Object-Oriented Software Design Course
a.a. 2018-2019

UNIVERSITÀ
DEGLI STUDI
DE L'AQUILA

traccia #1

PEDAGGIO AUTOSTRADALE

Studente	Matricola	E-mail
De Petris Vincenzo	249221	vincenzo.depetris@student.univaq.it

a. Requisiti

Introduzione al sistema

Lo scopo del sistema è quello di fornire una soluzione software che consente, ai fruitori dei servizi autostradali, di effettuare il pagamento del pedaggio presso i caselli d'uscita.

Arrivato al casello d'uscita l'utente inserisce il biglietto, ottenuto al casello d'entrata, all'interno dell'apposita macchina. Il sistema, preso atto della tratta percorsa dall'utente e del veicolo da lui guidato, calcola il prezzo del pedaggio e consente all'utente di procedere con il pagamento.

In oltre il sistema deve fornire un'interfaccia a gli amministratori che consente di effettuare le tipiche operazioni CRUD sui dati, quali autostrade e caselli.

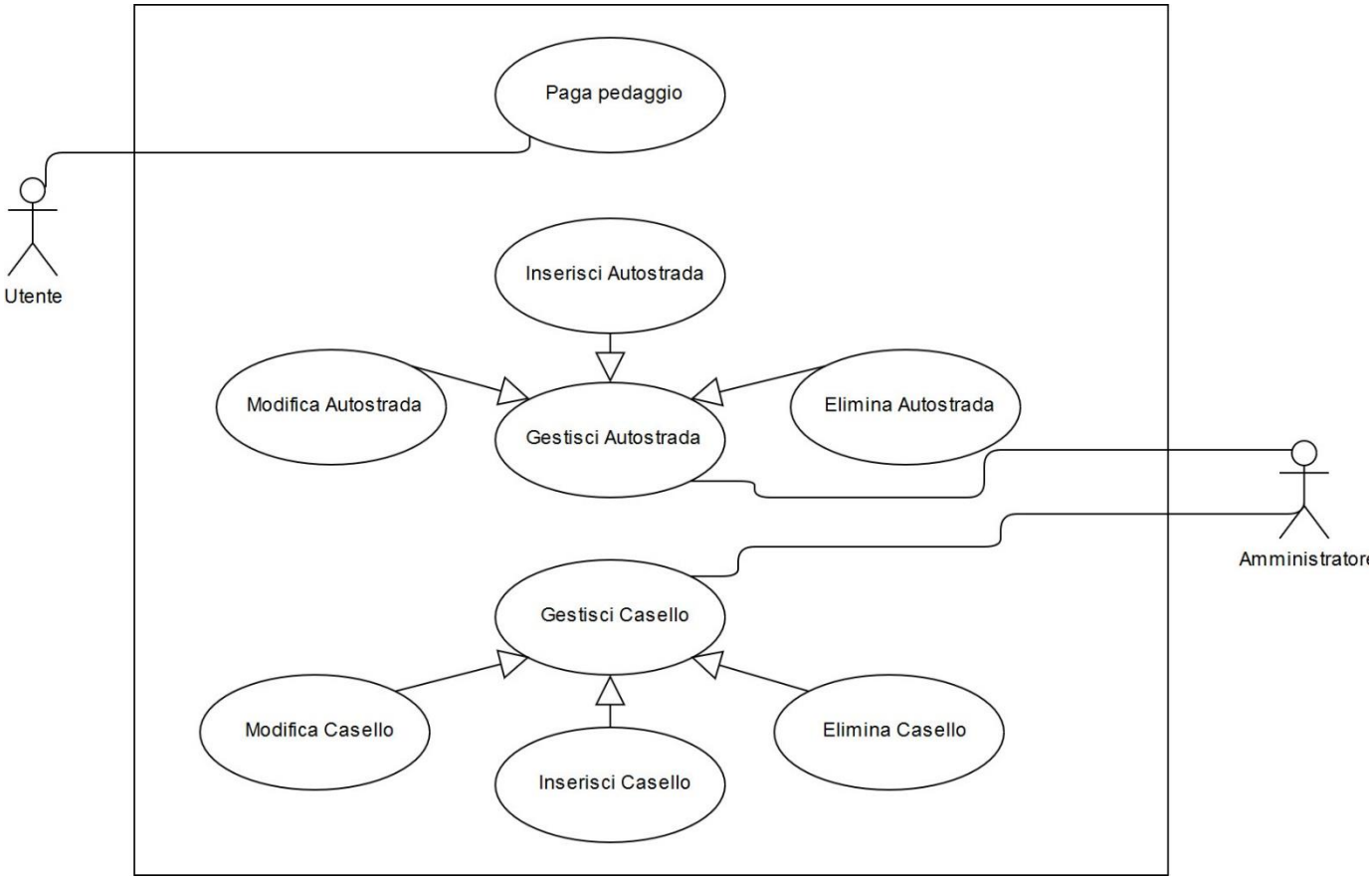
Requisiti funzionali

Utente

- Pagamento pedaggio

Amministratore

- Gestione autostrade
- Gestione caselli



Requisiti non funzionali

- Il sistema dev'essere mantenibile
- Il sistema deve essere implementato utilizzando tecnologie Java
- Il progetto dev'essere sviluppato all'interno dell'ambiente Eclipse
- La configurazione del progetto dev'essere gestita utilizzando Git e Github

