



Studente: Gabriele Romanato

Matricola: 258820

## Progetto finale – Traccia 1

### **CORSO DI OBJECT ORIENTED SOFTWARE DESIGN CORSO DI LAUREA IN INFORMATICA UNIVERSITA' DEGLI STUDI DELL'AQUILA A.A. 2019/2020**

**GitHub:** <https://github.com/gabrielromanato/univaq-pedaggio-autostradale>

**Nota:** Gli screenshot del codice presenti nel documento potrebbero non essere aggiornati con le recenti modifiche. Si faccia riferimento alla repository GitHub.

## **Package**

Descrizione dai file `package-info.java`.



Nome	Descrizione
app	Main application package
classes	Common classes
controller	Controller classes
helpers	Helper and utility classes
interfaces	Application interfaces
models	Model classes
views	View classes

## Creazione della base di dati

Poiché l'amministratore del sistema dovrà poter aggiornare l'elenco delle autostrade, si è ritenuto opportuno partire dall'elenco di autostrade italiane ufficialmente attive.

Collegandosi all'indirizzo <http://www.autostrade.it/autostrade-gis/percorso.do> si è innanzitutto provveduto ad estrarre dalla pagina web l'elenco delle autostrade con il relativo codice identificativo sfruttando la console JavaScript del browser.

```
let csv = 'codice,nome\n';
document.querySelector('select[name=cod]').querySelectorAll('option').forEach(opt => {
  csv += opt.getAttribute('value') + ',' + opt.innerText + '\n';
});
console.log(csv);
```



Quindi, poiché il progetto prevede l'uso di JDBC, si è provveduto a creare un database MySQL locale in cui si è importato il file CSV creato copiando l'output della console JavaScript del browser.

La tabella **autostrade** contiene tre colonne:

Nome	Tipo di dati	Lunghezza / Length
id	INT	11
nome	VARCHAR	255
codice	VARCHAR	3

Collegandoci all'indirizzo <https://www.autostrade.it/autostrade-gis/rete.do?op=caselli> possiamo estrarre l'elenco dei caselli con i relativi dati utilizzando su ciascuna tabella delle pagine di dettaglio il seguente codice JavaScript:

Quindi possiamo importare il file CSV creato nella tabella **caselli** che avrà la seguente struttura:



Nome	Tipo di dati	Lunghezza / Length
id	INT	11
autostrada	VARCHAR	3
progressiva_km	DECIMAL	10,2
nome	VARCHAR	255
codice	VARCHAR	255

Il campo `autostrada` corrisponde al codice di un'autostrada precedentemente salvato nella tabella `autostrade`. In questo modo effettuiamo il collegamento tra i due dati.

Il campo `progressiva_km` indica il numero di chilometri associati a ciascun casello. Con questo dato è possibile calcolare il pedaggio per ciascun casello.



```
function getData() {
  let parts = location.href.split('&');
  let id = parts[1].split('=')[1];
  let csv = '';
  const rows = document.querySelectorAll('table.elenco tr');
  rows.forEach(row => {
    const cells = row.querySelectorAll('td');
    let km = cells[0];
    let name = cells[1];
    let code = cells[2];

    if(km && name && code) {

      if(/^\\d+$/.test(code.innerHTML.trim())) {
        csv += id + ',' + km.innerHTML + ',' + name.innerHTML + ',' + code.innerHTML + '\\n';
      }

    }

  });
  console.log(csv);
};

getData();
```

Dato che il progetto prevede anche il pagamento del pedaggio, ne consegue che il database dovrà contenere una tabella per la registrazione dei pagamenti. Questa tabella, **pagamenti**, potrà avere la seguente struttura.

Nome	Tipo di dati	Lunghezza / Length
id	INT	11
id_veicolo	VARCHAR	255
importo	DECIMAL	10,2
orario	DATETIME	
tipologia	VARCHAR	255



**id\_veicolo** potrebbe essere rappresentato dalla targa del veicolo. In questo modo in un real case scenario il database centrale potrebbe dialogare con il database della motorizzazione per ulteriori implementazioni tramite API.

**id** nel nostro database è di tipo `AUTO_INCREMENT`<sup>1</sup> e quindi diventa facoltativo nel design delle classi che seguono.

L'ultimo dato da aggiungere è l'elenco delle classi tariffarie nella tabella che chiameremo **classi\_tariffarie**.

Nome	Tipo di dati	Lunghezza / Length
id	INT	11
nome	VARCHAR	255
esempi	TEXT	

## Models

Come premessa è utile ricordare due dei principi espressi in *Design Patterns. Elements of Reusable Object Oriented Software* (1995):

1. *Program to an interface, not an implementation.*
2. *Favor object composition over class inheritance.*

---

<sup>1</sup> <https://dev.mysql.com/doc/refman/8.0/en/example-auto-increment.html>



Il primo principio è immediatamente applicabile alla gestione dei dati. Se in futuro infatti volessimo sostituire una soluzione con database con una soluzione di diverso tipo (REST API, flat file ecc.), abbiamo bisogno di un'interfaccia che fornisca la necessaria genericità per sostituire un'implementazione con un'altra.

