1506
UNIVERSITÀ
DEGLI STUDI
DI URBINO
CARLO BO

Studenti

Berardi Martin
Bigelli Leonardo
Dragne Bogdan Andrei
Pecmarkaj Arlind
Petrelli Tommaso

PROGETTO

Corso di Programmazione e Modellazione ad Oggetti

Sessione invernale 2021/2022

Docente: *Prof.ssa Montagna Sara*

Indice

1. Analisi

Il progetto consiste nell'implementazione di un gestionale di parcheggi.

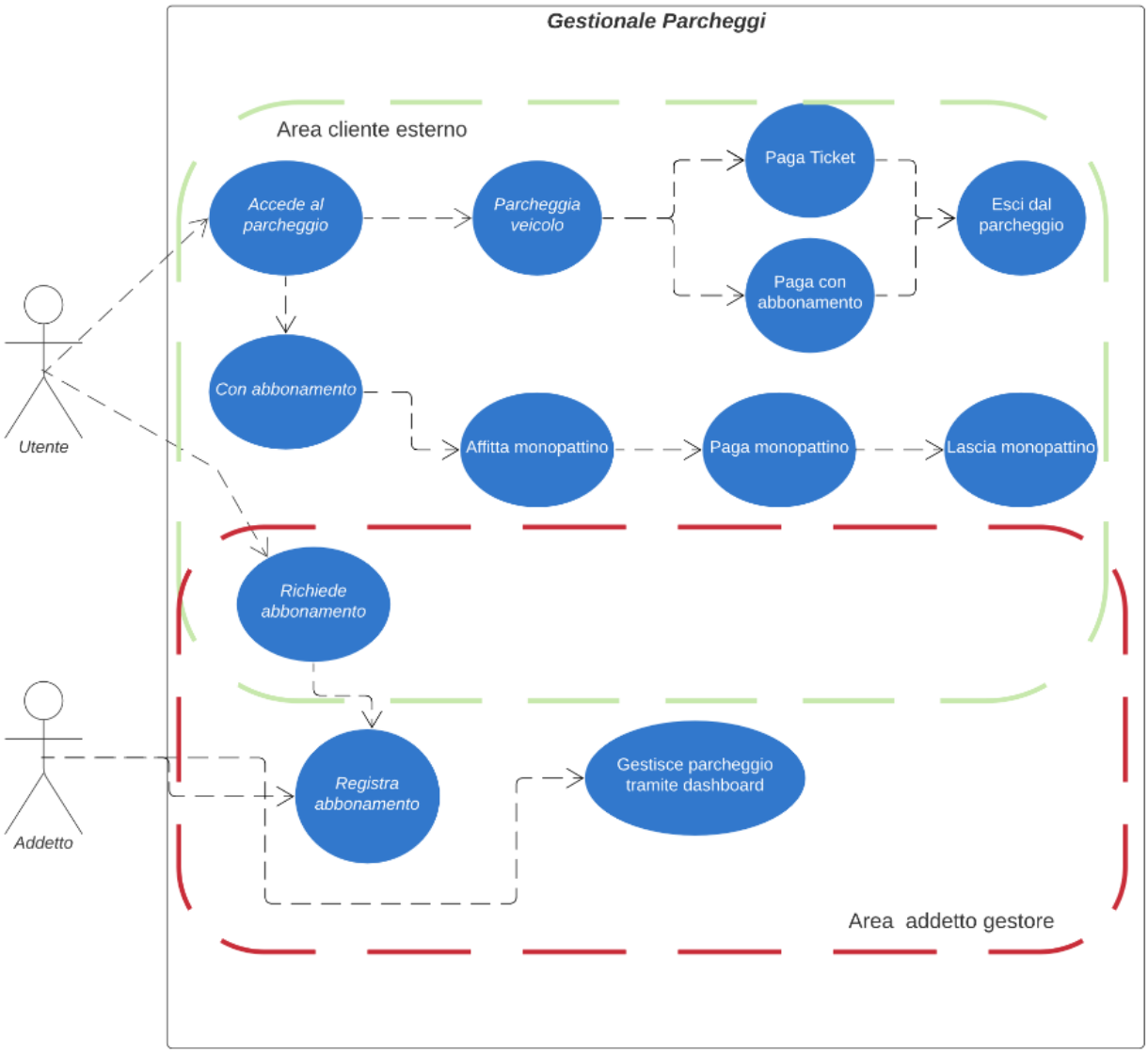
- Ogni parcheggio dispone di posti per auto e per moto, con la possibilità di effettuare il noleggio di monopattini elettrici.
- All'ingresso di ogni parcheggio è presente un sensore che rileva l'altezza di ogni auto, in modo tale da permettere l'ingresso al parcheggio solamente alle auto che non superano il limite massimo di altezza. In ogni posto è presente un sensore che, in base all'emissione dei gas di scarico, rileva il tipo di carburante utilizzato dall'auto. Nei parcheggi sotterranei non sarà possibile parcheggiare con macchine a GPL o a metano.
- Ogni proprietario ha la possibilità di fare un abbonamento per il parcheggio. L'abbonamento è strettamente associato ad un solo veicolo.
- Un utente, se dispone di abbonamento per il parcheggio, ha la possibilità di noleggiare un monopattino con una certa tariffa oraria. Si permette l'acquisto di un abbonamento premium che elimina la tariffa oraria di utilizzo del monopattino.
- Se all'uscita dal parcheggio il veicolo non risulta associato a nessun abbonamento, verrà calcolato il costo di utilizzo del parcheggio secondo la tariffa oraria imposta.
- I dati riguardanti il parcheggio vengono salvati in un file, che viene caricato durante l'apertura dell'applicazione e che viene aggiornato alla chiusura di quest'ultimo.
- È previsto l'utilizzo di un'interfaccia grafica (GUI).

