Progetto di Ingegneria del software

Anno accademico 2018/2019

Carcheck

Tabella Componenti		
Capriglione	**Francesco**	
D'Auria	**Aldo**	
De Falco	**Daniele**	0512104666
Iacovazzo	**Giovanni**	0512104774

1. Introduzione

Nel seguente documento vengono definiti tutti gli aspetti analizzati prima di procedere all'implementazione. Nelle fasi precedenti sono stati definiti i vari requisiti su cui modellare il sistema CarCheck. Il nostro scopo qui è andare ad interfacciarci con i Solution Objects, e tenere conto di aspetti di aspetti significativi riguardanti tempo di esecuzione di un'operazione, utilizzo di memoria e altre misure di costo.

1.1 Object design trade-off

Interfaccia vs Usabilità

L'interfaccia, dovendo essere semplice da utilizzare ed intuitiva, utilizzerà una serie di componenti standard. Saranno inoltre presenti delle etichette, suggerimenti ed altri elementi che faranno capire all'utente che l'azione che sta eseguendo ha prodotto un risultato oppure è incorretta. Eventualmente potranno essere utilizzate delle legende per una maggiore comprensione di simboli e notazioni.

Spazio di memoria vs Tempo di risposta

Si cercherà di mantenere un equilibrio tra questi due parametri. Infatti, da una parte abbiamo una grande quantità di informazioni da memorizzare su un numero molto alto di veicoli, dall'altra bisogna assicurare che i tempi delle risposte alle richieste degli utenti rispettino ciò che è stato espresso nella definizione dei requisiti non funzionali.

Sicurezza vs efficienza

Per gli utenti registrati sarà data priorità alla sicurezza. L'approccio utilizzato sarà quello di utilizzare come fattore di autenticazione una coppia E-mail-Password per evitare accessi non autorizzati. Altro aspetto, sempre relativo alla sicurezza, riguarda la criptazione dei dati sensibili trasmessi al server (e-mail, password, alcuni dati relativi a revisioni).

Per gli utenti non registrati tali problematiche non sono affatto presenti in quanto non vi sono informazioni sensibili da memorizzare su di loro. La priorità è quindi data interamente all'efficienza.

Prestazioni vs Costi:

Essendo un progetto universitario e non avendo risorse finanziarie, tutte le tecnologie da noi scelte sono distribuite sotto licenza open-source o free-ware. Per questo motivo i costi non sono un aspetto da tenere in considerazione.

Per quanto riguarda invece le prestazioni, tali tecnologie forniscono comunque dei buoni risultati sotto diversi aspetti e per questo rappresentano una buona soluzione per la realizzazione del sistema.

1.2 Interface documentation guidelines

Durante l'implementazione del progetto è necessario che siano seguite le guide linea qui sottoesposte.

È anche importante commentare ciò che viene fatto nel codice, tramite normali commenti o strumenti come ad esempio Javadoc che permette di ottenere una vera e propria documentazione.

Convenzioni sui nomi

I nomi utilizzati per la rappresentazione dei concetti principali, delle funzionalità e delle componenti generiche del sistema devono rispettare le seguenti condizioni:

Quando si utilizzano dei nomi per rappresentare componenti, concetti, operazioni o funzionalità di CarCheck bisogna stabilire che essi:

- Non devono avere una lunghezza "eccessiva", quando possibile;
- Siano in lingua inglese;
- Non siano composti da caratteri speciali (?@! Etc.).

È importante inoltre effettuare una distinzione tra **variabili** e **costanti**:

Per rappresentare una ***variabile*** bisogna:

- Utilizzare caratteri minuscoli, o almeno evitare che il primo carattere sia maiuscolo;
- Non utilizzare caratteri speciali (ad esempio #, @, $, %);
- Evitare la presenza di numeri, ove possibile.

Riassumendo si vuole evitare la Pascal Notation ed utilizzare invece la Camel Notation. Questo è un esempio corretto:

```
private int variableName;
```

Mentre questo è un esempio di errore:

```
private int variablename#2;
```

Per rappresentare una ***costante*** le convenzioni da seguire sono:

- Utilizzare soltanto caratteri maiuscoli;
- Per separare due parole distinte utilizzare il carattere underscore (“_”);
- Non utilizzare il carattere underscore all’inizio di una parola o parola composta;
- Non utilizzare caratteri speciali (ad esempio #, @, $, %);
- Evitare la presenza di numeri, ove possibile.

Questo è un esempio corretto:

```
private final int VARIABLE_NAME;
```

Questo è un esempio di errore:

```
private int _VARIABLENAME;
```

Una seconda distinzione viene fatta nella notazione tra **classi, interfacce e package**.

Classe

- Devono iniziare con una lettera maiuscola;
- Ogni parola che segue la prima deve iniziare con una lettera maiuscola;
- Il nome deve essere un ***sostantivo*** (Ad esempio Color, Button, System ecc.);
- Evitare di utilizzare acronimi.

Questo è un esempio corretto:

```
public class VehicleBean { … }
```

Questo è un esempio di errore:

```
public class vehiclebean { … }
```

Interfaccia

- Devono inizare con una I (i) maiuscola, seguita dalla prima lettera del nome in maiuscolo;
- Evitare di utilizzare acronimi

Questo è un esempio corretto:

```
public interface IMyInterface { … }
```

Questi sono esempi di errore:

```
public interface MyInterface { … }
public interface ImyInterface { … }
```

Package

- Il nome del package deve essere in minuscolo;
- Se esso contiene più parole, ciascuna di esse deve essere separata da un punto “.”
- Il nome ne deve ricordare il contenuto

Metodo

- Deve iniziare con una lettera minuscola;
- Dovrebbe essere un ***verbo*** (print(), doSomething());
- Se il nome contenesse più parole, la prima parola dovrebbe iniziare con una lettera minuscola mentre le altre con una lettera maiuscola. (Camel Notation);
- I nomi get() , set() sono riservati ai metodi getters and setters;
- Il nome is…() é riservato ai metodi getters dei valori booleani;

Enumeratori

- Vanno definiti come classi;
- Ogni valore che può assumere un enumeratore è descritto come una costante statica accessibile pubblicamente.

La scelta è dovuta alla cattiva gestione degli enumeratori nel linguaggio di implementazione scelto (Java). Per una maggiore comprensione, è allegato un esempio:

```
public class MyEnum {

  public static final String VALUE_X;
  public static final String VALUE_Y;
  public static final String VALUE_Z;
}
```

2. Packages

Il nostro sistema viene implementato secondo il pattern architetturale MVC (Model View Controller). Esso è suddiviso, dopo il primo package it, in due package principali

- carcheck
- dsoft

Il primo contiene tutti i sotto-package atti a gestire l'intero sistema. Il secondo contiene i sotto-package, implementati dal team, che si occupano di gestire le operazioni di base per la gestione dei dati persistenti (CRUD).

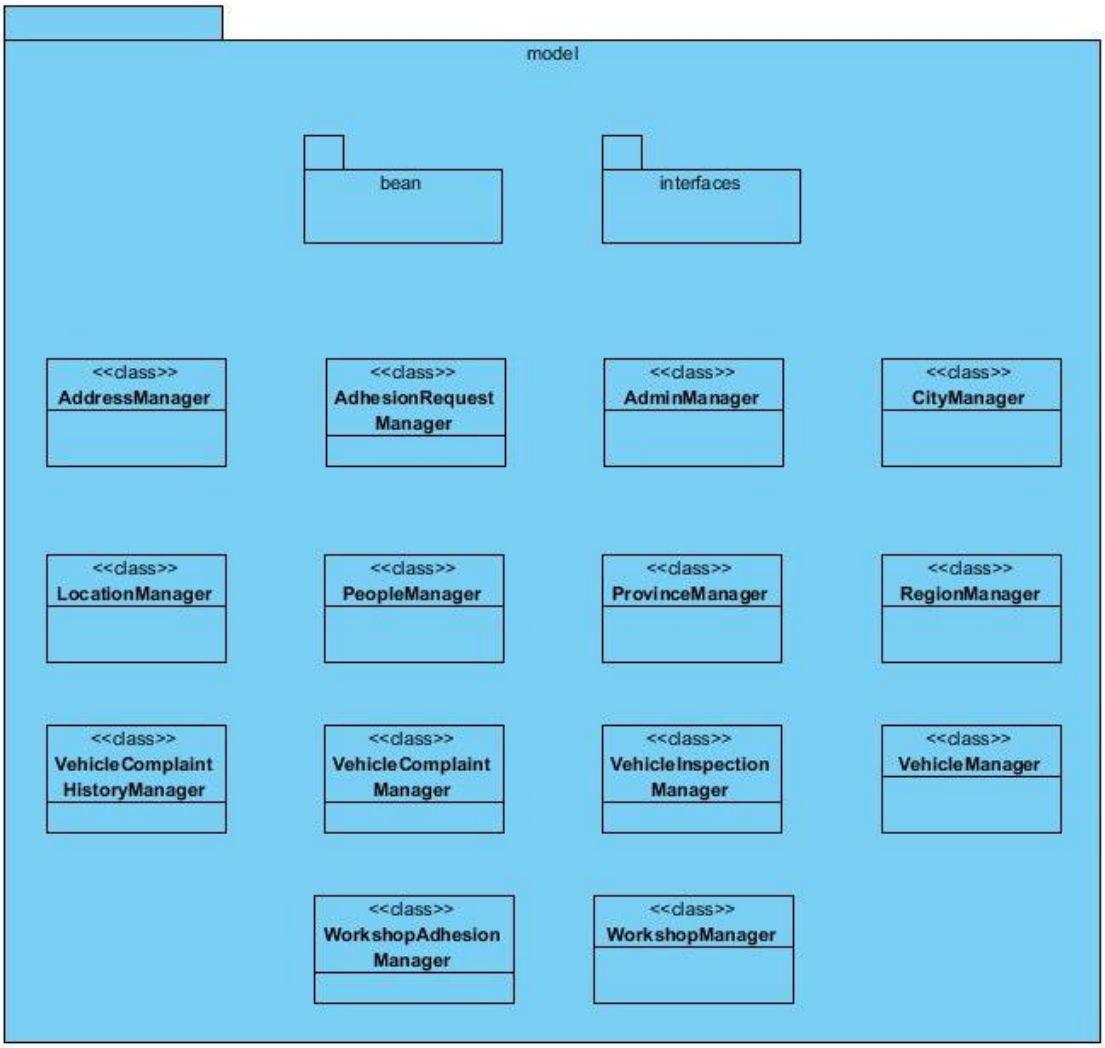
2.1 Package carcheck

Il package è suddiviso come segue:

- model
- control
- database
- utility

Inoltre, è presente una classe StartupListener che si occupa di gestire le operazioni da effettuare all'avvio e all'interruzione del server.

