Università degli studi di urbino

Dipartimento di Scienze Pure e Applicate
Facoltà di informatica applicata

Progetto di Programmazione e Modellazione ad oggetti

Bigelli Leonardo
Berardi Martin
Dragne Andrei Bogdan
Pecmarkaj Arlind
Petrelli Tommaso

Anno Accademico 2021/2022 - Sessione Invernale

13 febbraio 2022

Indice

Capitolo 1

Analisi

1.1 Cosa ci si pone di fare

Con la presenza in aumento di parcheggi privati nelle città unità al fatto che la mobilità green sta prendendo sempre più piede, come team ci siamo confrontati e ci siamo messi in mente di creare un software che permetta la gestione per un privato dei propri parcheggi diurni con la possibilità di gestire posti elettrici e il noleggio di monopattini. In dettaglio:

- 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 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).

1.2 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.

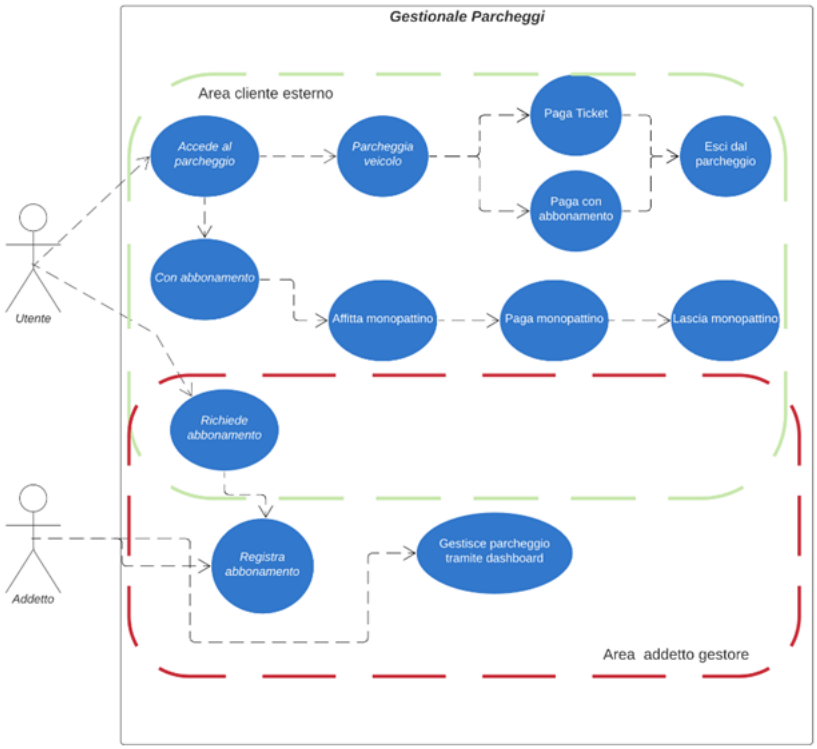

Figura 1.1: Use Case Diagram

1.3 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.

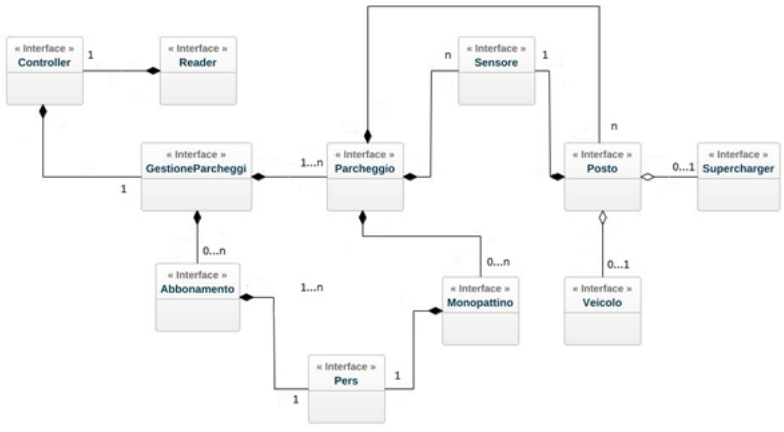

Figura 1.2: UML delle interfacce

Capitolo 2

Design

2.1 Design globale

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.
Per ultimo i parcheggi verranno contenuti nella classe GestioneParcheggi insieme agli abbonamenti. La classe si interfaccia ad ogni Parcheggio e fornirà gli abbonamenti ai parcheggi. Il controller si occupa di prelevare i dati e di fornirli a GestioneParcheggi e a lanciare la parte di interfaccia verso l'utente.

