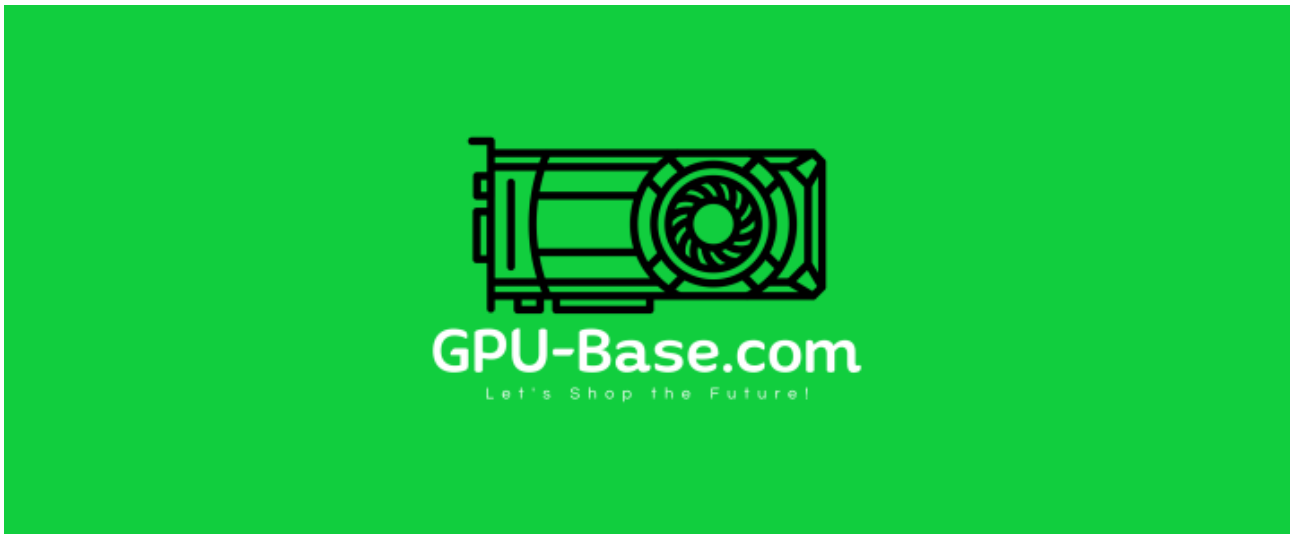
Università degli Studi di Salerno
Corso Ingegneria del Software

gpuBase
Object Design
Versione 1.9



Partecipanti:

Luigi Gallo	0512105704
Luigi Nacchia	0512105854
Giovanni Gaetano Nocerino	0512105974

INDICE

HYSTORY

Modifica	**Versione**	**Nome**	**Data**
Inserimento Object design trade offs	1.0	Nacchia	17/01/2021
Inserimento Componenti off-the-shelf	1.2	Gallo	18/01/2021
Inserimento Linee guida	1.4	Nocerino	19/01/2021
Inserimento Packages	1.6	Gallo	20/01/2021
Inserimento Test Control tables	1.8	Gallo	21/01/2021
Inserimento Testo Model tables	1.9	Nacchia	22/01/2021

1.1 Object design trade offs

Comprensibilità vs Tempo:

Il codice del sistema deve essere comprensibile, in modo da facilitare la fase di testing ed eventuali future modifiche da apportare.

Prestazioni vs Costi:

Dal momento che il budget allocato è spendibile principalmente in risorse umane e non consente l'acquisto di tecnologie proprietarie specifiche, verranno utilizzati template open source e componenti hardware di nostra proprietà

Interfaccia vs Usabilità:

Verrà realizzata un'interfaccia chiara e user friendly, usando form e pulsanti predefiniti che hanno lo scopo di rendere semplice l'utilizzo del sistema da parte dell'utente finale.

Sicurezza vs Efficienza:

La sicurezza per via di tempi di sviluppo molto limitati, ci limiteremo ad implementare sistemi di sicurezza basati su filtri, e-mail e password criptate in SHA-56.

1.2 Componenti off-the-shelf

Per l'implementazione del sistema utilizzeremo componenti software già disponibili e utilizzati per facilitare la creazione del software.

