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/gabrieleromanato/univaq-pedaggio-autostradale

Requisiti

Il sistema dovrà fornire:

1. Un’interfaccia grafica (GUI) che permetta ad un amministratore di gestire caselli ed autostrade.
2. Un’interfaccia grafica che permetta ad un utente di scegliere un percorso e pagare il pedaggio autostradale dopo aver fornito le caratteristiche del suo veicolo.
3. Un’interfaccia grafica che permetta di effettuare il login nel sistema.
4. Una logica di persistenza dei dati (database, file) che permetta di salvare le modifiche e le operazioni effettuate dagli utenti sul sistema.
5. La capacità del sistema di adattarsi a future modifiche, ad esempio nella modalità di pagamento del pedaggio autostradale o in altri futuri requisiti.
6. Un elevato livello di modularità che garantisca adattabilità, riusabilità e favorisca la continuous integration nel futuro.

Architettura del sistema

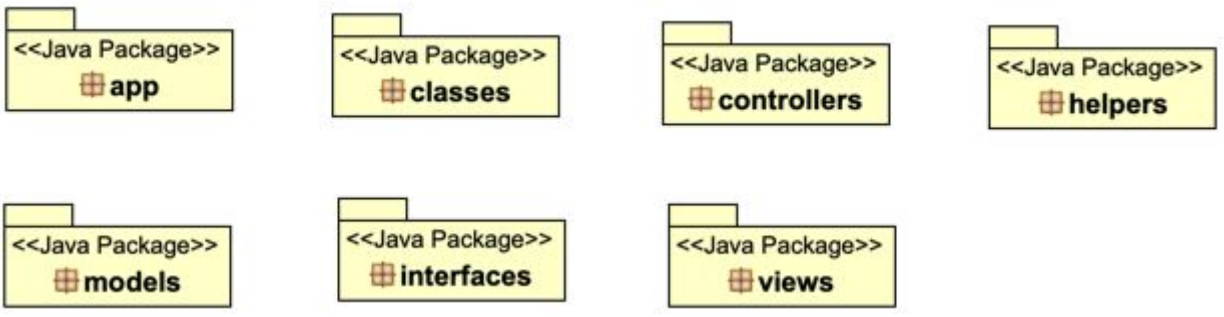
Obiettivi del design

Il sistema dovrà essere modulare rifuggendo dalla tentazione di creare entità monolitiche che in questo modo potrebbero racchiudere troppe funzionalità.

In tal senso ci si baserà su una modularizzazione basata sulla specializzazione dei componenti. Ciascun componente rivestirà un ruolo specifico nell'architettura del sistema senza eccedere in funzionalità.

Modello dell'architettura software

I package principali del sistema sono i seguenti:



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

Descrizione dell’architettura

1. **app** conterrà la logica per il bootstrap e l’inizializzazione dell’applicazione tenendo conto del threading di Java.

2. **classes** conterrà i componenti comuni a tutta l'applicazione, come ad esempio la classe responsabile della persistenza dei dati. In particolare tale classe implementerà il design pattern DAO estendendo un'interfaccia che fornisce la base per l'implementazione di questo design pattern strutturale.
3. **controllers** conterrà le classi controller onorando il design del pattern MVC. Le classi per la gestione delle autostrade, dei caselli, del percorso e del pedaggio ne faranno parte.
4. **helpers** conterrà delle classi di utility per la gestione dei dati (formattazione delle date, parsing di file testuali eccetera).
5. **interfaces** conterrà le interfacce che permetteranno alle classi che ne soddisfano il contratto di soddisfare il requisito principale della modularità senza essere di fatto vincolate ad una sola implementazione specifica.
6. **models** conterrà i modelli di dati secondo il pattern MVC. Ad esempio la classe **Veicolo** farà parte dei models. Tali modelli non conterranno metodi per le operazioni CRUD, come alcune implementazioni del pattern MVC invece consentono in altri linguaggi (ad esempio Laravel in PHP).
7. **views** implementerà la GUI dell'applicazione tramite il toolkit Swing di Java. In particolare vi sarà una classe base che le singole view estenderanno e che conterrà dei metodi che inseriranno i vari componenti dell'interfaccia. Una singola view potrà quindi applicare l'overloading e l'overriding dei metodi in modo da personalizzare la GUI finale.

