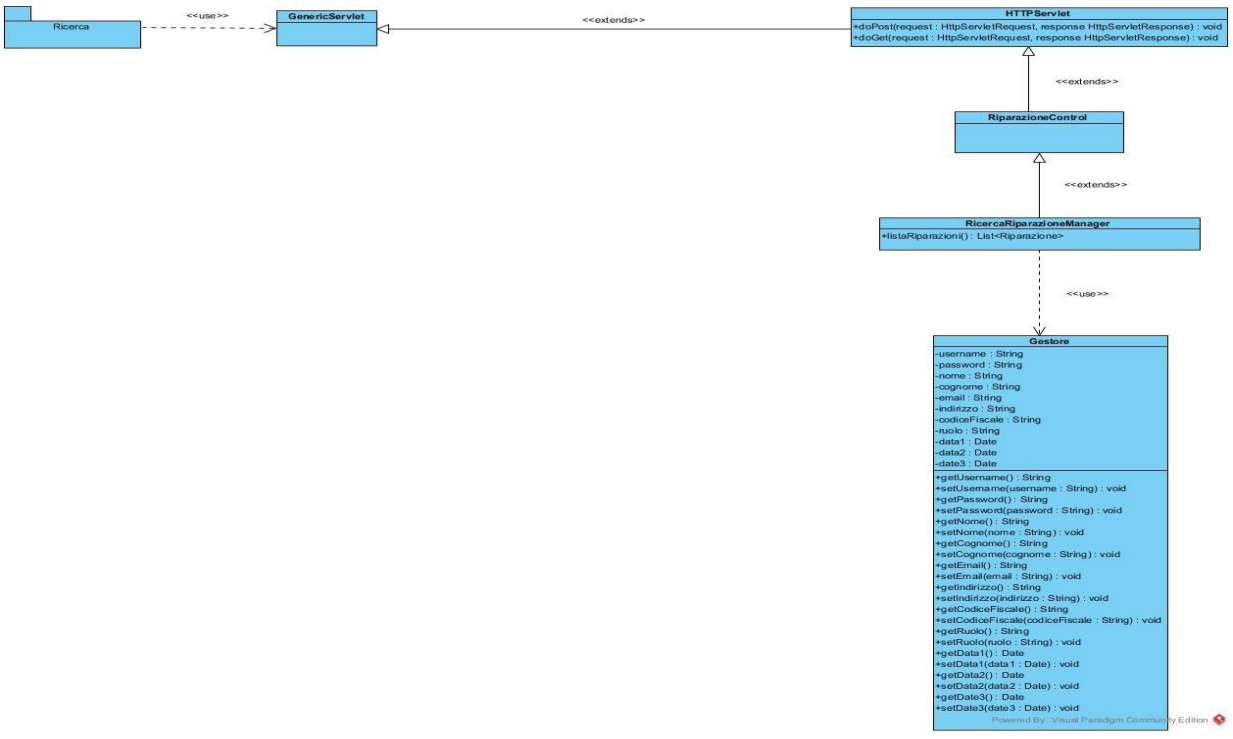
3.1.5 Prenotazione



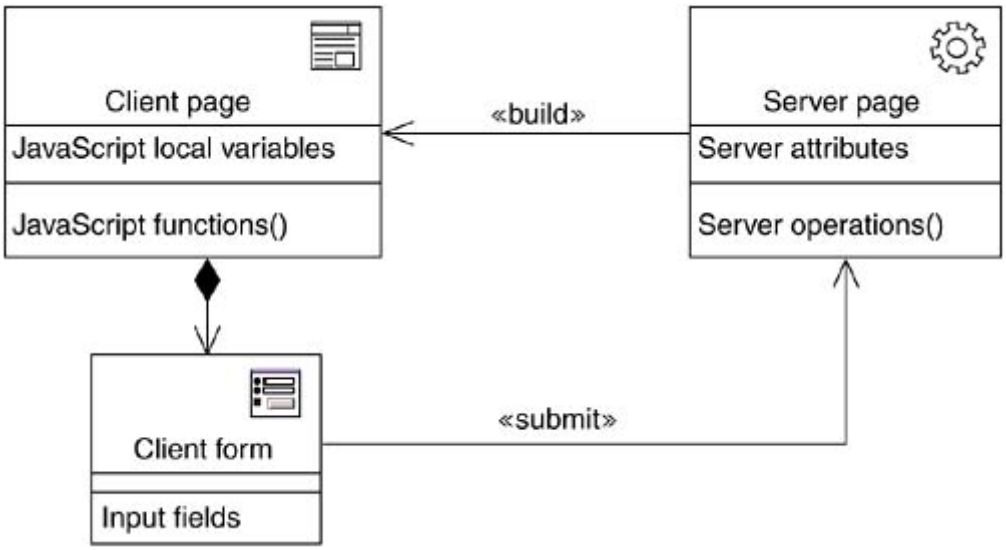
3.1.6 Ricerca prodotto

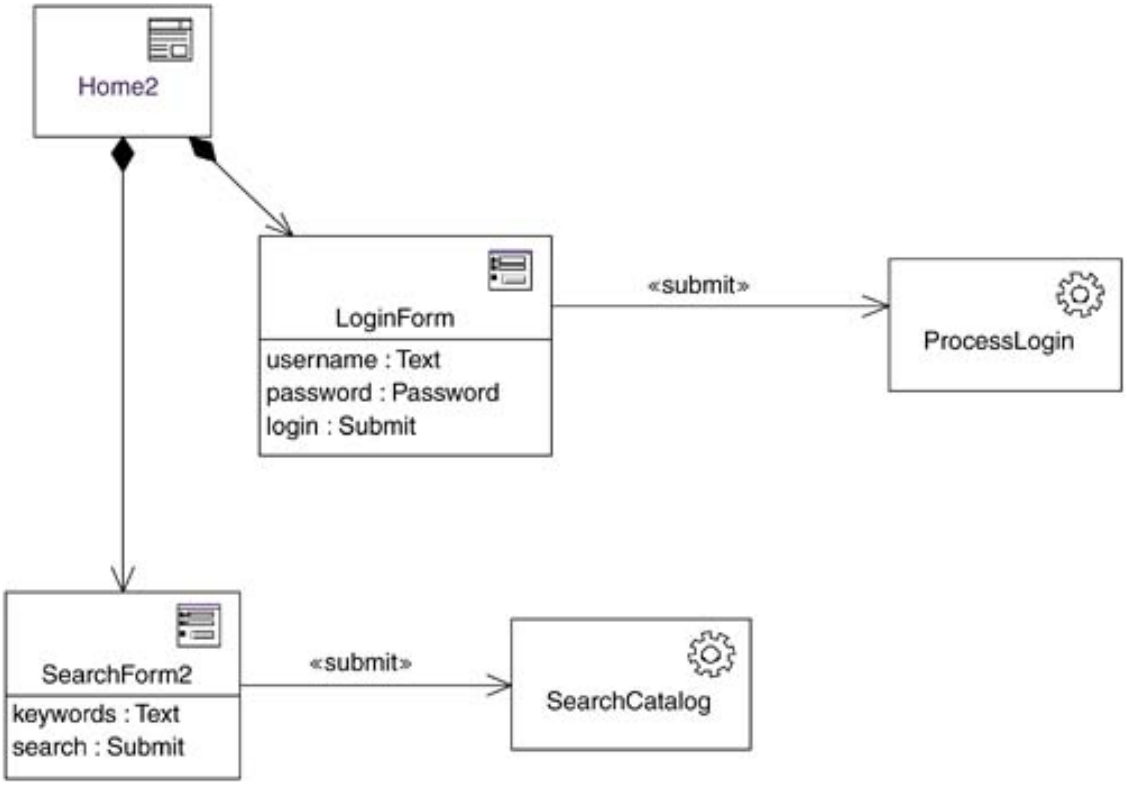


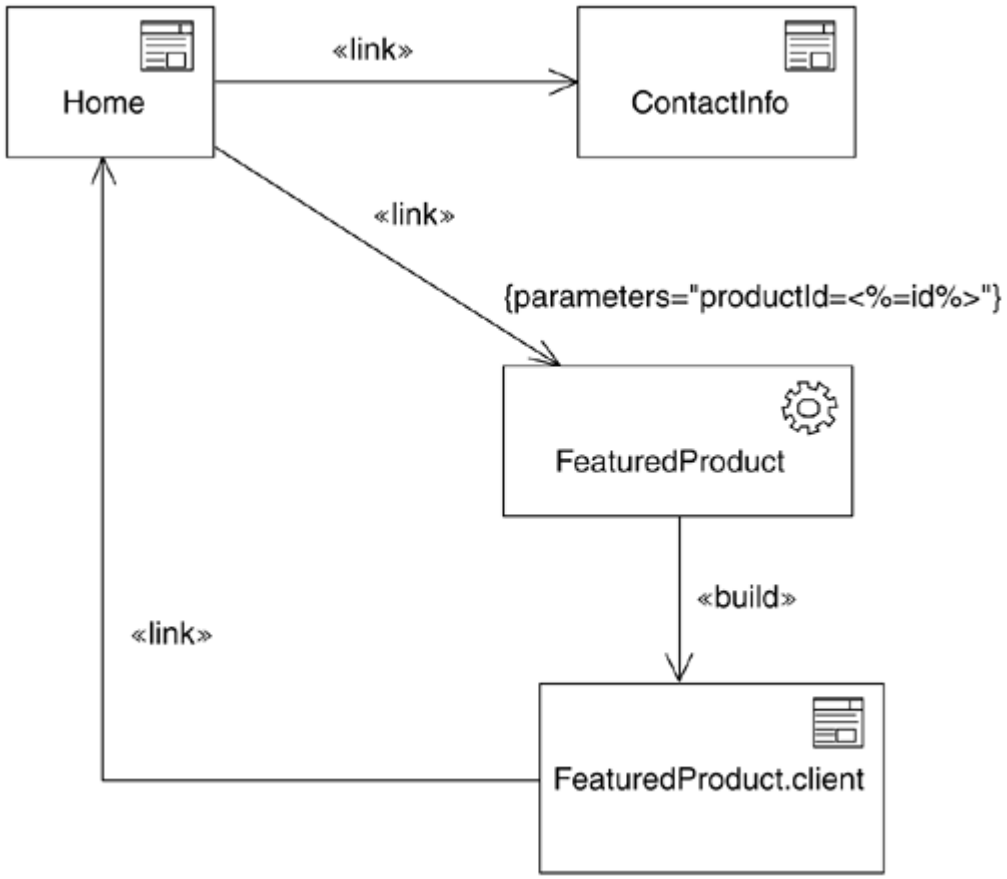
3.1.7 Ricerca riparazione

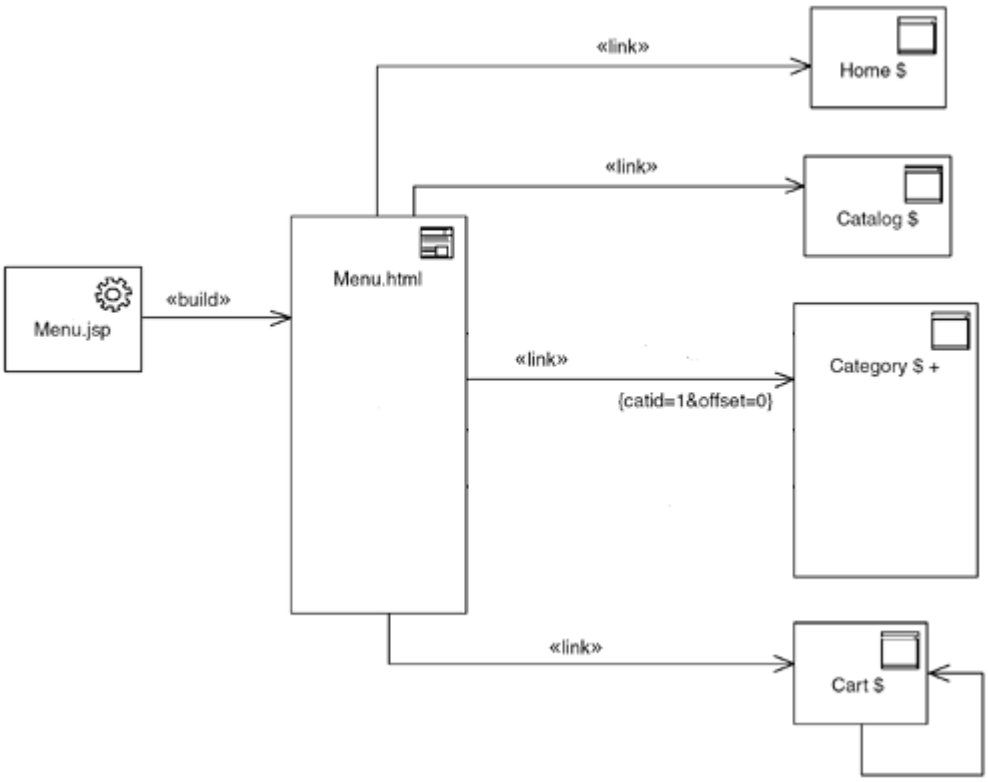