• Bootstrap, framework open source utilizzato per sviluppare progetti web. Verrà utilizzato insieme ad HTML, CSS e JavaScript.

• jQuery, libreria JavaScript che facilita la scrittura di scrip rendendo semplice la selezione e la manipolazione di elementi del DOM in pagine HTML.

• AJAX, tecnica di sviluppo software per la realizzazione di applicazioni web interattive che utilizzano scambi tra web browser e server. Consente l'aggiornamento dinamico di una pagina web senza caricamento esplicito da parte dell'utente.Verrà utilizzato insieme all'estensione JSON.

• JSON, formato di dati adatto allo scambio di informazioni in applicativi client/server. Usato in AJAX tramite l'API XHRHttp

• Selenium,una suite di tool utilizzati per automatizzare i test di sistema eseguendoli sul web browser.

• Junit, un framework di programmazione Java che viene utilizzato per implementare i test di unità e

1.3 Linee guida

Gli sviluppatori dovranno seguire precise linee guida per la stesura del codice:

Package

Il progetto verrà sviluppato con L'IDE Eclipse, sarà strutturato come segue:

• Il progetto avrà 3 package; model, control e test, i quali contengono i corrispettivi subpackage con rispettive classi.

Naming Convention

Per la documentazione delle interfacce bisognerà utilizzare nomi:

• Descrittivi;

• Pronunciabili;

• Di lunghezza medio-corta;

• Caratteri consentiti (a-z,A-Z,0-9)

Variabili

I nomi delle variabili dovranno seguire l'annotazione "Camel Notation", quindi prima lettera minuscola e le successive parole con l'iniziale maiuscola. Tipo camelCase.

In ogni riga dovrà esserci un'unica variabile dichiarata, eventualmente allineata con quelle del blocco dichiarativo.

Costanti

I nomi delle costanti dovranno seguire l’annotazione “CONSTANT_CASE”, tutte le lettere maiuscole, le parole saranno separate da underscore “_”.

Le costanti di classe dovranno essere dichiarate all’inizio della classe.

Metodi

I nomi dei metodi dovranno seguire la notazione “Camel Notation”, quindi prima lettera minuscola e le successive parole con l’iniziale maiuscola. Tipo camelCase.

Il nome del metodo sarà costituito da un verbo che ne identifica l’azione seguito da un sostantivo, eventualmente aggettivato.

Classi Java

I nomi delle classi seguiranno le convenzioni della OOP detta anche “Pascal Case”(camelCase con prima lettera maiuscola), quindi prima lettera maiuscola e le successive parole con l’iniziale maiuscola.

I nomi delle classi dovranno essere semplici, descrittivi e inerenti al dominio applicativo. Vietato usare underscore.

I nomi dei metodi accessori e modificatori seguiranno i pattern standard della OOP, rispettivamente saranno, getNomeVariabile e setNomeVariabile.

Le classi saranno strutturate nel seguente ordine:

- Dichiarazione della classe pubblica;
- Dichiarazioni di costanti;
- Dichiarazioni di variabili di classe;
- Dichiarazioni di variabili di istanza;
- Costruttore;
- Metodi pubblici;
- Metodi di servizio privati;

Pagine JSP

I nomi delle pagine JSP dovranno seguire la notazione “camelCase” descritta per i metodi delle classi Java.

Le pagine JSP quando eseguite dovranno produrre un documento conforme allo standard HTML5. Il codice java presente nelle JSP deve aderire alle convenzioni descritte precedentemente.

- Il tag di apertura (<%) dovrà essere seguito da un invio a capo;
- Il tag di chiusura (%>) dovrà trovarsi all’inizio della riga;
- Il codice tra i tag dovrà essere indentato;

- Nel caso di singola istruzione le tre regole precedenti possono essere evitate;

Pagine HTML

Le pagine HTML devono essere conformi allo standard HTML5 e il codice deve essere indentato, per facilitare lettura e modifiche, secondo le seguenti regole:

- Un’indentazione consiste in una tabulazione;

- Ogni tag deve essere un’indentazione maggiore del tag che lo contiene;

- Ogni tag di chiusura deve avere lo stesso livello di indentazione del corrispondente tag di apertura;
- I tag di commento devono seguire le stesse regole applicate ai tag normali;

- Per i tag che contengono breve testo è possibile mantenere tag di apertura e chiusura sulla stessa riga;
- Lo stile delle pagine deve essere impostato da fogli di stile CSS esterni;

- Gli script dovranno essere importati, a meno che non siano script di poche righe;

Script

Gli script dovranno essere scritti in JavaScript o in JQuery, dovranno essere ben indentati, di facile lettura e commentati.

