Laurea Triennale in informatica-Università di Salerno
Corso di *Ingegneria del Software*- Prof.ssa F. Ferrucci

Object Design Document

EasyLease

Riferimento	
Versione	0.8~~1.0~~
Data	15/12/2020
Destinatario	Azienda AutoErre S.r.l
Presentato da	Caprio Mattia Iodice Michele Attilio Mori Mattia Pepe Sara Sarro Antonio Torino Francesco Maria
Approvato da	Dario di Dario

Tabella formattata

Revision History

Data	Versione	Cambiamenti	Autori
15/12/2020	0.1	Stesura scheletro documento	Pepe Sara
17/12/2020	0.2	Aggiunta capitolo 1	Mori Mattia Sarro Antonio
18/12/2020	0.3	Aggiunta capitolo 2	Caprio Mattia Pepe Sara
18/12/2020	0.3	Aggiunta capitolo 4	Mori Mattia Sarro Antonio
20/12/2020	0.4	Modifica capitolo 2	Caprio Mattia
22/12/2020	~~1.0~~0.5	Aggiunta capitolo 3	Pepe Sara
28/01/2021	0.~~6~~5	Modifica capitolo 2	Pepe Sara
29/01/2021	0.~~7~~6	Modifica capitolo 4	Iodice Michele Attilio
30/01/2021	0.8	Revisione completa	Caprio Mattia

Sommario

ha formattato: Tipo di carattere: Century Gothic, 11 pt

1. Introduzione

1.1 Object Design Trade-Off

Abbiamo già discusso nelle precedenti fasi di analisi di quali saranno i criteri ed i compromessi su cui si baserà lo sviluppo del nostro sistema. Durante questa fase di Object Design, ci sono altri punti da analizzare, dunque altri compromessi da valutare, espressi di seguito:

~~Memoria~~ Estensibilità / ~~Estensibilità~~Memoria
Il sistema dovrà garantire, a discapito della memoria utilizzata, un'elevata estensibilità, così da poter essere flessibile a modifiche e variazioni, senza compromettere l'esperienza d'uso da parte dell'utente.

Affidabilità / Tempo di risposta
Come già sottolineato nei requisiti non funzionali (RADv3.2, punto 3.2.2) e nei Design Trade Off (SDD, punto 1.2.1), il sistema metterà al primo posto la correttezza delle informazioni piuttosto che il tempo di risposta, garantendo l'affidabilità dello stesso in ogni fase d'utilizzo.

Manutenzione / Performance
Il sistema verrà sviluppato prediligendo la portabilità e la manutenibilità del software, ~~cosi~~così da facilitare eventuali lavori di aggiornamento o estensione del sistema, a discapito però di una parte delle performance.

Costi / Robustezza
Non è garantita la massima robustezza del sistema per ridurre le ore uomo-lavoro e quindi non incorrere in un superamento del tetto massimo del budget. Quindi il sistema si avvarrà di un livello di sicurezza adatto a soddisfare le criticità basilari.

Nella seguente tabella, il Design Goal in **grassetto** indica il design goal prioritario.

Trade-off	
Estensibilità	Memoria
Affidabilità	Tempo di risposta
Manutenzione	Performance
Costi	Robustezza

1.2 Componenti Off-the-Shelf

Per l'implementazione del ~~nostro~~ sistema, verranno utilizzat~~e~~i diversi~~e~~ componenti Off-the-Shelf, ovvero componenti software già disponibili, così da facilitare la creazione del software~~.~~ del sistema.

Front End

Come già definito nei Requisiti non Funzionali (RAD, punto 3.2.2), ~~gestiremo~~ lo stile del sistema verrà gestito utilizzando HTML e CSS ~~insieme con~~con l'ausilio di Bootstrap (versione 5.0), un framework open-source contenente modelli di progettazione basati su HTML e CSS, i quali influiscono sulla tipografia e sulle varie componenti dell'interfaccia, estendendo inoltre le funzionalità di JavaScript.
Per il lato funzionale ~~usufruiremo~~ si fruirà del framework di JavaScript JQuery, il quale permette di semplificare l'animazione del sistema e le chiamate AJAX, oltre che a poter interagire direttamente su1 DOM del documento HTML.