1.1 Requisiti

L'applicazione dovrà essere in grado di gestire l'utilizzo di parcheggi per auto e per moto, con la possibilità di effettuare il noleggio di monopattini elettrici, se disponibili.

I dati immessi e successivamente elaborati dovranno essere salvati in un file di testo, che verrà caricato durante l'apertura dell'applicazione e che verrà aggiornato alla chiusura di quest'ultimo.

L'applicazione disporrà di un'interfaccia grafica (GUI) per visualizzare lo stato del programma e per semplificare l'interazione dell'utente verso l'applicazione stessa.



Use Case Diagram

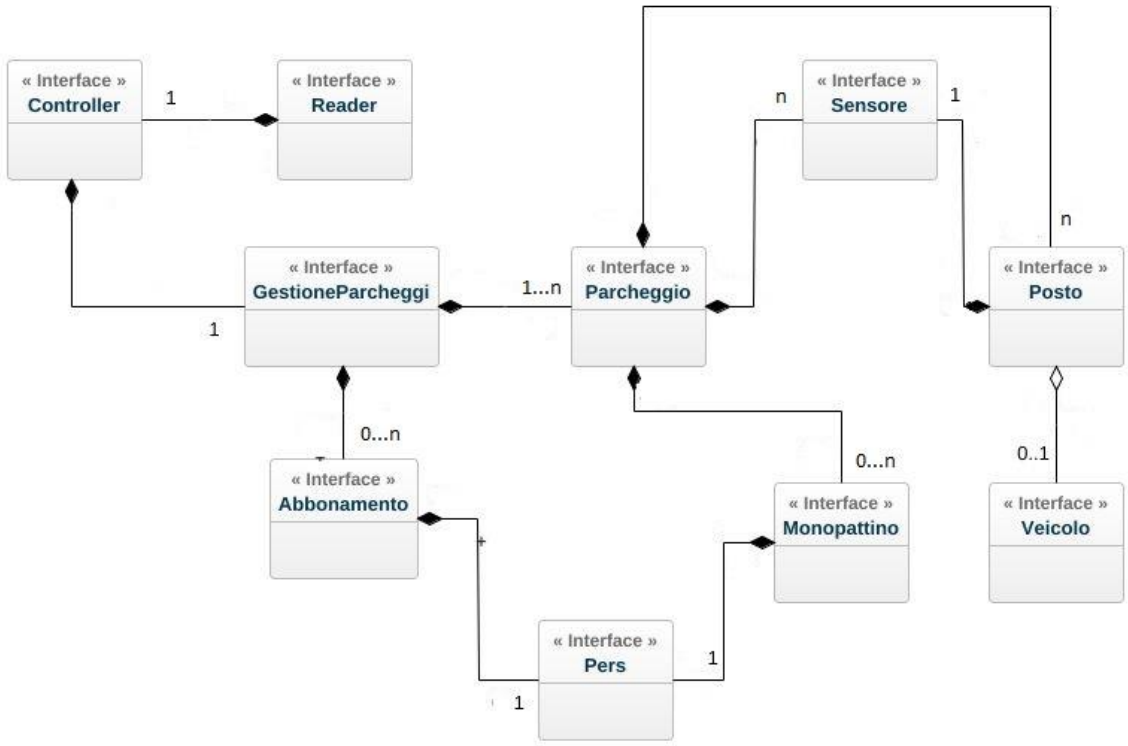
1.2 Modello del dominio

Ogni parcheggio dispone di posti per auto e per moto, con la possibilità di effettuare il noleggio di monopattini elettrici.

All'ingresso di ogni parcheggio è presente un sensore che rileva l'altezza di ogni auto, in modo tale da permettere l'ingresso al parcheggio solamente alle auto che non superano il limite massimo di altezza. Inoltre, in ogni posto è presente un sensore che, in base all'emissione dei gas di scarico, rileva il tipo di carburante utilizzato dall'auto. Nei parcheggi sotterranei non sarà possibile parcheggiare con macchine a metano.

Ogni utente ha la possibilità di effettuare un abbonamento per il parcheggio. L'abbonamento è strettamente associato ad un solo veicolo. Un utente, se dispone di abbonamento per il parcheggio, ha la possibilità di noleggiare un monopattino secondo una certa tariffa oraria. È possibile acquistare un abbonamento premium che elimina la tariffa oraria di utilizzo del monopattino. Se all'uscita dal parcheggio il veicolo uscente non risulta associato a nessun abbonamento, verrà calcolato il costo di utilizzo del parcheggio secondo la tariffa oraria imposta.

L'immagine che segue rappresenta il diagramma delle classi del nostro scenario.



2. Design

La classe principale della struttura è Parcheggio. In essa vengono istanziati i vari posti per le auto e quelli per le moto, quest'ultimi definiti tramite delle classi apposite, PostoAuto e PostoMoto, esse sono realizzate partendo da una classe astratta utilizzata come template (Posto). La scelta di utilizzare uno scheletro di questo tipo è data dal fatto che non ha senso istanziare un posto generico, in quanto i posti potranno essere per le auto o per le moto, che avranno proprietà diverse ma anche qualche metodo in comune (motivazione principale dell'utilizzo di una classe astratta più generica).