b. Architettura

Obiettivi di design

Il principale obiettivo di design ricavato dai requisiti non funzionali è la modificabilità. Le funzionalità del sistema devono essere facilmente modificabili al verificarsi di cambiamenti all'interno delle regole di business, come ad esempio l'introduzione di una nuova normativa che può definire nuove classi veicolari, o comunque riconsiderare la logica dietro il calcolo dei pedaggi.

Altri obiettivi di design secondari sono l'estensibilità, ovvero la facilità di aggiungere nuove classi e/o funzionalità al sistema, e la leggibilità, cioè la facilità di capire il sistema leggendo il codice sorgente.

Decomposizione del sistema

Il sistema è stato inizialmente decomposto in 3 layer, addottando quindi un'architettura 3-tier chiusa:

Layer	Scopo
Presentation layer	Presentare dati all'utente e raccogliere gli input
Business layer	Implementazione logica di business
Data layer	Gestione e persistenza dati

Il Presentation Layer è stato partizionato ulteriormente in 2 componenti. Una componente si occupa della logica di presentazione per gli amministratori, mentre l'altra implementa la logica di presentazione per gli utenti.

Anche il Business Layer è stato diviso in due componenti. La componente Model che racchiude gli oggetti di dominio, e la componente Manager, che implementa la logica di business e fornisce servizi al layer sovrastante utilizzando a sua volta i sevizi offerti dal Data Layer.

La mansione del Data Layer è quella di gestire i dati persistenti del sistema.

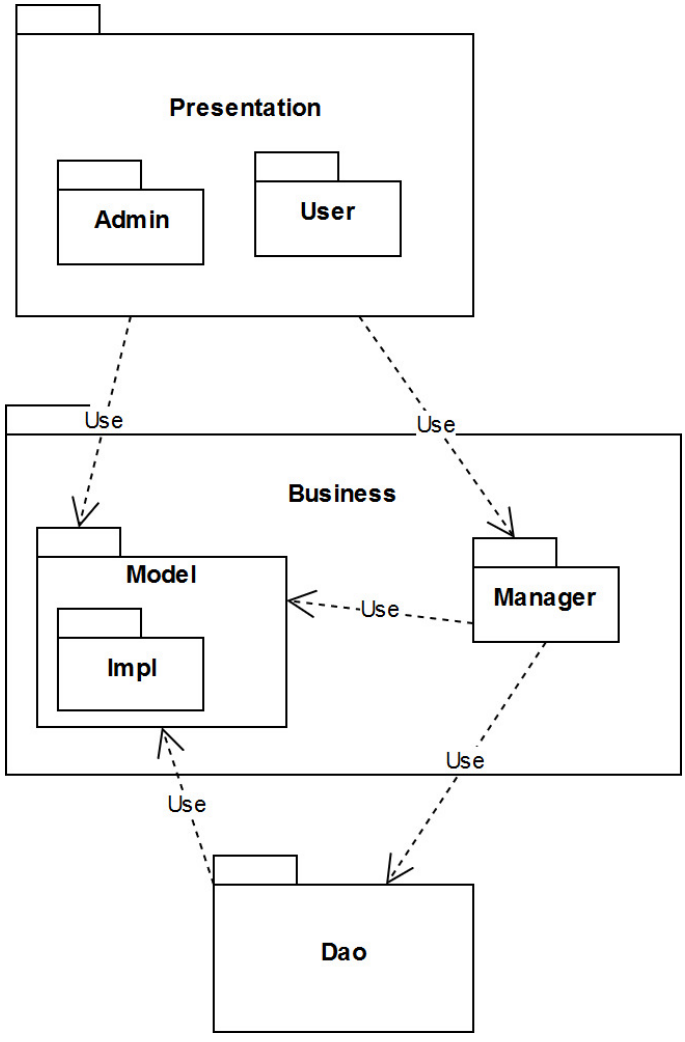
Strategie adottate per la costruzione del sistema

Per quanto riguarda le componenti Presentation e Model è stata utilizzata la tecnologia JavaFX. Questo ha consentito una buona "Separation of Concern" adottando un approccio MVC, permettendo di specificare l'interfaccia grafica ed i suoi controlli in maniera statica (per mezzo di un linguaggio di markup) e di racchiudere in altri file la logica necessaria per gestire gli input degli utenti. La componente di Model segue la convenzione dei Beans specificata da JavaFX, facendo utilizzo delle Properties. Questo ha consentito di non preoccuparsi della sincronizzazione tra i dati in memoria e quelli presenti sull'interfaccia grafica, sfruttando l'implementazione del Observer Pattern fornita da tali Properties.

