odd – object design document



|  |  |
| --- | --- |
| Riferimento |  |
| Versione | 1.0 |
| Data | 28/12/2020 |
| Destinatario | Filomena Ferrucci |
| Presentato da | Fabio D’Anello  Ciro Malafronte Vincenzo Malafronte |
| Approvato da | Cesaro Manholo D’Anello Fabio Iannelli Pierpaolo Malafronte Ciro Malafronte Vincenzo |

# Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| **Data** | **Versione** | **Descrizione** | **Autori** |
| 14/12/2020 | 0.1 | Introduzione, paragrafi 1.1, 1.2, 1.3.1 | D’Anello Fabio, Malafronte Vincenzo, Iannelli Pierpaolo |
| 15/12/2020 | 0.2 | Completato capitolo 1 | Cesaro Manholo, Malafronte Ciro |
| 16/12/2020 | 0.3 | Capitolo 2 | Malafronte Ciro, Malafronte Vincenzo |
| 17/12/2020 | 0.4 | Class Interfaces(capitolo 3) | TUTTI |
| 18/12/2020 | 0.5 | Paragrafi 4.1, 4.2 | Malafronte Ciro, Malafronte Vincenzo |
| 28/12/2020 | 1.0 | 4.3, 4.4, Revisione | D’Anello Fabio, Cesaro Manholo, Iannelli Pierpaolo |

Sommario

[Revision History 1](#_Toc62840281)

[1. Introduzione 3](#_Toc62840282)

[*1.1* *Object Design Trade-offs* 3](#_Toc62840283)

[*1.2* *Componenti off -the-shelf* 3](#_Toc62840284)

[*1.3* *Linee guida per l’interfaccia* 3](#_Toc62840285)

[*1.3.1* *Package Model e Controller* 3](#_Toc62840286)

[*1.3.2* *JavaScript* 4](#_Toc62840287)

[*1.3.3* *Interfaccia grafica* 4](#_Toc62840288)

[*1.3.4* *DataBase SQL* 5](#_Toc62840289)

[2. Packages 5](#_Toc62840290)

[*2.1.* *View* 5](#_Toc62840291)

[*2.2.* *Model* 6](#_Toc62840292)

[*2.3.* *Controller* 7](#_Toc62840293)

[3. Class Interfaces 8](#_Toc62840294)

[4. Design patterns e Class diagram 13](#_Toc62840295)

[*4.1.* *MVC Design Pattern* 13](#_Toc62840296)

[*4.2.* *Proxy Pattern* 14](#_Toc62840297)

[*4.3.* *Singleton Pattern* 14](#_Toc62840298)

[*4.4.* *Class diagram package* 15](#_Toc62840299)

# Introduzione

## ***Object Design Trade-offs***

Dopo la realizzazione del documento RAD (Requirement Analysis Document) e SDD (System Design Document), abbiamo descritto in linea di massima, quello che sarà il nostro sistema e quindi i nostri obiettivi, tralasciando gli aspetti implementativi.

*Prestazioni vs. Costi:* Considerando il sistema che stiamo realizzando, possiamo dire che il non eccessivo budget a nostra disposizione ci ha consentito di realizzare il prodotto utilizzando materiale open source partendo da zero minimizzando così i costi e rendendo il sistema più che soddisfacente.

*Interfaccia vs. Tempo di risposta*: I tempi di risposta tra server e interfaccia sono più che sufficienti (rapidi) a soddisfare le esigenze dei vari dipendenti e clienti collegati al Sistema. Ovviamente maggiore sarà la grandezza del database e maggiore sarà il tempo di risposta e ricerca nel database.

*Interfaccia vs. Easy-use:* L’interfaccia, grazie a form e ad una impostazione semplice e intuitiva, permette un facile utilizzo del sistema.

*Costi vs. Mantenimento:* Grazie a un uso di materiale open source il sistema può essere facilmente modificato, implementato con nuove funzioni o corretto in presenza di errori.

