Object-Oriented Software Design

a.a.2018-2019

PEDAGGIO AUTOSTRADALE

Gruppo composto da

|  |  |  |
| --- | --- | --- |
| NOME | COGNOME | MATRICOLA |
| Cimoroni | Marco |  |
| Di Egidio | Laura | 248831 |
| Gentile | Fabio | 248809 |

1. REQUISITI

Lo scopo del Sistema è quello di fornire un software capace di calcolare il pedaggio autostradale a seconda del percorso svolto. Il sistema inoltre deve fornire anche meccanismi di calcolo differenti a seconda del tipo di calcolo messo a disposizione.

L’utente in base alla normativa stabilita dall’amministratore ha la possibilità di calcolare il percorso effettuato o che vorrà effettuare semplicemente inserendo la sua targa ed il percorso desiderato.

Il sistema deve inoltre fornire un’interfaccia grafica per l’utente e per l’amministratore, il quale dovrà gestire operazioni CRUD riguardanti il casello.

* Requisiti funzionali

Utente: Conoscenza della tariffa dal casello di entrata al casello d’uscita

Amministratore: gestione dei caselli (CRUD)

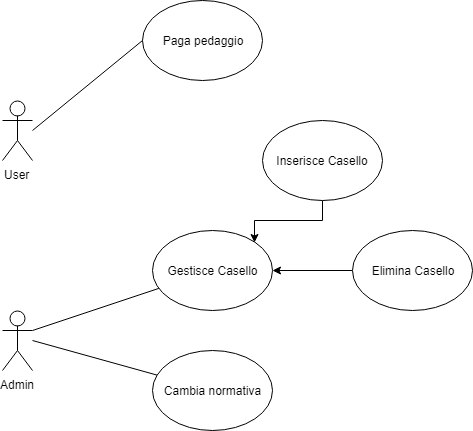
* Requisiti non funzionali

Il sistema deve essere scalabile

Il sistema deve essere svolto nell’ ambiente Java

Il sistema deve essere affidabile

USE CASE DIAGRAM



1. ARCHITETTURA

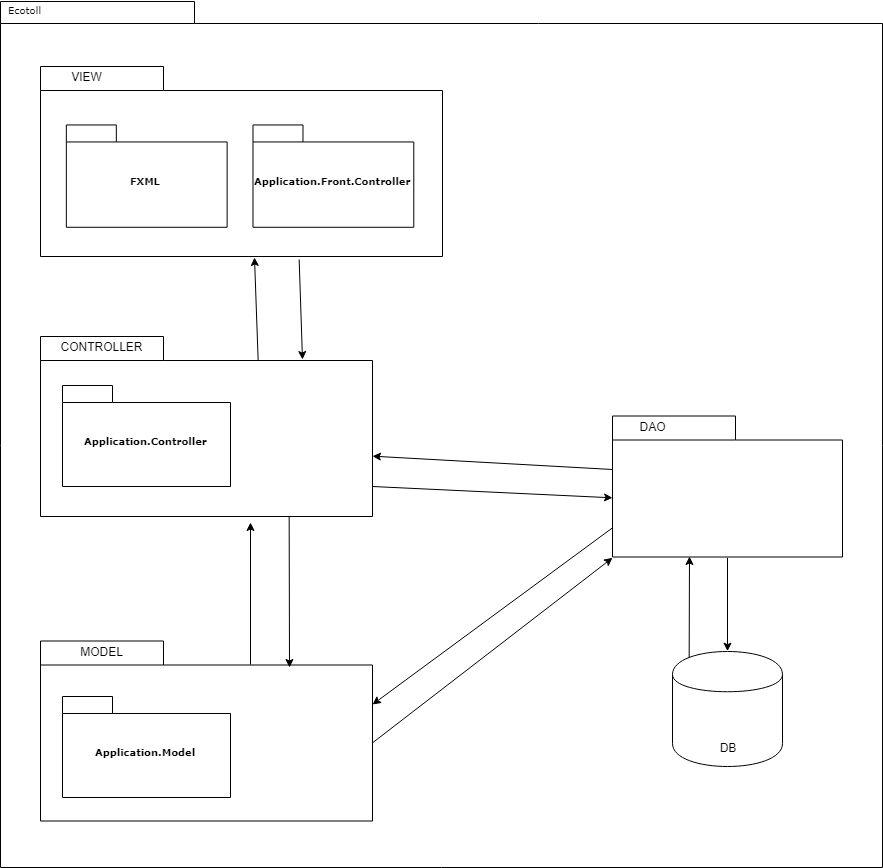
Obiettivi di design

Il principale obiettivo di design del sistema riguardante i requisiti non funzionali è la scalabilità. Il sistema deve essere infatti in grado di poter gestire eventuali normative che porteranno a modifiche all’interno del sistema. Per ora ci è sembrato opportuno mettere a discrezione dell’amministratore la scelta di quale normativa applicare per far calcolare il percorso all’utente.