Non è stata adottata nessuna policy particolare per regolare l'accesso ai dati, sia perché il sistema è pensato per essere deployato in apposite macchine presso i caselli autostradali (e questo di per sé limita l'accesso al sistema e di conseguenza ai dati) e sia perché in tal modo è stato possibile alleggerire il carico di lavoro e diminuire i tempi di sviluppo.

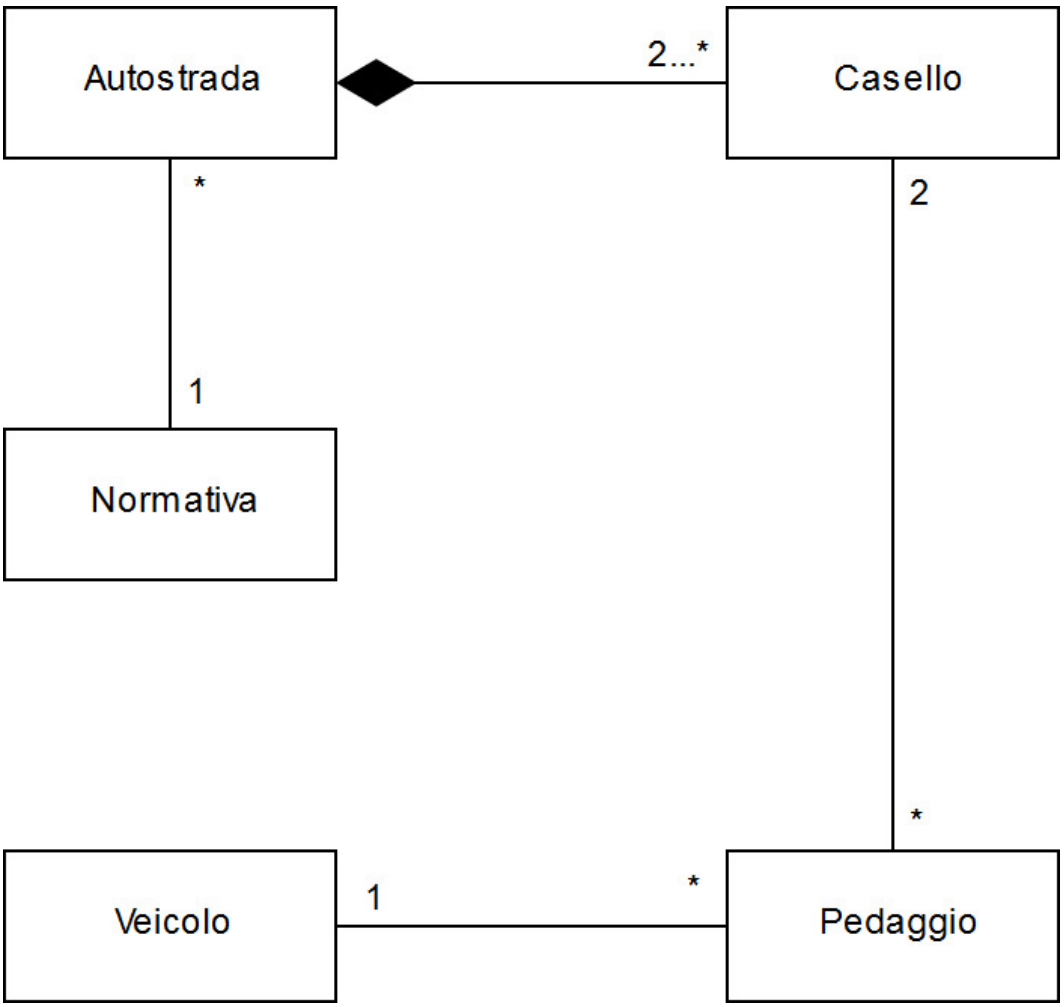
Per gestire la persistenza dei dati è stato scelto di utilizzare i servizi offerti da un DBMS, per l'esattezza MySQL. Anche questo ha consentito di abbassare i tempi di sviluppo, grazie all'utilizzo del suddetto DBMS in progetti passati.

Package diagram



c. Classi e Interfacce

Domain model



Le entità ricavate dall'attività di elicitazione dei requisiti e della loro analisi sono Autostrada, Casello, Normativa, Pedaggio e Veicolo.

Un'autostrada è identificata univocamente dal nome ed è associata ad un insieme di Caselli e ad una Normativa che vige correntemente sull'Autostrada.

Un casello è identificato univocamente da un numero identificativo ed è definito da attributi quali il nome e il numero di chilometri di distanza dall'inizio dell'unica Autostrada a cui è associato.

Un pedaggio è associato ad un Casello di entrata, ad uno di uscita e ad un Veicolo. In oltre contiene un attributo che rappresenta il prezzo da pagare in base al percorso ed al veicolo.