## ***Componenti off -the-shelf***

Per lo sviluppo del sistema utilizziamo le seguenti componenti off-the-shelf, ovvero componenti software messe a disposizione dal mercato che offrono pacchetti di soluzioni mirate per la risoluzione di specifici problemi:

* *Gson*, una libreria java open source per serializzare e deserializzare oggetti Java in (e da) JSON.
* *jQuery.js,* una libreria JavaScript per applicazioni web il cui obiettivo è quello di semplificare la selezione, la manipolazione, la gestione degli eventi e l'animazione di elementi DOM in pagine HTML. Il suo utilizzo ha lo scopo di ridurre la complessità del codice JavaScript durante l’implementazione.
* *AJAX,* una tecnica di sviluppo software per la realizzazione di applicazioni web interattive che si basa su uno scambio di dati in background fra web browser e server, che consente l'aggiornamento dinamico di una pagina web senza esplicito ricaricamento da parte dell'utente. Il suo utilizzo consente di effettuare richieste asincrone al database.
* *Selenium IDE,* un framework che consente la registrazione, l'editing e il debug di test funzionali.
* *CheckStyle,* strumento di analisi statica del codice utilizzato nello sviluppo software per verificare se il codice sorgente Java è conforme alle regole di codifica specificate.
* *SpringFramework,* un framework open source per lo sviluppo di applicazioni su piattaforma Java. Il suo utilizzo consente di effettuare i test sulle componenti funzionali del sistema.

## ***Linee guida per l’interfaccia***

### ***Package Model e Controller***

Il progetto verrà sviluppato tramite l’IDE di sviluppo Eclipse e sarà strutturato nel seguente modo:

* Nel progetto vi sono due package(Model e Controller), i quali contengono le rispettive classi.
* Le classi Java, oltre a prevedere le convenzioni più comuni del linguaggio, presentano il seguente schema:
  + Clausole import;
  + Dichiarazione della classe;
  + Variabili di istanza;
  + Costruttore;
  + Metodi.
* Il nome di una classe deve rispettare il formato “NomeClasse”.
* I nomi delle variabili di istanza e dei metodi rispettano il formato “nomeVariabile”, “nomeMetodo”.
* L’insieme delle variabili d’istanza è preceduto e seguito da una riga vuota.
* Un intero metodo, compreso di intestazione ed istruzioni, è preceduto e seguito da una riga vuota.
* I nomi utilizzati per identificare classi, attributi e metodi devono risultare quanto più possibile significativi del loro scopo.
* Una parentesi graffa aperta non è preceduta da uno spazio ed è seguita da un carattere di new-line che comincia con un carattere di indentazione.
* Una parentesi graffa chiusa è preceduta e seguita da un carattere di new line.
* Un costrutto di tipo “if” che prevede una sola istruzione utilizza le parentesi graffe.
* I commenti vengono utilizzati per chiarire concetti che potrebbero non essere chiari al momento della lettura. Nel caso in cui un commento si estenda per una sola riga, esso presenta il formato: // commento; se, invece, un commento si estende su più righe diverse, esso presenta il formato /\* commento \*/.
* Il package Model contiene tutte le classi che fanno riferimento ad entità persistenti (Bean e DAO).
* Il package Controller contiene tutte le classi Servlet che si occupano della logica di business del sistema e agiscono da interlocutore tra le classi contenute nel package Model, che contiene la classe DriverManagerConnectionPool che si occupa della connessione del sistema al Database.
* Dov’è possibile, in modo particolare per i Bean contenuti nel package Model, le classi sono corredate da opportuni metodi GETTER, SETTER e, eventualmente, sovrascrivono il metodo toString().
* Una singola indentazione consiste di un solo carattere di tabulazione e, qualunque sia il linguaggio usato per la produzione di codice, ogni istruzione risulta opportunamente indentata.
* Le dichiarazioni si trovano sempre all’inizio dei corrispondenti blocchi di codice, poiché dichiarare le variabili soltanto in corrispondenza del loro primo utilizzo può provocare confusione.
* Qualsiasi variabile presente in un blocco più interno non è mai dichiarata con lo stesso nome di una variabile presente in un blocco più esterno.

### ***JavaScript***

Nel progetto vengono utilizzate pagine JavaScript, che fanno da intermediari tra pagine JSP e Servlet. Gli script in JavaScript devono rispettare le seguenti conversioni:

* Il codice Javascript deve seguire le stesse convenzioni per il layout e i nomi del codice Java.
* Le funzioni Javascript devono essere documentate in modo analogo ai metodi Java.