Questa interfaccia dovrà fornire due metodi:

1. Un metodo per la lettura dei dati.
2. Un metodo per la scrittura dei dati.

Possiamo chiamare questa interfaccia `DataHandler` e assegnarle la seguente struttura.

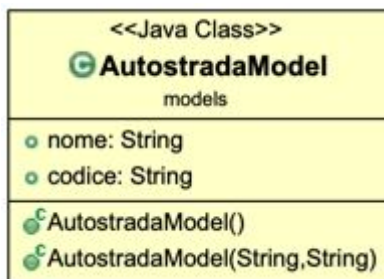




Nell'ottica di seguire il pattern MVC, dobbiamo definire dei modelli (Model) per i dati che andremo a gestire.

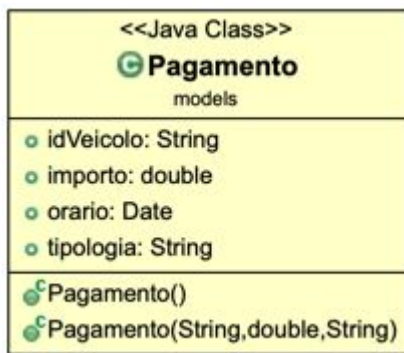
Per procedere dobbiamo creare una classe astratta che rappresenti un modello generico di dati e le operazioni disponibili su di essi.

Il primo dato, gestibile dall'amministratore del sistema, verrà rappresentato dalla classe `AutostradaModel` che estende la classe astratta `Model`.

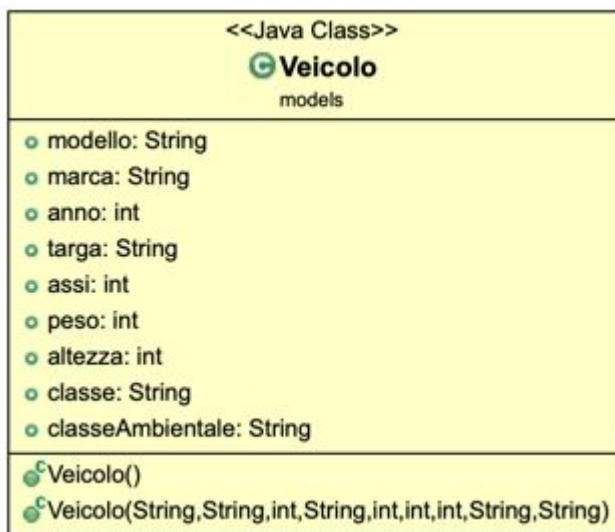


Il modello per i pagamenti, la classe `Pagamento`, seguirà il modello ereditario della classe `AutostradaModel` e avrà la seguente struttura.





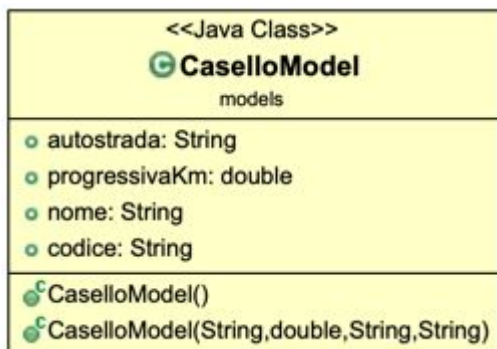
La classe `Veicolo` costituirà a sua volta il modello per tutti i veicoli.



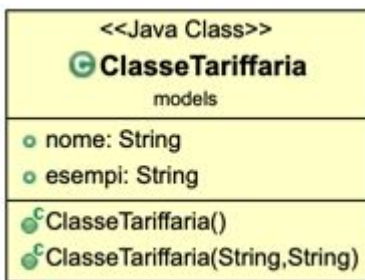


È utile ricordare che la proprietà `classeAmbientale` è fondamentale per gestire il passaggio alla nuova tassazione europea nel calcolo del pedaggio.

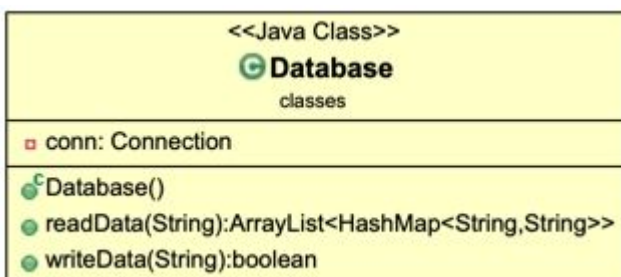
Poiché nel progetto esiste già una classe chiamata `Casello`, il modello che gestirà il tipo di dati casello si chiamerà `CaselloModel` in modo da evitare ambiguità nello sviluppo e aiutare la funzionalità di suggerimento dell'IDE. Avrà la seguente struttura:



Definiamo a questo punto il modello per una classe tariffaria.