Un veicolo è identificato univocamente dal numero di targa e contiene attributi quali modello, marca, numero di assi, altezza e classe ambientale.

Una normativa è associata ad un'Autostrada e definisce quali sono le classi veicolari, i criteri di appartenenza dei Veicolo alle suddette classi e una tariffa unitaria per ogni classe veicolare. E' l'entità più soggetta a cambiamenti all'interno del dominio.

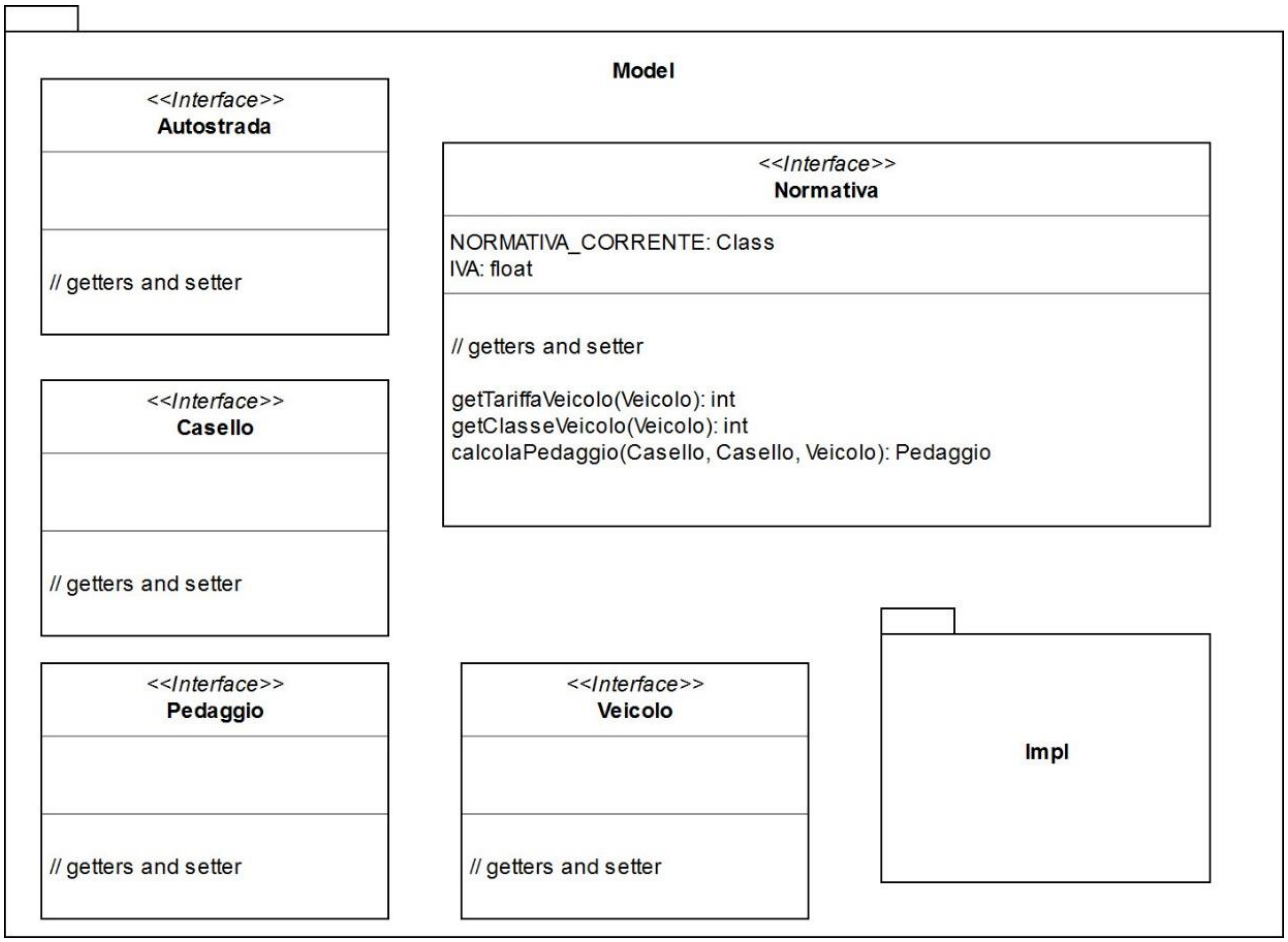
Le autostrade (e di conseguenza i caselli) e le normative che ne definiscono le tariffe unitarie, fanno parte dei dati persistenti. (I veicolo sono stati inseriti nel database con l'unico scopo di simulare il riconoscimento delle targhe e delle caratteristiche del veicolo).

NOTA: la normativa vigente dovrebbe essere la stessa su ogni autostrada in un tempo dato (cioè definisce le stesse classi veicolari e la stessa logica di calcolo dei pedaggi). Quello che cambia sono le tariffe unitarie, che possono dipendere da fattori esterni, quali il gestore dell'autostrada e/o il fatto di essere situata in zone montuose ed avere maggior necessità di manutenzione.