Per rappresentare tutti i veicoli in generale è stata creata una classe Veicolo. La creazione delle Auto e delle Moto è data dall'estensione (`extends`) della classe Veicolo. Per rappresentare tutti i possibili tipi di carburante è stato scelto di utilizzare un tipo `enum` Alimentazione, in modo tale da avere accesso esplicito al tipo di carburante utilizzato dal veicolo.

Il parcheggio comunicherà con i singoli posti di quest'ultimo, che a sua volta dialogherà con i veicoli che ci sosteranno. I vari sensori sono presenti nei posti per parcheggiare i veicoli. Il parcheggio potrà gestirli attraverso i posti che saranno presenti in esso.

2.1 Architettura

L'architettura generale del software e il pattern seguito per la progettazione prende spunto dal Model View Controller (indicato come MVC) che viene modificato per le nostre esigenze.
Il model è una descrizione dei tipi usati per risolvere il problema e per fornire le funzionalità, mentre il view è l'interfaccia grafica che si presenta all'utente.
Mentre nel MVC il controller si occupa di interfacciarsi con la View e fornire le funzionalità della parte di Model, nel nostro caso il Controller si occupa di ricevere la directory per la lettura dei dati e di fornire la classe di gestione alla View che potrà usare solo quello che fornisce il Controller.
Questo è stato fatto semplicemente per avere (a parer nostro) una maggiore semplicità di lettura globale del software le cui dimensioni relativamente limitate giustificano questo approccio. C'è anche da considerare che l'intero sistema è progettato per parcheggi di dimensioni discrete, i quali offrono funzionalità basi integrate con la mobilità green e nella fase di manutenzione futura non si prevedono stravolgimenti totali al programma.

2.2 Design dettagliato

Arlind Pecmarkaj

Classi realizzate:

- GestioneParcheggi.java
- ReaderWriter.java e sottotipi
- Controller.java
- GUIGestione.java

Per prima cosa abbiamo bisogno della classe fondamentale che tiene traccia di tutti i parcheggi e che tenga in memoria gli abbonamenti che vengono inseriti.
Come pattern si usa il singleton, ossia per l'intero software si tiene una singola istanza di Gestione in quanto tenersi più Gestionali per gli stessi parcheggi sarebbe inutile.
La classe fornisce i metodi per inserire Parcheggi e per inserire abbonamenti e per ottenere tutta la lista di essi o una singola istanza.
Per memorizzare sia gli abbonamenti che i parcheggi si è scelti di usare gli ArrayList in quanto si combinano i vantaggi delle Collections con quelle degli array.
La classe si occupa anche di aggiornare gli abbonamenti (eliminando quelli scaduti) e di trasferirli ai vari parcheggi.
Si è deciso che i parcheggi alla fine vengono inseriti tramite file. Per questo si è reso necessario l'uso di un interfaccia che si occupa di ciò: ReaderWriter.java;

L'interfaccia è fondamentale in quanto oltre a leggere i parcheggi da un file seguendo una stretta formattazione (si assume che il file sia ben formattato al primo avvio) li riscrive, permettendo un riutilizzo consono dei parcheggi che vengono gestiti. L'interfaccia è parametrizzata in caso in futuro si voglia leggere da file nuovi tipi di oggetti e come interfaccia permette alle classi che la implementano di gestire autonomamente la lettura difatti basti pensare nel caso volessimo aggiornare l'intero software con la lettura da database basterebbe semplicemente creare una nuova classe.
Chi si occupa di gestire entrambe le cose è la classe Controller.java che si tiene una istanza del lettore e del gestore e passa all'interfaccia grafica (spiegato successivamente) gli oggetti necessari al funzionamento. Infatti il controller si occupa pure di far partire l'interfaccia grafica.
GestioneParcheggi è l'unica classe nella Model a non implementare un'interfaccia poiché per come è progettata (mettendola in parole semplice è un contenitore di parcheggi e abbonamenti) non si è reso necessario avere un tipo che fornisse un contratto da seguire. Andando a vedere la classe di gestione fornisce gli abbonamenti e i parcheggi e non si riesce a pensare a cosa potrebbe fornire in più rispetto a ciò. Per come è strutturato il software e alla 'simbiosi' che c'è tra interfaccia grafica e gestione dei tipi si è visto come il contratto da seguire per fornire le funzionalità di gestione è unico e non si prevede che ciò possa effettivamente essere modificato.
Il controller, come spiegato nella sezione precedente, si occupa di tenersi il lettore e scrittore e di tenersi la classe GestioneParcheggi; esso fa la lettura dei dati, crea un istanza dei parcheggi che son presenti e con essi apre l'interfaccia grafica che ottiene i dati che gli servono.
In questa situazione la View, sì lavora direttamente con i tipi che rappresentano i dati, ma li riceve esclusivamente dal Controller.
GUIGestione.java all'apertura mostra i parcheggi che ha ricevuto e un form per l'inserimento degli abbonamenti. Cliccando sui singoli parcheggi viene aperto il parcheggio selezionato da cui il dipendente (l'utente del software) lavorerà

Tommaso Petrelli

Classi e *Interfacce* realizzate:

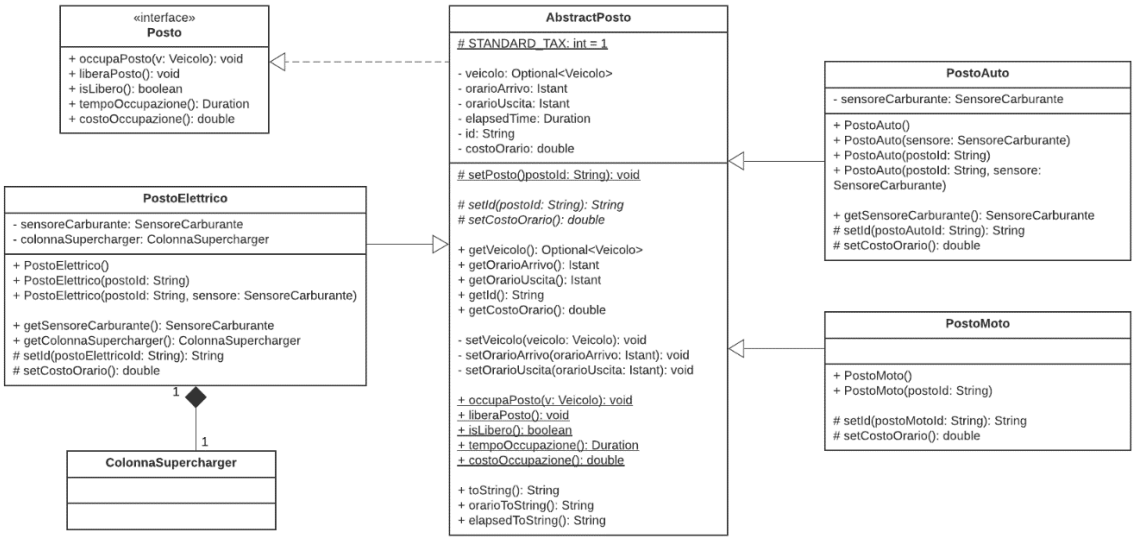
- Posto.java
    - AbstractPosto.java
        - PostoAuto.java
        - PostoElettrico.java

        - PostoMoto.java
- Supercharger.java
    - ColonnaSupercharger.java
- GUIParcheggio.java
- GUIRicaricaAuto.java

Posto del parcheggio

1. In questa sezione il problema posto è quello di fornire all'intero ecosistema dell'applicazione delle aree inizialmente vuote in cui potranno essere parcheggiati i veicoli. Il compito di ognuna di queste aree, che chiameremo "posto", sarà quello di fornire informazioni raccolte durante la permanenza di un veicolo. Per cui, ogni posto deve:
    a. registrare il tempo di arrivo e di uscita del veicolo;
    b. calcolare il tempo di occupazione del posto;
    c. calcolare il prezzo da pagare per aver occupato il parcheggio sulla base del tempo di occupazione delle tariffe del parcheggio;
    d. fornire l'informazione per cui il posto è libero oppure occupato.

    Per cui notiamo che la natura di questo problema diverge da quelle gestionali dei problemi di più alto livello.
2. La soluzione valutata più positivamente per risolvere tale problema è stata quella di generalizzare il concetto di posto e renderlo indipendente da ciò che dovrà ospitare, in quanto anche nell'enunciato del problema non si dichiarano vincoli dovuti alla tipologia del veicolo che richiede la sosta. Utilizzando questo approccio è possibile sfruttare appieno il vantaggio della programmazione in team in quanto si applica una netta separazione tra le diverse sezioni in cui è stato diviso il lavoro. Si vuole risolvere questo problema andando a costruire delle entità da utilizzare in classi più generali e che implementano le logiche di gestione dei parcheggi. Come altro vantaggio abbiamo che una soluzione di questo tipo mi consente di utilizzare il noto design pattern detto *Template Method* (descritto nel punto 4).
3. Passiamo ora a vedere come possiamo implementare questa soluzione e come possiamo legare tutte le entità che concorrono alla risoluzione del problema. A questo proposito useremo uno schema UML:

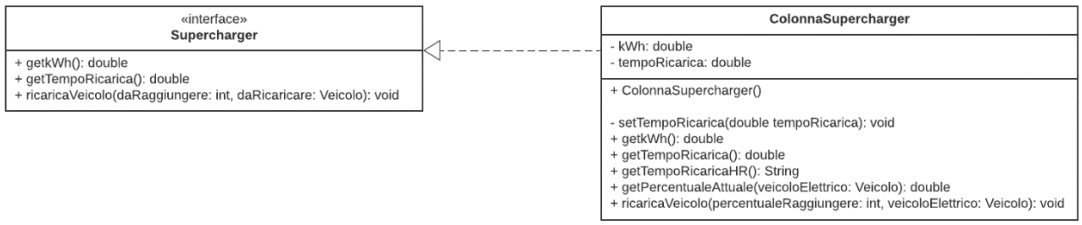


4. Tenendo sott'occhio il diagramma UML, osserviamo come il pattern Template Method venga utilizzato a partire dalla classe astratta `AbstractPosto`. Questa classe astratta va ad implementare l'interfaccia `Posto` che fornisce il contratto che un qualsiasi tipo di posto dovrà rispettare. In questo caso, sarà solo `AbstractPosto` ad implementare tale interfaccia poiché abbiamo detto di volerci mantenere il più indipendenti e generali possibili. In tale classe astratta andiamo a specificare la struttura algoritmica per la creazione di un posto, e tale operazione dovrà differire per alcuni passi e dettagli a seconda delle sottoclassi. Questo lo facciamo in accordo con la progettazione indicata dal pattern scelto. Il Template Method, infatti, mi consente di ridefinire certi passi di un algoritmo senza cambiare la struttura di esso. In particolare, tale algoritmo viene definito all'interno del *metodo template* `setPosto()` che viene dichiarato `final` per ribadire l'immutabilità della struttura essenziale dell'algoritmo. Le sottoclassi che estendono la classe astratta sono `PostoAuto`, `PostoElettrico` e `PostoMoto`, che rispettivamente specializzano il concetto di posto in posto per autovetture, posto per auto a motore elettrico ed in posto per motocicli. Di nuovo in accordo con il pattern, queste sottoclassi hanno il compito di implementare le operazioni primitive dell'algoritmo che devono essere ridefinite, oltre ovviamente a definire nuovi metodi specifici.