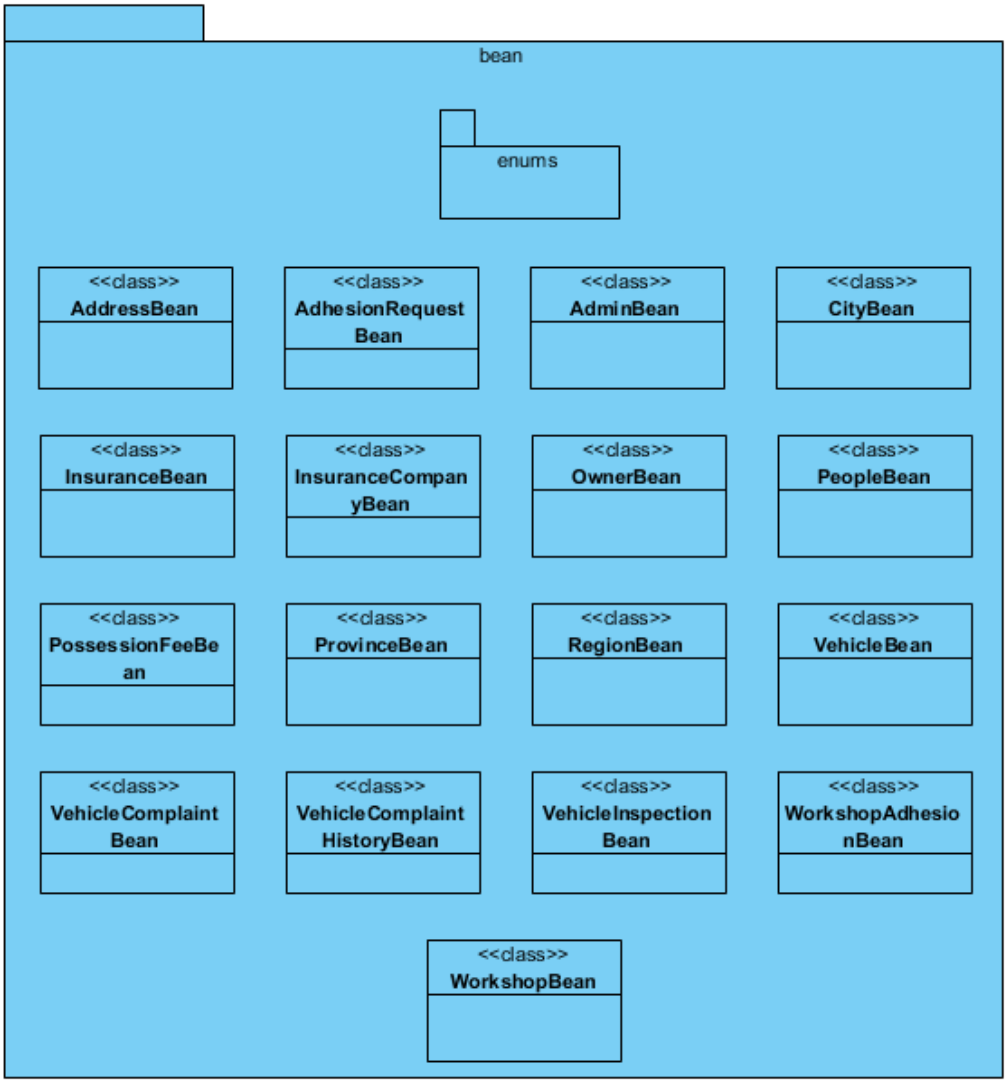
2.1.1 MODEL



Questo package si occupa di fornire classi che consentono l'accesso ai dati utili all'applicazione. Esso è composto dalle seguenti classi:

Classe	Descrizione
AddressManager	Manager che si occupa della gestione dei dati degli indirizzi.
AdhesionManager	Manager che si occupa della gestione dei dati delle richieste di adesione
AdminManager	Manager che si occupa della gestione dei dati di un amministratore del sistema
CityManager	Manager che si occupa della gestione dei dati di una città
LocationManager	Classe che si occupa di gestire AddressManager, CityManager, ProvinceManager e RegionManager
PeopleManager	Manager che si occupa della gestione dei dati di una persona fisica nel sistema
ProvinceManager	Manager che si occupa della gestione dei dati delle province
RegionManager	Manager che si occupa della gestione dei dati delle regioni
VehicleComplaintHystoryManager	Manager che si occupa della gestione dei dati riguardo lo storico delle denunce di un veicolo
VehicleComplaintManager	Manager che si occupa della gestione dei dati delle denunce presenti attualmente su un veicolo
VehicleInspectionManager	Manager che si occupa della gestione dei dati delle revisioni di un veicolo
VehicleManager	Manager che si occupa della gestione dei dati dei veicoli memorizzati nel sistema
WorkshopAdhesionManager	Manager che si occupa della gestione dei dati ottenuti dall'unione (JOIN) tra un'officina e una richiesta di adesione
WorkshopManager	Manager che si occupa della gestione dei dati delle officine

MODEL – Package bean



Questo package contiene tutte le cassi che rappresentano le tabelle, con eventuali join, presenti nel database. Come suggerisce il nome del package, queste classi sono oggetti Bean e quindi hanno un costruttore vuoto e sono composte da soli metodi getter e setter. Nello specifico è composto dalle seguenti classi:

Classe	Descrizione
AddressBean	Classe che rappresenta la struttura della tabella address
AdhesionRequestBean	Classe che rappresenta la struttura della tabella adhesionrequestmanager
AdminBean	Classe che rappresenta la struttura della tabella admin
CityBean	Classe che rappresenta la struttura della tabella city

InsuranceBean	Classe che rappresenta la struttura della tabella insurance
InsuranceCompanyBean	Classe che rappresenta la struttura della tabella insurancecompany
OwnerBean	Classe che rappresenta la struttura della tabella owner
PeopleBean	Classe che rappresenta la struttura della tabella people
PossessionFeeBean	Classe che rappresenta la struttura della tabella possessionfee
ProvinceBean	Classe che rappresenta la struttura della tabella province
RegionBean	Classe che rappresenta la struttura della tabella adhesionrequestmanager
VehicleBean	Classe che rappresenta la struttura della tabella vehicle
VehicleComplaintHystoryBean	Classe che rappresenta la struttura della tabella vehiclecomplainthystory
VehicleComplaintBean	Classe che rappresenta la struttura della tabella vehiclecomplaint
VehicleInspectionBean	Classe che rappresenta la struttura della tabella vehicleinspection
WorkshopAdhesionBean	Classe che rappresenta la struttura della tabella workshop unita (JOIN) con la tabella adhesion
WorkshopBean	Classe che rappresenta la struttura della tabella workshop

MODEL – Package bean.enums

Questo package contiene tutti gli enumeratori che sono riferiti al contesto del package bean. In particolare:

Enum	Descrizione
EuroClass	Collezione di valori possibili per le varie categorie EURO di un veicolo
Fuel	Collezione di valori possibili per le categorie di alimentazione
Gender	Collezioni di valori possibili per indicare il genere di una persona
Grade	Collezione di valori possibili per indicare i permessi di un amministratore
RequestStatus	Collezione di valori possibili per indicare lo stato di una richiesta di adesione

VehicleCategory	Collezione di valori possibili per indiare la categoria di un veicolo

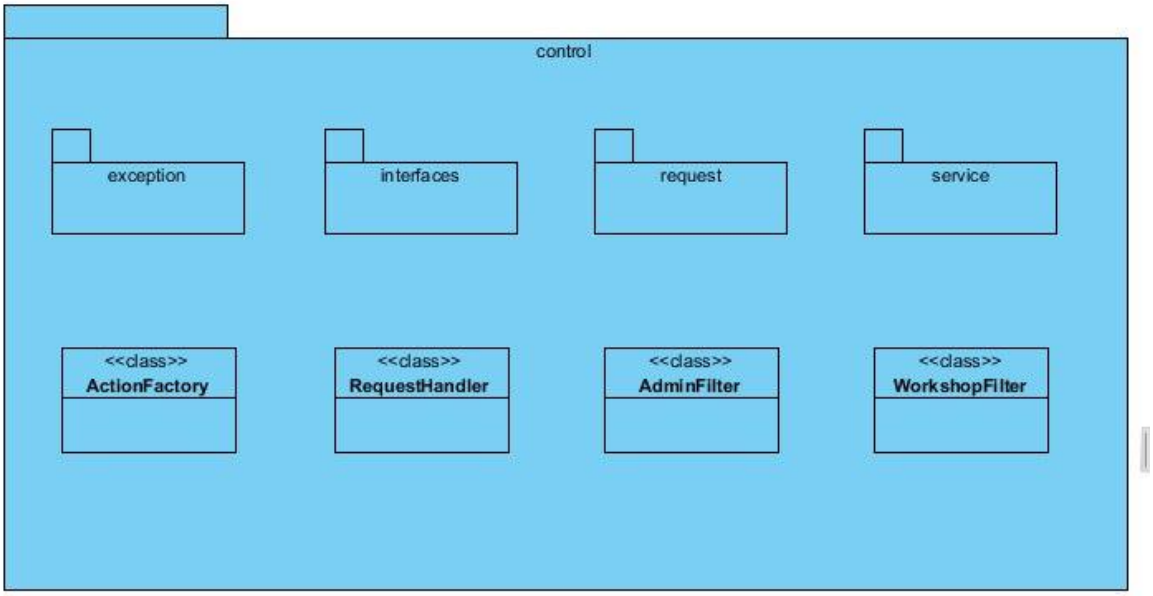
MODEL – Package interfaces

Questo package contiene tutte le interfacce per le classi che si occupano di consentire l'accesso ai dati utili all'applicazione (Manager). In particolare, ci sono le seguenti interfacce:

Interfaccia	Descrizione
IAddress	Interfaccia che deve essere implementata da tutte le classi che intendono gestire i dati relativi agli indirizzi.
IAdhesionRequest	Interfaccia che deve essere implementata da tutte le classi che intendono gestire i dati relativi alle richieste di adesione
IAdmin	Interfaccia che deve essere implementata da tutte le classi che intendono gestire i dati relativi agli admin
ICity	Interfaccia che deve essere implementata da tutte le classi che intendono gestire i dati relativi alle città
IDatabaseOperation	Interfaccia che definisce tutte le operazioni di base per l'accesso ai dati
IPeople	Interfaccia che deve essere implementata da tutte le classi che intendono gestire i dati relativi alle persone nel database
IProvince	Interfaccia che deve essere implementata da tutte le classi che intendono gestire i dati relativi alle province
IRegion	Interfaccia che deve essere implementata da tutte le classi che intendono gestire i dati relativi alle regioni
IUser	Interfaccia che definisce le operazioni di base per gli utenti del sistema
IVehicle	Interfaccia che deve essere implementata da tutte le classi che intendono gestire i dati relativi ai veicoli
IVehicleComplaint	Interfaccia che deve essere implementata da tutte le classi che intendono gestire i dati relativi alle denunce sui veicoli
IVehicleComplaintHistory	Interfaccia che deve essere implementata da tutte le classi che intendono gestire i dati relativi allo storico delle denunce sui veicoli

IVehicleInspection	Interfaccia che deve essere implementata da tutte le classi che intendono gestire i dati relativi alle revisioni dei veicoli
IWorkshop	Interfaccia che deve essere implementata da tutte le classi che intendono gestire i dati relativi alle officine
IWorkshopAdhesion	Interfaccia che deve essere implementata da tutte le classi che intendono gestire i dati relativi ai veicoli uniti (JOIN) con le richieste di adesione

2.1.2 CONTROL



Questo package si occupa di gestire la logica di controllo del sistema. E' composto dalle seguenti classi:

Classe	Descrizione
ActionFactory	Classe che si occupa di restituire l'implementazione dell'interfaccia IAction (vedi interfaces di questo package) in base al metodo (GET/POST) e al pathinfo della richiesta.
RequestHandler	Rappresenta l'unico controller del sistema che fornisce un punto di ingresso centralizzato per tutte le richieste.
AdminFilter	Filtro che impedisce l'acceso alle pagine riservate agli amministratori se non si è loggati come tali.
WorkshopFilter	Filtro che impedisce l'acceso alle pagine riservate alle officine se non si è loggati come tali.