I nomi dei file degli script dovranno seguire la notazione “camelCase”

Fogli di stile CSS

I fogli di stile dovranno essere formattati come segue:

- I selettori della regola dovranno trovarsi sulla stessa riga;
- L’ultimo selettore della regola è seguito dalla parentesi graffa aperta “{“;
- Le proprietà che costituiscono la regola saranno una per riga e sono indentate rispetto ai selettori;
- La regola è terminata da una graffa chiusa “}” collocata da sola su una sola riga.

Database SQL

- I costrutti sql devono essere scritti con sole lettere maiuscole
- I nomi delle tabelle devono essere costituiti da solo lettere minuscole.
- I nomi devono appartenere al dominio del problema ed esplicare correttamente ciò che intendono rappresentare.
- I nomi delle colonne delle tabelle devono seguire la notazione “camelCase”, anche loro devono esplicare correttamente la parte del dominio del problema che intendono rappresentare.

Design Pattern MVC

Il design pattern MVC consente la suddivisione del sistema in tre blocchi principali: Model, View e Controller. Il Model modella i dati del dominio applicativo e fornisce i metodi di accesso ai dati persistenti, il View si occupa della presentazione dei dati all’utente e di ricevere da quest’ultimo gli input, infine il Controller riceve i comandi dell’utente attraverso il View e modifica lo stato di quest’ultimo e del Model

DAO(Data Access Object) Pattern

Il DAO pattern è utilizzato per il mantenimento di una rigida separazione tra le componenti Model e Controller, in questo tipo di applicazioni basate sul paradigma MVC.

Data-Access Object:classe che implementa i metodi degli oggetti che rappresenta.
Classi Bean:classi che contengono i getters/setters degli oggetti che rappresentano e saranno usati dai DAO.

2.Packages

View

Nel folder WebContent contiene le JSP per la visualizzazione delle pagine.
Le JSP sono divise in base alle funzionalità che offrono::

- La **SellerView.jsp** è la pagina dedicata al venditore. In questa pagina il venditore può aggiungere un prodotto al catalogo ,modificare lo stato di un ordine (evadere l'ordine) e modificare le caratteristiche di un prodotto (es. Prezzo,quantità)
- La **UserView.jsp** (*not Implemented)* è la pagina dedicata all'area utente e solo l'utente registrato può accedervi. In questa pagina l'utente registrato può visionare i propri dati personali,controllare lo stato degli ordini effettuati e stampare una ricevuta degli ordini evasi.
- L**'Header.jsp** è una componente comune di tutte le view che visualizza scorciatoie verso altre pagine sotto forma di menu e facilita la navigazione del sito.
- Il **Footer.jsp** è uan componente comune di tutte le view che visualizza informazioni di contatto del negozio fisico come link ai socialnetwork e indirizzo e numero di cellulare del negozio.
- La **HomeView.jsp** è una pagina in cui gli utenti possono accedere per visualizzare i prodotti in vendita ed eventualmente aggiungerli al carrello
- La **ChartView.jsp** è una pagina in cui gli utenti possono controllare i prodotti aggiunti precedentemente al carrello e gestirne la quantità.
- La **OrderView.jsp** (*not Implemented)* è una pagina che permette all'utente di effettuare l'ordine e ne mostra l'esito . Sarà richiesto di inserire i dati di pagamento per ogni ordinazione. I dati di pagamenti non saranno inseriti nel database.
- La **PaymentView.jsp** (*not Implemented)*è una componente della pagina OrderView che gestisce l'acquisizione dei dati di pagamento.
  (è racchiusa in OrderView)
- La **RegistrationView.jsp** (*not Implemented)*è una pagina che permette al utente non registrato di registrarsi al sito inserendo mail,password,cellulare e indirizzo di spedizione.
- La **LoginView.jsp** (*not Implemented)* è una pagina che permette all'utente di effettuare il login per autenticarsi tramite mail e password.