Descrizione delle scelte e strategie adottate

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.*

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.

Il secondo principio verrà applicato in particolar modo alla gestione del rapporto tra controller e base dati. Invece di far estendere la classe di cui al punto 2 da ogni controller secondo il paradigma IS-A, si applicherà un tipo di rapporto HAS-A in modo da favorire la flessibilità del design.

Il punto focale del sistema è la gestione del calcolo del pedaggio. Per ottenere la flessibilità richiesta, possiamo ispirarci alla scelta del design del database operata dal team di WordPress. La domanda era: come è possibile generare infiniti tipi di post senza modificare il numero di tabelle del database? La risposta è stata molto semplice: modificare il ruolo di un post cambiando il suo tipo, ossia il campo post_type. Ragionando in termini di classi, lo stato di una classe è determinato dalle sue proprietà. Quando una proprietà cambia, cambia anche lo stato della classe. Questo stato può essere usato dai suoi metodi per modificarne il comportamento. Di conseguenza la classe che gestirà il calcolo del pedaggio potrebbe avere queste implementazioni:

1. **Classe con proprietà booleana**: effettuando lo switch della proprietà, il calcolo del pedaggio cambia aggiungendo una nuova condizione.
2. **Classe che implementa un’interfaccia**: in questo caso la classe implementerà solo il metodo per il calcolo del pedaggio, quindi potremmo avere una classe che implementa il calcolo standard, una che implementa i nuovi requisiti europei e così via.
3. **Classe che estende una classe**: in questo caso la classe base implementerà il calcolo standard ed in futuro una classe figlia potrebbe effettuare l’overriding del metodo preposto al calcolo.

2 e 3 presentano lo svantaggio di dover creare nuove classi e componenti, quindi in caso di modifiche occorrerebbe modificare più componenti aumentando il rischio di

errori nelle dipendenze. 1 ha il vantaggio di concentrare le modifiche su un unico componente, tuttavia ci sarebbe il problema futuro dell'aggiunta di nuovi requisiti per cui il singolo flag booleano potrebbe risultare insufficiente.

La situazione così presentata ci offre la possibilità di utilizzare il design pattern creazionale **Factory** per quello che concerne la creazione di istanze specifiche di una stessa classe di tipo Pedaggio, una per il calcolo predefinito ed una per il calcolo con l'aggiunta della tassa basata sulla classe ecologica del veicolo.

Progettazione di classi e interfacce

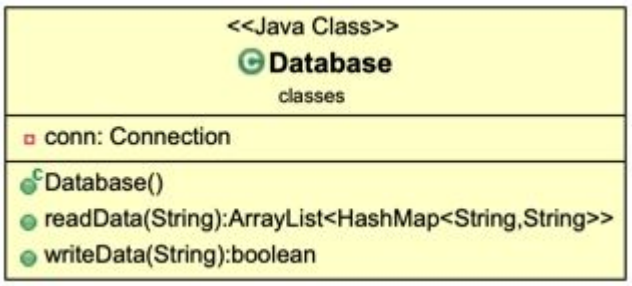
Descrizione di classi, interfacce e membri

L’interfaccia per la base di dati 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.

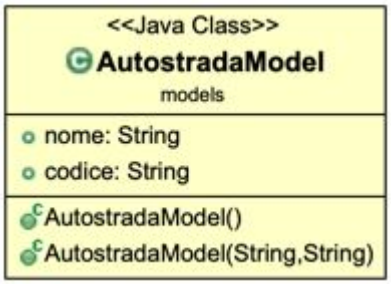
La classe per la gestione della banca dati implementerà tale interfaccia fornendo un’implementazione semplificata del pattern DAO.



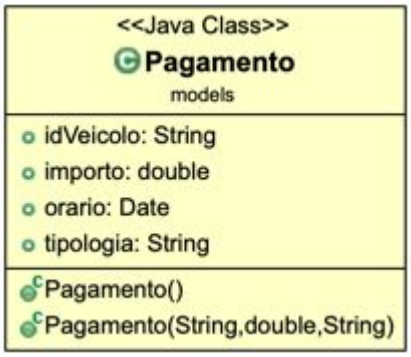
Nell'ottica di seguire il pattern MVC, dobbiamo definire dei modelli (Model) per i dati che andremo a gestire.