2.2 Architettura

Per la progettazione dell'architettura del software per la gestione di parcheggi, abbiamo scelto si seguire un pattern che facesse leva sulle fondamenta del pattern architetturale chiamato MVC (Model – View - Controller), ossia a partire da questo lo vogliamo adattare alle nostre esigenze di progetto.

Il Model viene elaborato in modo tale da riuscire a dare una descrizione della metodologia di progettazione utilizzata per risolvere il problema che ci siamo posti. A questo, si delega a questa sezione del progetto la gestione degli accessi alle funzionalità e delle modifiche dei dati che l'applicazione mette a disposizione.
La View è l'interfaccia grafica che si presenta all'utente. A questa sezione del pattern utilizzato viene incapsulato il compito di gestire l'interazione tra utente e dati/funzionalità, interfacciandosi ad esso attraverso diverse GUI. La progettazione delle interfacce grafiche e stata fatta seguendo le linee guida del framework per Java utilizzato: Swing.

Dove abbiamo modificato il pattern MVC? Nell'interazione tra la View e il Controller. Infatti, mentre nel pattern MVC il Controller si occupa di interfacciarsi con la View e fornire le funzionalità della parte di Model, nel nostro caso l'approccio cambia.
Il Controller diventa, nel nostro caso, la componente che si occupa di ricevere dalla directory contenente un file di testo tutti i dati sui quali si basa la View, ma anche il Model. Per cui la View potrà utilizzare solo quello che gli fornisce il Controller attraverso una classe di gestione.

La decisione di assumere una variante del pattern MVC viene suggerita dal fatto di voler rendere l'architettura del software più snella e semplice nel suo design complessivo.
Il vantaggio principale che si porta dietro tale scelta è infatti quello di rendere più agevole la lettura del software. Quello che dobbiamo considerare è che per prendere questa scelta assumiamo anche che le dimensioni di questo software non avranno un considerevole aumento in quanto abbiamo già pensato di implementare, oltre alle funzionalità di base, quei servizi aggiuntivi che potremmo trovare nei parcheggi.
C'è anche da considerare che l'intero sistema è progettato per parcheggi di dimensioni discrete, e ciò ha dato maggiore supporto all'approccio di sviluppo utilizzato. Quindi per sopperire la mancanza di scalabilità nel caso di progettazione di un applicativo per parcheggi più primitivo, offriamo già funzionalità integrate come la mobilità green (sensori e parcheggi elettrici) in modo

tale che nella fase di manutenzione futura non si prevedono stravolgimenti totali al programma.

2.3 Design dettagliato

Arlind Pecmarkaj
Classi e intefacce 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 e ColonninaSupercharger.java
- GUIRicarica.java
- Lavorato insieme a Leonardo Bigelli in GUIParcheggio.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).
   Per calcolare la tariffa oraria per ogni tipologia diversa di parcheggio mi è sembrato opportuno definire un tipo enumeratore TassaParcheggio, il quale verrà moltiplicato per un costo prefissato globalmente (1€) per tutti i posti del parcheggio. Per il costo orario viene considerato calcolo adatto a quella che è una simulazione, per cui si tenga conto (anche nei test) che 1 secondo equivale ad 1 ora.

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 della figura 2.1 nella pagina successiva.

4. Tenendo sott'occhio il diagramma UML, osserviamo come il pattern Template Method venga utilizzato a partire dalla classe astratta AbstractPosto.

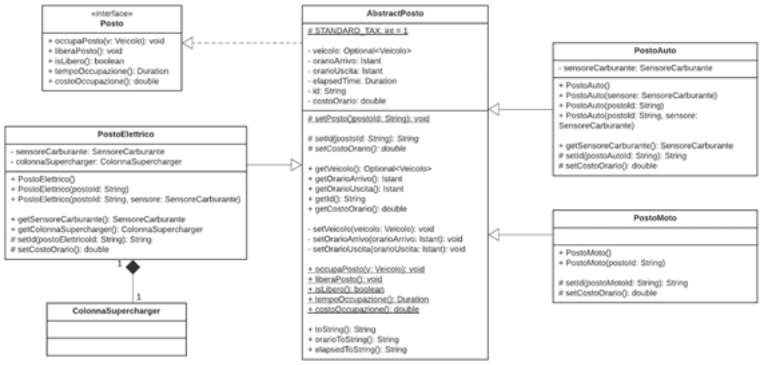

Figura 2.1: UML dei posti

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.