3.1.8 Web application extension





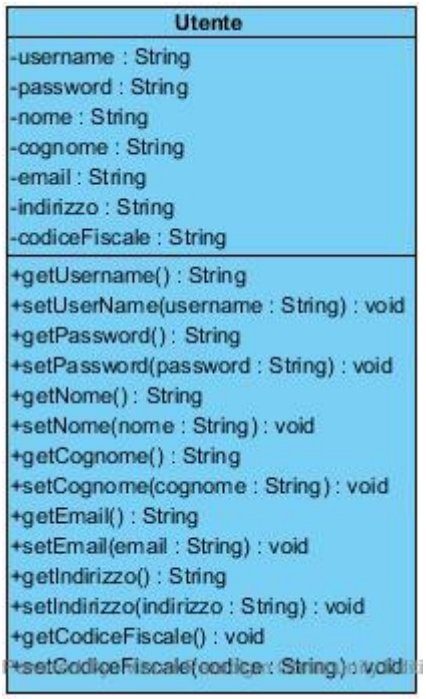

Home
«link»
ContactInfo
«link»
{parameters="productId=<%=id%>"}
FeaturedProduct
«build»
«link»
FeaturedProduct.client


«link»
Home $
«link»
Catalog $
Menu.html
Menu.jsp
«build»
«link»