CONTROL – Package service

Il sotto-package service si occupa delle richieste che, una volta elaborate, restituiscono un risultato in JSON. E' composto dalle seguenti classi:

Classe	Descrizione
AddressAction	Classe che si occupa della restituzione delle informazioni, in formato JSON, relative agli indirizzi presenti nel sistema
AdminAction	Classe che si occupa delle operazioni di gestione degli admin. Ogni operazione restituisce l'esito in formato JSON
AdminStatisticsAction	Classe che si occupa della restituzione delle informazioni, in formato JSON, delle statistiche del sistema. Alcuni esempi sono il numero di veicoli registrati e il numero di officine aderenti.
ChangePasswordAction	Classe che si occupa dell'operazione di cambio password per gli utenti del sistema. L'operazione restituisce l'esito in formato JSON.
CityAction	Classe che si occupa della restituzione delle informazioni, in formato JSON, relative alle città presenti nel sistema
EmailCheckerAction	Classe che si occupa delle operazioni relative alla convalida delle e-mail. I risultati vengono restituiti in formato JSON
PeopleAction	Classe che si occupa delle operazioni di gestione (aggiunta, modifica e rimozione) delle persone fisiche memorizzate nel sistema. Ogni operazione restituisce l'esito in formato JSON
ProvinceAction	Classe che si occupa della restituzione delle informazioni, in formato JSON, relative alle province presenti nel sistema
RegionAction	Classe che si occupa della restituzione delle informazioni, in formato JSON, relative alle regioni presenti nel sistema
RequestsAction	Classe che si occupa delle operazioni di gestione (appuntamento, approvazione, rifiuto) delle richieste di adesioni da parte delle officine. Ogni operazione restituisce l'esito in formato JSON
VehicleAction	Classe che si occupa della restituzione delle informazioni, in formato JSON, relative ai veicoli presenti nel sistema

CONTROL – Package request

Il sotto-package request si occupa delle richieste che, una volta elaborate, rimandano le informazioni elaborate a pagine JSP. E' composto dalle seguenti classi:

Classe	Descrizione
AdmiList	Classe che si occupa di restituire alla pagina admins.jsp la lista di admin presenti nel sistema.
AdminLoginAction	Classe che si occupa di effettuare l'operazione di login dell'admin.
AdminSignupAction	Classe che si occupa di effettuare la registrazione di un nuovo amministratore nel sistema.
AdminViewAction	Classe che si occupa di reindirizzare le informazioni necessarie per aggiungere o modificare un admin alla pagina adminForm.jsp
LogoutAction	Classe che si occupa di eseguire le operazioni di logout per l'utente connesso.
ShowVehicleAction	Classe che si occupa di reindirizzare le informazioni relative ai veicoli memorizzati alla pagina vehicles.jsp
VehicleFindAction	Classe che si occupa di elaborare le informazioni dei veicoli agli utenti data una determinata targa. Le informazioni vengono inoltrate alla pagina result.jsp
ViewAdhesionAction	Classe che si occupa di elaborare le informazioni relative alle adesioni, inoltrando le informazioni di quest'ultime alla pagina adhesionView.jsp
WorkshopEditInspectionAction	Classe che si occupa di elaborare le informazioni da modificare di un determinato veicolo, inoltrandole alla pagina inspectionView.jsp
WorkshopInsertInspectionAction	Classe che si occupa di gestire le informazioni che devono essere visualizzate nel caso si voglia aggiungere una nuova revisione. Tali informazioni vengono inoltrate alla pagina inspectionView.jsp
WorkshopInspectionOperationAction	Classe che si occupa di visualizzare le informazioni relative alle revisioni su un determinato veicolo.
WorkshopLoginAction	Classe che si occupa della restituzione delle informazioni, in formato JSON, relative alle regioni presenti nel sistema
WorkshopSignupAction	Classe che si occupa della registrazione di una nuova officina.
WorkshopEditInspectionAction	Classe che si occupa di elaborare le informazioni da visualizzare di una determinata revisione, inoltrandole alla pagina inspectionView.jsp

CONTROL – Package interfaces

Questo sotto-package contiene tutte le interfacce utili ai sotto-package di control. In particolare, contiene la seguente interfaccia:

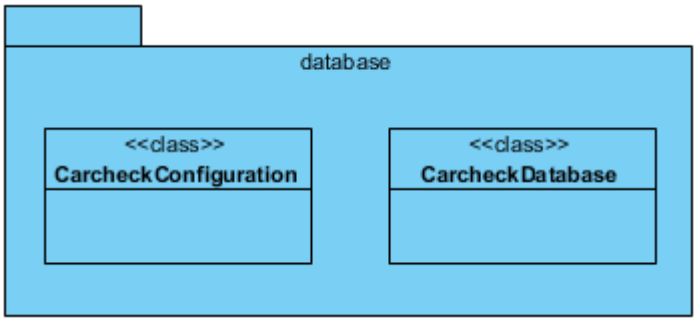
Interfaccia	Descrizione
IAction	Interfaccia che deve essere implementata da tutte le classi che vogliono rappresentare una richiesta.

CONTROL – Package exception

Questo sotto-package contiene tutte le eccezioni che possono essere lanciate dalle varie classi presenti nel package control. In particolare, contiene le seguenti classi:

Classe	Descrizione
ActionException	Eccezione che rappresenta una generica eccezione catturata durante l'esecuzione di una classe che implementa IAction
ActionNotFoundException	Eccezione che viene catturata quando non esiste nessuna action con quel determinato metodo e quel determinato pathInfo
ActionBadFormedException	Eccezione catturata quando una richiesta non è formata nella maniera in cui la si aspetta.

2.1.3 DATABASE



Questo package contiene le classi utili alla configurazione del database. Le classi qui contenute estendono tutte quelle che si trovano nel package dsoft. Le classi contenute sono di seguito descritte:

Classe	Descrizione
CarcheckConfiguration	Classe che rappresenta la configurazione del database.
CarcheckDatabase	Classe che implementa le operazioni CRUD del database

2.1.4 UTILITY

Package che contiene classi utili al sistema che non trovano locazione in nessuno dei package precedenti. Le classi sono di seguito descritte:

Classe	Descrizione
EmailSender	Classe che si occupa di fornire i metodi per inviare e-mail.
JSonResponse	Classe che si occupa di generare le risposte in formato JSON
PasswordHasher	Classe che si occupa di criptare stringhe

2.2 Package dsoft

Il package è composto di un package principale fastcrud che è suddiviso in sotto-package come segue:

- core
- enums
- exceptions
- interfaces
- utility

Il package fastcrud contiene queste classi:

Classe	Descrizione
DatabaseConfiguration	Classe rappresentante la configurazione di un Database
FastCrud	Classe che fornisce i metodi principali per l'interazione con il Database
QueryGenerator	Classe di supporto per la generazione di Query
StatementFactory	Classe di supporto che genera gli Statement necessari per l'esecuzione di una query

2.2.1 CORE

Package che contiene al suo interno tutta la struttura portante del CRUD mediante la quale è possibile effettuare tutte le operazioni di base. Le classi sono di seguito descritte:

Classe	Descrizione
BeanEntity	Classe che mette a disposizione metodi per la gestione dei Bean
BeanEntityFactory	Classe che si occupa della creazione di BeanEntity
BeanManager	Classe che mette a disposizione una serie di metodi utili a popolare un BeanEntity
ConnectionPool	Classe necessaria per la creazione e rilascio di connessione SQL

CORE - Package Annotations

Package contenente tutte le Java Annotation utilizzate nei Bean per definire la struttura di una tabella nel Database. Le classi sono di seguito descritte:

Classe	Descrizione
Entity	Annotazione utile nel caso in cui un campo venga chiamato in modo diverso nel database
PrimaryKey	Annotazione che definisce quale proprietà rappresenta la chiave primaria nel database
Table	Annotazione che definisce il nome della tabella nel database

CORE - Package Enums

Package contenente enumeratori utilizzati nel package Core. Le classi sono di seguito descritte:

Classe	Descrizione
PrimaryKeyOption	Enumeratore che definisce i vari tipi di primary key da poter utilizzare

CORE - Package Interfaces

Package contenente tutte le interfacce utilizzate dalle classi del package Core. Le interfacce sono di seguito descritte:

Classe	Descrizione
IBeanEntity	Interfaccia che contiene tutte le firme dei metodi utilizzati dalla classe BeanEntity
IBeanEntityFactory	Interfaccia che contiene tutte le firme dei metodi utilizzati dalla classe BeanEntityFactory
IConnectionPool	Interfaccia che contiene tutte le firme dei metodi utilizzati dalla classe ConectionPool

2.2.2 ENUMS

Package contenente tutti gli enumeratori utilizzati dal Package fastcrud. Le classi sono di seguito descritte:

Classe	Descrizione
OperationType	Enumeratore che definisce le possibili operazioni da effettuare per la creazione delle query
StatementType	Enumeratore che definisce le possibili operazioni da effettuare per la creazione degli statement

2.2.3 EXCEPTIONS

Package contenente classi che estendono la classe Exception utilizzate per identificare tutte le eccezioni che possono avvenire mediante l'utilizzo del FastCrud. Le classi sono di seguito descritte:

Classe	Descrizione
ConfigurationException	Classe che rappresenta una eccezione scatenata da una errata configurazione del database
DatabaseUpdateException	Classe che rappresenta una eccezione scatenata da un update inaspettato sul database
ObjectMapException	Classe che rappresenta una eccezione scatenata da un mapping degli oggetti errato
TableNameException	Classe che rappresenta una eccezione scatenata quando un Bean non possiede il nome della tabella che rappresenta

2.2.4 INERFACES

Package contenente tutte le interfacce in uso all'interno del package fastcrud. Le interfacce sono definite come segue:

Classe	Descrizione
IDatabase	Interfaccia che mette a disposizione la firma di tutti i metodi di base per la gestione di un database

2.2.5 UTILITY

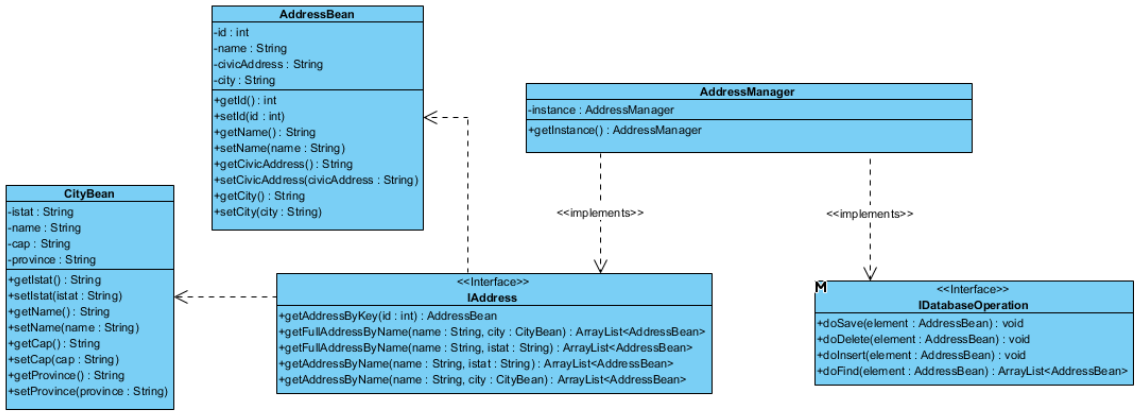
Package contenente tutte le classi che non hanno collocazione stabilita in altri package. Le classi sono definite come segue:

Classe	Descrizione
CharacterChecking	Classe che si occupa di rimuovere tutti i caratteri non validi per una possibile query e li sostituisce con caratteri utilizzabili in una query
Tuple	Classe che rappresenta un oggetto Tupla con Chiave - Valore

3. Class interfaces

In questa sezione vengono descritte le **classi manager** appartenenti al package model. Vengono illustrate le loro dipendenze, i loro metodi e i loro attributi. Le classi Bean vengono ommesse in quanto è possibile estrapolare le informazioni sopra citate dai manager stessi.