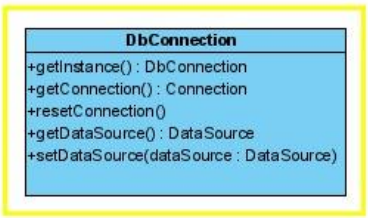
Back End

Per la gestione del lato Back-End, verranno utilizzati Java Enterprise come Web Container, ed Apache Tomcat come WebServer. Inoltre, la funzionalità per l'invio delle e-mail verrà implementata tramite l'utilizzo della libreria JavaMail.

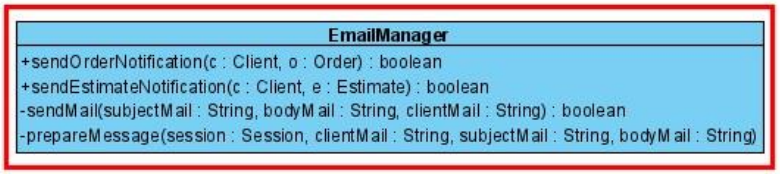
1.3 Design Pattern

Per velocizzare lo sviluppo del sistema senza dover riscrivere del codice già esistente ogni volta, il team si avvarrà di alcuni design pattern~~s~~ che aiutano a risolvere dei problemi comuni riscontrati anche nel nostro progetto.

- Singleton Pattern
  - Il singleton è un design pattern di tipo creazionale, ed ha la funzione di garantire all'interno di un ambiente software un'unica istanza di una determinata classe, che nel caso del progetto EaseLease è la classe che si occuperà della connessione al DB.



- Proxy Pattern
  - Il proxy è un design pattern di tipo strutturale che fornisce un'interfaccia per oggetti che richiedono risorse e tempo per essere creati, e nel nostro caso verrà utilizzato per inviare delle e-mail agli utenti della piattaforma tramite la libreria "JavaMail".



ha formattato: Nessuna sottolineatura

ha formattato: Nessuna sottolineatura

ha formattato: Nessuna sottolineatura

Formattato: Normale, Giustificato, Rientro: Sinistro: 0,2 cm

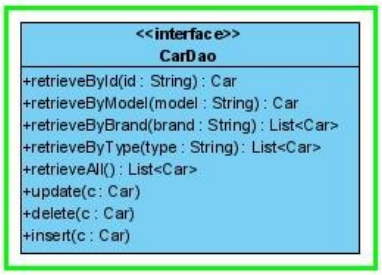
Formattato: Allineato al centro, Nessun elenco puntato o numerato

Formattato: Allineato al centro, Nessun elenco puntato o numerato

Formattato: Rientro: Sinistro: 1,27 cm, Nessun elenco puntato o numerato

- DAO Pattern

  Il DAO è un design pattern di tipo architetturale per il basso livello di accesso ai dati e quindi per la gestione della persistenza. Nel nostro progetto saranno quindi delle classi con i relativi metodi che andranno a rappresentare le entità individuate nel DB, con lo scopo quindi di separare la logica di business e l'accesso diretto ai dati persistenti.



1.4 Linee guida per la documentazione delle interfacce

Di seguito verranno stabilite le linee guida da seguire da parte degli sviluppatori del sistema al fine di essere consistenti per l'intero progetto e quindi facilitare la comprensione di ogni funzionalità del sistema all'intero team.

Classi e Interfacce Java

Lo standard che verrà utilizzato nella definizione di classi o interfacce Java è il code style definito da Google. Quindi ogni relativa documentazione dovrà seguire le relative linee guida.

Esempio Corretto

```
return () -> {
  while (condition()) {
    method();
  }
};

return new MyClass() {
  @Override public void method() {
    if (condition()) {
      try {
        something();
      } catch (ProblemException e) {
        recover();
      }
    } else if (otherCondition()) {
      somethingElse();
    } else {
      lastThing();
    }
  }
};
```

Esempio Errato

```
return () -> {
  while (condition()) { method();}
};

return new MyClass() {
@Override public void method() {
    if (condition()) {
      try {something();}
      catch (ProblemException e) {
        recover();
      }
    } else if (otherCondition())
      somethingElse();
    else
      lastThing();
}};
```

1.4.31.4.2 Java Servlet Pages (JSP)

Per quanto riguarda le convenzioni adottate per la stesura delle Servlet e le relative JSP è stato adottato lo standard Oracle.