### ***Interfaccia grafica***

Le interfacce del progetto vengono implementate attraverso i file JSP, HTML e CSS:

* Le pagine JSP devono, quando eseguite, produrre un documento conforme allo standard HTML 5. Il codice Java delle pagine deve aderire alle convenzioni per la codifica in Java, con le seguenti puntualizzazioni:
  + Il tag di apertura (<%\*) si trova all'inizio di una riga;
  + Il tag di chiusura (%>) si trova all'inizio di una riga;
* Non sono presenti pagine statiche, ma solo pagine dinamiche che, però, possono anche non contenere codice Java.
* Esiste un solo foglio CSS. Ogni regola CSS deve essere formattata come segue:
  + I selettori della regola si trovano a livello 0 di indentazione;
  + L'ultimo selettore della regola è seguito da parentesi graffa aperta ({);
  + Le proprietà che costituiscono la regola sono listate una per riga e sono indentate rispetto ai selettori;
  + La regola è terminata da una parentesi graffa chiusa (}), collocata da sola su una riga.

### ***DataBase SQL***

* I nomi delle tabelle del Database SQL devono seguire le seguenti regole:
  + Devono essere costituiti da sole lettere minuscole;
  + Il nome deve essere un sostantivo singolare tratto dal dominio del problema ed esplicativo del contenuto.
* I nomi dei campi devono seguire le seguenti regole:
  + Devono essere costituiti da sole lettere minuscole;
  + Se il nome è costituito da più parole, è previsto l’uso di underscore (\_);
  + Il nome deve essere un sostantivo singolare tratto dal dominio del problema ed esplicativo del contenuto.

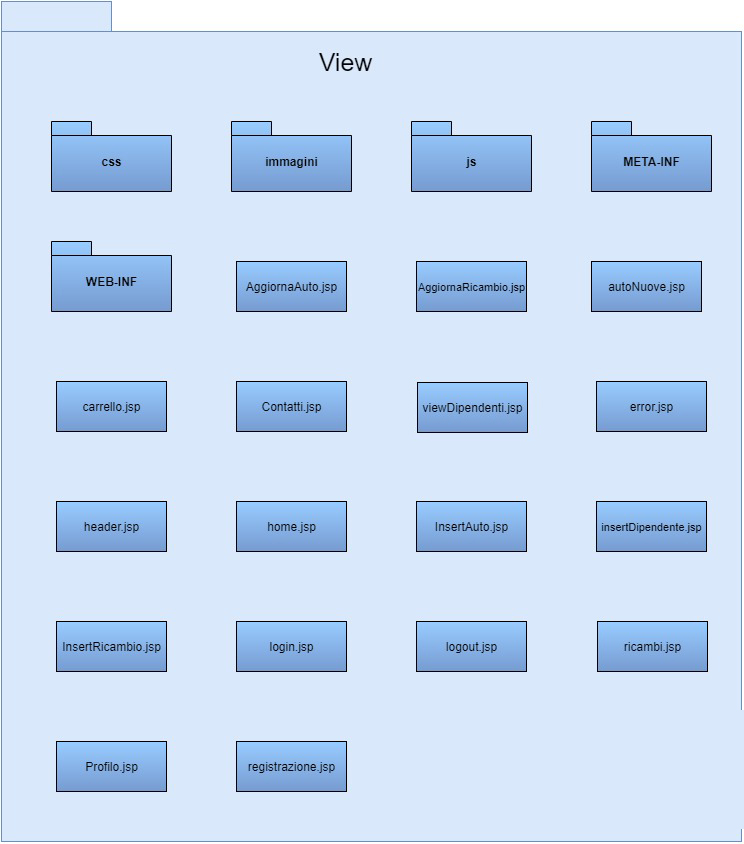
# Packages

## ***View***

Per package View intendiamo la cartella WebContent nella quale abbiamo le seguenti sottocartelle:

* *Css*, nella quale abbiamo il documento “style.css” che, a sua volta, contiene tutte le regole di stile della nostra pagina;
* *Immagini*, nella quale sono inserite tutte le immagini utilizzate per le pagine;
* Js*,* nella quale sono inseriti i fogli con le funzioni JavaScript;
* *META-INF* e WEB-INF*,* sono cartelle autogenerate dall’IDE;