<u>Supercharger per l'elettrico</u>

1. Il problema che ci poniamo per questa sezione del progetto è quello di trovare un modo per riuscire a ricaricare un'auto elettrica che si parcheggia in un posto riservato alle auto elettriche.

2. Da come viene posto il problema si sceglie di adottare come soluzione quella di sfruttare il concetto di *Reuse By Composition* fornito dal paradigma della programmazione a oggetti. Infatti, viene specificato che le auto elettriche potranno ricaricarsi non in un qualsiasi posto ma solo in quelli riservati all'elettrico. Allora possiamo specializzare la classe `PostoElettrico` aggiungendo un campo per comporre quello che è stato definito come Supercharger, ossia un entità che rappresenta una comune colonnina di ricarica per autovetture. Tale Supercharger dovrà poi essere caratterizzato e progettato in modo tale da poter garantire funzionalità come: avvisare l'utente del tempo di ricarica e, ovviamente, permettere la ricarica del veicolo. Per l'azione di ricaricare il veicolo è stato scelto di dare la possibilità all'utente di impostare la percentuale di ricarica che si vuole far raggiungere alla vettura, e sulla base di questa scelta poi il Supercharger restituirà il tempo necessario per completare la ricarica.
3. Vediamo come possiamo implementare questa soluzione usando uno schema UML:



4. Data la natura semplice della soluzione non abbiamo la necessità di passare all'utilizzo di uno specifico design pattern. Quello che andiamo ad utilizzare è invece il meccanismo delle interfacce, messo a disposizione dal paradigma di programmazione utilizzato. Nell'interfaccia `Supercharger` specifichiamo il contratto, e quindi le operazioni fondamentali che un Supercharger deve garantire e che sono state definite in fase di progettazione. Separatamente definiamo una classe che rappresenta effettivamente l'astrazione di un reale Supercharger. In tale classe vediamo aggiungersi altri metodi che serviranno ad esempio, se consideriamo `getPercentualeAttuale()`, a mostrare all'utente informazioni sempre relative al caricamento dell'auto elettrica, in questo caso, la percentuale della batteria del veicolo prima di ricaricare.

Leonardo Bigelli

Classi realizzate:

- Parcheggio.java

    - ParcheggioImpl.java
- Sensore.java
- Monopattino.java
    - MonopattinoImpl.java
- GUIParcheggio.java (in collaborazione con Arlind Pecmarkaj e Tommaso Petrelli)

Parcheggio e ParcheggioImpl:

La classe presa in considerazione è il cuore dell'intero sistema, in quanto è colei che va ad utilizzare tutte le altre componenti realizzate. In esse sono presenti un numero finito di posti generici (per le auto o per le moto), un insieme di abbonamenti e anche un numero di monopattini disponibili per il noleggio. Tutte queste componenti che vanno ad astrarre il parcheggio in sé sono gestite in maniere dinamica, tramite l'utilizzo di collezioni che il linguaggio ci mette a disposizione. Il parcheggio è definito con caratteristiche il cui numero può variare a seconda di come viene richiesta l'istanza di un nuovo oggetto. La classe Parcheggio.java non è altro che l'interfaccia dove al suo interno sono presenti i messaggi che più comunemente un parcheggio scambierà con altre entità. Un parcheggio permette di aggiungere un nuovo veicolo e quindi di parcheggiarlo, liberare un posto occupato, noleggiare un monopattino e, di conseguenza, anche di restituirlo, fornire una lista di veicoli parcheggiati in esso e fornire anche il numero di posti specifici al suo interno. In particolare quest'ultimi messaggi sono stati realizzati sfruttando l'esistenza degli stream a cui veniva applicato un filtro. Alcuni filtri sono stati implementati seguendo il Pattern Strategy, infatti ad essi gli viene passata una strategia generica permettendo al singolo metodo di funzionare per diverse situazioni (es. *getNPostiSpecifici(Predicate<Posto> filtro)*). Il metodo citato restituisce il numero di posti di cui è composto il parcheggio, che siano posti per le moto o per le auto. Il suo valore di ritorno dipenderà dal tipo di strategia passata come parametro d'ingresso.

Il parcheggio è identificato da un codice alfanumerico univoco (id). Una caratteristica fondamentale è data dal primo carattere di questo codice:

1. 'S' → Identifica che il parcheggio sarà sotterraneo con conseguenza che le auto a metano e a GPL non potranno sostarsi;
2. 'Qualsiasi altra lettera' → Identifica un generico parcheggio all'aperto.

Successivamente sono descritte le problematiche e le loro gestioni, vengono citate solamente questioni più complesse dal punto di vista algoritmico.

Aggiunta di un veicolo

Aggiungere un qualsiasi veicolo è il metodo basilare del parcheggio insieme a quello per liberare un posto. Questo sarà uno dei messaggi più scambiati con la classe di gestione per controllare più parcheggi.