1.4.41.4.3 Pagine HTML

Le pagine HTML devono essere conformi allo standard HTML5 e CSS3. Inoltre, il codice deve essere conforme alle linee guida descritte da Google.

Esempio Corretto

```
<div>
  <span>
    <ul>
      <li>
        Esempio
      </li>
      <li>
        esatto
      </li>
    </ul>
  </span>
</div>

```

Esempio Errato

```
<div>
<span>
<ul>
<li>Esempio</li> <li> errato</li>
</ul>
</span>
</div>
```

Gli script Javascript e la relativa documentazione seguiranno le linee guide definite da Google nel proprio CodeStyle.

Esempio Corretto

```
class InnerClass {
  constructor() {}

  /** @param {number} foo */
  method(foo) {
    if (condition(foo)) {
      try {
        something();
      } catch (err) {
        recover();
      }
    }
  }
}
```

Esempio Errato

```
class InnerClass {
  constructor() {}

  /** @param {number} foo */
  method(foo){
    if (condition(foo)){
      try{
        something();
      }catch(err){
        recover();}
    }}
}
```

1.4.71.4.5 Script SQL

Gli script SQL e il relativo Database seguiranno le linee guida descritte nel libro "SQL Programming Style" di Joe Celko.

Esempio Corretto

```
SELECT first_name AS fn
  FROM staff AS s1
  JOIN students AS s2
    ON s2.mentor_id = s1.staff_num;
```

Esempio Errato

```
SELECT firstName AS fn FROM Staff AS s1
  JOIN students
  AS s2
   ON students.mentorId = s1.staffNum;
```

1.5 Definizioni, acronimi, abbreviazioni

Definizioni

DOM = Document Object Model, rappresenta la strutturazione dei documenti come modello orientato ad oggetti.
SDD = System Design Document
RAD = Requirement Analysis Document
CSS = Cascading Style Sheet
HTML = HyperText Markup Language
AJAX = Asynchronous JavaScript And XML
SQL = Structured Query Language
DAO = Data Access Object

1.6 Riferimenti

Google Coding Style: Java, Javascript, HTML/CSS
Oracle Coding Style: Servlet/JSP,
Simon Holywell: SQL

Codice campo modificato
Codice campo modificato
Codice campo modificato
Codice campo modificato