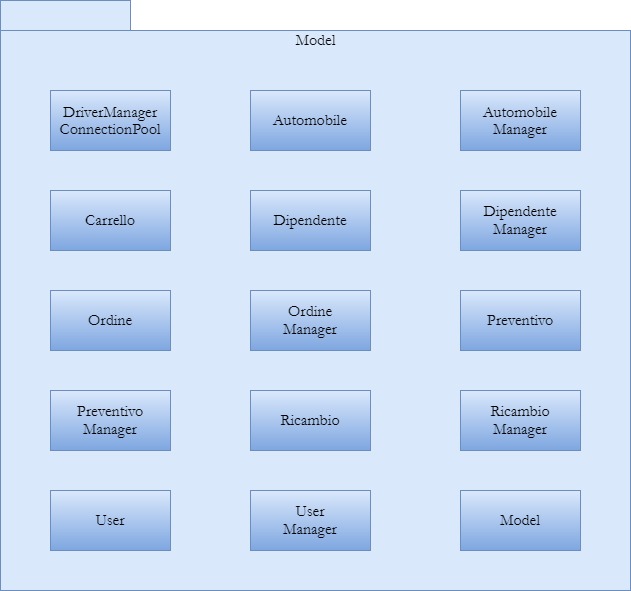
Oltre a queste sottocartelle, il package contiene tutte le JSP necessarie per le pagine: “home.jsp” viene utilizzata per mostrare la prima interfaccia appena si accede al sito; grazie ad essa sarà possibile accedere a tutte le altre pagine, nonché accedere o registrarsi tramite “login.jsp” e “registrazione.jsp”, può effettuare il logout tramite “logout.jsp” e sarà possibile visualizzare le proprie informazioni tramite la pagina “Profilo.jsp”. Tramite la pagina “ricambi.jsp” sarà possibile visualizzare il catalogo dei pezzi di ricambio, i quali possono essere inseriti all’interno del carrello e verranno visualizzati in “carrello.jsp”. La pagina “autoNuove.jsp” contiene il catalogo delle auto per le quali è possibile richiedere un preventivo mentre la pagina “Contatti.jsp” contiene alcune informazioni riguardanti l’attività. L’utente col ruolo di dipendente e/o amministratore potrà accedere alla pagina “InsertRicambio.jsp” e “InsertAuto.jsp” le quali permettono di aggiungere nuovi prodotti e, per modificarne le informazioni, dovranno accedere alla pagina “AggiornaRicambio.jsp” o “AggiornaAuto.jsp”. L’utente col ruolo di amministratore potrà anche accedere alla pagina “insertDipendente.jsp” e “viewDipendenti.jsp”.



## ***Model***

Il package Model contiene la classe DriverManagerConnectionPool che consente la connessione al database l’interfaccia “Model.java” che fornisce i metodi essenziali per dialogare col database, oltre alle seguenti classi:

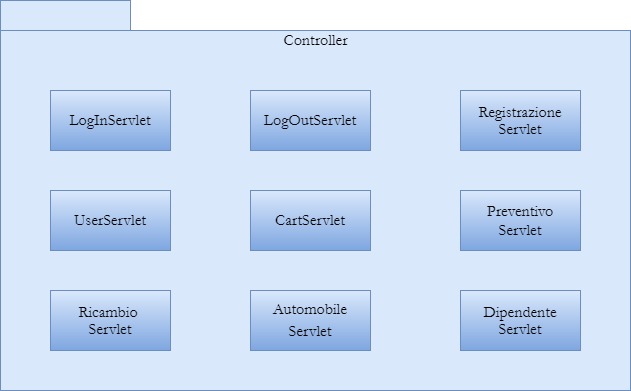
* Classe “Automobile.java” che rappresenta l’oggetto automobile;
* Classe “Carrello.java” che rappresenta l’oggetto carrello;
* Classe “Dipendente.java” che rappresenta l’oggetto dipendente;
* Classe “Ordine.java” che rappresenta l’oggetto ordine;
* Classe “Preventivo.java” che rappresenta l’oggetto preventivo;
* Classe “Ricambio.java” che rappresenta l’oggetto ricambio;
* Classe “User.java” che rappresenta l’oggetto user.

Per ogni classe sopra indicata (eccetto la classe “Carrello.java”) abbiamo una classe “…Manager.java”, la quale implementa l’interfaccia “Model.java”, che sfrutta i metodi dell’interfaccia per dialogare con la tabella del database desiderata. 