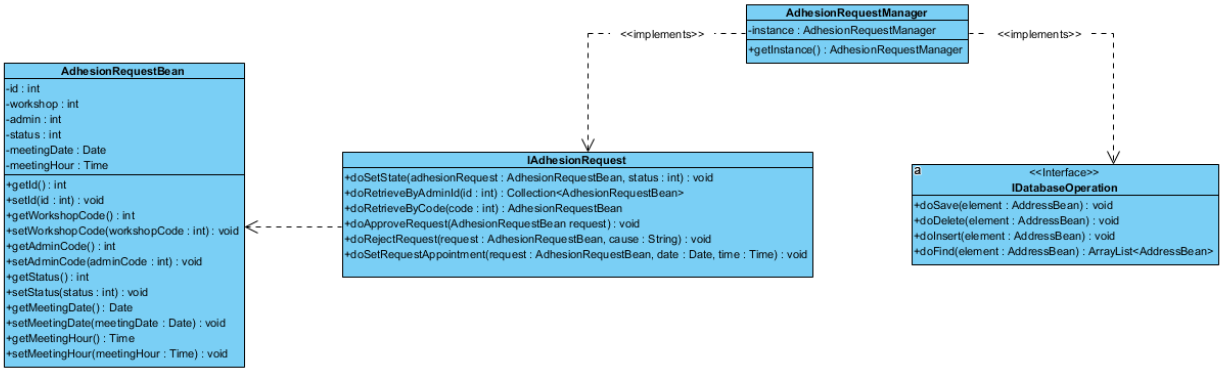
3.1 AddressManager



Classe	AddressManager
Descrizione	Manager che si occupa della gestione dei dati degli indirizzi
Pre-condizioni	**Context** AddressManager::getAddressByKey(id: int) **Pre: id > 0** **Context** AddressManager::getFullAddressByName(name: String, city: CityBean) **Pre: name != null && city != null** **Context** AddressManager::getFullAddressByName(name:String, istat:String) **Pre: name != null && istat != null** **Context** AddressManager::getAddressByName(name:String, istat:String) **Pre: name != null && istat != null** **Context** AddressManager::getAddressByName(name:String, city:CityBean) **Pre: name != null && city != null**
Post-condizioni	**Context** AddressManager::getAddressByKey(id: int) **Post:** address.getId() = id **Context** AddressManager::getFullAddressByName(name: String, istat:

	String) **Post:** ∀ x in addresses, x.getCity() = istat && x.getName() ≃ name **Context** AddressManager::getFullAddressByName(name: String, city: CityBean) **Post:** ∀ x in addresses, x.getCity() = city.getIstat() && x.getName() ≃ name **Context** AddressManager::getAddressByName(name:String, istat:String) **Post:** ∀ x in addresses, x.getCity() = istat && x.getName() ≃ name **Context** AddressManager::getAddressByName(name:String, city:CityBean) **Post:** ∀ x in addresses, x.getCity() = city.getIstat() && x.getName() ≃ name
Invarianti	

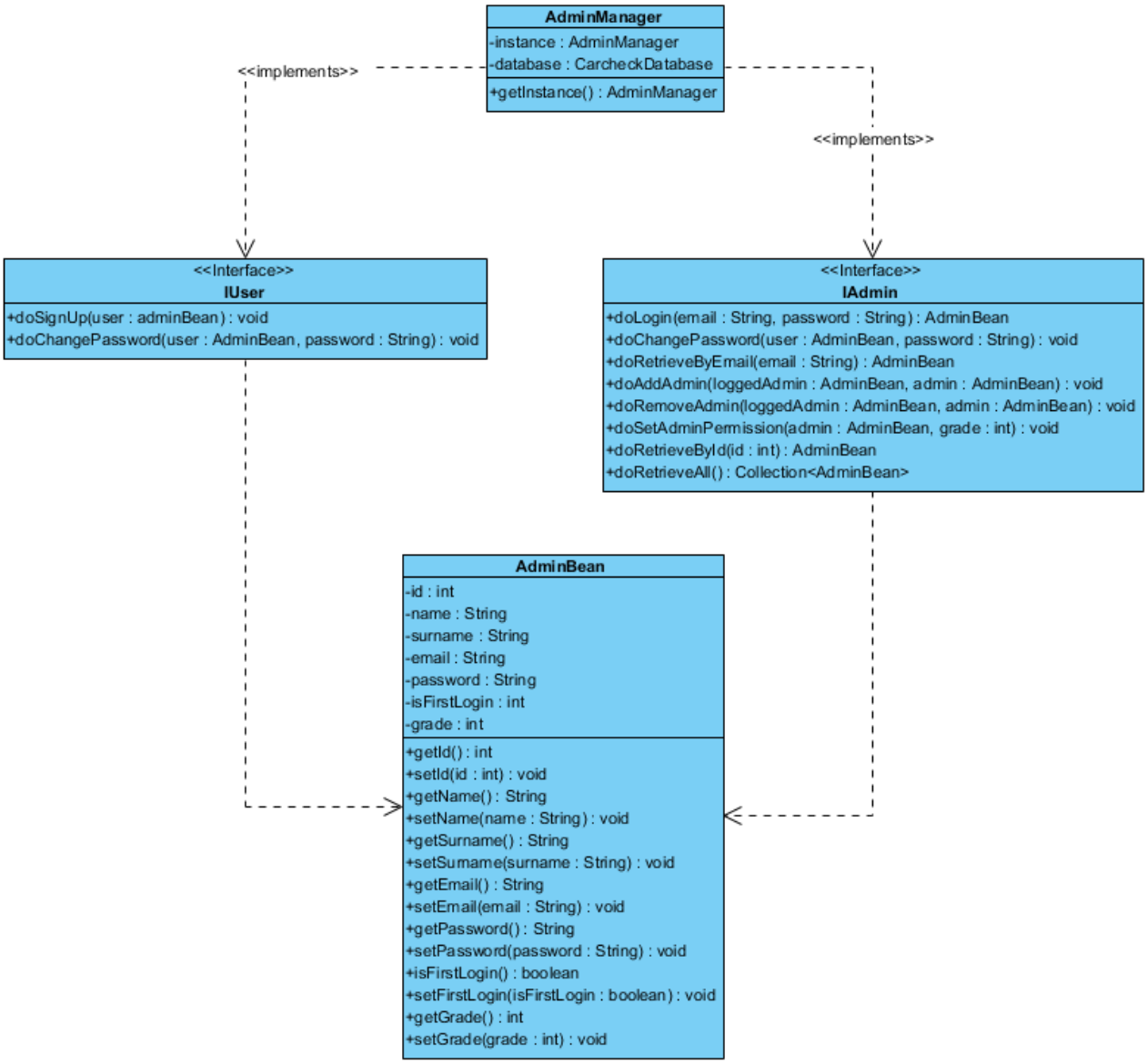
3.2 AdhesionRequestManager



Classe	AdhesionRequestManager
Descrizione	Manager che si occupa della gestione dei dati delle richieste di adesione
Pre-condizioni	**Context** AdhesionRequestManager::doSetState(adhesionRequest:AdhesionRequestManager, status: int) **Pre: adhesionRequest != null && (status >= 0 && status <=2)** **Context** AdhesionRequestManager::doRetrieveByAdminId (id:int) **Pre: id > 0** **Context** AdhesionRequestManager::doRetrieveByCode(code:int) **Pre: id > 0**

	Context AdhesionRequestManager::doApproveRequest(request: AdhesionRequestBean) **Pre: request != null** **Context** AdhesionRequestManager::doRejectRequest(request: AdhesionRequestBean) **Pre: request != null** **Context** AdhesionRequestManager::doSetRequestAppointment(request:AdhesionRequestBean, date:Date, time:Time) **Pre: request != null && date != null && time != null**
Post-condizioni	**Context** AdhesionRequestManager::doSetState(adhesionRequest:AdhesionRequestManager, status: int) **Post:** adhesionRequest.getStatus() = status **Context** AdhesionRequestManager::doRetrieveByAdminId (id:int) **Post:** $\forall$ x in adhesionRequests, x.getAdmin() = id **Context** AdhesionRequestManager::doRetrieveByCode(code:int) **Post:** adhesionRequest.getId() = code **Context** AdhesionRequestManager::doApproveRequest(request: AdhesionRequestBean) **Post:** request.getState() = 3 **Context** AdhesionRequestManager::doRejectRequest(request: AdhesionRequestBean) **Post:** request.getState() = 1 **Context** AdhesionRequestManager::doSetRequestAppointment(request:AdhesionRequestBean, date:Date, time:Time) **Post:** request.getState() = 2 && request.getDate() = date && request.getTime() = time
Invarianti	

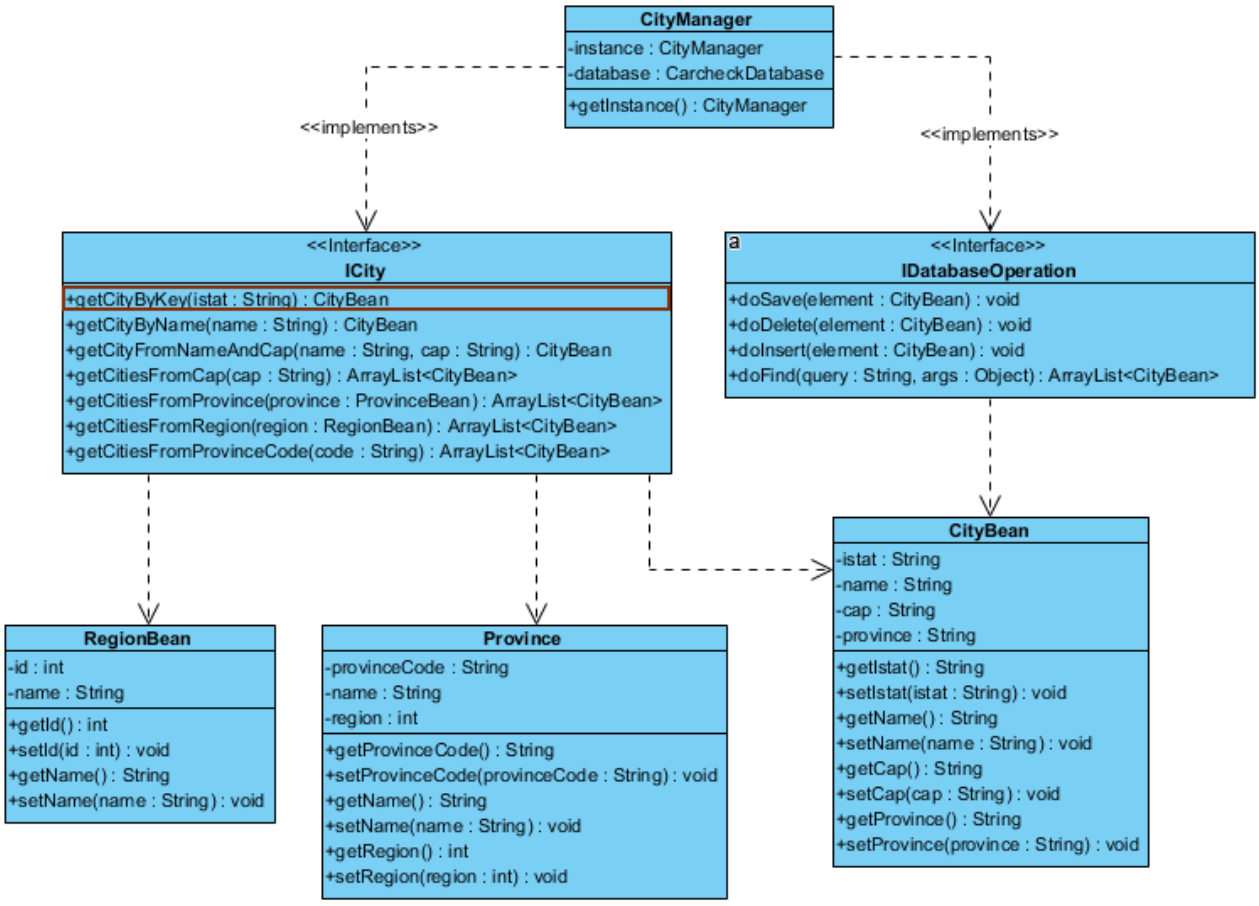
3.3 AdminManager



Classe	AdminManager
Descrizione	Manager che si occupa della gestione dei dati degli amministratore del sistema
Pre-condizioni	**Context** AdminManager::doLogin(email: String, password: String) **Pre: email != null && password != null** **Context** AdminManager::doChangePassword (user:AdminBean, password:String) **Pre: user != null && password != null** **Context** AdminManager::doRetrieveByEmail (email:String) **Pre: email != null** **Context** AdminManager::doAddAdmin(loggedAdmin: AdminBean, admin:

	AdminBean) **Pre: request != loggedAdmin != null && admin != null && loggedAdmin != admin** **Context** AdminManager::doRemoveAdmin(loggedAdmin: AdminBean, admin: AdminBean) **Pre: request != loggedAdmin != null && admin != null && loggedAdmin != admin** **Context** AdminManager::doSetRequestAppointment(request:AdhesionRequestBean, date:Date, time:Time) **Pre: request != null && date != null && time != null**
Post-condizioni	**Context** AdhesionRequestManager::doSetState(adhesionRequest:AdhesionRequestManager, status: int) **Post:** adhesionRequest.getStatus() = status **Context** AdhesionRequestManager::doRetrieveByAdminId (id:int) **Post:** ∀ x in adhesionRequests, x.getAdmin() = id **Context** AdhesionRequestManager::doRetrieveByCode(code:int) **Post:** adhesionRequest.getId() = code **Context** AdhesionRequestManager::doApproveRequest(request: AdhesionRequestBean) **Post:** request.getState() = 3 **Context** AdhesionRequestManager::doRejectRequest(request: AdhesionRequestBean) **Post:** request.getState() = 1 **Context** AdhesionRequestManager::doSetRequestAppointment(request:AdhesionRequestBean, date:Date, time:Time) **Post:** request.getState() = 2 && request.getDate() = date && request.getTime() = time
Invarianti	

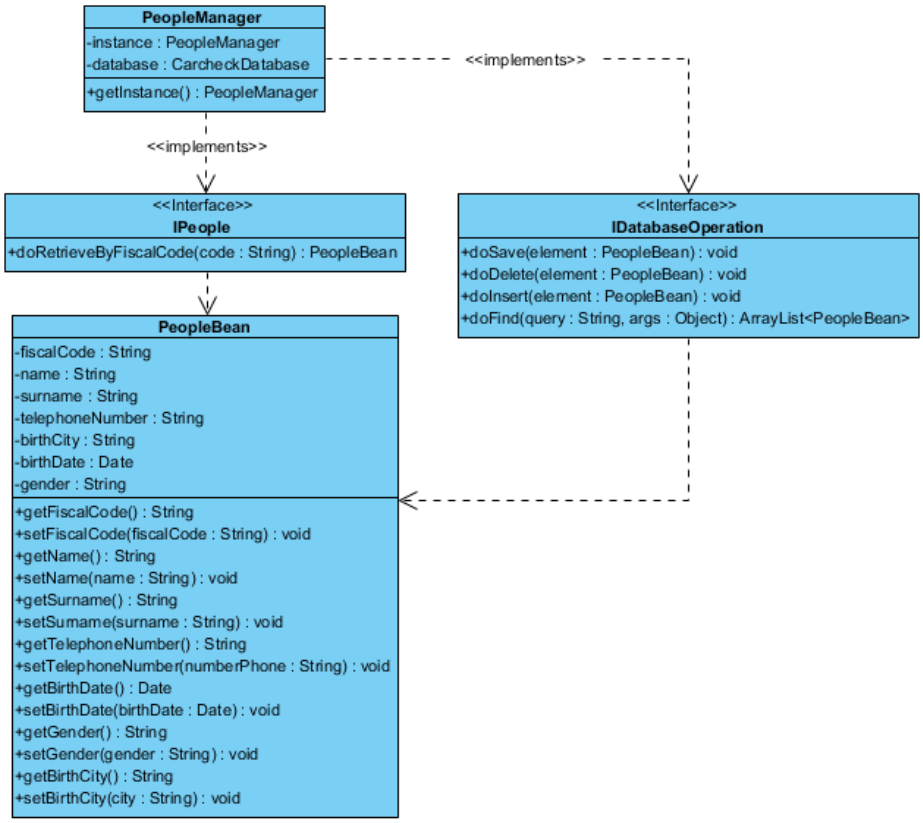
3.4 CityManager



Classe	CityManager
Descrizione	Manager che si occupa della gestione dei dati di una città
Pre-condizioni	**Context** CityManager::getCityByKey(istat: String) **Pre: istat != null** **Context** CityManager::getCityByName(name: String) **Pre: name != null** **Context** CityManager::getCityFromNameAndCap (name:String, cap:String) **Pre: name != null && cap != null** **Context** CityManager::getCitiesFromCap (cap:String) **Pre: cap != null** **Context** CityManager::getCitiesFromProvince (province:ProvinceBean) **Pre: province != null**

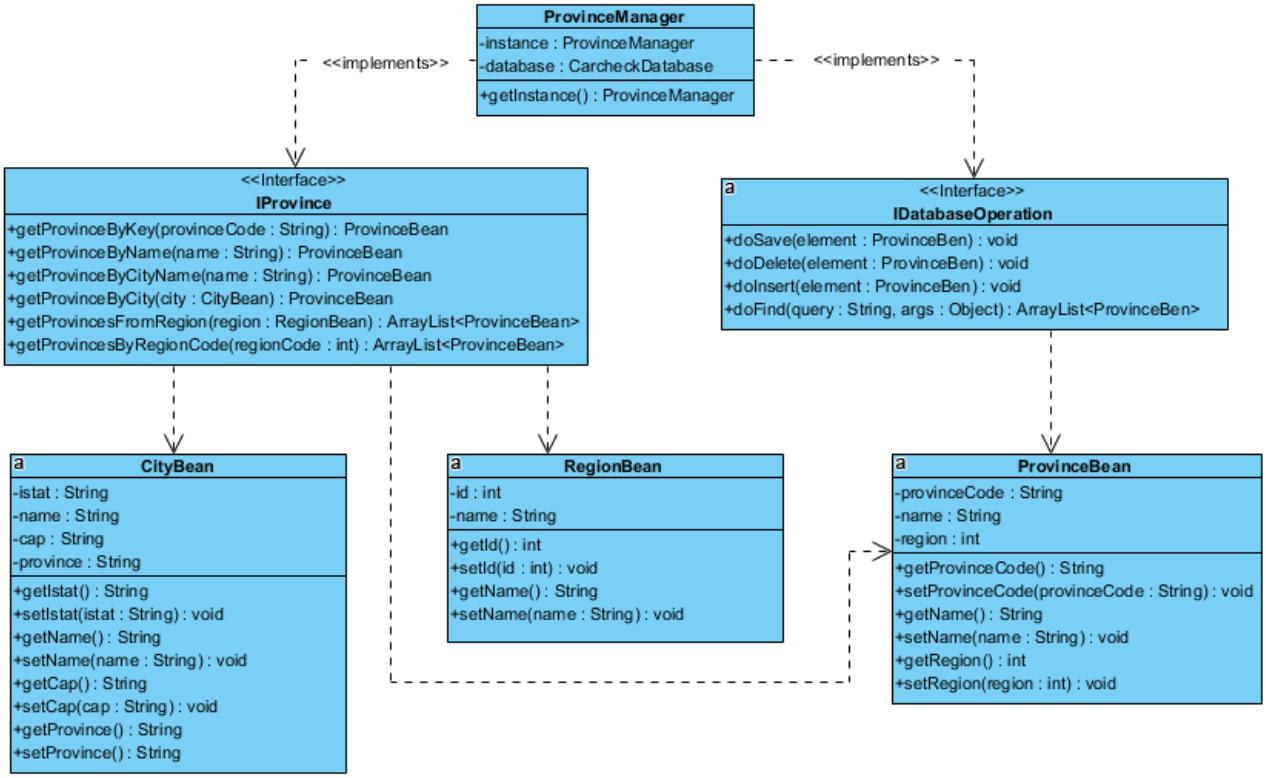
	Context CityManager::getCitiesFromRegion (region:RegionBean) **Pre: region != null** **Context** CityManager::getCitiesFromProvinceCode(code:int) **Pre: code > 0**
Post-condizioni	**Context** CityManager::getCityByKey(istat: String) **Post:** city.getIstat() = istat **Context** CityManager::getCityByName(name: String) **Post:** city.getName() = name **Context** CityManager::getCityFromNameAndCap (name:String, cap:String) **Post:** city.getName() = name && city.getCap() = cap **Context** CityManager::getCitiesFromCap (cap:String) **Post:** ∀ x in cities, x.getCap() = cap **Context** CityManager::getCitiesFromProvince (province:ProvinceBean) **Post:** ∀ x in cities, x.getProvince() = province.getProvinceCode() **Context** CityManager::getCitiesFromRegion(region:RegionBean) **Post:** ∀ x in cities, x.getProvince() = **Context** CityManager::getCitiesFromProvinceCode(code:int) **Post:** ∀ x in cities, x.getProvince() = code
Invarianti	

3.5 PeopleManager



Classe	PeopleManager
Descrizione	Manager che si occupa della gestione dei dati di una persona fisica nel sistema
Pre-condizioni	**Context** PeopleManager::doRetrieveByFiscalCode(code: String) **Pre: code != null**
Post-condizioni	**Context** PeopleManager::doRetrieveByFiscalCode(code: String) **Post:** people.getFiscalCode() = code
Invarianti	

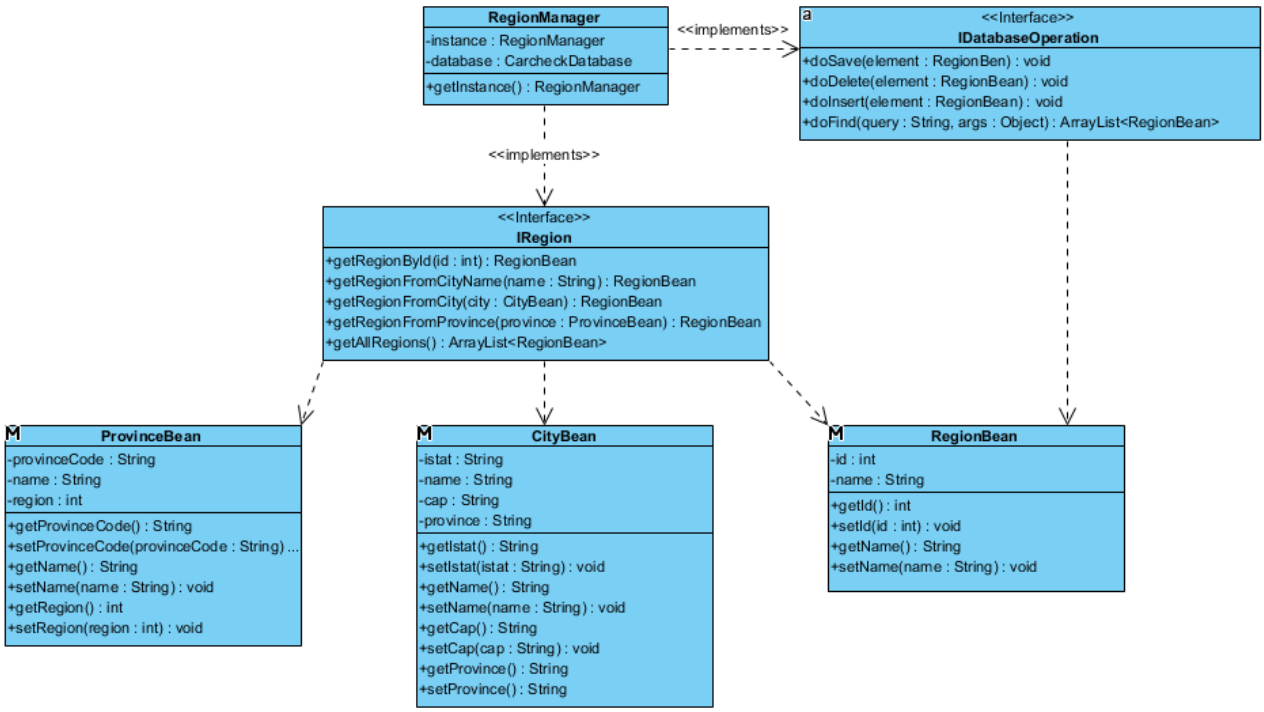
3.6 ProvinceManager



Classe	ProvinceManager
Descrizione	Manager che si occupa della gestione dei dati di una provincia
Pre-condizioni	**Context** ProvinceManager::getProvinceByKey(provinceCode: String) **Pre: provinceCode != null** **Context** ProvinceManager::getProvinceByName(name: String) **Pre: name != null** **Context** ProvinceManager::getProvinceByCityName(name:String) **Pre: name != null** **Context** ProvinceManager::getProvinceByCity (city:CityBean) **Pre: city != null** **Context** ProvinceManager::getProvincesFromRegion(region:RegionBean) **Pre: region != null** **Context** ProvinceManager::getProvincesByRegionCode(regionCode:int) **Pre: int > 0**
Post-condizioni	**Context** ProvinceManager::getProvinceByKey(provinceCode: String) **Post:** province.getProvinceCode() = provinceCode **Context** ProvinceManager::getProvinceByName(name: String)