Descrizione dettagli di Design

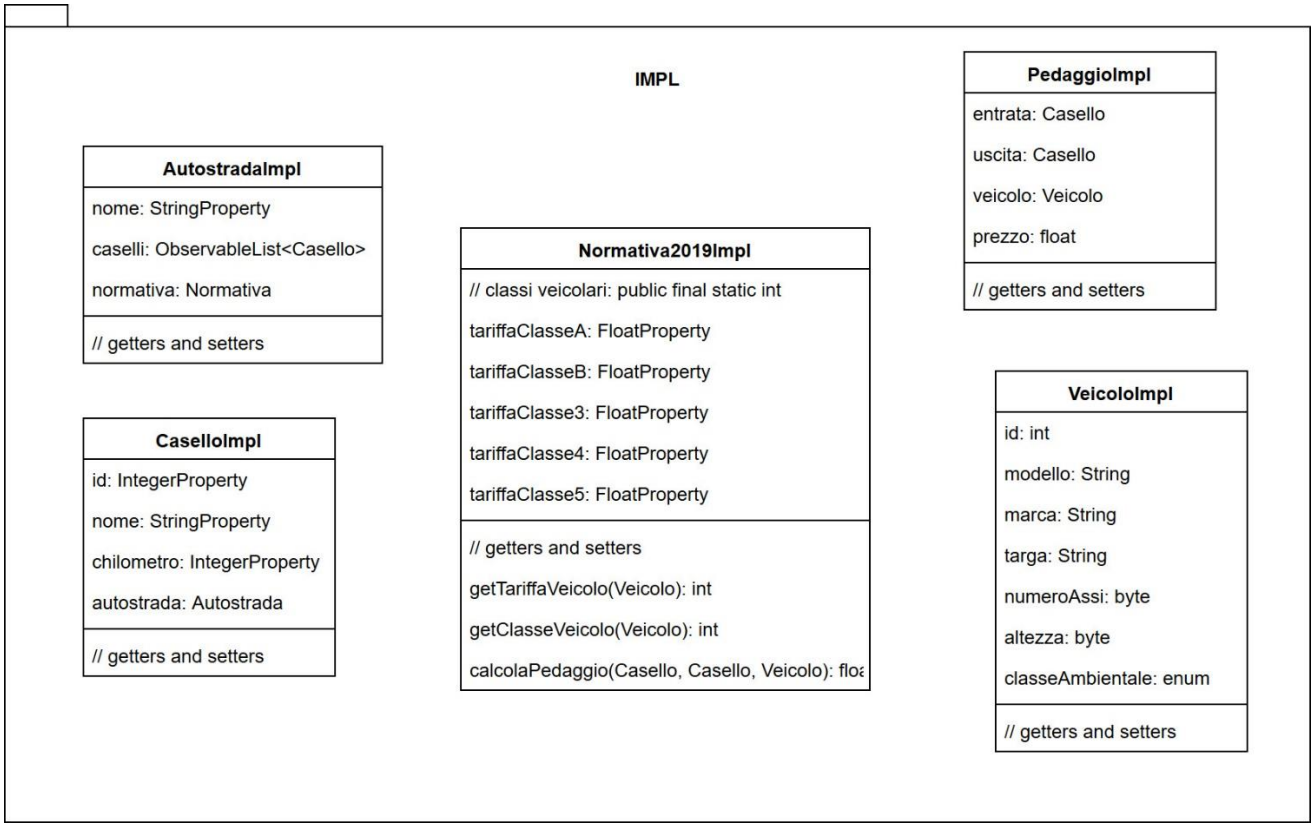
Business Layer

Model package



NOTA: i commenti "// getters and setters" all'interno del package Model si riferiscono a gli attributi delle implementazioni concrete (vedi Model.Impl package). Le segnature dei metodi sono state omesse dal diagramma per questione di spazio.