Gestione

Per evitare di implementare metodi diversi per ciascun tipo di veicolo (auto, auto elettrica e moto) ho realizzato un unico metodo dove al suo interno andrà a invocare un metodo privato chiamato *filtraAggiungi()*. Quindi si è utilizzato il pattern DRY (Don't Repeate Yourself). Il metodo sfrutta gli stream e i filtri applicati a essi per identificare di quale veicolo si tratti, con lo scopo di parcheggiarlo nel posto giusto. Al metodo si passa come parametro una strategia di filtraggio utilizzata poi nello stream (Pattern Strategy). Con questo procedimento evitiamo di scrivere codice specifico per trovare il posto pertinente alla tipologia di veicolo. Il metodo lancerà delle eccezioni per identificare eventuali problemi:

- **AltezzaMassimaConsentitaException** → il veicolo è troppo alto (per parcheggi sotterranei), la rilevazione viene effettuata tramite un sensore posto all'ingresso del parcheggio;
- **TipologiaCarburanteNonConsentitaException** → L'auto è a metano e non sono ammesse per i parcheggi sotterranei;
- **PostiFinitiException** → Posti non più disponibili.

Noleggio monopattini

Ogni parcheggio ha a disposizione un numero di monopattini da poter noleggiare alle persone che lasciano il proprio veicolo parcheggiato. Il numero dei monopattini può variare da zero a un numero potenzialmente elevato, a seconda del tipo di parcheggio che si vuole rappresentare.

Gestione

La gestione del noleggio è simulata, ovvero l'oggetto del monopattino non uscirà mai dal parcheggio e per gestire questa caratteristica è presente un flag di tipo booleano come campo della classe rappresentante il monopattino, che permetterà al sistema di capire se il monopattino è già stato noleggiato o meno. Anche in questo caso l'uso degli stream è stato fondamentale per poter identificare i monopattini disponibili e per poi restituirli (es. *restituisciMonopattino()*).

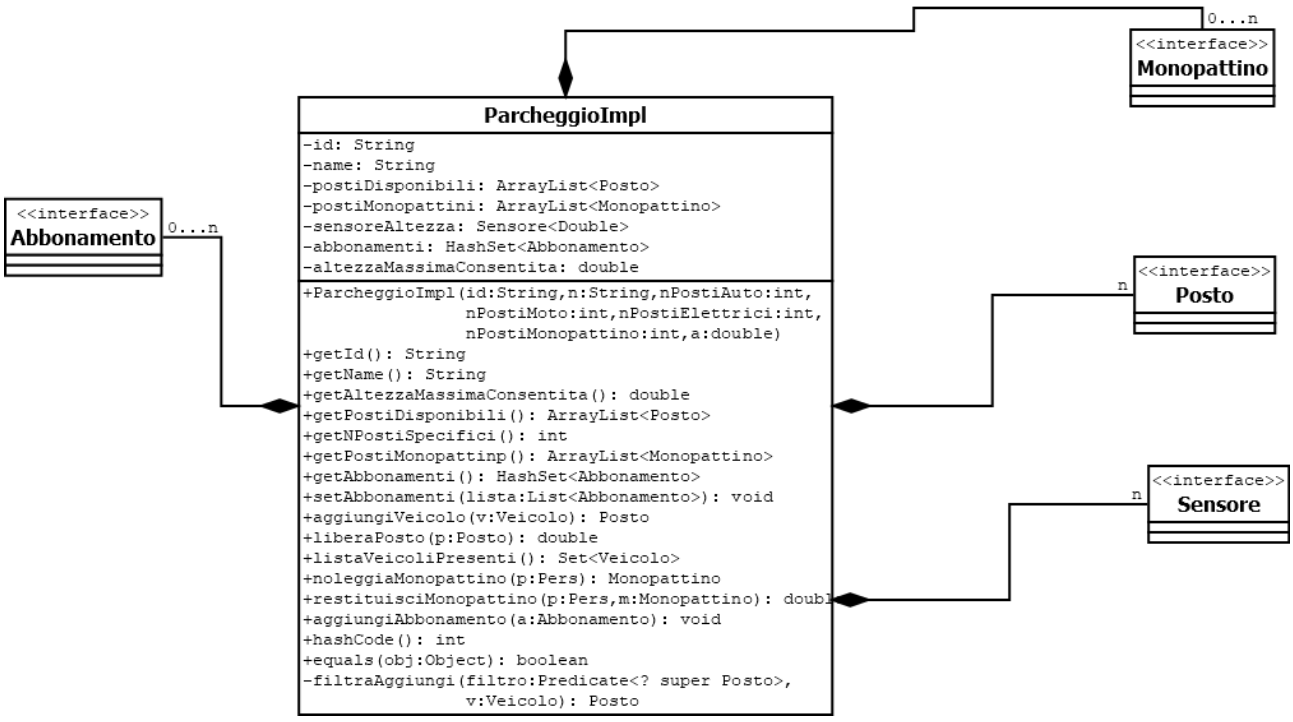
Rilevazione carburante

Ogni singolo posto auto è munito di un sensore che effettuerà una rilevazione non appena si cerca di parcheggiare un veicolo. A seconda della tipologia di parcheggio (sotterraneo o esterno) la rilevazione avrà un effetto diverso.

Gestione

Nel caso di un parcheggio esterno il sensore non influirà in nessun modo sull'azione di parcheggiare. La situazione cambia nel caso di un parcheggio sotterraneo, una volta rilevato il carburante (operazione simulata eseguendo un get() sul tipo di carburante del veicolo) se quest'ultimo dovesse essere a metano verrà generata un eccezione e l'auto non verrà parcheggiata.