## ***Controller***

Il package controller riceve, tramite il pacchetto View, i comandi dall’utente. È formato da:

* *LoginServlet,* gestisce il login di un utente;
* *LogoutServlet,* gestisce il logout di un utente;
* *RegistrazioneServlet,* gestisce la registrazione di un nuovo utente;
* *UserServlet,* gestisce la modifica della password da parte di un utente;
* *CartServlet*, gestisce l’aggiunta e la rimozione di un pezzo di ricambio al carrello, la pulizia totale del carrello e l’acquisto degli elementi presenti nel carrello;
* *PreventivoServlet,* gestisce la richiesta di un preventivo;
* *RicambioServlet,* gestisce l’aggiunta, la rimozione e la modifica delle informazioni relativa ai ricambi;
* *AutomobileServlet,* gestisce l’aggiunta, la rimozione e la modifica delle informazioni relativa alle automobili;
* *DipendenteServlet,* gestisce l’aggiunta di un nuovo dipendente, la rimozione e la modifica delle informazioni relativa ad un dipendente;



# Class Interfaces

|  |  |
| --- | --- |
| **Nome Classe** | **LoginServlet** |
| **Descrizione** | Questa classe crea una sessione per l’utente registrato |
| **Pre:** | **Context** LoginServlet::doGet(request, response); **Pre:** request.getSession().getAttribute(“user”)==null &&  request.getParameter(“email”) != null && request.getParameter(“password”)!=null; |
| **Post:** | **Context** LoginServlet::doPost(request, response); **Post:** |
| **Invarianti:** |  |

|  |  |
| --- | --- |
| **Nome Classe** | **LogoutServlet** |
| **Descrizione** | Questa classe invalida la sessione dell’utente che ha effettuato il login. |
| **Pre:** | **Contex** LogoutServlet::doGet(request, response) **Pre:** request.getSession().getAttribute(“user”)!=null; |
| **Post:** | **Contex** LogoutServlet::doGet(request, response) **Post:** |
| **Invarianti:** |  |

|  |  |
| --- | --- |
| **Nome Classe** | **RegistrazioneServlet** |
| **Descrizione** | Questa classe viene utilizzata per la registrazione di un nuovo utente |
| **Pre:** | **Context** RegistrazioneServlet::doGet(request, response) **Pre:** request.getParameter(“email”)!=null && request.getParameter(“password”)!=null &&  request.getParameter(“username”)!=null && request.getParameter(“nome”)!=null && request.getParameter(“cognome”)!=null &&  request.getParameter(“indirizzo”)!=null ; |
| **Post:** | **Context** RegistrazioneServlet::doGet(request, response) **Post:** è stato salvato un nuovo utente nel db |
| **Invarianti:** |  |

|  |  |
| --- | --- |
| **Nome Classe** | **UserServlet** |
| **Descrizione** | Questa classe gestisce la modifica dei dati personali da parte di un utente registrato. |
| **Pre:** | **Context** UserServlet::doGet(request, response) **Pre:** request.getSession().getAttribute(“user”)!=null &&  request.getParameter(“password”)!=null &&  request.getparameter(“nuovaPassword”)!=null; |
| **Post:** | **Context** UserServlet::doGet(request, response) **Post:** |
| **Invarianti:** |  |

|  |  |
| --- | --- |
| **Nome Classe** | **CartServlet** |
| **Descrizione** | Questa classe permette di aggiungere un prodotto al carrello |
| **Pre:** | **Context** UserServlet::doGet(request, response) **Pre:** request.getSession().getAttribute(“user”)!=null &&  request.getParameter(“action”)== “add” &&  request.getParameter(“id”)>0; |
| **Post:** | **Context** UserServlet::doGet(request, response) **Post:** |
| **Invarianti:** |  |

|  |  |
| --- | --- |
| **Nome Classe** | **CartServlet** |
| **Descrizione** | Questa classe permette di rimuovere un prodotto dal carrello |
| **Pre:** | **Context** UserServlet::doGet(request, response) **Pre:** request.getSession().getAttribute(“user”)!=null &&  request.getParameter(“action”)== “remove” &&  request.getParameter(“id”)>0; |
| **Post:** | **Context** UserServlet::doGet(request, response) **Post:** |
| **Invarianti:** |  |

|  |  |
| --- | --- |
| **Nome Classe** | **CartServlet** |
| **Descrizione** | Questa classe permette di svuotare il carrello |
| **Pre:** | **Context** UserServlet::doGet(request, response) **Pre:** request.getSession().getAttribute(“user”)!=null &&  request.getParameter(“action”)!= “clear” &&  request.getParameter(“id”)==0; |
| **Post:** | **Context** UserServlet::doGet(request, response) **Post:** |
| **Invarianti:** |  |