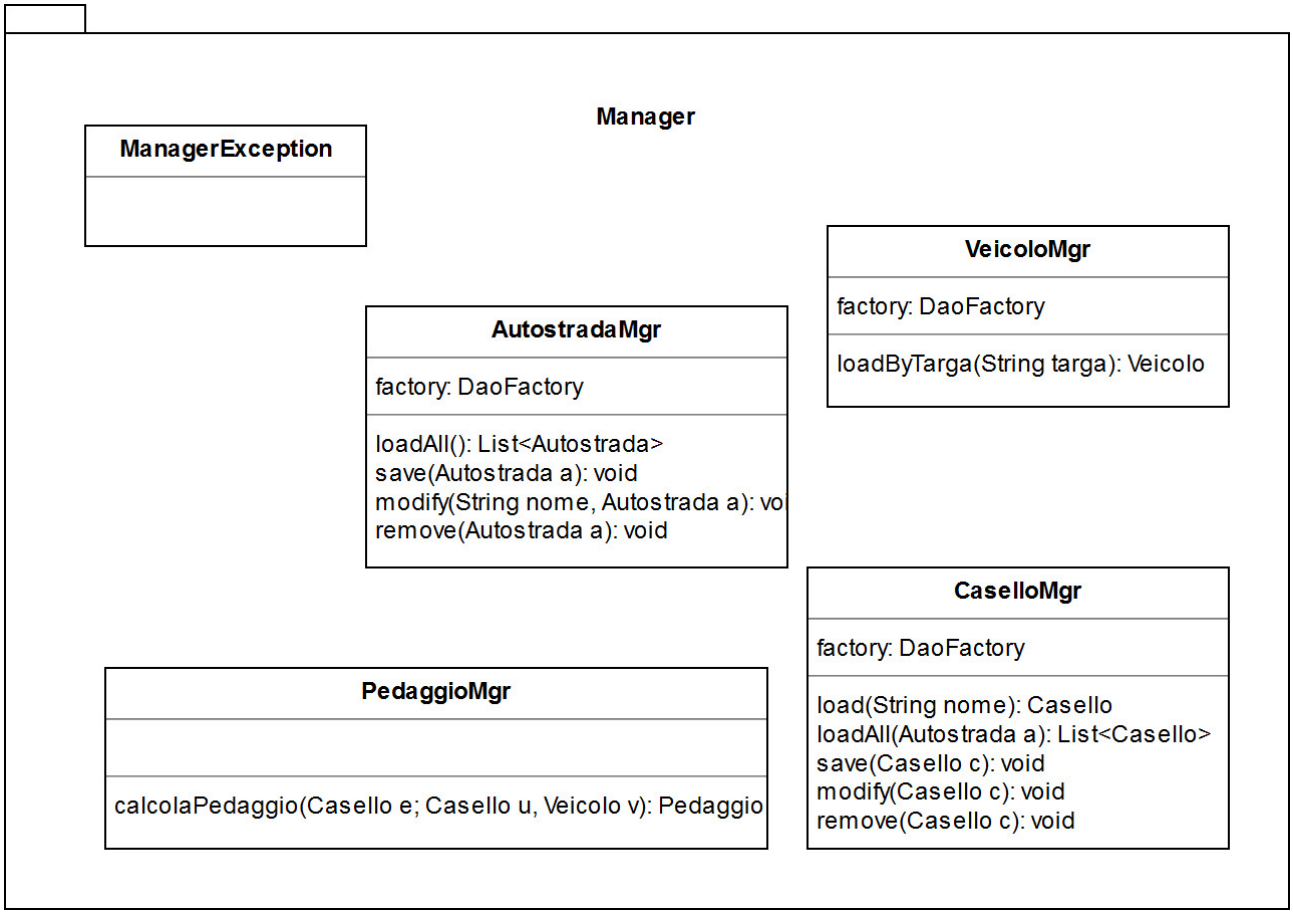
Model.Impl package



Il package "Model.Impl" contiene le implementazioni degli oggetti di dominio, di cui ogni classe implementa la rispettiva interfaccia presente all'interno del package "Model".

Le implementazioni dell'interfaccia Normativa devono definire le classi veicolari, la logica di assegnazione dei veicoli alle classi e la logica del calcolo dei pedaggi. In caso subentri una nuova normativa che ridefinisce la logica nel calcolo dei pedaggi (senza andare a modificare le classi veicolari) è sufficiente estendere l'implementazione precedente sovrascrivendo il metodo in questione. In caso a cambiare siano anche le classi veicolari è necessario fornire una nuova implemetazione che ridefinisce le classi e la logica. In oltre l'interfaccia Normativa contiene un campo che determina la normativa vigente al momento (ovvero l'implementazione da instanziare) diminuendo così l'accoppiamento tra le componenti che utilizzano l'interfaccia e la sua attuale implementazione.