Segue un diagramma UML dettagliato, contenente anche un metodo privati, rappresentante l'implementazione del parcheggio.



Sensore

La classe sensore è una interfaccia con lo scopo di astrarre, in questo scenario, due tipologie di sensori:

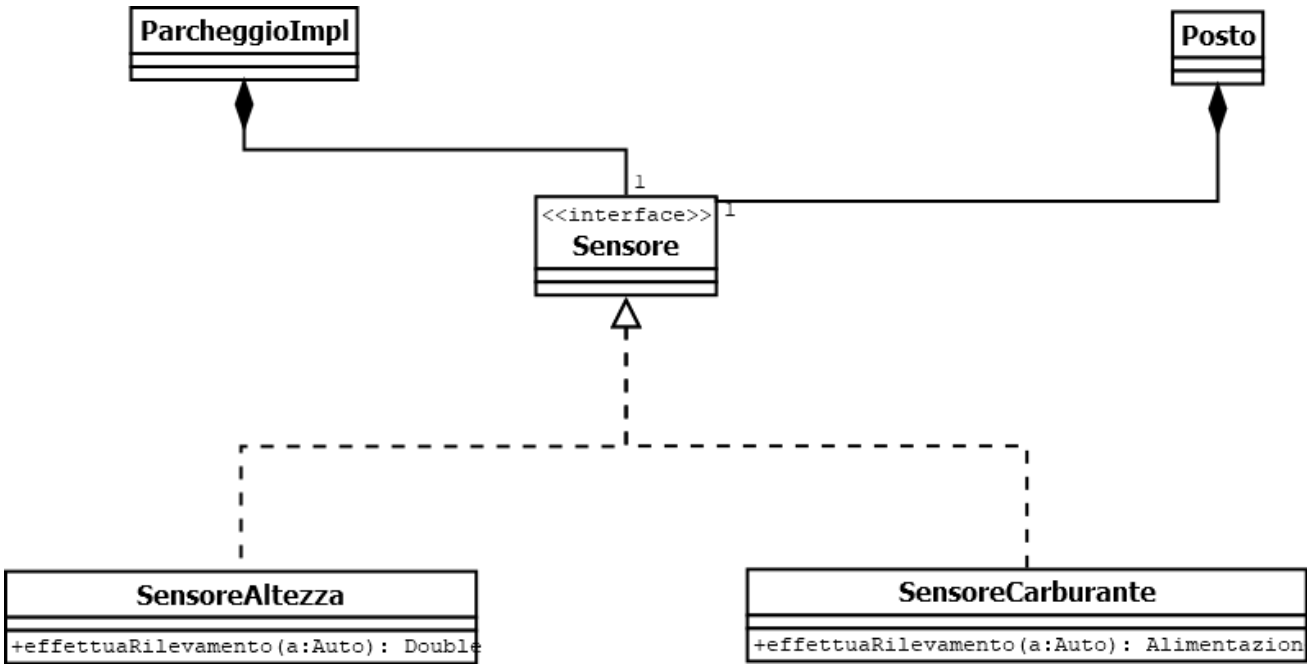
- Sensore per rilevare il carburante di un veicolo;
- Sensore per rilevare l'altezza di un veicolo.

La prima tipologia di sensore è utilizzata per negare l'accesso al parcheggio, se sotterraneo come descritto in precedenza, in caso si trattasse di un'alimentazione a metano. Mentre la seconda tipologia è utilizzata per vietare l'entrata a un veicolo con altezza non consentita. Il limite di quest'ultima sarà prestabilita da chi gestirà i parcheggi.

Il sensore non è altro che un'interfaccia parametrizzata con un unico metodo al suo interno anch'esso parametrizzato, chiamato '*effettuaRilevazione()*'. La scelta di utilizzare una classe parametrizzata è dovuta perché i due sensori presi in questione avranno scambieranno con il resto del sistema lo stesso messaggio, ma esso avrà un valore di ritorno differente a seconda della tipologia del sensore. Infatti nel caso del sensore d'altezza, il metodo restituirà un valore di tipo Double (classe wrapper in quanto le classi parametrizzate non accettano tipi di dato primitivi). Mentre l'altro

tipo di sensore dovrà comunicare il tipo di carburante, quindi un tipo Alimentazione (enum).