Supercharger per auto elettriche

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 pa-

radigma 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. Tale soluzione prevede un'interazione attiva dell'utente per cui è stata definita un'apposta eccezione per gestire i casi in cui sbadatamente l'utente voglia raggiungere una percentuale di ricarica minore da quella attuale.
La classe in questione viene chiamata IllegalChargerException e garantisce anche che la percentuale specificata non superi il 100

3. Vediamo come possiamo implementare questa soluzione usando uno schema UML:

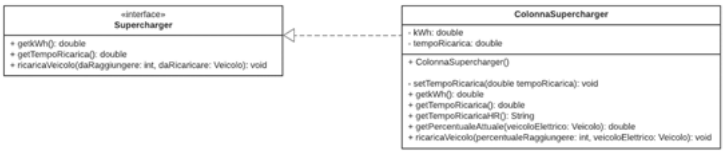

Figura 2.2: UML del supercharger

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 in-

formazioni sempre relative al caricamento dell'auto elettrica, in questo caso, la percentuale della batteria del veicolo prima di ricaricare.

Leonardo Bigelli
Classi e interfacce realizzate:

- Parcheggio e ParcheggioImpl.java
- Sensore.java
- Monopattino e MonopattinoImpl.java
- GUIParcheggio.java

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 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’ ci dentifica che il parcheggio sarà sotterraneo con conseguenza che le auto a metano e a GPL non potranno sostarsi;

2. ‘Qualsiasi altra lettera’ ci 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.
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;

- AutoMetanoNonAmmessaException: l’auto è a metano e non sono ammesse per i parcheggi sotterranei;

- PostiFinitiException: posti non più disponibili.

- TargaNonPresenteException: Il veicolo ha una targa vuota.

- TargheUgualiException: Il veicolo ha una targa che è già presente nel parcheggio.

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 monopattini
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. 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 privato, rappresentante l'implementazione del parcheggio.

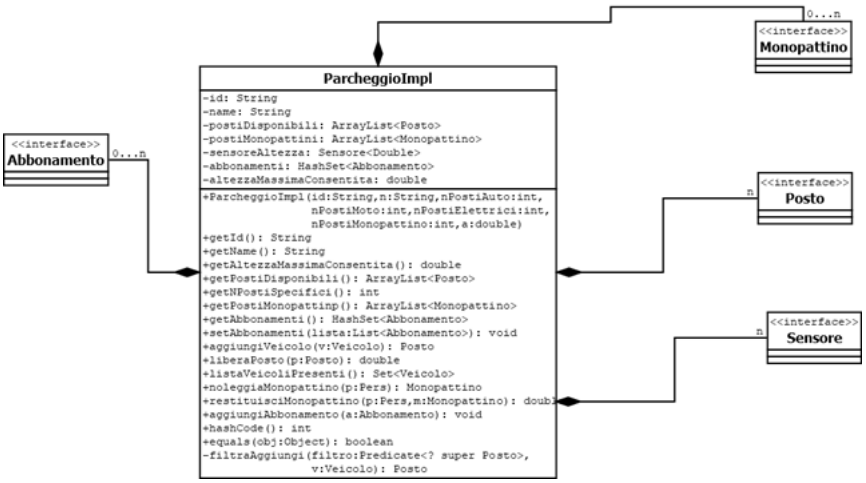

Figura 2.3: UML del parcheggio

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 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, ma non per le classi che lo utilizzano:

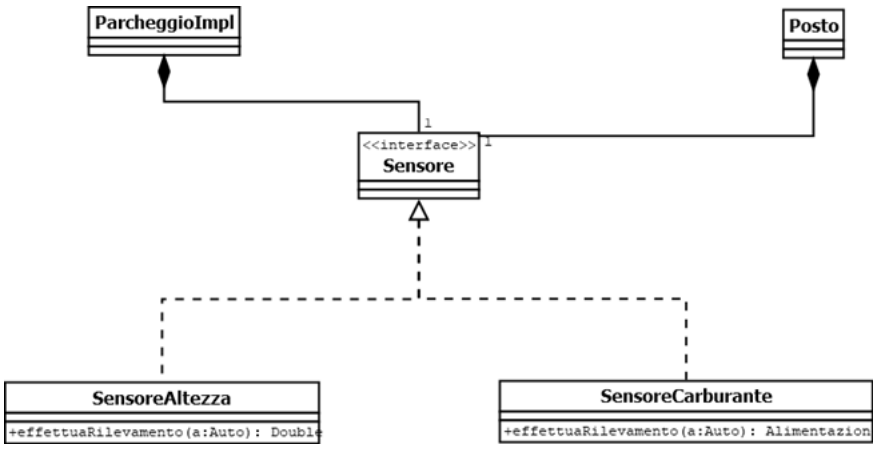

Figura 2.4: UML dei sensori

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.

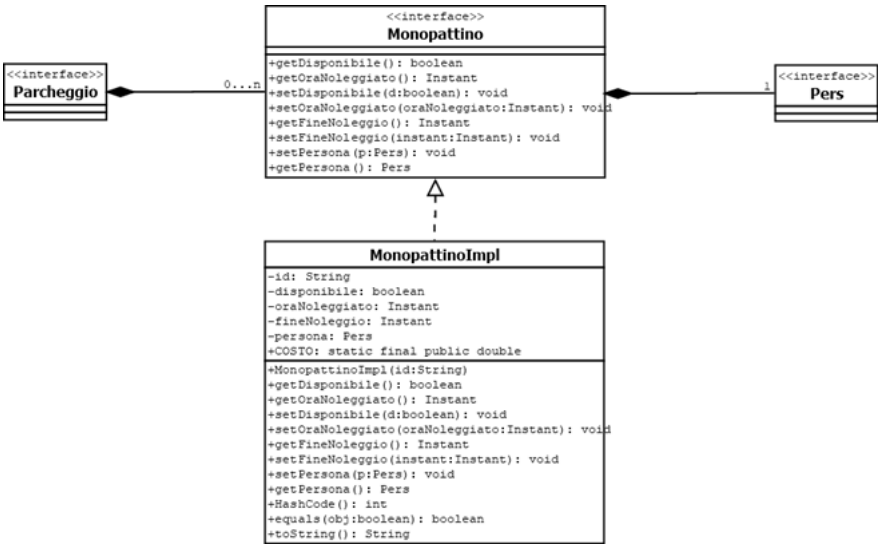

Figura 2.5: UML dei monopattini

Bogdan Andrei Dragne
Classi e interfacce realizzate

- Pers.java e Persona.java
- Abb.java e Abbonamento.java

Per l'implementazione dei servizi sopracitati è stato adottato una forma limitata del Design by Contract (DbC), una metodologia di progettazione molto comune e innovativa.

Vista l'occasione della realizzazione del progetto da un Team di persone, ho optato per questa metodologia in modo da realizzare una verifica di sicurezza più solida, infatti, il DbC è innovativo nel riconoscere che questi contratti sono cruciali per la correttezza del software.
Questo ci porta a diversi benefici:

- Consente una maggior comprensione della software construction.

- Garantisce un approccio sistematico nell'individuare eventuali bugs.
- Assicura che qualunque classe voglia implementare l'interfaccia abbia quei metodi obbligatori imposti dall'interfaccia.
- L'esecuzione del programma è vista come l'interazione tra moduli vincolati dai contratti.

Perchè implementare un'interfaccia per ogni classe?
Sviluppando contratti per una classe, l'interazione delle diverse parti di un programma può essere facilmente mappata e prevista.
La presenza dei contratti, garantisce anche che il programma non tenterà di eseguire in alcun modo, se almeno una classe viola il contratto. Il modello è costituito da una tripla {Q} S {R}:

- Q : predicato detto precondizione.
- S : la nostra classe costituita dal listato del software.
- R : predicato detto postconsizione.