Model

- **VenditoreDAO.java** è una classe che interagisce con l'entità "Seller", contenuta sul DataBase, e si occupa di fornire la funzionalità di aggiornamento dati personali degli amministratori, ed è utilizzato per effettuare controlli di autenticazione di questi ultimi.
- **UtenteDAO.java** è una classe che interagisce con l'entità "Utente", contenuta sul DataBase e si occupa di fornire la funzionalità di aggiornamento dei dati personali dello stesso. Viene utilizzato per effettuarne, inoltre i controlli sull'autenticazione.
- **ProdottoDAO.java** è una classe che interagisce con l'entità "Prodotto" contenuta sul DataBase e si occupa di fornire le funzionalità di aggiunta del prodotto (senza possibilità di eliminazione, in quanto i prodotti sono legati agli ordini.) e di modifica dei dettagli dello stesso.
- **OrdineDAO.java** è una classe che interagisce con l'entità "Ordine", contenuta sul DataBase, e contiene i dati relativi ai prodotti che sono stati acquistati, oltre alla data di acquisto e lo stato stesso dell'ordine. Ogni ordine rappresenta un singolo prodotto. Quando un utente effettua una ordinazione, di fatto effettua una serie di ordini singoli per ciascun prodotto acquistato (indipendentemente dalla quantità di ogni tipo di prodotto) .
  La relazione tra l'utente e l'insieme dei singoli ordini, è rappresentato da un "Ordinazione".
- **OrdinazioneDAO.java** è una classe che interagisce con l'entità "Ordinazione", contenuta sul DataBase, ed è una entità intermedia che rappresenta la relazione tra utenti ed ordini. Ogni ordinazione può essere collegata a più ordini, e di fatto, rappresenta l'insieme dei singoli ordini dei prodotti acquistati dal singolo utente.

Control

- **SellerControl.java** è una classe che interagisce col **VenditoreDAO.java**, e offre funzioni di registrare, leggere e modificare i dati personali del venditore su richiesta di **SellerView.jsp**.
- **UserControl.java** è una classe che interagisce con **UtenteDAO.java** e offre funzioni di registrare, leggere e modificare i dati personali dell'utente registrato su richiesta di **UserView.jsp.**
- **ChartControl.java** è una classe che interagisce con **CarrelloDAO.java** e offre funzioni di aggiungere prodotti al carrello ,eliminare prodotti dal carrello e modificarne le quantità su richiesta di **ChartView.jsp**.
- **OrderControl.java** è una classe che interagisce con **OrrdinazioneDAO** e **OrdineDAO.java** ed offre funzioni per effettuare l'ordine dei prodotti nel carrello e ne permette la lettura e l'evasione.Le richiesta di effettuare l'ordine dei prodotti , la richiesta di effettuarne la lettura e l'evasione vengono fatte da **SellerView.jsp** ; Inoltre una richiesta di lettura può essere fatta anche da **UserView.jsp**.
- **ProductControl.jsp** è una classe che interagisce con **ProdottoDAO.java** e offre funzioni per effettuare modifiche,inserimenti e lettura . Modifica ,inserimento e lettura saranno richieste dalla **SellerView.java** ; la lettura sarà richiesta anche da **HomeView.jsp**.

3.Test

Questo package contiene i test di unità del sistema in due package e la test suite:

- **AllTestsControl & AllTestsDAO:**sono le test suite.
- **control:**è il package che contiene i test delle servlet seguendo la tecnica **blackbox**
- **model**: è il package che contiene i test dei dao.