2. Packages

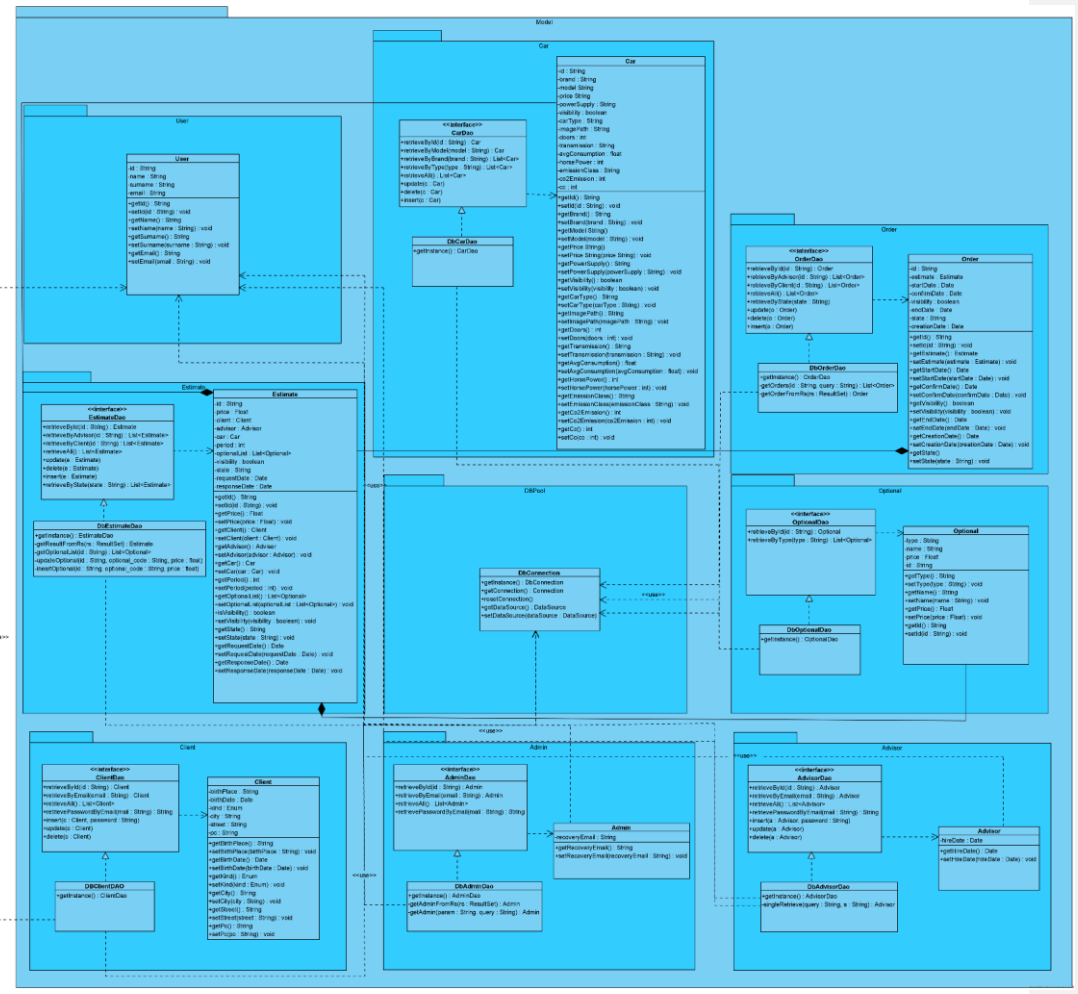
2.1 Model

Il package denominato "Model" contiene le classi adoperate per implementare la logica di business che caratterizza il sistema. Tale package è suddiviso in ulteriori package, ognuno corrispondente all'entità che sfrutta i relativi file DAO, DATABASE e POJO. Di seguito sono riportate le definizioni delle categorie dei file utilizzati per ogni package inferiore:

- **DAO**: Tale categoria contiene le interfacce dei Data Access Object (o DAO), ovvero classi adibite alla gestione dei dati persistenti presenti all'interno di un Database, i metodi per fare ciò permettono operazioni di lettura (Read), creazione (Create), aggiornamento (Update) ed eliminazione (Delete) dei valori delle tabelle del suddetto DB.
- **DATABASE**: Tale categoria contiene le implementazioni delle interfacce DAO.
- **POJO**: Tale categoria contiene le classi necessarie alla gestione delle funzionalità offerte dal sistema, ognuna delle quali dispone dei relativi metodi Getter e Setter per ognuna delle variabili di istanza della classe stessa.

Di seguito sono riportati tutti i package relativi alle entità e il package corrispondente alla funzionalità di connessione al Database:

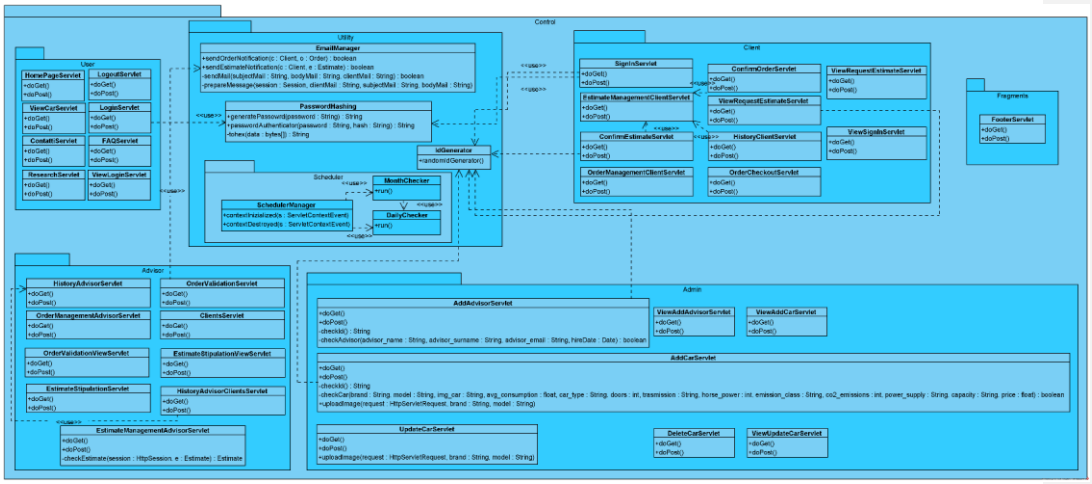
- **User**: UserDao, DbUserDao, User;
- **Client**: ClientDao, DBClientDao, Client;
- **Advisor**: AdvisorDao, DbAdvisorDao, Advisor;
- **Admin**: AdminDao, DbAdminDao, Admin;
- **Order**: OrderDao, DbOrderDao, Order;
- **Estimate**: EstimateDao, DbEstimateDao, Estimate;
- **Optional**: OptionalDao, DbOptionalDao, Optional;
- **Car**: CarDao, DbCarDao, Car;
- **DBPool**: ConnectionManagerInterface, ConnectionManager, ConnectionPool.