Abbiamo definito in precedenza l'interfaccia `DataHandler`. Ora dobbiamo creare una classe che la implementi e che possa essere riutilizzata. Chiameremo questa classe `Database`.



Il metodo `readData()`, da implementare per soddisfare il contratto con l'interfaccia `DataHandler`, compie diverse operazioni sui risultati di una query SQL. La prima consiste nell'estrarre i nomi delle colonne e il tipo di dati associati ad esse. Con queste



informazioni si andranno a reperire i dati usando i metodi appropriati della classe `ResultSet` e uniformandoli nel formato `String`. Queste operazioni si sono rese necessarie in quanto non è possibile riutilizzare un oggetto `ResultSet` dopo che la transazione viene chiusa.

Il metodo `writeData()` esegue una query di aggiornamento sul database.

Questa classe verrà usata dai model sfruttando il principio della composition<sup>2</sup> e troverà un'applicazione nelle classi che la usano.

## Views

Abbiamo tre view principali e quattro sottoview:

1. Login
2. Area amministrativa
  - 2.1 Lista autostrade
  - 2.2 Lista caselli
  - 2.3 Aggiunta autostrada
  - 2.4 Aggiunta casello
3. Area utente

Ciascuna di queste view dovrà estendere una classe base `View` che gestisce la creazione della GUI e dei suoi componenti. A differenza di altre soluzioni, per ottenere un

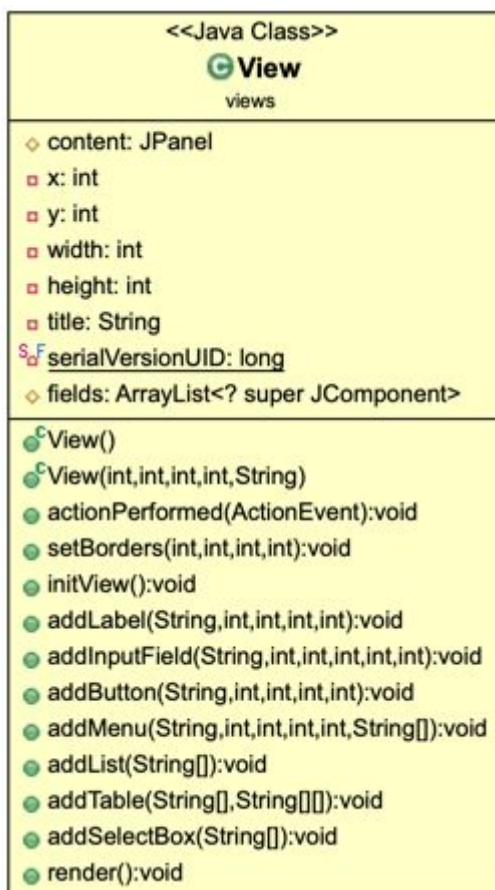
---

<sup>2</sup> *Design Patterns: Elements of Reusable Object-Oriented Software*, p. 20



adeguato livello di agnosticismo circa il tipo e il numero di componenti della GUI, dobbiamo ragionare tenendo conto della gerarchia delle classi del toolkit Swing.

Poiché ogni componente di un pannello (label, pulsanti, campi di testo ecc.) eredita dalla classe `JComponent`, possiamo usare i generics di Java per creare una proprietà nella classe base `protected ArrayList<? super JComponent> fields` in cui inserire i riferimenti ai componenti man mano che questi vengono aggiunti dai metodi della classe `View`. In questo modo tali componenti possono essere riutilizzati dalle classi figlie di `View`.



Come si può notare dallo schema, questa classe presenta molti metodi di utility che semplificano il rendering finale di ciascuna view. Si tratta di scegliere i componenti più comuni di Swing con i loro attributi e parametri (posizione sull'asse x e y, larghezza, altezza ecc.) e inserirne la logica di creazione nei metodi a loro dedicati. L'elemento radice



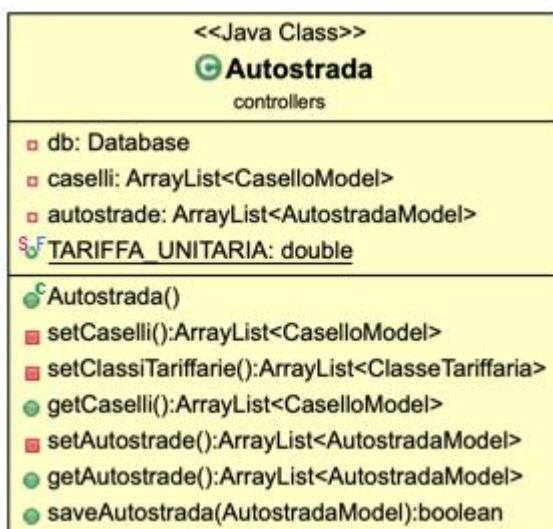
di ciascuna view è `JFrame`. I componenti vengono quindi aggiunti alla proprietà `content` di tipo `JPanel`.