{catid=1&offset=0}
Category $ +
«link»
Cart $

3.2 Descrizione delle classi

3.2.1 Utente

La classe contiene le informazioni relative ad un generico utente

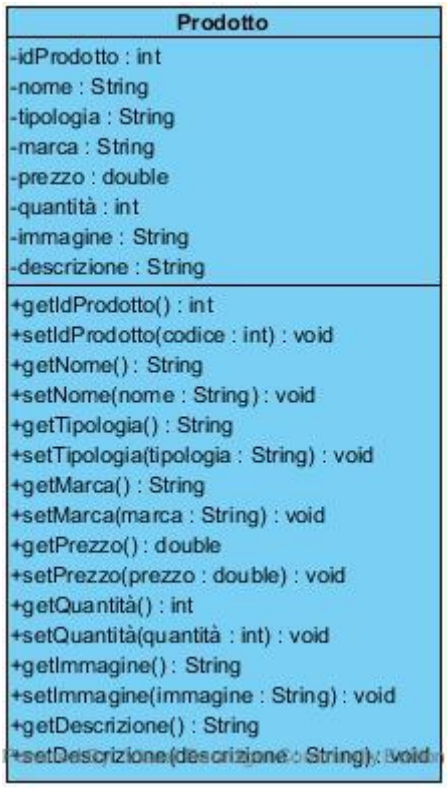


- Private username as string
  Username dell'utente
- Private password as string
  Password dell'utente
- Private nome as string
  Nome dell'utente
- Private cognome as string
  Cognome dell'utente
- Private email as string
  Email dell'utente
- Private indirizzo as string
  Indirizzo dell'utente
- Private 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