|  |  |
| --- | --- |
| **Nome Classe** | **CartServlet** |
| **Descrizione** | Questa classe permette di acquistare tutti i prodotti presenti nel carrello |
| **Pre:** | **Context** UserServlet::doGet(request, response) **Pre:** request.getSession().getAttribute(“user”)!=null &&  request.getParameter(“action”)!= “buy”; |
| **Post:** | **Context** UserServlet::doGet(request, response) **Post:** |
| **Invarianti:** |  |

|  |  |
| --- | --- |
| **Nome Classe** | **PreventivoServlet** |
| **Descrizione** | Questa classe permette di gestire la generazione e il salvataggio di un preventivo |
| **Pre:** | **Context** PreventivoServlet::doGet(request, response)  **Pre:** request.getSession().getAttribute(“user”)!=null &&  Request.getparameter(“id”)>0; |
| **Post:** | **Context** PreventivoServlet::doGet(request, response)  **Post:** |
| **Invarianti:** |  |

|  |  |
| --- | --- |
| **Nome Classe** | **RicambioServlet** |
| **Descrizione** | Questa classe permette di aggiungere un nuovo pezzo di ricambio |
| **Pre:** | **Context** RicambioServlet::doGet(request, response)  **Pre:** request.getSession().getAttribute(“dip”)!=null &&  request.getParameter(“action”)==”add” &&  request.getParameter("marca") != null &&  request.getParameter("tipo") != null &&  request.getParameter("qnt") != null &&  request.getParameter("prezzo"!= null; |
| **Post:** | **Context** RicambioServlet::doGet(request, response)  **Post:** |
| **Invarianti:** |  |

|  |  |
| --- | --- |
| **Nome Classe** | **RicambioServlet** |
| **Descrizione** | Questa classe permette di rimuovere un pezzo di ricambio dal catalogo |
| **Pre:** | **Context** RicambioServlet::doGet(request, response)  **Pre:** request.getSession().getAttribute(“dip”)!=null &&  request.getParameter(“action”)==”remove”; |
| **Post:** | **Context** RicambioServlet::doGet(request, response)  **Post:** |
| **Invarianti:** |  |

|  |  |
| --- | --- |
| **Nome Classe** | **RicambioServlet** |
| **Descrizione** | Questa classe permette di modificare le informazioni di un pezzo di ricambio già presente nel catalogo |
| **Pre:** | **Context** RicambioServlet::doGet(request, response)  **Pre:** request.getSession().getAttribute(“dip”)!=null &&  request.getParameter(“id”)!=null && request.getParameter(“action”)==”edit”; |
| **Post:** | **Context** RicambioServlet::doGet(request, response)  **Post:** |
| **Invarianti:** |  |

|  |  |
| --- | --- |
| **Nome Classe** | **AutomobileServlet** |
| **Descrizione** | Questa classe permette di aggiungere una nuova automobile all’interno del catalogo auto. |
| **Pre:** | **Context** AutomobileServlet::doGet(request, response)  **Pre:** request.getSession().getAttribute(“dip”) != null &&  request.getParameter(“action”)== “add” &&  request.getParameter("marca")!=null &&  request.getParameter("modello")!=null &&  request.getParameter("prezzo")!=null &&  request.getParameter("img")!=null; |
| **Post:** | **Context** AutomobileServlet::doGet(request, response)  **Post:** |
| **Invarianti:** |  |

|  |  |
| --- | --- |
| **Nome Classe** | **AutomobileServlet** |
| **Descrizione** | Questa classe permette di rimuovere una nuova automobile all’interno del catalogo auto. |
| **Pre:** | **Context** AutomobileServlet::doGet(request, response)  **Pre:** request.getSession().getAttribute(“dip”) != null &&  Request.getParameter(“action”)== “remove”; |
| **Post:** | **Context** AutomobileServlet::doGet(request, response)  **Post:** |
| **Invarianti:** |  |

|  |  |
| --- | --- |
| **Nome Classe** | **AutomobileServlet** |
| **Descrizione** | Questa classe permette di modificare le informazioni riguardanti un’automobile già presente all’interno del catalogo auto |
| **Pre:** | **Context** AutomobileServlet::doGet(request, response)  **Pre:** request.getSession().getAttribute(“dip”) != null &&  request.getParameter(“action”)== “edit” &&  request.getParameter(“id”)!=0; |
| **Post:** | **Context** AutomobileServlet::doGet(request, response)  **Post:** |
| **Invarianti:** |  |