Model
Car
Car
User
User
CarDao
DbCarDao
Order
OrderDao
Order
DbOrderDao
Estimate
EstimateDao
Estimate
DbEstimateDao
DBPool
DbConnection
Optional
OptionalDao
Optional
DbOptionalDao
Client
ClientDao
Client
DBClientDAO
Admin
AdminDao
Admin
DbAdminDao
Advisor
AdvisorDao
Advisor
DbAdvisorDao

2.2 Control

Il package denominato "Control" contiene le classi che estendono la classe HttpServlet, adoperate per gestire le richieste svolte dai client. Tale package è suddiviso in diverse categorie, di seguito riportate:

- **User**: contiene le servlet per effettuare le operazioni possibili comuni a tutti i tipi di utente (registrato e non);
- **Client**: contiene le servlet per effettuare le operazioni peculiari all'account di tipo Cliente e la registrazione di un account di tipo Cliente;
- **Advisor**: contiene le servlet per effettuare le operazioni peculiari all'account di tipo Consulente;
- **Admin**: contiene le servlet per effettuare le operazioni peculiari all'account di tipo Amministratore;
- **Fragmtens**: contiene le servlet per effettuare le operazioni peculiari al footer;
- **Utility**: contiene le classi per effettuare le operazioni di supporto al sistema



2.3 View

Il package denominato "View" contiene tutte le pagine JSP utilizzate per permettere l'interazione tra utente e sistema. Il package si suddivide nelle seguenti sottocategorie:

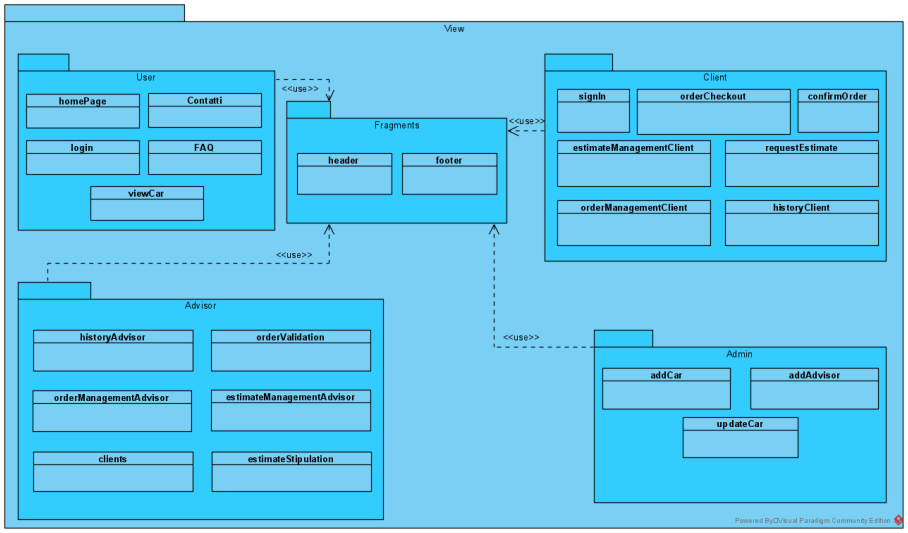
• **User**: contiene le pagine JSP attraverso le quali è possibile effettuare le operazioni possibili comuni a tutti i tipi di utente (registrato e non);