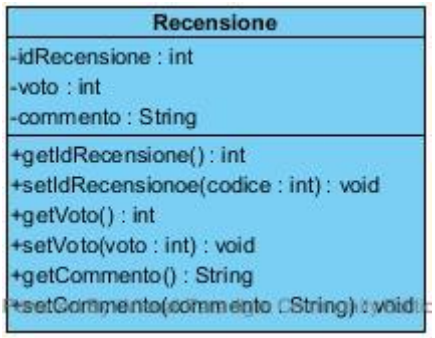


- Private idProdotto as int
  Indica l'Id del prodotto
- Private nome as string
  Nome del prodotto
- Private tipologia as string
  Tipologia del prodotto
- Private marca as string
  Marca del prodotto
- Private prezzo as double
  Indica il prezzo del prodotto
- Private quantità as int
  Indica la quantità 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

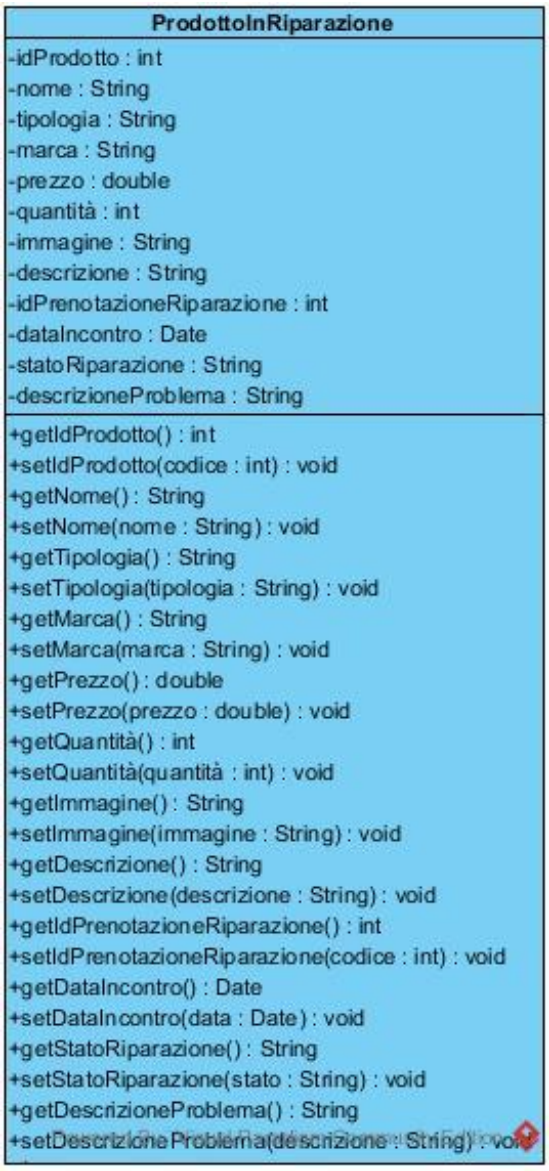


- Private idRecensione as int
  Indica l'id della recensione
- Private voto as string
  Voto della recensione
- Private commento as string
  Commento per uno specifico prodotto