|  |  |
| --- | --- |
| **Nome Classe** | **DipendenteServlet** |
| **Descrizione** | Questa classe permette all’utente amministratore di aggiungere un nuovo dipendente. |
| **Pre:** | **Context** DipendenteServlet::doGet(request, response)  **Pre:** l’utente è un amministratore &&  request.getParameter(“action”)== “add” &&  request.getParameter("username")!= null &&  request.getParameter("email")!= null &&  request.getParameter("password")!= null &&  request.getParameter("nome")!= null &&  request.getParameter("cognome")!= null &&  request.getParameter("stipendio")!= null &&  request.getParameter("ruolo")!= null; |
| **Post:** | **Context** DipendenteServlet::doGet(request, response)  **Post:** l’utente è un amministratore |
| **Invarianti:** |  |

|  |  |
| --- | --- |
| **Nome Classe** | **DipendenteServlet** |
| **Descrizione** | Questa classe permette all’utente amministratore di rimuovere un dipendente già presente all’interno del db. |
| **Pre:** | **Context** DipendenteServlet::doGet(request, response)  **Pre:** l’utente è un amministratore &&  request.getParameter(“action”)== “remove” ; |
| **Post:** | **Context** DipendenteServlet::doGet(request, response)  **Post:** |
| **Invarianti:** |  |

|  |  |
| --- | --- |
| **Nome Classe** | **DipendenteServlet** |
| **Descrizione** | Questa classe permette all’utente amministratore di modificare i dati relativi ad un dipendente già presente all’interno del db. |
| **Pre:** | **Context** DipendenteServlet::doGet(request, response)  **Pre:** l’utente è un amministratore &&  request.getParameter(“action”)== “edit”%%  request.getParameter(“chiave”)!=null; |
| **Post:** | **Context** DipendenteServlet::doGet(request, response)  **Post:** |
| **Invarianti:** |  |

# Design patterns e Class diagram

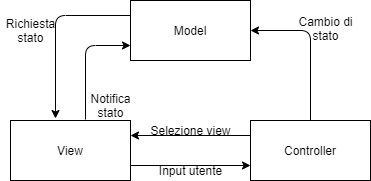
Di seguito vengono descritti i design pattern utilizzati per l’implementazione del sistema proposto.

## ***MVC Design Pattern***

BuyCar fa uso del MVC Design Pattern per separare i componenti software che implementano il modello delle funzionalità di business dai componenti che implementano la logica di presentazione e di controllo che utilizzano tali funzionalità, così da semplificare la progettazione e da avere una separazione dei ruoli più semplice. Vengono, dunque, definite tre tipologie di componenti:

* Il *Model,* che implementa le funzionalità di business;
* La *View,* che implementa la logica di presentazione;
* Il *Controller,* che implementa la logica di controllo.

La comunicazione attraverso i vari componenti avviene tramite l’implementazione dell’ Observer Pattern il quale permette di avere sempre delle View aggiornate attraverso, l’osservazione del model e la possibilità di una registrazione dinamica a runtime, attraverso un’osservazione della View da parte del controller.



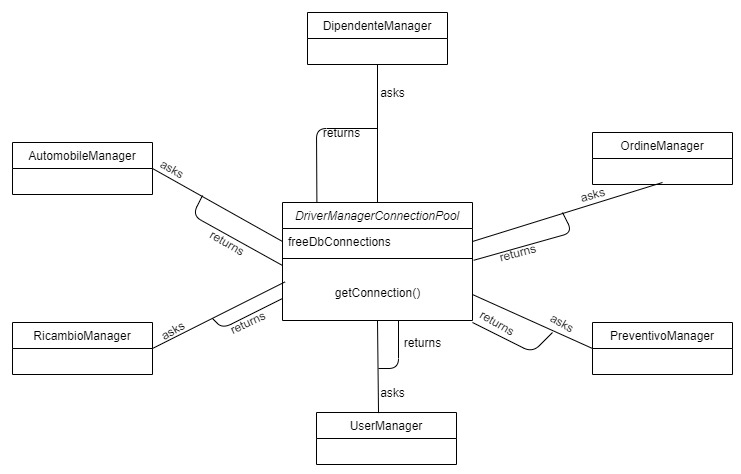
## ***Proxy Pattern***

Il proxy pattern in linea generale viene utilizzato per fornire il controllo dell’accesso all’oggetto reale. Nel nostro sistema l’utilità di questo pattern nasce dalla necessità che utenti diversi abbiano diritti di accesso e di visualizzazione diverse per le stesse pagine. Nel caso specifico di BuyCar i possibili utenti saranno il dipendente, l’amministratore, l’utente registrato e infine l’utente non registrato.