Il metodo `initView()` ha lo scopo di effettuare il setup del pannello principale. `render()` inserisce i componenti aggiunti al pannello principale.

L'area utente consentirà di scegliere il casello di partenza, il casello di destinazione e di caricare un file CSV con i dati del veicolo. Quindi verrà mostrato il totale del pedaggio da pagare e verrà effettuato il pagamento.

## Controllers

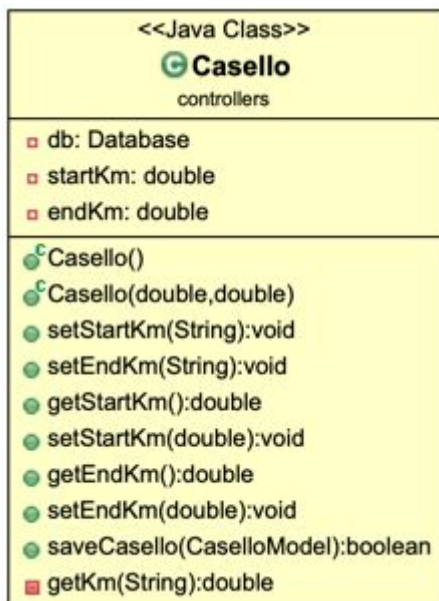
La classe `Autostrada`, come da documento funzionale, gestirà i dati di un'autostrada, compresa la tariffa unitaria e i caselli. Inoltre potrà salvare una nuova autostrada nel database.



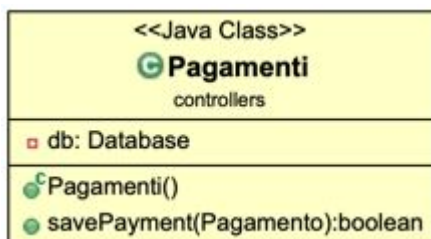
I metodi setter di questa classe reperiscono i dati dal database ed impostano le proprietà relative ai caselli ed alle autostrade.



Casello, come da documento funzionale, gestirà il chilometraggio. Dato che abbiamo l'elenco dei caselli nella base di dati con il relativo chilometraggio, dobbiamo prevedere dei metodi getter per reperire i km in base al nome del casello. Su questi getter potremo sfruttare l'overloading dei metodi.

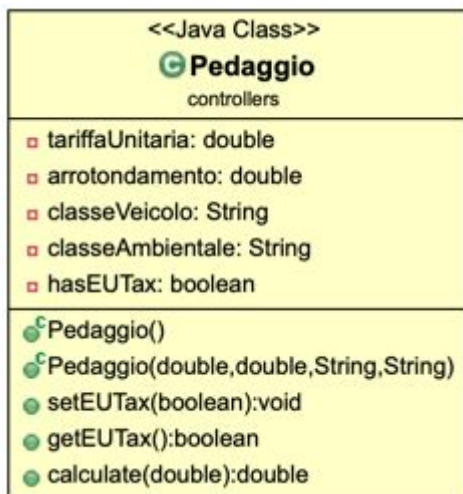


Pagamenti i gestirà il salvataggio dei pagamenti nel database.






`Pedaggio` gestirà il calcolo del pedaggio in base ai parametri definiti nel documento funzionale. Per venire incontro alle future direttive europee, questa classe ha una proprietà booleana che se impostata su `true` attiva l'aggiunta della tassa al calcolo del pedaggio usando la proprietà `classeAmbientale` di ciascun veicolo.







`Percorso` calcola la distanza tra la partenza e l'arrivo usando un metodo dedicato che imposta la proprietà `travelKm`.

<<Java Class>>	
 <b>Percorso</b>	
controllers	
<ul style="list-style-type: none"><li>▣ <code>arrivalKm: double</code></li><li>▣ <code>departureKm: double</code></li><li>▣ <code>travelKm: int</code></li></ul>	
<ul style="list-style-type: none"><li>● <code>Percorso()</code></li><li>● <code>Percorso(double,double)</code></li><li>● <code>calculateTravelKm():void</code></li><li>● <code>getArrivalKm():double</code></li><li>● <code>setArrivalKm(double):void</code></li><li>● <code>getDepartureKm():double</code></li><li>● <code>setDepartureKm(double):void</code></li><li>● <code>getTravelKm():int</code></li><li>● <code>setTravelKm(int):void</code></li></ul>	