• **Client**: contiene le pagine JSP attraverso le quali è possibile effettuare le operazioni peculiari all'account di tipo Cliente e la registrazione di un account di tipo Cliente;

• **Advisor**: contiene le pagine JSP attraverso le quali è possibile effettuare le operazioni peculiari all'account di tipo Consulente;

• **Admin**: contiene le pagine JSP attraverso le quali è possibile effettuare le operazioni peculiari all'account di tipo Amministratore;

• **Fragmtens**: contiene le JSP attraverso le quali è possibile effettuare le operazioni peculiari all'header e al footer.



3. Interfacce delle Classi

Il seguente capitolo fornirà le interfacce delle classi del package "**View**". L'interfaccia delle classi dei package restanti verrà introdotta tramite JavaDoc una volta implementate.

Formattato: Giustificato

Formattato: Giustificato, Rientro: Sinistro: 0,2 cm, SpazioPrima: 1,75 pt, Interlinea: multipla 1,08 ri

Nome Classe:	homePage
Descrizione:	Tale classe è la home page del sistema e contiene l'elenco delle automobili presenti nel database. Tale elenco può essere filtrato mediante apposito form.
Pre-condizione:	
Post-condizione:	
Invariante:	

Nome Classe:	logIn
Descrizione:	Questa classe permette agli utenti di fare il login a condizione che siano già registrati.
Pre-condizione:	
Post-condizione:	
Invariante:	

Nome Classe:	viewCar
Descrizione:	Questa classe permette di visualizzare un'automobile.
Pre-condizione:	
Post-condizione:	
Invariante:	

Nome Classe:	requestEstimate
Descrizione:	Questa classe permette al cliente di richiedere un preventivo.
Pre-condizione:	doGet(request, response): il cliente dev'essere autenticato come tale.
Post-condizione:	doGet(request, response): il sistema invia la richiesta di preventivo.
Invariante:	

Nome Classe:	Contatti
Descrizione:	Questa classe permette agli utenti di mettersi in contatto con l'azienda.
Pre-condizione:	
Post-condizione:	
Invariante:	

Nome Classe:	**signIn**
Descrizione:	Questa classe permette all'utente non registrato di registrarsi alla piattaforma.
Pre-condizione:	
Post-condizione:	
Invariante:	

Nome Classe:	**FAQ**
Descrizione:	Questa classe permette agli utenti di ricevere informazioni generali sulla policy dell'azienda.
Pre-condizione:	
Post-condizione:	
Invariante:	

Nome Classe:	**estimateManagementClient**
Descrizione:	Questa classe permette al cliente di visualizzare e gestire un preventivo da egli richiesto.
Pre-condizione:	doGet(request, response): il cliente dev'essere autenticato come tale AND aver richiesto almeno un preventivo.
Post-condizione:	doGet(request, response): il sistema mostra i dettagli del preventivo.
Invariante:	

Nome Classe:	**orderManagementCli ent**
Descrizione:	Questa classe permette al cliente di gestire un ordine.
Pre-condizione:	doGet(request, response): il cliente dev'essere autenticato come tale AND deve avere almeno un ordine nel suo storico.
Post-condizione:	doGet(request, response): il sistema mostra l'ordine.
Invariante:	

Nome Classe:	**orderCheckout**
Descrizione:	Questa classe permette al cliente di pagare un ordine.
Pre-condizione:	doGet(request, response): il cliente dev'essere autenticato come tale AND deve aver confermato un ordine.
Post-condizione:	doGet(request, response): Il sistema mostra una pagina che permette al cliente di pagare un ordine.
Invariante:	

Nome Classe:	**confirmOrder**
Descrizione:	Questa classe permette al cliente di confermare un ordine.
Pre-condizione:	doGet(request, response): Il cliente deve essere loggato come tale.
Post-condizione:	doGet(request, response): Il sistema mostra una pagina che permette al cliente di confermare un ordine.
Invariante:	