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

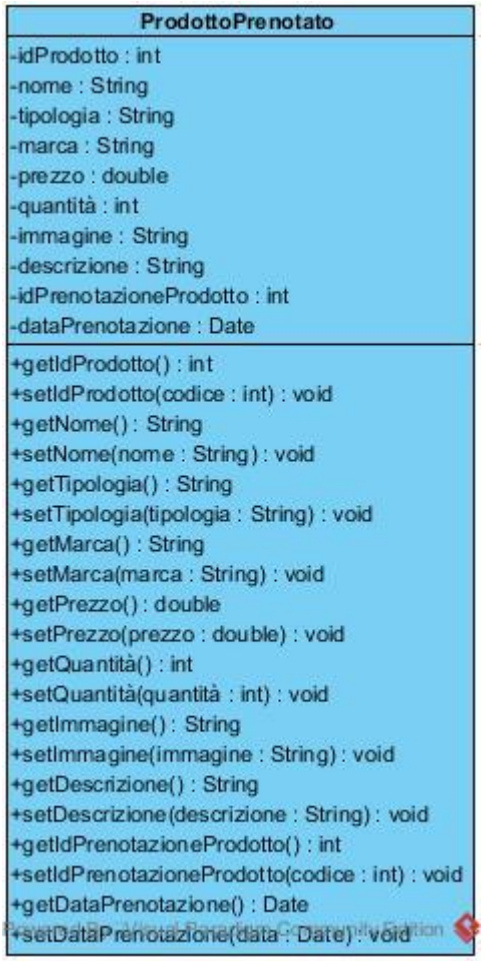
Descrizione del prodotto

- 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

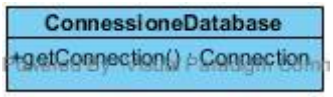
Email dell'utente

- 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



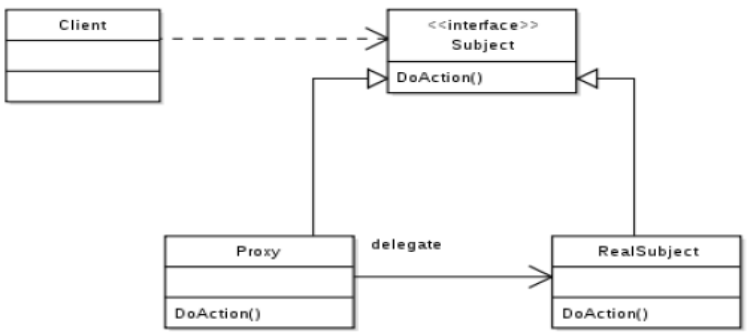
- Public getConnection() as Connection
  Serve per accedere al database

4. Design Pattern

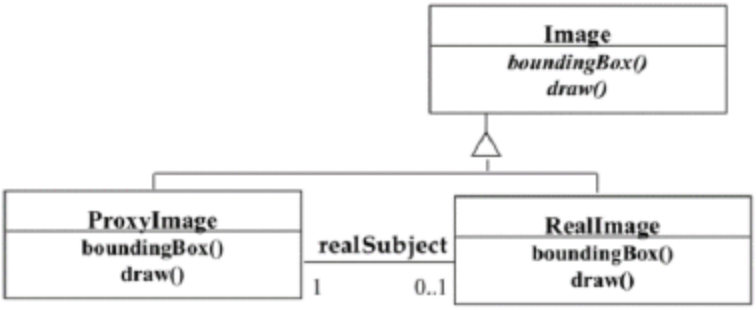
Proxy Pattern

Semplice ma allo stesso tempo molto utile, il Proxy Pattern nasce principalmente per risolvere problematiche legate all'accesso ad un oggetto che richiede tempi importanti per la creazione o per essere raggiunto.
Tale pattern prevede quindi la creazione di un oggetto "proxy" che viene usato al posto dell'oggetto reale e che quindi deve avere necessariamente la stessa "forma" dell'oggetto che sostituisce.
Insomma, crea una sorta di un "surrogato" (o segnaposto) per un altro oggetto di cui si desidera controllare l'accesso.
Ecco i principali benefici dati dal suo utilizzo:

- • Distanza tra "Concetto" ed Implementazione minimizzata.
- • Maggiore portabilità del codice del client. Le modifiche sono incapsulate nell'oggetto Proxy.



Avremo quindi senza dubbio vantaggio in termini di tempi di sviluppo e di eventuale modifica degli oggetti durante la programmazione, e al tempo stesso vi saranno vantaggi per gli utenti che utilizzeranno la piattaforma; dato che questo pattern consente di posticipare l'effettivo accesso alle risorse dell'oggetto quando è davvero necessario, consentendo maggiore fluidità all'elaborazione. Definito "proxy di sincronizzazione", questo pattern ci sarà ancora utile in quanto capace di regolare l'accesso ad un oggetto sottoposto a più richieste. Infine, ci avvaremo soprattutto del "proxy remoto", capace di avere l'accesso a risorse distribuite sulla rete come se fossero accessibili come oggetto locale (è il meccanismo Remote Methode Invocation di Java). Nello specifico, verranno gestite le foto dei docenti sulla piattaforma.



Le immagini verranno memorizzate e caricate separatamente dal testo. Se non viene caricata una "*RealImage*" (immagine reale), una "*ProxyImage*" visualizza un rettangolo grigio in luogo dell'immagine. Il client non può distinguere che si tratta di un "*ProxyImage*" invece di una "*RealImage".*

5. Glossario

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


DIPARTIMENTO DI INFORMATICA
UNISA.IT