	Post: province.getName() = name **Context** ProvinceManager::getProvinceByCityName(name:String) **Post:** La provincia ottenuta è quella della città che ha quel determinato nome **Context** ProvinceManager::getProvinceByCity (city:CityBean) **Post:** province.getProvinceCode() = city.getProvince() **Context** ProvinceManager::getProvincesFromRegion(region:RegionBean) **Post:** Ogni provincia ottenuta appartiene alla regione specificata **Context** ProvinceManager::getProvincesByRegionCode(regionCode:int) **Post:** Ogni provincia ottenuta appartiene alla regione con il codice specificato
Invarianti	

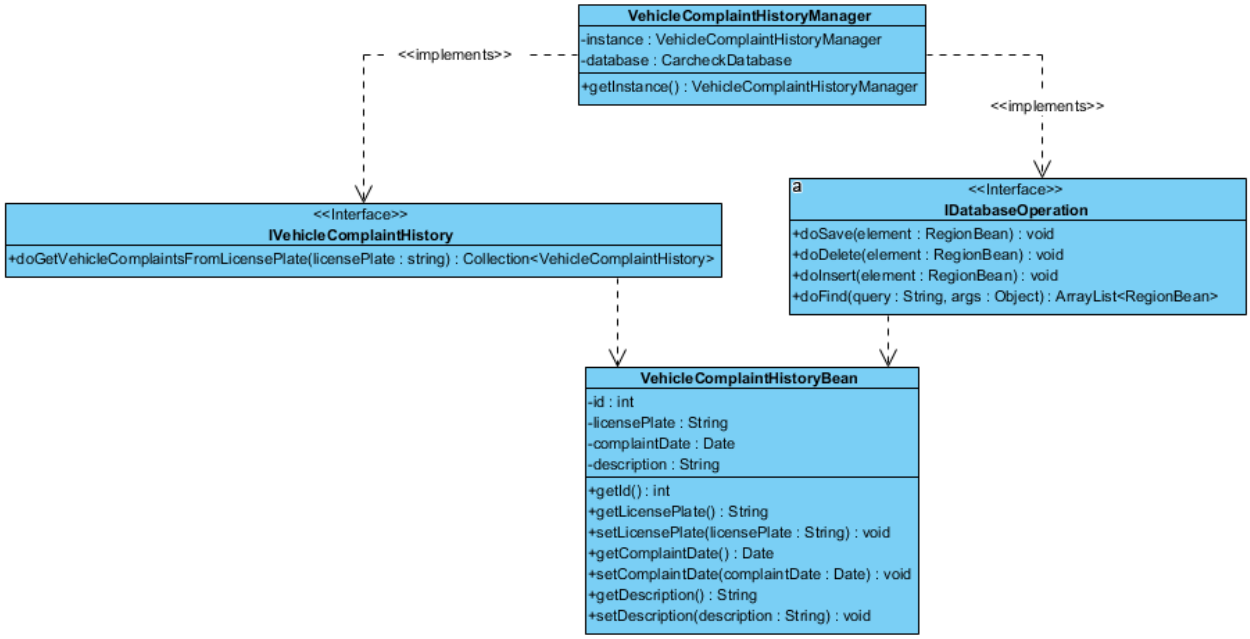
3.7 RegionManager



Classe	RegionManager
Descrizione	Manager che si occupa della gestione dei dati di una regione
Pre-condizioni	**Context** RegionManager::getRegionById(id: int) **Pre: id > 0** **Context** RegionManager::getRegionFromCityName(name: String) **Pre: name != null** **Context** RegionManager ::getRegionFromCity(city:CityBean) **Pre: city != null** **Context** RegionManager::getRegionFromProvince(province:ProvinceBean) **Pre: province != null** **Context** RegionManager::getAllRegions() **Pre:** *none*
Post-condizioni	**Context** RegionManager::getRegionById(id: int) **Post:** region.getId() = id **Context** RegionManager::getRegionFromCityName(name: String) **Post:** la regione ottenuta è quella della citta con il nome uguale a “name”

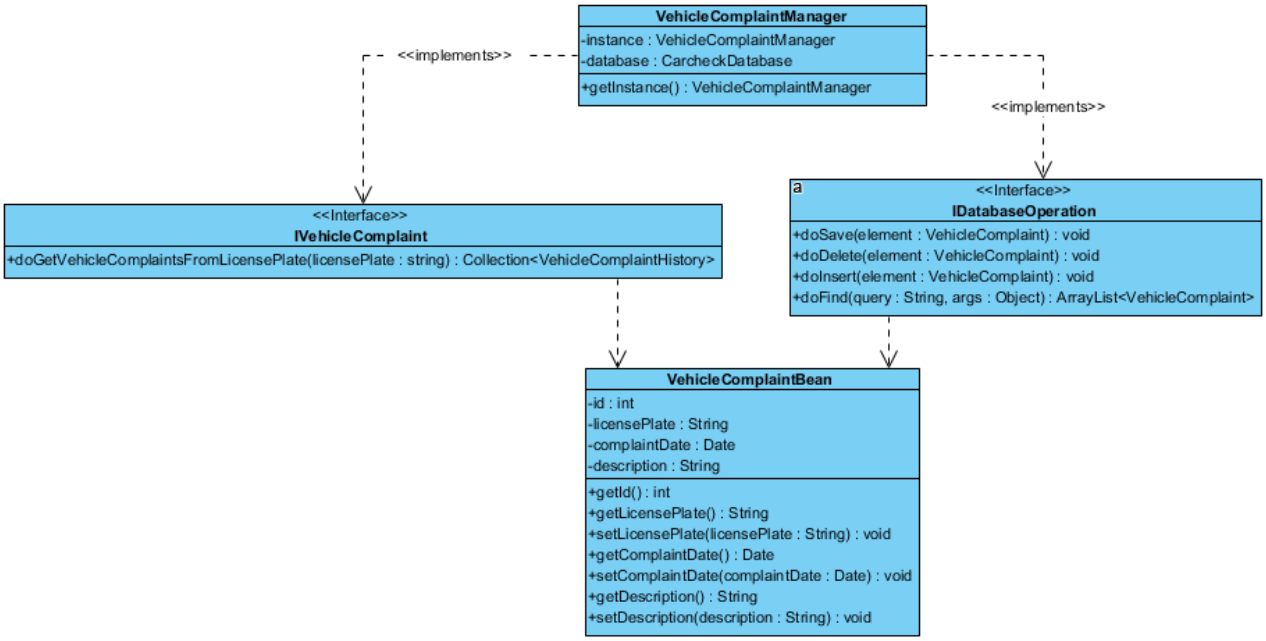
	Context RegionManager ::getRegionFromCity(city:CityBean) **Post:** La regione ottenuta è quella in cui si trova la città specificata **Context** RegionManager::getRegionFromProvince(province:ProvinceBean) **Post:** La regione ottenuta è quella della provincia specificata **Context** RegionManager::getAllRegions() **Post:** ∀ x in regions, x ∈ doFind("SELECT * FROM region");
Invarianti	

3.8 VehicleComplaintHistoryManager



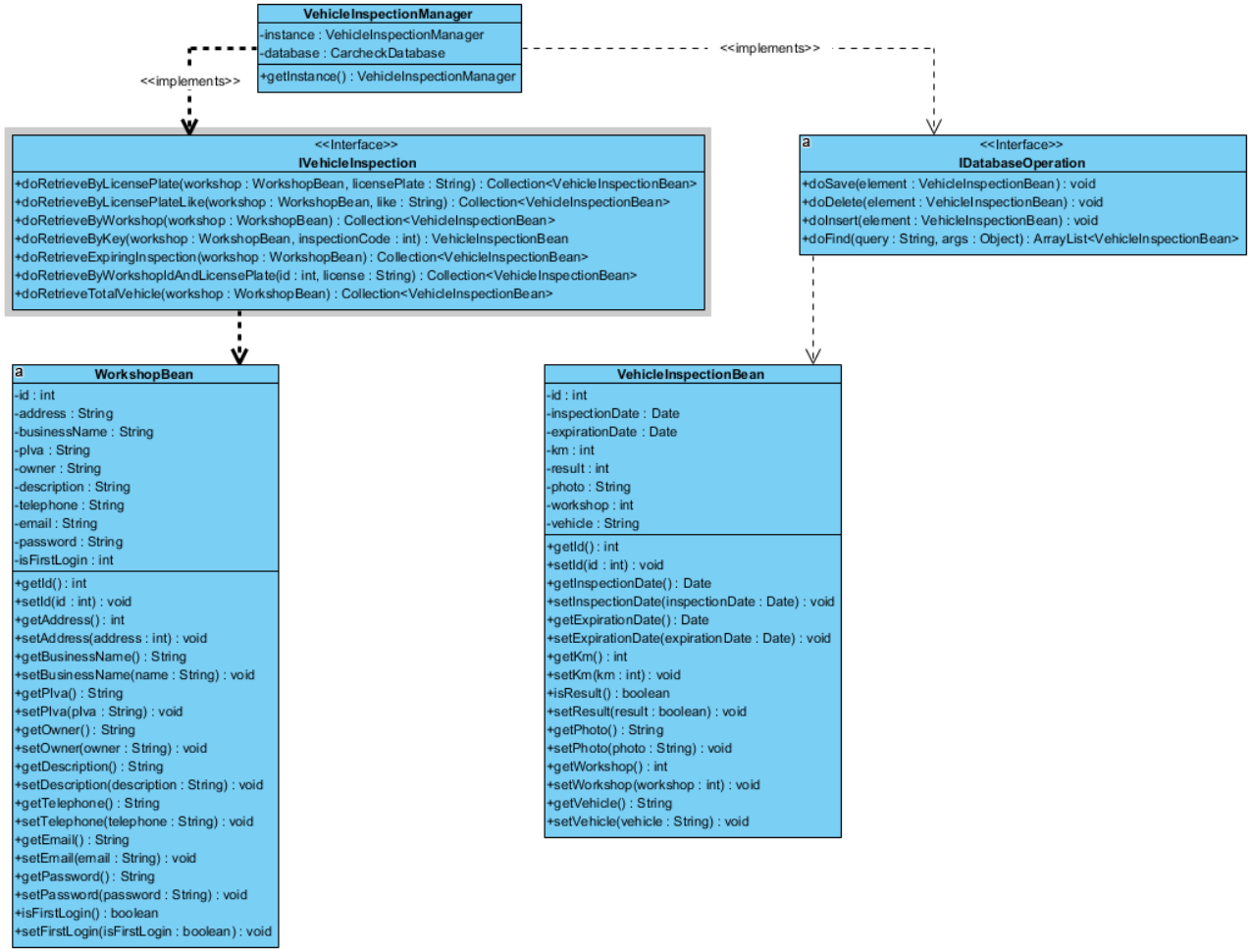
Classe	VehicleComplaintHistoryManager
Descrizione	Manager che si occupa della gestione dei dati relativi allo storico delle denunce su un veicolo
Pre-condizioni	**Context** VehicleComplaintHistoryManager::doGetVehicleComplaintsFromLicensePalte (licensePlate: String) **Pre: licensePlate != null**
Post-condizioni	**Context** VehicleComplaintHistoryManager::doGetVehicleComplaintsFromLicensePalte (licensePlate: String) **Post:** vehicleComplaintHistory.getVehicle() = licensePlate
Invarianti	