TestControl:

Nome Classe	`addToChart`
Descrizione	Permette di aggiungere un prodotto al carrello.
Pre-condizione	request.getSession().getAttribute("doingTest").equals(“true”)&& request.getParameter("action").equals(“addtoChart”)&& request.getParameter("idProdotto").equals(“1”)&& request.getSession().getAttribute("productChartList").size()>0;
Post-condizione	ChartControl::doPost(request, response);

Nome Classe	`updatePiecesChart`
Descrizione	Permette di aggiornare il numero di pezzi di un determinato prodotto presente nel carrello.
Pre-condizione	request.getSession().getAttribute("doingTest").equals("true")&& request.getParameter("idProdotto").equals(“1”)&& request.getParameter("action").equals("updatePiecesChart")&& request.getParameter("pezzi").equals("10") && request.getSession().getAttribute("productChartList").size()>0;
Post-condizione	ChartControl::doPost(request, response);

Nome Classe	`deleteFromChart`
Descrizione	Permette di eliminare un prodotto dal carrello.
Pre-condizione	request.getSession().getAttribute("doingTest").equals("true")&& request.getParameter("action").equals("deleteFromChart")&& request.getSession().getAttribute("loginType") .equals("registeredUser")&& request.getParameter("idProdotto").equals(1+"")&& request.getSession().getAttribute("productChartList").size()>0;
Post-condizione	ChartControl::doPost(request, response);

Nome Classe	performOrder
Descrizione	Permette di effettuare un ordinazione di tutti i prodotti presenti nel carrello.
Pre-condizione	request.getSession().getAttribute("doingTest").equals("true")&& request.getParameter("action").equals("performOrder")&& request.getSession().getAttribute("loginType") .equals("registeredUser")&& request.getParameter("idOrdinazione").equals(1+"")&& request.getSession().getAttribute("productChartList").size()>0;
Post-condizione	OrderControl::doPost(request, response);

Nome Classe	evadeOrder
Descrizione	Permette al venditore di evadere un ordine.
Pre-condizione	request.getSession().getAttribute("doingTest").equals("true")&& request.getParameter("action").equals("evadeOrder")&& request.getSession().getAttribute("loginType") .equals("registeredUser")&& request.getParameter("idOrdinazione").equals(1+"")&& request.getSession().getAttribute("productChartList").equals (productChartList);
Post-condizione	OrderControl::doPost(request, response);

Nome Classe	getAllProducts
Descrizione	Restituisce tutti i prodotti presenti nel catalogo
Pre-condizione	request.getSession().getAttribute("doingTest").equals("true")&& request.getParameter("action").equals("getAllProducts")&& request.getParameter("idProdotto").equals("1");
Post-condizione	ProductControl::doPost(request, response);

Nome Classe	deleteProduct
Descrizione	Permette di eliminare un prodotto dal catalogo.
Pre-condizione	request.getSession().getAttribute("loginType").equals("seller")&& request.getSession().getAttribute("doingTest").equals("true")&& request.getParameter("idProdotto").equals(1+"")&& request.getParameter("action").equals("deleteProduct");
Post-condizione	ProductControl::doPost(request, response);

Nome Classe	`insertProduct`
Descrizione	Permette di inserire un prodotto nel catalogo
Pre-condizione	request.getSession().getAttribute("loginType").equals("seller")&& request.getSession().getAttribute("doingTest").equals("true")&& request.getParameter("action").equals("insertProduct")&& request.getParameter("idProdotto").equals(1+"")&& request.getSession().getAttribute("loginMail").equals("userTest1")&& request.getParameter("nome").equals("name")&& request.getParameter("descrizione").equals("desc")&& request.getParameter("numeroPezzi").equals("1")&& request.getParameter("prezzo").equals("1");
Post-condizione	ProductControl::doPost(request, response);

Nome Classe	`updateProduct`
Descrizione	Permette di aggiornare un prodotto contenuto nel catalogo
Pre-condizione	request.getSession().getAttribute("loginType").equals("seller")&& request.getSession().getAttribute("doingTest").equals("true")&& request.getParameter("action").equals("insertProduct")&& request.getParameter("idProdotto").equals(1+"")&& request.getSession().getAttribute("loginMail").equals("userTest1")&& request.getParameter("nome").equals("name")&& request.getParameter("descrizione").equals("desc")&& request.getParameter("numeroPezzi").equals("1")&& request.getParameter("prezzo").equals("1");
Post-condizione	ProductControl::doPost(request, response);