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

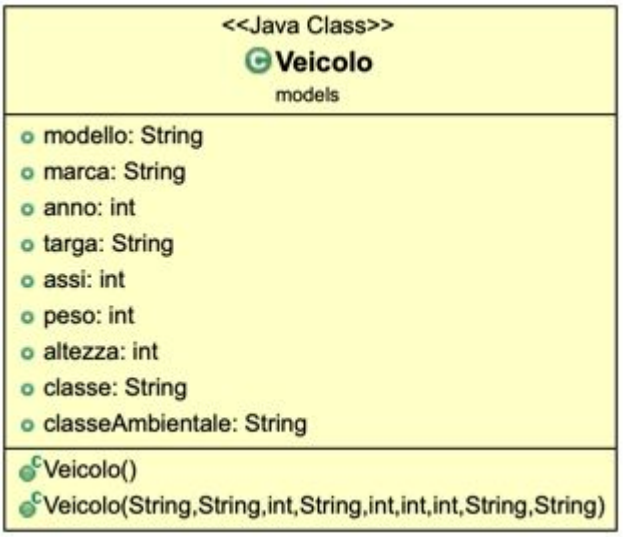
Il primo dato, gestibile dall'amministratore del sistema, verrà rappresentato dalla classe `AutostradaModel` che estende la classe astratta `Model`.



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

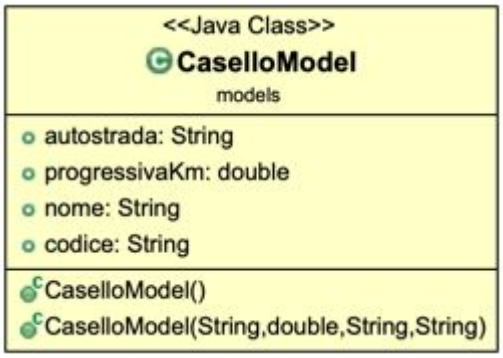


La classe `Veicolo` costituirà a sua volta il modello per tutti i veicoli.

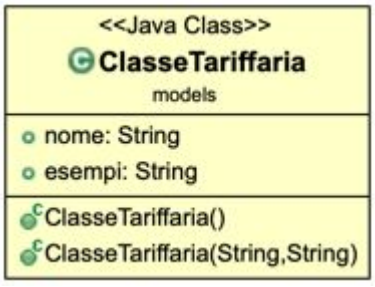


È utile ricordare che la proprietà `classeAmbientale` è fondamentale per gestire il passaggio alla nuova tassazione europea nel calcolo del pedaggio.

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.



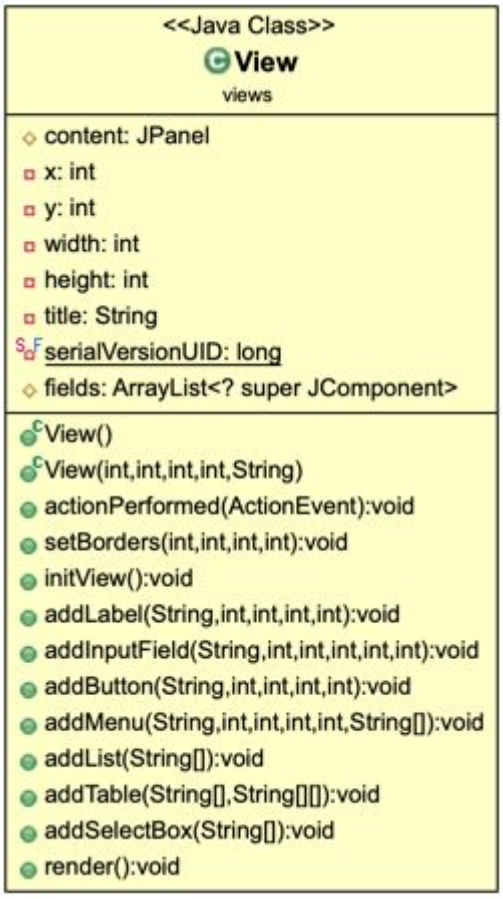
Per le view, abbiamo tre view principali e quattro sottoview:

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 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`.