Nome Classe:	**historyClient**
Descrizione:	Questa classe permette al cliente di vedere uno storico dei suoi ordini e preventivi.
Pre-condizione:	doGet(request, response): Il cliente deve essere loggato come tale.
Post-condizione:	doGet(request, response): Il sistema mostra al cliente la lista degli ordini e dei preventivi.
Invariante:	

Nome Classe:	**addCar**
Descrizione:	Questa classe permette all'amministratore di aggiungere un'auto al database.
Pre-condizione:	doGet(request, response): L'amministratore deve essere loggato come tale.
Post-condizione:	doGet(request, response): Il sistema aggiunge una nuova auto al database.
Invariante:	

Nome Classe:	**addAdvisor**
Descrizione:	Questa classe permette all'amministratore di aggiungere un consulente.
Pre-condizione:	doGet(request, response): L'amministratore deve essere loggato come tale.
Post-condizione:	doGet(request, response): Il sistema aggiunge un nuovo consulente al database.
Invariante:	

Nome Classe:	**updateCar**
Descrizione:	Questa classe permette all'amministratore di modificare un'auto.
Pre-condizione:	doGet(request, response): L'amministratore deve essere loggato come tale.
Post-condizione:	doGet(request, response): Il sistema modifica l'auto correttamente.

Invariante:	

Nome Classe:	**historyAdvisor**
Descrizione:	Questa classe permette al consulente di visualizzare la lista di ordini e preventivi da lui presi in carico, nonché la lista di preventivi non presi in carico ancora da nessun consulente.
Pre-condizione:	doGet(request, response): il consulente deve essere autenticato come tale.
Post-condizione:	doGet(request, response): la pagina mostra la lista di ordini e preventivi.
Invariante:	

Nome Classe:	**clients**
Descrizione:	Questa classe permette al consulente di visualizzare un elenco dei clienti.
Pre-condizione:	doGet(request, response): Il consulente dev'essere autenticato come tale.
Post-condizione:	doGet(request, response): la pagina mostra la lista dei clienti.
Invariante:	

Nome Classe:	**estimateStipulation**
Descrizione:	Questa classe permette al consulente di stipulare un preventivo.
Pre-condizione:	doGet(request, response): il consulente dev'essere autenticato come tale AND nel sistema dev'essere presente un preventivo ancora non preso in carico.
Post-condizione:	doGet(request, response): la pagina mostra il form che permette di stipulare un preventivo.
Invariante:	

Nome Classe:	**estimateManagemen tAdvisor**
Descrizione:	Questa classe permette ad un consulente di visualizzare un preventivo.
Pre-condizione:	doGet(request, response): il consulente dev'essere autenticato come tale.
Post-condizione:	doGet(request, response):

	la pagina mostra i dettagli di un preventivo.
Invariante:	

Nome Classe:	**orderManagement Advisor**
Descrizione:	Questa classe permette ad un consulente di visualizzare un ordine.
Pre-condizione:	doGet(request, response): il consulente dev'essere autenticato come tale.
Post-condizione:	doGet(request, response): la pagina mostra i dettagli di un ordine.
Invariante:	