3.9 VehicleComplaintManager



Classe	VehicleComplaintManager
Descrizione	Manager che si occupa della gestione dei dati delle denunce presenti su un veicolo
Pre-condizioni	**Context** VehicleComplaintManager::doGetVehicleComplaintsFromLicensePalte (licensePlate: String) **Pre: licensePlate != null**
Post-condizioni	**Context** VehicleComplaintManager::doGetVehicleComplaintsFromLicensePalte (licensePlate: String) **Post:** vehicleComplaintHistory.getVehicle() = licensePlate
Invarianti	

3.10 VehicleInspectionManager

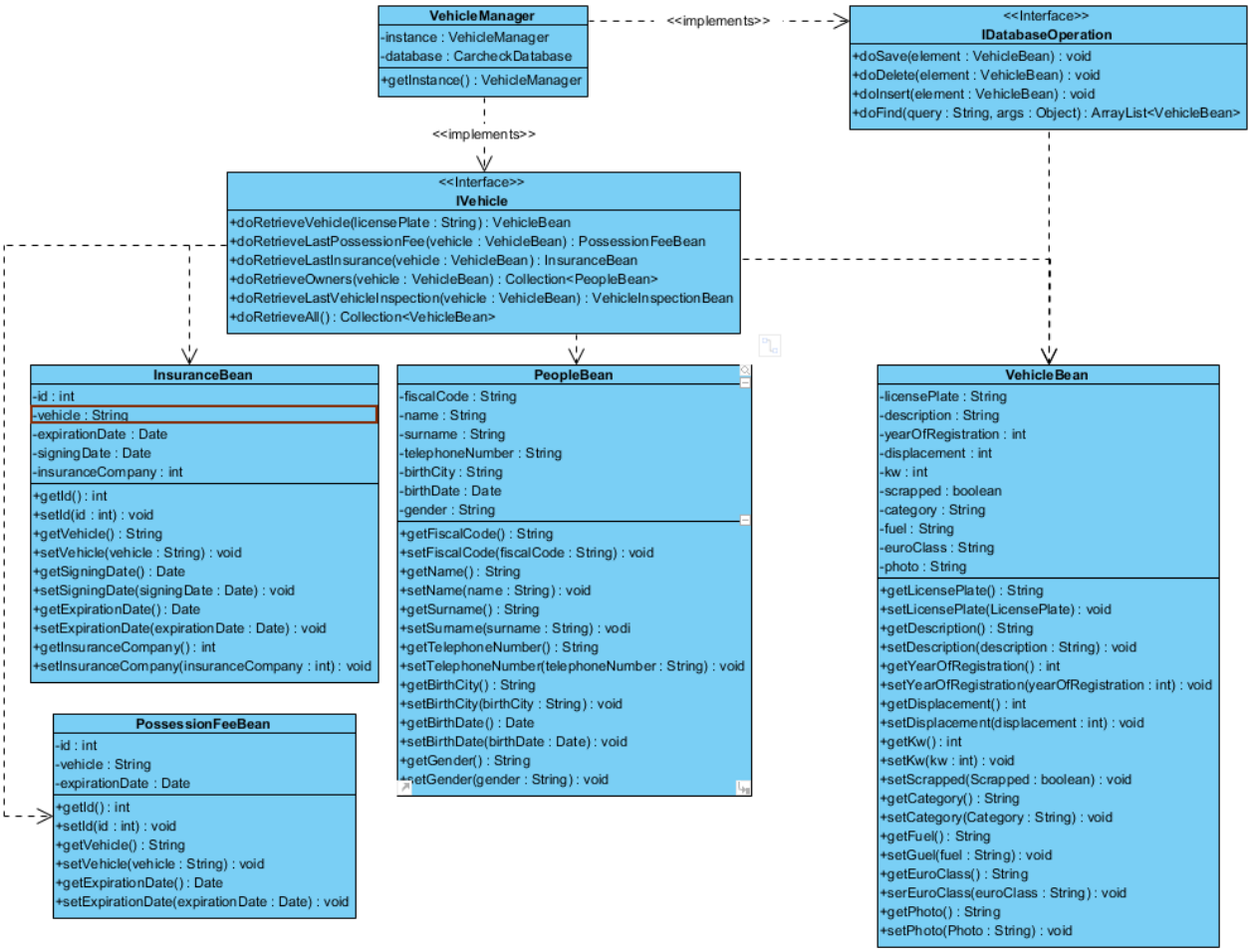


Classe	VehicleInspectionManager
Descrizione	Manager che si occupa della gestione dei dati delle revisioni di un veicolo
Pre-condizioni	**Context** VehicleInspectionManager::doRetrieveByLicensePlate(workshop:WorkshopBean, licensePlate: String) **Pre: licensePlate != null && workshop != null** **Context** VehicleInspectionManager::doRetrieveByLicensePlateLike(workshop:WorkshopBean, like: String) **Pre: like != null && workshop != null** **Context** VehicleInspectionManager::doRetrieveByWorkshop(workshop:WorkshopBean) **Pre: workshop != null**

	Context VehicleInspectionManager::doRetrieveByKey(workshop: WorkshopBean, inspectionCode: int) **Pre: workshop != null && inspectionCode > 0** **Context** VehicleInspectionManager::doRetrieveExpiringInspection (workshop: WorkshopBean) **Pre: workshop != null** **Context** VehicleInspectionManager::doRetrieveByWorkshopIdAndLicensePlate(id: int, license: String) **Pre: license != null && id > 0** **Context** VehicleInspectionManager::doRetrieveByWorkshopIdAndLicensePlate(id: int, license: String) **Pre: license != null && id > 0** **Context** VehicleInspectionManager::doRetrieveTotalVehicle (workshop: WorkshopBean) **Pre: workshop != null**
Post-condizioni	**Context** VehicleInspectionManager::doRetrieveByLicensePlate(workshop:WorkshopBean, licensePlate: String) **Post:** ∀ x in vehicleInspection, x.getVehicle() = licensePlate && x.getWorkshop() = workshop.getId() **Context** VehicleInspectionManager::doRetrieveByLicensePlateLike(workshop:WorkshopBean, like: String) **Post:** ∀ x in vehicleInspection, x.getVehicle() ≃ licensePlate && x.getWorkshop() = workshop.getId() **Context** VehicleInspectionManager::doRetrieveByWorkshop (workshop:WorkshopBean) **Post:** ∀ x in vehicleInspection, x.getWorkshop() = workshop.getId() **Context** VehicleInspectionManager::doRetrieveByKey(workshop: WorkshopBean, inspectionCode: int) **Post:** vehicleInspection.getWorkshop() = workshop.getId() && vehicleInspection.getId() = inspectionCode

	Context VehicleInspectionManager::doRetrieveExpiringInspection (workshop: WorkshopBean) **Post:** ∀ x in vehicleInspection, x.getWorkshop() = workshop.getId() && x.getExpirationDate() >= Now() && x.getExpriationDate() <= Now() + 30 giorni **Context** VehicleInspectionManager::doRetrieveByWorkshopIdAndLicensePlate(id: int, license: String) **Post:** ∀ x in vehicleInspection, x.getWorkshop() = workshop.getId() && x.getVehicle() = license **Context** VehicleInspectionManager::doRetrieveTotalVehicle (workshop: WorkshopBean) **Post:** ∀ x,y in vehicleInspection, x.getWorkshop() = workshop.getId() && x.getId() != y.getId()
Invarianti	

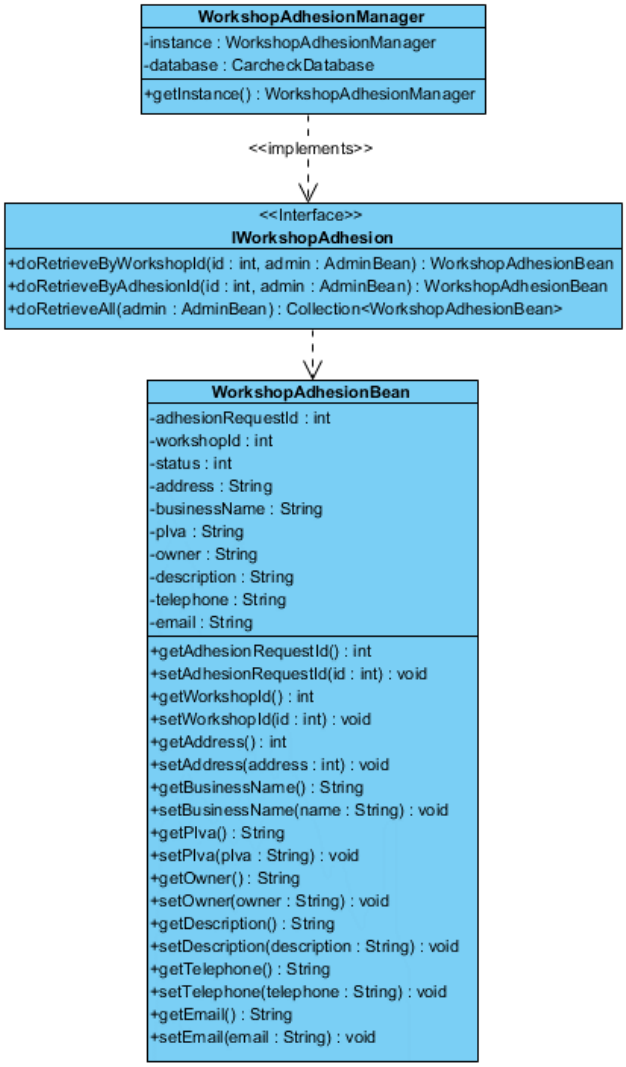
3.11 VehicleManager



Classe	VehicleManager
Descrizione	Manager che si occupa della gestione dei dati dei veicoli
Pre-condizioni	**Context** VehicleManager::doRetrieveLastPossessionFee(vehicle:VehicleBean) **Pre: vehicle != null** **Context** VehicleManager::doRetrieveLastInsurance(vehicle:VehicleBean) **Pre: vehicle != null** **Context** VehicleManager::doRetrieveOwners(vehicle:VehicleBean) **Pre: vehicle != null** **Context** VehicleManager::doRetrieveLastVehicleInspection(vehicle:VehicleBean) **Pre: vehicle != null** **Context** VehicleManager::doRetrieveVehicle (licensePlate: String) **Pre: licensePlate != null**

	Context VehicleManager::doRetrieveAll() **Pre:** *none*
Post-condizioni	**Context** VehicleManager::doRetrieveLastPossessionFee(vehicle:VehicleBean) **Post:** ∀x ∈ doFind(SELECT * FROM possessionFee Where vehicle = ?", vehicle.getLicensePlate()), x.getExpirationDate() < possessionFee.getExpirationDate **Context** VehicleManager::doRetrieveLastInsurance(vehicle:VehicleBean) **Post:** ∀x ∈ doFind(SELECT * FROM insurance Where vehicle = ?", vehicle.getLicensePlate()), x.getExpirationDate() < insurance.getExpirationDate **Context** VehicleManager::doRetrieveOwners(vehicle:VehicleBean) **Pre:** ∀x ∈ owners, x è proprietario del veicolo vehicle **Context** VehicleManager::doRetrieveLastVehicleInspection(vehicle:VehicleBean) **Post:** ∀x ∈ doFind(SELECT * FROM vehicleinspection Where vehicle = ?", vehicle.getLicensePlate()), x.getExpirationDate() <vehicleInspection.getExpirationDate() **Context** VehicleManager::doRetrieveVehicle (licensePlate: String) **Post:** vehicle.getLicensePlate() = licensePlate **Context** VehicleManager::doRetrieveAll() **Post:** ∀ x in vehicles, x ∈ doFind("SELECT * FROM vehicle");
Invarianti	