Nome Classe	getProductFoto
Descrizione	Restituisce la foto di un dato prodotto

Pre-condizione	`request.getSession().getAttribute("doingTest").equals("true")&&`
	`request.getParameter("action").equals("getProductFoto")&&`
	`request.getParameter("idProdotto").equals(1+"");`

Post-condizione	ProdottoDAO::doPost(request, response);

TestModel:

Nome Classe	OrdineDAO.DoSave
Descrizione	Permette di salvare un ordine preso in input nel database.
Pre-condizione	UserDao!=null&&userBean!=null&& sellerDao!=null&&sellerBean!=null&& prodottoDao!=null&&prodottoBean!=null&& ordinazioneDao!=null&&ordinazioneBean!=null&& testDao!=null&&testBean!=null;
Post-condizione	OrdineDAO:: testDao.doRetrieveAll().contains(testBean)

Nome Classe	OrdineDAO.RetriveByKey
Descrizione	Ci fa ottenere un ordine con un dato id preso in input dal database(se quest'ultimo e' presente).
Pre-condizione	TestDao!=null && testBean!=null
Post-condizione	OrdineDAO:: testBean.equals(testDao.doRetrieveByKey(testBean.getIdOrdine()));

Nome Classe	OrdineDAO.RetriveAll
Descrizione	Permette di ottenere tutti gli ordini memorizzati nel database.
Pre-condizione	collection = new ArrayList<OrdineBean>();
Post-condizione	OrdineDAO::!(collection.equals(testDao.doRetrieveAll()));

Nome Classe	OrdineDAO.DoDelete
Descrizione	Permette di eliminare un ordine(contenuto nel database) con un dato id .
Pre-condizione	TestBean!=null&& ordinazioneBean!=null&& prodottoBean!=null&& sellerBean!=null&& userBean!=null;
Post-condizione	OrdineDAO::testDao.doDelete(testBean.getIdOrdine()).equals(true);

Nome Classe	OrdinazioneDAO.DoSave
Descrizione	Permette di salvare un ordinazione presa in input nel database.
Pre-condizione	venditoreBean!=null&& testBean!=null;
Post-condizione	OrdinazioneDAO:: testDao.doRetriveAll().contains(testBean);

Nome Classe	OrdinazioneDAO.RetriveByKey
Descrizione	Ci fa ottenere un ordinazione con un dato id preso in input dal database(se quest'ultimo e' presente).
Pre-condizione	`TestDao!=null && testBean!=null`
Post-condizione	OrdinazioneDAO:: testBean.equals(testDao.doRetriveByKey((testDao.doRetrieveAll(),size())));

Nome Classe	OrdinazioneDAO.RetriveAll
Descrizione	Permette di ottenere tutti le ordinazioni memorizzatenel database.
Pre-condizione	voidList = new LinkedList<ProdottoBean>();
Post-condizione	OrdinazioneDAO::!(voidList.equals(testDao.doRetrieveAll()));

Nome Classe	OrdinazioneDAO.DoEvade
Descrizione	Permette al venditore di evadere l'ordinazione aggiornandone lo stato.
Pre-condizione	TestBean!=null&& userBean!=null testDao!=null;
Post-condizione	OrdinazioneDAO:: !(testDao.doRetriveByKey(testBean.getIdOrdinazione()).equals(testBean));

Nome Classe	OrdinazioneDAO.DoDelete
Descrizione	Permette di eliminare un ordinazione(contenuta nel database) con un dato id .
Pre-condizione	TestBean!=null&& userBean!=null testDao!=null;
Post-condizione	OrdinazioneDAO:: testDao.doDelete(testBean.getIdOrdinazione()).equals(true);

Nome Classe	ProdottoDAO.DoSave
Descrizione	Permette di salvare un prodotto preso in input nel database.
Pre-condizione	venditoreBean!=null&& testBean!=null;
Post-condizione	ProdottoDAO:: testDao.doRetrieveAll().contains(testBean);