La precondizione stabilisce se è possibile invocare il metodo ovvero è il prerequisito per l'attivazione La postcondizione stabilisce se il metodo restituisce il valore atteso.
Avremo quindi errori del codice: dal chiamante (client) se le precondizioni non sono soddisfatte e da chi ha chiamato (server) se le postcondizioni non sono soddisfatte.
Il contratto dunque rappresenta precondizione e postcondizione, mentre la variante non è altro che il vincolo che deve valere per ogni stato stabile di un oggetto, durante tutto il suo ciclo di vita. Una volta istanziato quel oggetto rimane tale.
In dettaglio avremo:

- *Pers e Persona*: l'interfaccia obbliga a chi la implementa di rispettare il contratto.
  La classe persona deve implementare tutti I metodi dell'interfaccia Pers, altrimenti viola il contratto.
  Di conseguenza, il costruttore della classe Persona è unico, ciò significa che per istanziare un oggetto della classe Persona, bisogna specificare tutti quei parametri. Così facendo, si è sicuri di avere un oggetto della classe Persona con tutti quei requisiti (codice fiscale, nome, cognome ecc...).
  Nella classe Persona è presente il metodo "illegalArgumentPersona", che prevede di controllare i parametri passati al costruttore. Ogni

persona si distingue da un'altra per avere il Codice Fiscale differente, quindi l'hash code e equals sono stati implementati esclusivamente per questo campo.

- *Abb e Abbonamento*: l'interfaccia prevede il medesimo compito della precedente.
  La classe Abbonamento, oltre ai campi che la caratterizzano, si compone di un oggetto di tipo Persona.
  Dato che la specifica consente abbonamenti Premium e Standard, si è deciso di implementare due costruttori.
  Il primo permette di creare un abbonamento specificando anche se l'abbonamento è Premium, l' altro invece consente di istanziare un abbonamento Standard, non specificando se l'abbonamento è Premium, che di default è "false", cioè non è Premium.
  Nella classe Abbonamento è stato introdotto il metodo illegalArgumentAbbonamento che prevede di controllare il riempimento dei campi e che la lunghezza del campo "targa" sia 7.
  Le eccezioni vengono gestite da chi si occupa della gestione degli abbonamenti.
  Ogni abbonamento si distingue da un altro se l'id è unico.
  L'hash code e equals sono stati esclusivamente implementati per questo campo.

Segue l'UML dell'implementazione .

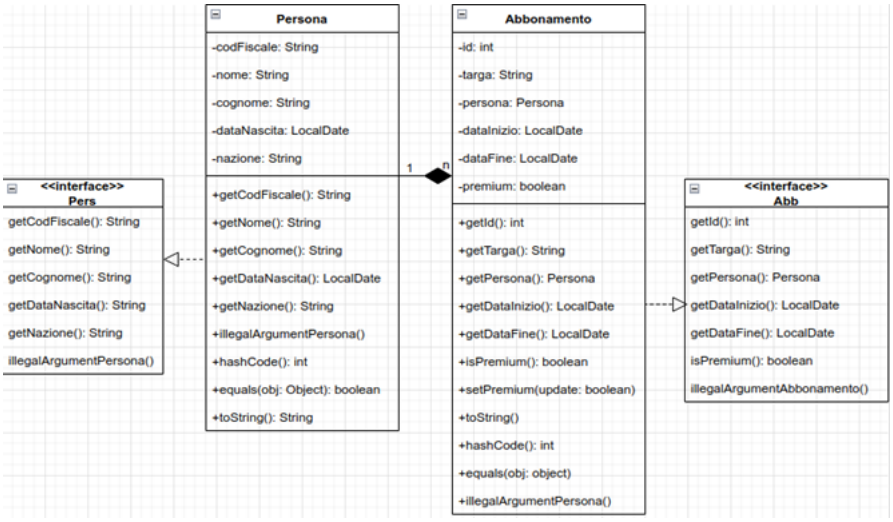

Figura 2.6: UML di Pers e Abb e i loro sottotipi

Martin Berardi
Classi e interfacce realizzate:

- VeicoloInt.java
  - Veicolo.java
  - Auto.java
  - Moto.java
- Tutte le eccezioni contenute nel package parcheggio.exceptions
- GUIHelp.java
- Alcuni accorgimenti aggiunti nelle classi GUIParcheggio.java e GUIGestione.java

1. *Problema da risolvere:* la mia parte di progetto si concentra principalmente sulla modellazione dei Veicoli, che considero le entità principali del nostro scenario, gli oggetti che interagiscono maggiormente con le altre entità del dominio. Inoltre è risultata necessaria la creazione e la gestione delle eccezioni, utilizzate prevalentemente in ParcheggioImpl.java.