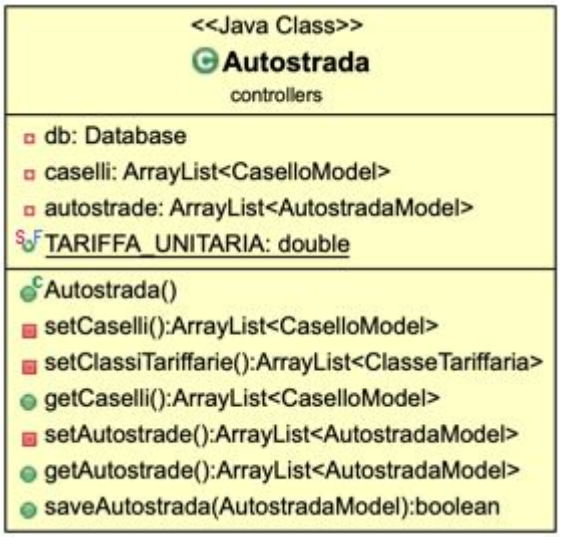


Come si può notare dallo schema, questa classe presenta molti metodi di utility che semplificano il rendering finale di ciascuna view. Si tratta di scegliere i componenti più comuni di Swing con i loro attributi e parametri (posizione sull'asse x e y, larghezza, altezza ecc.) e inserirne la logica di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

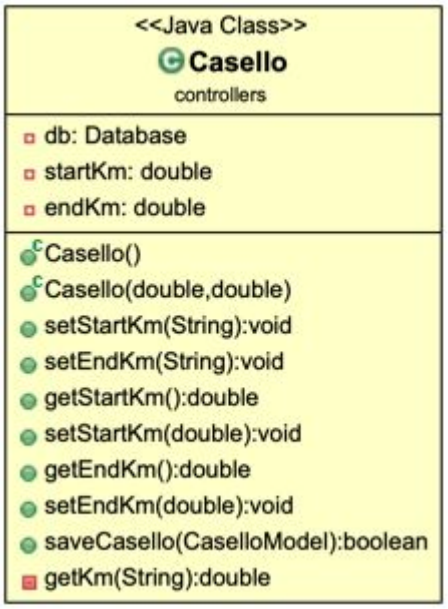
L’area utente consentirà di scegliere il casello di partenza, il casello di destinazione e di caricare un file CSV con i dati del veicolo. Quindi verrà mostrato il totale del pedaggio da pagare e verrà effettuato il pagamento.

Per i controller, 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.

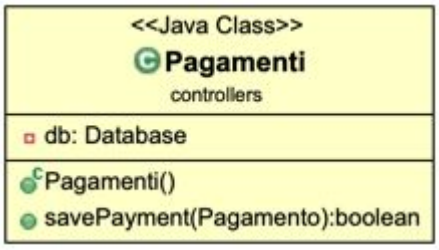


I metodi setter di questa classe reperiscono i dati dal database ed impostano le proprietà relative ai caselli ed alle autostrade.

`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.



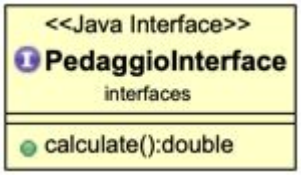
`Pagamenti` gestirà il salvataggio dei pagamenti nel database.



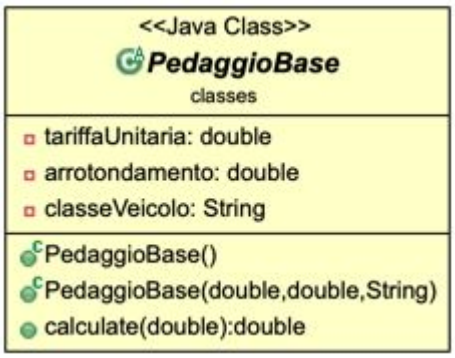
`Pedaggio` gestirà il calcolo del pedaggio in base ai parametri definiti nel documento funzionale. Per venire incontro alle future direttive europee, questa classe ha una proprietà booleana che se impostata su `true` attiva l'aggiunta della tassa al calcolo del pedaggio usando la proprietà `classeAmbientale`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

Un'implementazione priva della proprietà booleana risulterebbe più flessibile ed implementabile in due modi:

1. Con interfaccia.

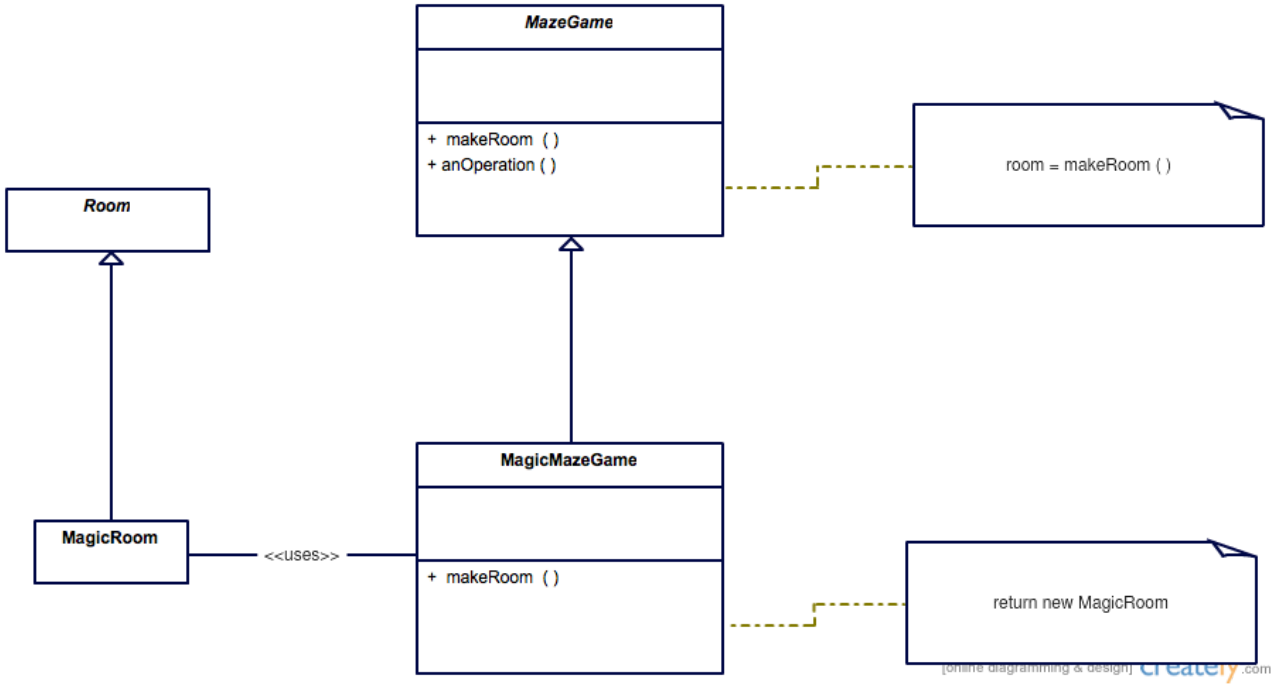


2. Con classe di base.

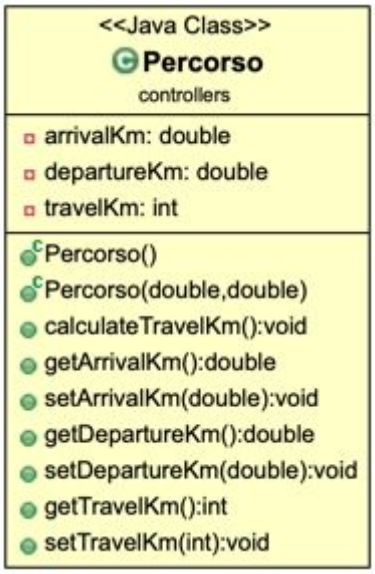


Qui può essere inserita l'implementazione del Factory Pattern in modo da garantire un buon livello di flessibilità e agnosticismo nella creazione di istanze di classi non predeterminate.

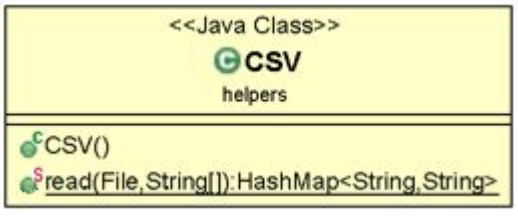
L'esempio che seguiremo sarà basato sullo schema illustrato nel libro *Design Patterns*.



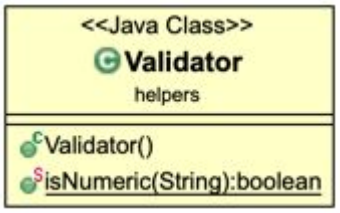
`Percorso` calcola la distanza tra la partenza e l'arrivo usando un metodo dedicato che imposta la proprietà `travelKm`.