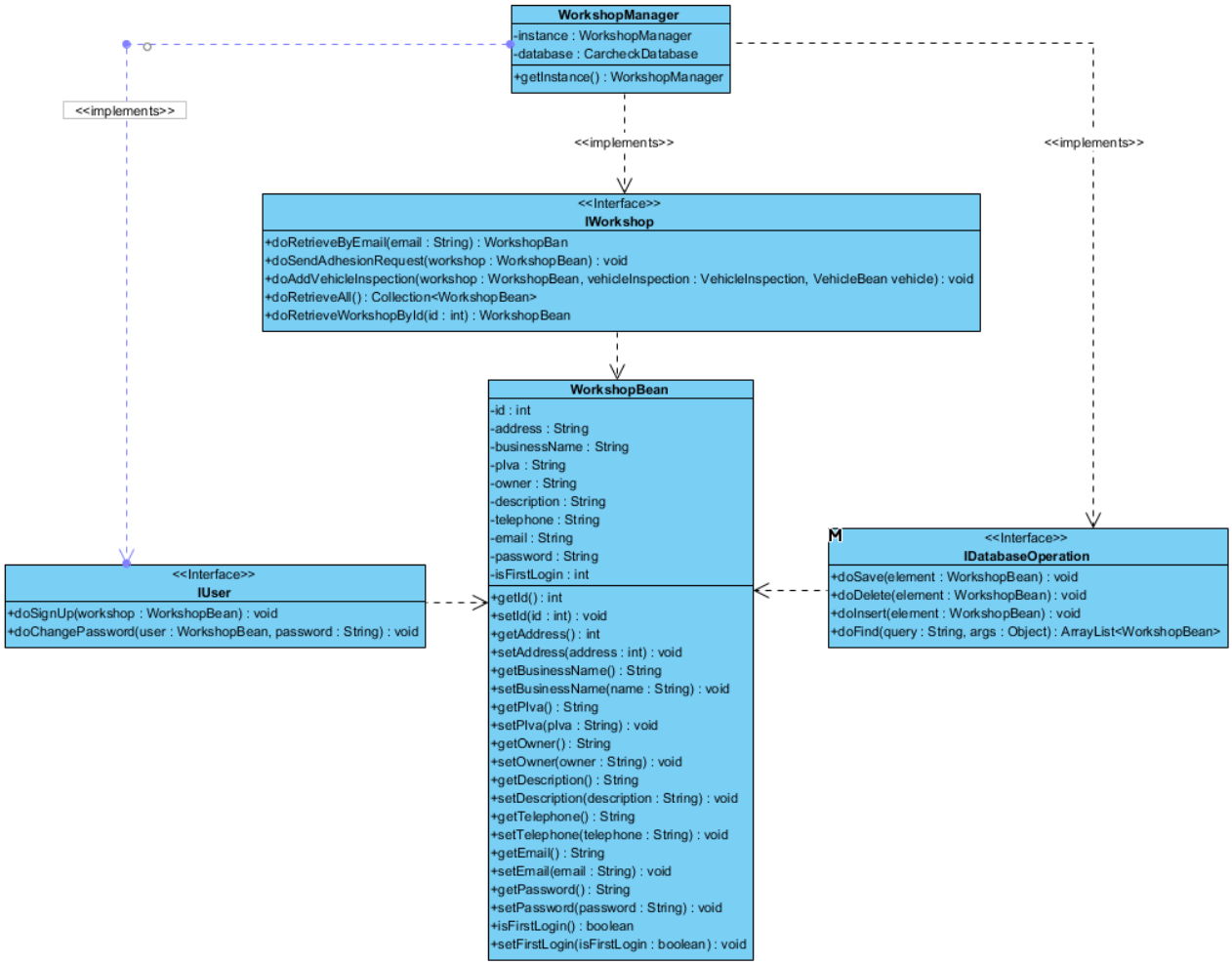
3.12 WorkshopAdhesionManager



Classe	WorkshopAdhesionManager
Descrizione	Manager che si occupa della gestione dei dati ottenuti dall'unione (JOIN) tra un'officina e una richiesta di adesione
Pre-condizioni	**Context** WorkshopAdhesionManager::doRetrieveByWorkshopId(id: int, admin: AdminBean) **Pre: admin != null && id > 0** **Context** WorkshopAdhesionManager::doRetrieveByAdhesionId(id: int, admin: AdminBean) **Pre: admin != null && id > 0** **Context** WorkshopAdhesionManager::doRetrieveAll(admin: AdminBean) **Pre: admin != null**

Post-condizioni	**Context** WorkshopAdhesionManager::doRetrieveByWorkshopId(id: int, admin: AdminBean) **Post:** workshopAdhesion.getAdminId() = admin.getId() && workshopAdhesion.getWorkshopId() = id **Context** WorkshopAdhesionManager::doRetrieveByAdhesionId(id: int, admin: AdminBean) **Post:** workshopAdhesion.getAdminId() = admin.getId() && workshopAdhesion.getWorkshopId() = id **Context** WorkshopAdhesionManager::doRetrieveAll(admin: AdminBean) **Post:** ∀ x in workshopAdhesions, x ∈ doFind(SELECT * FROM workshop, adhesionrequest WHERE workshop.id = adhesionrequest.workshop)
Invarianti	

3.13 WorkshopManager



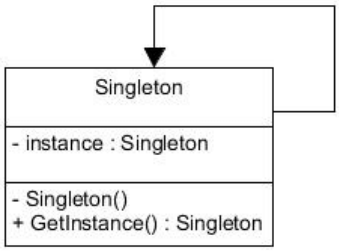
Classe	WorkshopManager
Descrizione	Manager che si occupa della gestione dei dati delle officine del sistema
Pre-condizioni	**Context** WorkshopManager::doLogin(email: String, password: String) **Pre: email != null && password != null** **Context** WorkshopManager::doChangePassword (user:AdminBean, password:String) **Pre: user != null && password != null** **Context** WorkshopManager::doRetrieveByEmail (email:String) **Pre: email != null** **Context** WorkshopManager::doSendAdhesionRequest(workshop: WorkshopBean) **Pre: workshop != null**

	Context WorkshopManager::doAddVehicleInspection(workshop: WorkshopBean, vehicleInspection: VehicleInspectionBean, vehicle: VehicleBean) **Pre: workshop != null && vehicleInspection != null && vehicle != null** **Context** WorkshopManager::doRetrieveWorkshopById(id:int) **Pre: id > 0** **Context** WorkshopManager::doRetrieveAll() **Pre:** *none*
Post-condizioni	**Context** WorkshopManager::doLogin(email: String, password: String) **Post:** workshop.getEmail() = email && workshop.getPassword() = password **Context** WorkshopManager::doChangePassword (user:AdminBean, password:String) **Post:** user.getPassword() = password **Context** WorkshopManager::doRetrieveByEmail (email:String) **Post:** workshop.getEmail() = email **Context** WorkshopManager::doSendAdhesionRequest(workshop: WorkshopBean) **Post:** esiste una richiesta di adesione presentata da quell'officina **Context** WorkshopManager::doAddVehicleInspection(workshop: WorkshopBean, vehicleInspection: VehicleInspectionBean, vehicle: VehicleBean) **Post:** viene aggiunta la revisione vehicleInspection al veicolo vehicle **Context** WorkshopManager::doRetrieveWorkshopById(id:int) **Post:** workshop.getId() = id **Context** WorkshopManager::doRetrieveAll() **Post:** ∀ x in workshops, x ∈ doFind(SELECT * FROM workshop)
Invarianti	

4. Design Patterns

In questa sezione vengono descritti tutti i design pattern che dovranno essere utilizzati per la realizzazione del sistema. In particolare, per realizzare l'architettura MVC verranno utilizzati il Front Controller Pattern, Strategy Pattern e il Factory Pattern. Troverà spazio anche il singleton pattern per tutte le classi che avranno bisogno di una sola istanza, come ad esempio i manager.

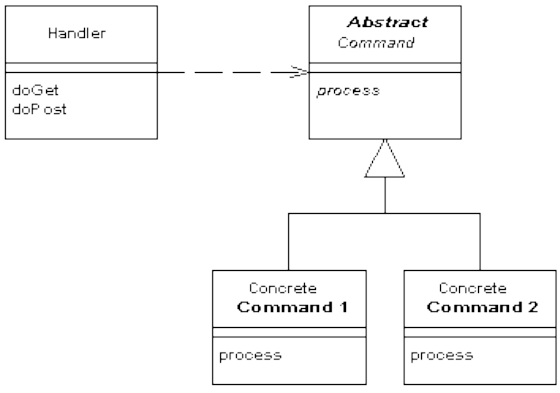
4.1 Singleton Pattern



Il singleton pattern verrà utilizzato nell'implementazione delle classi manager o, più in generale, in tutte quelle classi in cui si ha la necessità di creare una ed una sola istanza di esse.

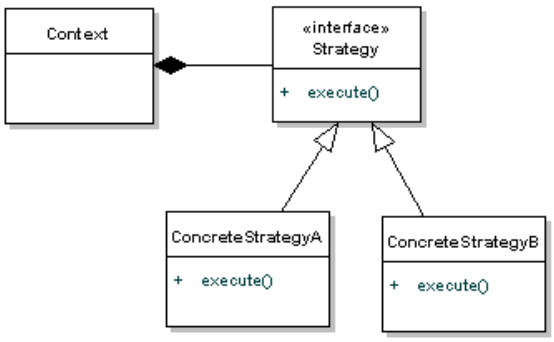
Ogni classe che utilizzerà questo design pattern avrà quindi un costruttore privato e per ottenere l'istanza si utilizzerà il metodo statico "getInstance()". Questo metodo restituirà (ed eventualmente inizializzerà se quella effettuata è la prima chiamata) l'istanza contenuta nella variabile statica instance.

4.2 Front Controller Pattern



Il Front Controller Pattern verrà utilizzato per implementare la logica di controllo del sistema. La logica di controllo è composta da una sola servlet che fornisce un punto di accesso centralizzato per tutte le richieste.

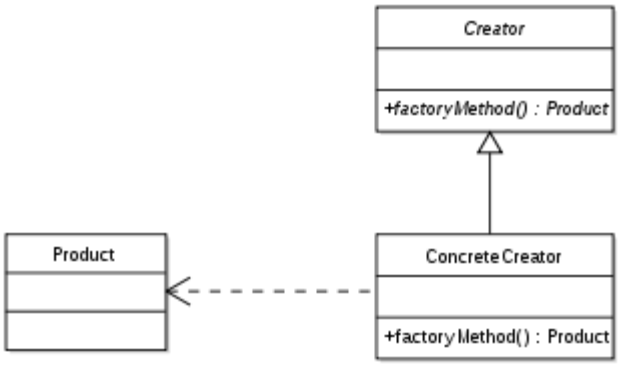
4.3 Strategy pattern



Visto che la logica di controllo non può sapere a priori quale “comando” eseguire, viene utilizzato lo strategy pattern. Il comando da eseguire viene quindi stabilito a runtime.

Nello specifico, questo pattern viene utilizzato nel sistema per definire l’interfaccia IAction che, definisce il metodo execute che prende come parametri la richiesta e la risposta http.

4.4 Factory pattern



Il Factory pattern è un pattern creazionale che fornisce un’implementazione concreta di un tipo astratto. In particolare, nel MVC si occupa di restituire l’implementazione concreta dell’interfaccia Action in base al metodo e al pathinfo della richiesta.

5. Glossario

Query: In informatica il termine **query** viene utilizzato per indicare l'interrogazione da parte di un utente di un database, strutturato tipicamente secondo il modello relazionale, per compiere determinate operazioni sui dati

Cookie: Piccoli file di testo inviati da un web server al browser dell'utente e che vengono memorizzati sul dispositivo di quest'ultimo con lo scopo di identificarlo.

Browser: Programma che consente di navigare ed interagire con le pagine web, i testi, le immagini ed altri elementi multimediali che formano internet o una rete locale.

Pathinfo: rappresenta la parte nell'URL della richiesta che si trova dopo il contesto e il percorso della servlet.