Manager package

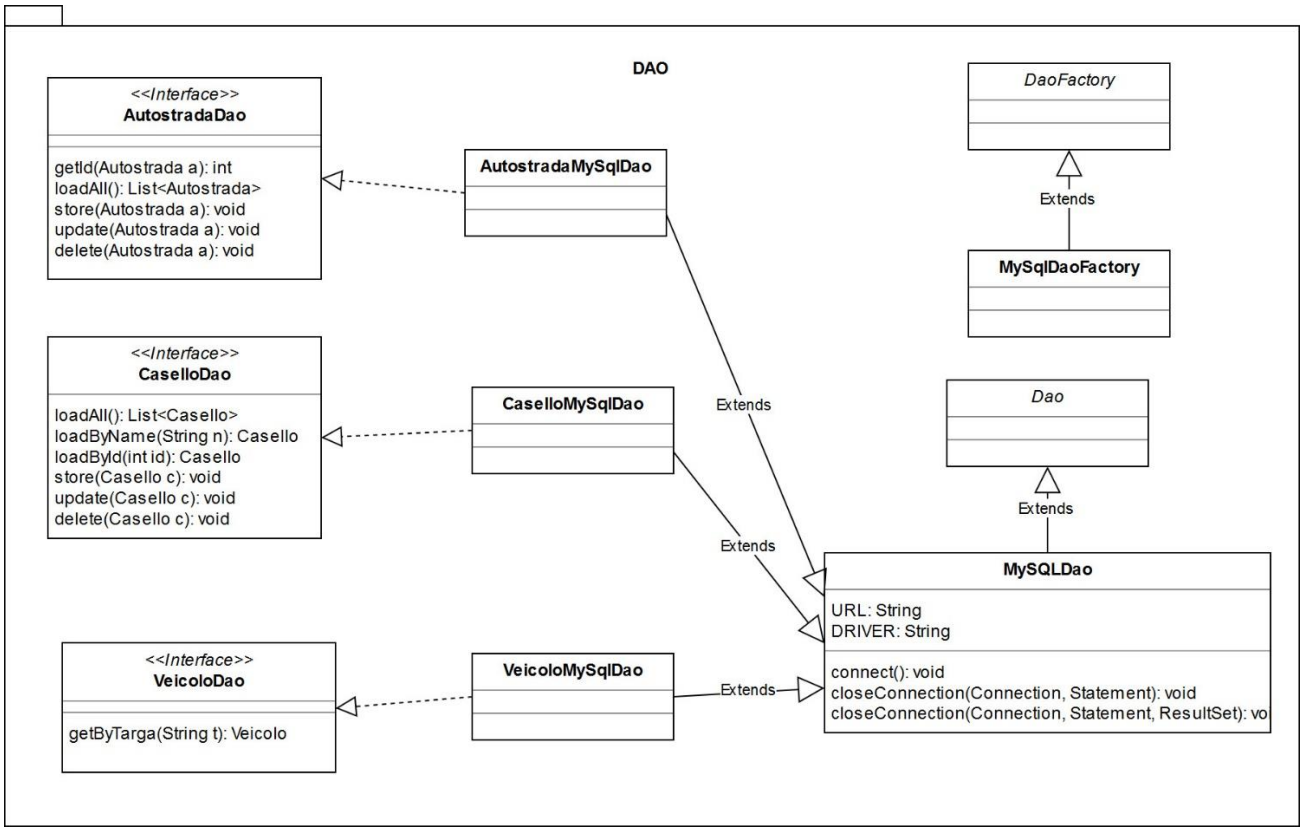


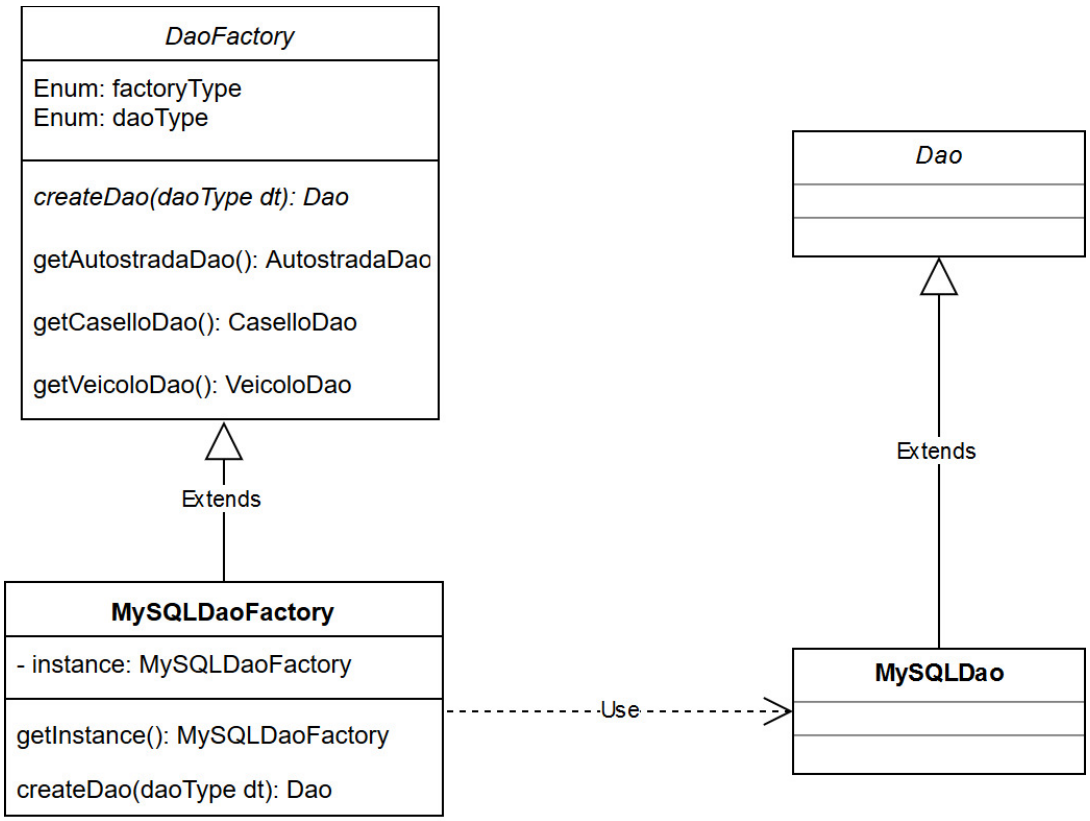
Il package Business.Manager fornisce al Presentation Layer un'interfaccia orientata alle funzionalità. Sostanzialmente è il controller dell'applicazione. Le classi controller di JavaFX nel Presentation Layer effettuano chiamate alle componenti del Manager che andranno ad eseguire le funzionalità richieste dall'utente ed eventualmente a modificare le componenti del Model.