Strategie adottate per la costruzione del sistema

All’interno del package application.front.fxml troviamo tutte quelle che sono le viste messe a disposizione per l’utente. In questo componente è stata utilizzata la tecnologia JavaFX.  
Abbiamo ritenuto utile adottare il pattern MVC per una buona “Separation of Concern” gestendo quelli che sono gli input dell’utente e la parte statica del sistema.  
La componente Model in application.model fa uso delle Properties consentendo quindi di avere una sincronizzazione tra dati in memoria con quelli presenti sull’interfaccia grafica.  
Per la persistenza dei dati abbiamo scelto l’utilizzo di un database MYSQL.  
E’ stato scelto di costruire anche un’interfaccia DAO che permette l’accesso al db e l’implementazione per ogni componente del Model utilizzato.

PACKAGE DIAGRAM



Design Patterns

Ciò che ha guidato il nostro progetto è stato sostanzialmentedividere quello che è la parte statica del progetto da ciò che invece nel tempo potrebbe mutare in quanto soggetto a cambiamenti.

I principali Design Patterns utilizzati sono:

**DAO FACTORY**: Il pattern **Data Access Object** (DAO) è un **pattern architetturale** utilizzato per separare i servizi di business dell’application processing layer dalle operazioni di accesso ai dati del data management layer. Il Data Access Object nasconde completamente i dettagli dell’interazione con la sorgente dati. L’interfaccia esposta dal DAO al client non cambia quando l’implementazione dell’origine dati sottostante cambia, e questo consente al DAO di adattarsi a diversi schemi di archiviazione senza dover modificare nulla sugli altri layer.   
 Il DAO implementa le operazioni CRUD (Create, Read, Update, Delete).

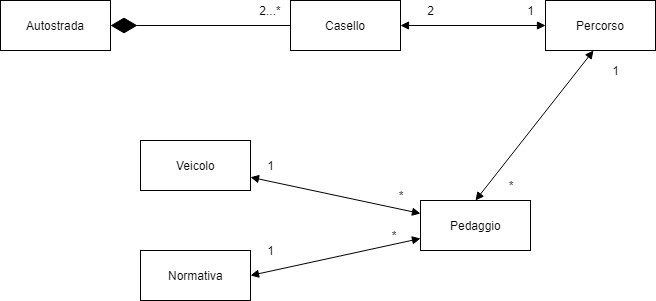
**MVC**: **Model-View-Controller** (MVC) è un pattern utilizzato in programmazione per dividere il codice in blocchi dalle funzionalità ben distinte.  
L'applicazione deve separare i componenti software che implementano il modello delle funzionalità di business, dai componenti che implementano la logica di presentazione e di controllo che utilizzano tali funzionalità. Vengono quindi definiti tre tipologie di componenti che soddisfano tali requisiti:   
- il Model, View, Controller.  
In particolare:

* Model: contiene i metodi di accesso ai dati.
* View: si occupa di visualizzare i dati all’utente e gestisce l’interazione fra quest’ultimo e l’infrastruttura sottostante.
* Controller: riceve i comandi dell’utente attraverso il View e reagisce eseguendo delle operazioni che possono interessare il Model e che portano generalmente ad un cambiamento di stato del View.

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

1. CLASSI E INTERFACCE

Domain model



Le entità che sono state utili durante lo sviluppo del sistema ordinandole per importanza sono Autostrada, Casello, Normativa, Veicolo, TipoVeicolo, ClasseVeicolo, Utente e Ruolo.

L’entità Autostrada composta dal suo nome, codice Autostrada (PK) , ‘da’ e ‘a’ che rispettivamente indicano l’inizio e la fine dell’autostrada, la lunghezza e la tariffa per km.

Un’ autostrada è formata da caselli i quali hanno un proprio nome, un id (PK) , un’altezza e ovviamente l’id dell’autostrada.

La Normativa è utile nel momento in cui dobbiamo andare a calcolare il percorso. L’Entità normativa indica appunto la normativa vigente in quel momento impostata dall’amministratore. Nel momento in cui l’amministratore aggiornerà la normativa nel DB verrà fatto un update del campo NormativaVigente.

L’entità veicolo invece è stata scelta in quanto l’utente ha la possibilità di inserire la propria targa del veicolo (Considerando ovviamente una situazione in cui il sistema ha registrato tutte le targhe in circolazione) e da li risalire dunque al tipo e alla classe.

Il Tipo e Classe Veicolo sono due entità che abbiamo scelto di inserire per maggiore chiarezza e perché si possa risalire al moltiplicatore di ogni tipo/classe partendo dalla targa del veicolo.

L’entità Utente è stata utile in quanto il nostro sistema gestisce un login con controllo credenziali ed una eventuale registrazione.

Dall’utente ovviamente il suo Ruolo scelto come entità sempre per un discorso di chiarezza dei dati.

Descrizione dettagli di design

MODEL PACKAGE

IMMAGINE DI TUTTE LE INTERFACCE MODEL!!!