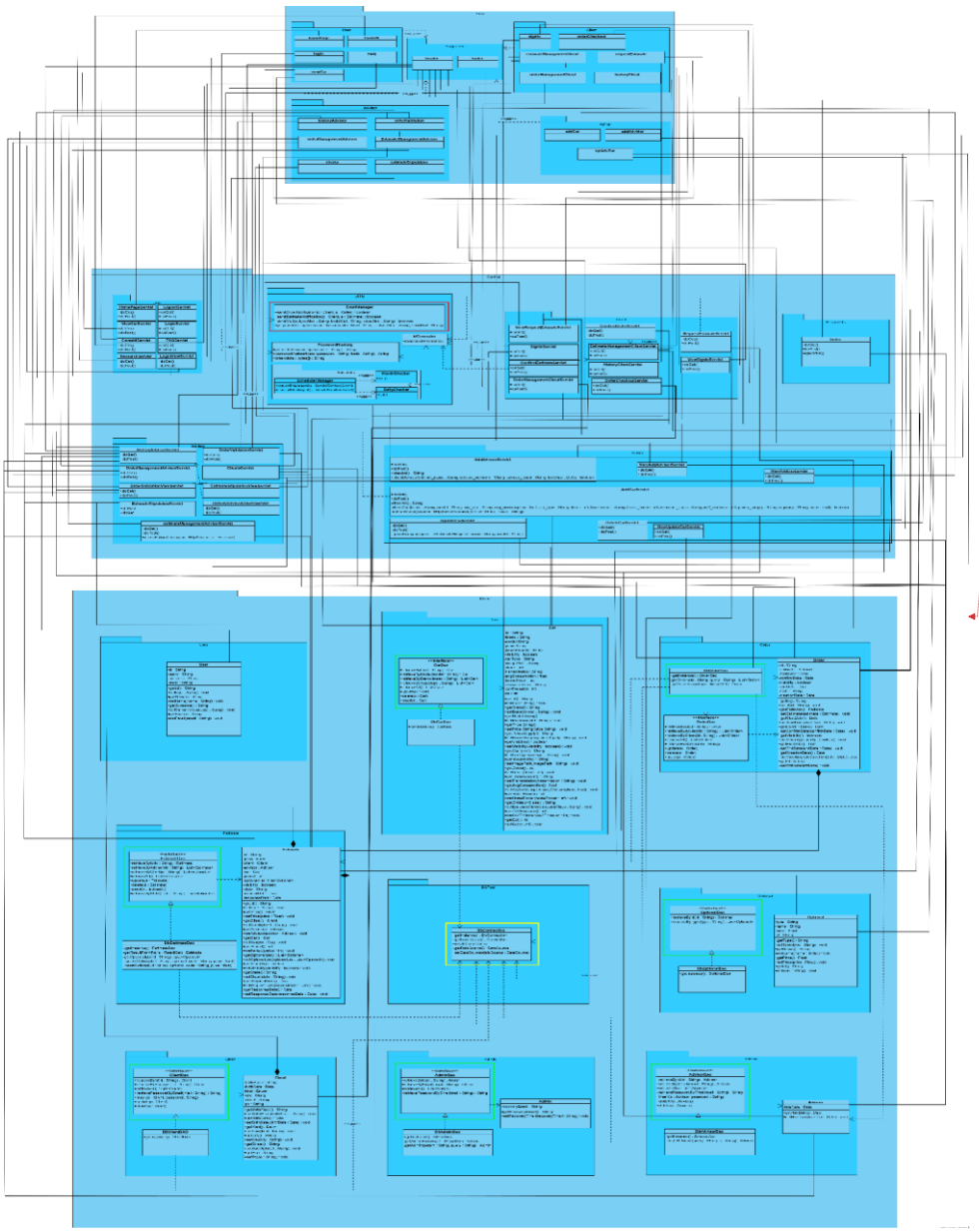
Nome Classe:	**orderValidation**
Descrizione:	Questa classe permette ad un consulente di convalidare un ordine.
Pre-condizione:	doGet(request, response): il consulente dev'essere autenticato come tale AND deve trovarsi nella pagina che permette di visualizzare un preventivo.
Post-condizione:	doGet(request, response): il sistema permette al consulente di convalidare un ordine.
Invariante:	

Nome Classe:	**footer**
Descrizione:	Questa classe viene inclusa da tutte le altre pagine JSP, e rappresenta il piè di pagina.
Pre-condizione:	
Post-condizione:	
Invariante:	

Nome Classe:	**header**
Descrizione:	Questa classe viene inclusa da tutte le altre pagine JSP, e rappresenta l'intestazione della pagina. Essa contiene link per accedere alle altre pagine, riguardanti in particolar modo l'autenticazione (Login, Logout, Registrazione), azioni generali (accesso ai Contatti e alle FAQ) e azioni peculiari dell'utente, una volta che questi ha effettuato l'autenticazione.
Pre-condizione:	
Post-condizione:	
Invariante:	

4. Class Diagram

Legenda Colori: DAO Pattern, Singleton Pattern, Proxy Pattern



Formattato: Rientro: Sinistro: 0 cm

5 Glossario

Standard di sicurezza minimo:

1. La password di ciascun utente deve essere criptata prima di essere memorizzata nel database;
2. Le pagine devono essere accessibili solo agli utenti che hanno il permesso di visualizzarla;
3. Le query al database devono essere sempre prima controllate per evitare attacchi di natura "SQL Injection";

Formattato: Giustificato