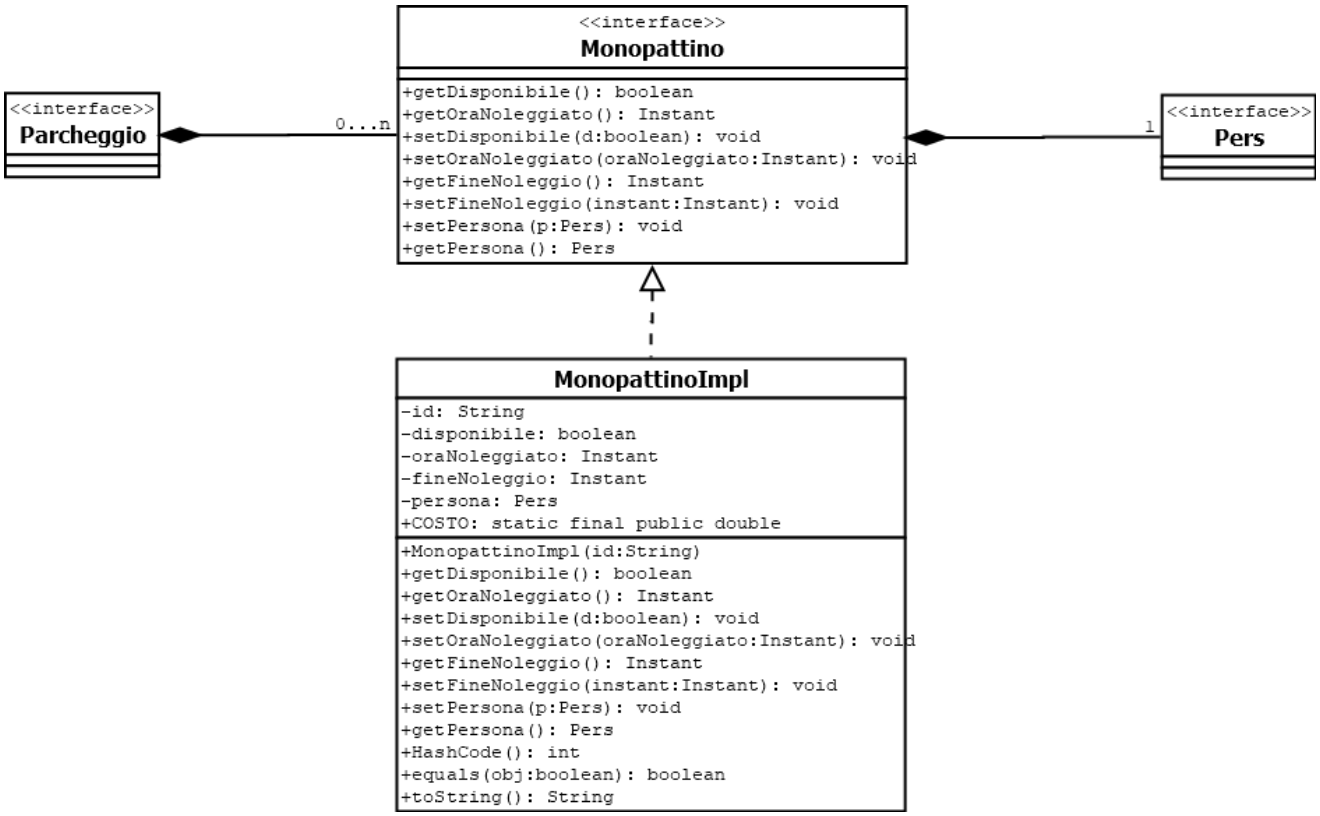
L'immagine successiva rappresenta l'UML del Sensore e delle classi che lo utilizzano. Il diagramma è dettagliato solo per il sensore non anche per le classi che sono composte da esso.



Monopattino e MonopattinoImpl

La classe MonopattinoImpl è l'astrazione di un monopattino elettrico. Essa implementa l'interfaccia Monopattino. Come campo è anche presente una costante che rappresenta la tariffa di noleggio. L'interfaccia citata garantisce la modellizzazione di future nuove tipologie di monopattini.

Segue il diagramma UML.


<<interface>>
Parcheggio
0...n
<<interface>>
Monopattino
+getDisponibile(): boolean
+getOraNoleggiato(): Instant
+setDisponibile(d:boolean): void
+setOraNoleggiato(oraNoleggiato:Instant): void
+getFineNoleggio(): Instant
+setFineNoleggio(instant:Instant): void
+setPersona(p:Pers): void
+getPersona(): Pers
1
<<interface>>
Pers
MonopattinoImpl
-id: String
-disponibile: boolean
-oraNoleggiato: Instant
-fineNoleggio: Instant
-persona: Pers
+COSTO: static final public double
+MonopattinoImpl(id:String)
+getDisponibile(): boolean
+getOraNoleggiato(): Instant
+setDisponibile(d:boolean): void
+setOraNoleggiato(oraNoleggiato:Instant): void
+getFineNoleggio(): Instant
+setFineNoleggio(instant:Instant): void
+setPersona(p:Pers): void
+getPersona(): Pers
+HashCode(): int
+equals(obj:boolean): boolean
+toString(): String

3 Sviluppo

.

.

.

3.1 Testing automatizzato

.

.

.

3.2 Metodologia di lavoro (boh?)

.

.

.

3.3 Note di sviluppo

Funzionalità avanzate di Java utilizzate per la realizzazione delle classi implementate:

<u>Tommaso Petrelli</u>

- Stream: utilizzati per generare collezioni specifiche. Utilizzati nella classe `GUIParcheggio`.
- Labda Expressions: utilizzati per definire i filtri di ricerca degli stream. Utilizzate nella classe `GUIParcheggio`.
- Optional: utilizzati in quei campi o risultati che non sempre garantivano la presenza di un valore. Troviamo l'uso di Optional nella classa `AbstractPosto`.
- Algoritmi: progettato un algoritmo per la ricarica del veicolo elettrico e per calcolare il tempo necessario a completare la ricarica. Questo algoritmo viene implementato come metodo nella classe `ColonninaSupercharger`.

<u>Leonardo Bigelli</u>

1. Stream → fondamentali per la ricerca nelle collezioni. Utilizzati in:
    a. ParcheggioImpl.java;
    b. GUIParcheggio.java.
2. Lambda Expressions → nei filtri di ricerca degli stream. Utilizzati in:
    a. ParcheggioImpl.java;
    b. GUIParcheggio.java.
3. Optional → sfruttati per la presenza o meno di posti, veicoli e monopattini disponibili. Utilizzati in:
    a. ParcheggioImpl.java.
4. Classi generiche → per gestire diversi tipi di valori di ritorno. Utilizzato in:

a. Sensore.java.