Nome Classe	ProdottoDAO.RetriveByKey
Descrizione	Ci fa ottenere un prodotto ,con un dato id preso in input, dal database(se quest'ultimo e' presente).
Pre-condizione	TestDao!=null && testBean!=null
Post-condizione	ProdottoDAO:: testBean.equals(testDao.doRetriveByKey((testDao.doRetrieveAll(),size())));

Nome Classe	ProdottoDAO.RetriveAll
Descrizione	Permette di ottenere tutti le ordinazioni memorizzate nel database.
Pre-condizione	voidList = new LinkedList<ProdottoBean>();
Post-condizione	ProdottoDAO::!(voidList.equals(testDao.doRetrieveAll()));

Nome Classe	ProdottoDAO.DoUpdatePezzi
Descrizione	Permette di aggiornare il numero di pezzi di un certo prodotto nel catalogo.
Pre-condizione	pezzi!=null && testDao!=null&& testBean1=null;
Post-condizione	ProdottoDAO:: (testDao.doRetrieveByKey(testBean.getIdProdotto()).getNumeroPezzi()).equals(pezziPrec+1);

Nome Classe	ProdottoDAO.DoUpdate
Descrizione	Permette di aggiornare le caratteristiche di un certo prodotto nel catalogo.
Pre-condizione	TestDao!=null&& testBean!=null
Post-condizione	ProdottoDAO:: (testDao.doRetrieveByKey(testBean.getIdProdotto()).getDescrizione().equals(buf.getDescrizione()))

Nome Classe	ProdottoDAO.DoDelete
Descrizione	Permette di eliminare un certo prodotto nel catalogo.
Pre-condizione	TestDao!=null&& testBean!=null&& venditoreDAO;
Post-condizione	ProdottoDAO:: (testDao.doDelete(testBean.getIdProdotto()))==true;

Nome Classe	UtenteDAO.DoSave
Descrizione	Permette di salvare un utente preso in input nel database.
Pre-condizione	venditoreBean!=null&& testBean!=null&& testBean2!=null;
Post-condizione	UtenteDAO:: testBean.equals(testBean2);

Nome Classe	UtenteDAO.RetriveByKey
Descrizione	Ci fa ottenere un utente ,con un dato id preso in input, dal database(se quest'ultimo e' presente).
Pre-condizione	TestDao!=null && testBean!=null
Post-condizione	UtenteDAO:: testBean.equals(testDao.doRetriveByKey((testDao.doRetrieveAll(),size())));

Nome Classe	UtenteDAO.RetriveAll
Descrizione	Permette di ottenere tutti gli utenti memorizzati nel database.
Pre-condizione	collection = new ArrayList<UtenteBean>();
Post-condizione	UtenteDAO::!(collection.equals(testDao.doRetrieveAll()));

Nome Classe	UtenteDAO.DoDelete
Descrizione	Permette di eliminare un certo utente.
Pre-condizione	TestDao!=null&& testBean!=null;
Post-condizione	UtenteDAO:: (testDao.doDelete(testBean.getIdProdotto()))==true;

Nome Classe	VenditoreDAO.DoSave
Descrizione	Permette di salvare un venditore preso in input nel database.
Pre-condizione	testBean!=null&& testBean2!=null&& testDao!=null;
Post-condizione	VenditoreDAO:: testBean.equals(testBean2);

Nome Classe	VenditoreDAO.RetriveByKey
Descrizione	Ci fa ottenere un venditore ,con un dato id preso in input, dal database(se quest'ultimo e' presente).
Pre-condizione	TestDao!=null && testBean!=null
Post-condizione	VenditoreDAO:: testBean.equals(testDao.doRetriveByKey((testDao.doRetrieveAll(),size())));

Nome Classe	VenditoreDAO.RetriveAll
Descrizione	Permette di ottenere tutti gli utenti memorizzati nel database.
Pre-condizione	collection = new ArrayList<UtenteBean>();
Post-condizione	VenditoreDAO::!(collection.equals(testDao.doRetrieveAll()));

Nome Classe	VenditoreDAO.DoDelete
Descrizione	Permette di eliminare un certo utente.
Pre-condizione	TestDao!=null&& testBean!=null;
Post-condizione	VenditoreDAO:: (testDao.doDelete(testBean.getIdProdotto()))==true;