E' facile notare la relazione tra l'API definita dalle classi AutostradaMgr, PedaggioMgr, Casello Mgr ed i requisiti funzionali del sistema.

Le componenti del Manager utilizzano i servizi offerti dal layer inferiore, ovvero il DataLayer, e dipendono unicamente dalle interfacce definite in esso. In oltre definiscono un punto ottimale per gestire le eccezioni nel caso in cui non sia possibile fornire la funzionalità richiesta dall'utente.

Data Layer


DAO
<<Interface>>
AutostradaDao
getId(Autostrada a): int
loadAll(): List<Autostrada>
store(Autostrada a): void
update(Autostrada a): void
delete(Autostrada a): void
AutostradaMySqlDao
DaoFactory
Extends
MySqlDaoFactory
<<Interface>>
CaselloDao
loadAll(): List<Casello>
loadByName(String n): Casello
loadById(int id): Casello
store(Casello c): void
update(Casello c): void
delete(Casello c): void
CaselloMySqlDao
Extends
Dao
Extends
Extends
MySQLDao
URL: String
DRIVER: String
connect(): void
closeConnection(Connection, Statement): void
closeConnection(Connection, Statement, ResultSet): vo
<<Interface>>
VeicoloDao
getByTarga(String t): Veicolo
VeicoloMySqlDao
Extends


DaoFactory
Enum: factoryType
Enum: daoType
createDao(daoType dt): Dao
getAutostradaDao(): AutostradaDao
getCaselloDao(): CaselloDao
getVeicoloDao(): VeicoloDao
Extends
MySQLDaoFactory
- instance: MySQLDaoFactory
getInstance(): MySQLDaoFactory
createDao(daoType dt): Dao
Use
Dao
Extends
MySQLDao

Per implementare ed isolare il Data Layer è stato scelto il pattern architetturale Data Access Object. I vari oggetti DAO (che in questo caso rappresentano un'entità tabellare e forniscono i metodi per effettuare query su di essa) sono generati attraverso una Factory, diminuendo così l'accoppiamento tra le componenti Dao e Manager. Infatti il Data Layer offre, alle componenti esterne che lo utilizzano, solo le interfacce dei Dao (e non le implementazioni concrete) e la classe Factory astratta.

Design Patterns e Design Principles

La fase di Object Design del sistema è stata guidata dai seguenti principi di design:

1. **Identificare gli aspetti dell'applicazione soggetti a cambiamenti e separarli da quelli che restano.**
   Classi veicolari e logica dietro il calcolo dei pedaggi sono stati incapsulati nella classe Normativa.
   Essi infatti, da specifica, sono gli aspetti soggetti a cambiamenti in caso di riforme (vedi riforma 2021 e riforma 2026). Inoltre è stato scelto di associare Normative e Autostrade a causa della stretta relazione tra classi veicolari e tariffe unitarie.

2. **Programmare su interfacce e non implementazioni concrete.**
   Si è cercato di utilizzare il meno possibile classi concrete, a favore delle sole interfacce. Esempi dell'applicazione di questo principio di design sono all'interno del package Model e del package DAO, il quale nasconde al proprio interno, per mezzo dei modificatori, tutte le classi concrete che contiene.

3. **Classi aperte a estensioni e chiuse a modifiche.**
   Tutta la progettazione è stata guidata da questo principio di design, al fine di consentire l'accomodamento dei cambi tramite aggiunta di codice e non tramite la modifica di esso.

4. **Manenere un basso grado di accoppiamento tra oggetti che interagiscono tra di loro.**

Inoltre i principali Design Patterns utilizzati sono:

- **Singleton Pattern**
  Utilizzato per implementare la DaoFactory. Infatti è sufficiente un'unica istanza della factory per generare le varie classi DAO.

- **Factory Method Pattern**
  E' stato scelto il factory method pattern per generare gli oggetti DAO. Per accomodare un cambiamento nella strategia di persistenza dei dati (come l'utilizzo di un altro DBMS) è sufficiente introdurre nuove classi DAO concrete che ereditano da DAO ed implementano la corretta interfaccia, ed una nuova factory concreta, implementando correttamente il metodo creazionale.

- **Observer Pattern**
  Al fine di gestire il Data Binding è stato utilizzando l'observer pattern, sfruttando l'implementazione fornita da JavaFX.