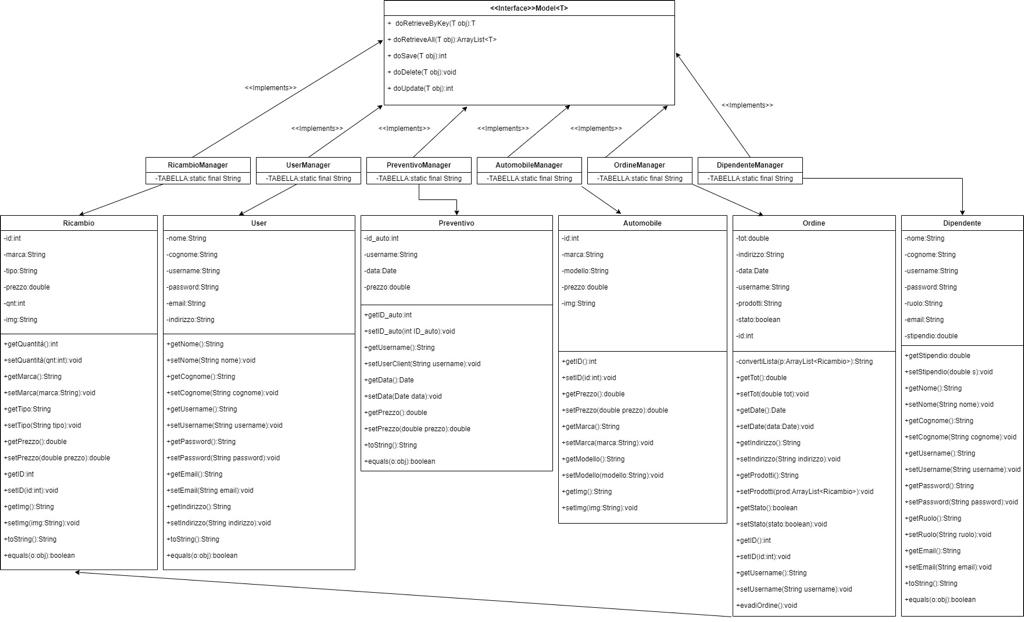
## ***Singleton Pattern***

Il Singleton è un design pattern di tipo creazionale, utilizzato nelle situazioni in cui è necessario istanziare un singolo oggetto della classe di interesse, costituendo un punto di accesso globale per tutte le altre classi del sistema, quindi possiede uno scope di applicazione e supporta la gestione degli accessi concorrenti dei metodi che offre, attraverso la mutua esclusione. Presenta un unico metodo pubblico getConnection() utilizzato per creare un’istanza nel caso non esistesse ancora oppure restituire il riferimento di essa qualora fosse già stata creata. Le variabili di istanza, così come il costruttore, sono private. Il design pattern verrà utilizzato per accedere alla classe. Qualsiasi classe che intende utilizzare i metodi della classe DriverManagerConnectionPool, lo farà attraverso l’utilizzo dell’unica istanza della classe presente all’interno del sistema.

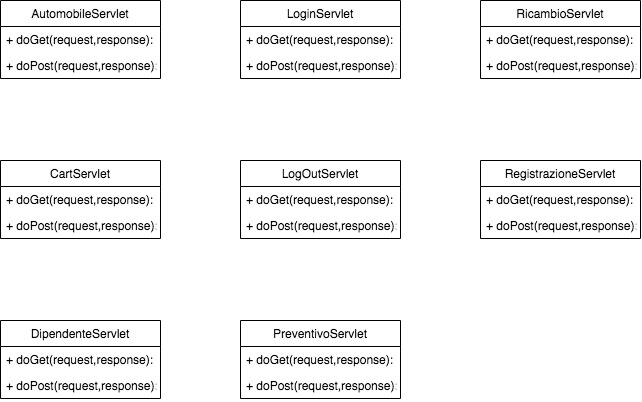


## ***Class diagram package***

*Model:*

**

*Controller:*

**