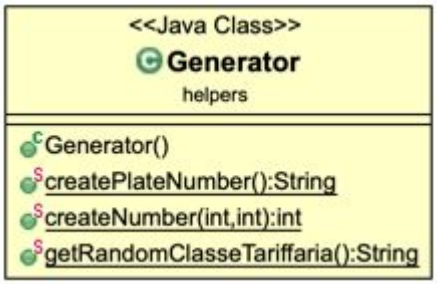
Poiché il documento funzionale prevede l'acquisizione dei dati da file, sceglieremo il formato standard CSV per l'elaborazione dei dati del veicolo forniti dall'utente. A tal proposito creeremo una classe helper dedicata che effettui la lettura di un file CSV e che restituisca una struttura dati coerente.



Poiché la GUI prevede l'input utente, dovremo creare una classe helper deputata alla validazione dei dati in ingresso inseriti come stringhe.



Poiché in futuro le funzionalità del progetto potrebbero prevedere l'impiego di un approccio TDD, abbiamo infine necessità di una classe helper che generi dati fittizi per i test da usare ad esempio nei costruttori predefiniti delle classi.



Descrizione dei dettagli di design scelti

Il progetto non mira a creare un'applicazione perfetta, ma un'applicazione **perfettibile**. Non è raro che il committente richieda in futuro integrazioni e modifiche.

Questa filosofia è alla base del nostro design. Se ad esempio volessimo aggiungere una nuova view (una mappa interattiva), un design monolitico avrebbe fatto si che avremmo dovuto ricopiare tutti i componenti Swing da una classe all'altra. Avendo invece una view di base, questo non è più necessario perché ci troviamo di fatto a riutilizzare funzionalità esistenti e sufficientemente generiche. Se volessimo inserire un logo in tutte le view, dovremmo solo modificare la view di base per fare in modo che il logo a cascata venga ereditato da tutte le view discendenti.

Nel pattern MVC, non abbiamo inserito metodi CRUD nei modelli. L'applicazione è di tipo desktop, quindi di fatto manca un componente fondamentale che da senso a metodi come `get()`, `update()`, `delete()` e `save()`: il passaggio di dati tramite routing. In un framework web ha senso una chiamata come `Model.get(id)` perché il parametro viene passato nella richiesta (ad esempio `/pagamenti/10`). Inoltre sarebbe ridondante definire tali metodi in quanto il nostro design prevede l'implementazione del pattern DAO che già svolge questi compiti. Il risultato finale sarebbe quello di generare confusione tra i programmatori, come spesso accade nei framework web scritti in altri linguaggio dove non è mai semplice scegliere tra le due opzioni (Laravel in PHP ne è un esempio).

In tal senso applichiamo il principio del *favor object composition over inheritance* alla classe che implementa il pattern DAO quando si tratta di riusare tale classe in altri componenti. Questo facilita il loose coupling ed evita così di avere bug a cascata nelle classi che la utilizzano.

Abbiamo creato classi helper allo scopo di non sovraccaricare le classi principali MVC di compiti che non dovrebbero essere di loro competenza. Validare i dati o effettuare il parsing di un file testuale non è ad esempio compito di un controller. Questo invece accadrebbe se non implementassimo le classi con la delegazione dei compiti ma seguissimo un approccio monolitico.

Il compito principale dell'applicazione è calcolare un pedaggio autostradale in base ai dati definiti nel documento principale del progetto. L'incognita è rappresentata dalle modifiche future da applicare all'algoritmo del calcolo. Il fatto che attualmente tali modifiche riguardino una normativa europea è incidentale: le modifiche future infatti potrebbero essere di qualsiasi tipo.

Per questo motivo, la classe `Pedaggio` deve essere sufficientemente flessibile per adattarsi ai cambiamenti futuri. In generale, tuttavia, tutti i componenti dovrebbero seguire lo stesso design di base di questa classe.

Il pattern Factory garantisce un certo livello di astrazione nella creazione degli oggetti. Se si opta per un approccio non basato su un cambio di stato e comportamento della classe ma su un approccio creazionale, questo pattern è sufficientemente robusto per far si che in futuro una modifica al sistema di calcolo non determini un refactoring dell'intero code base.