2. *Soluzione proposta:* I veicoli vengono modellati attraverso un'apposita classe Veicolo. Anche se non esplicitamente dichiarato nel codice, si tratta di una classe astratta a tutti gli effetti. Si è scelto quindi di utilizzare una classe astratta al posto di un'interfaccia per vari motivi:

   a. È composta da campi che sono in comune tra Auto e Moto, le quali poi otterranno questi campi tramite una semplice estensione (extends).
   b. Un'interfaccia dispone solamente delle firme dei metodi, senza implementazione.
   c. Non ha senso istanziare la classe Veicolo, dato che questa non viene utilizzata propriamente, ma sono Auto e Moto che interagiscono con le altre classi.

   Le classi Auto e Moto sfruttano il concetto di Reuse By Inheritance fornito dal paradigma della programmazione a oggetti. Questo per evitare spreco di codice dato che entrambe le classi fanno parte della categoria Veicolo.
   Le eccezioni presenti nel package parcheggio.exceptions sono state realizzate per far fronte ad alcuni problemi sorti durante la stesura del

codice in ParcheggioImpl.java. Le eccezioni sono tutte sottoclassi di RuntimeException perché sono tutte di tipo unchecked.
L'unica eccezione diversa dalle altre è IllegalChargerException, che è di tipo checked.
La gestione delle eccezioni in ParcheggioImpl è molto intuitiva: viene eseguito un controllo tramite un if in base al contesto in cui ci si trova, e a seconda dell'esito del controllo viene lanciata o meno l'eccezione tramite throw.

3. Siccome la soluzione proposta è molto semplice, non è necessario utilizzare uno specifico design pattern. Piuttosto utilizzo il meccanismo delle interfacce.
   Nell'interfaccia VeicoloInt specifichiamo il contratto che deve rispettare un Veicolo: si è scelto quindi di limitarsi ai setter/getter.
   La classe Veicolo implementerà quest'interfaccia senza aggiungere ulteriori metodi.
   La classe Auto, che eredita dalla classe Veicolo, dispone di un ulteriore metodo set/getAltezza() per controllare che l'altezza effettiva dell'auto non superi la soglia imposta dal parcheggio al momento dell'entrata.

4. L'UML della soluzione:

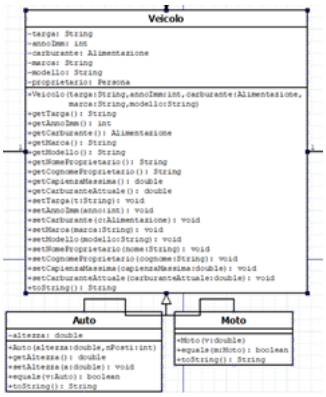

Figura 2.7: UML dei veicoli

Capitolo 3

Sviluppo

3.1 Metodologia di lavoro

La fase iniziale dello sviluppo del progetto ha coinvolto tutti i componenti del team. Ci siamo fin da subito voluti concentrare nell'analisi del problema che volevamo risolvere e che ci ha portato alla definizione del dominio nel quale abbiamo individuato le entità fondamentali a trovare una soluzione.
Il passo successivo è stato quello di fissare le decisioni principali che avrebbero rappresentato i cardini dell'implementazione della soluzione in software rappresentata attraverso un diagramma UML delle interfacce.
Una volta definito il dominio e lo scheletro dell'architettura siamo passati alla suddivisione dei lavori.
Per lavorare nel modo più indipendente possibile abbiamo scelto di utilizzare una metodologia di sviluppo parallela in cui ogni componente del team veniva incaricato di implementare solo alcune entità e logiche di funzionamento del software in base al dominio. Ossia, abbiamo suddiviso il dominio in aree indipendenti per poi assegnare un'area ad ogni componente.
L'indipendenza veniva fornita dal fatto che durante la prima fase di analisi avevamo già definito tutte le interfacce che sicuramente sarebbero state utilizzate a livello globale.
Abbiamo investito diverso tempo sulla suddivisione dei compiti per rendere le aree in cui abbiamo suddiviso il dominio le più indipendenti possibili. In questo modo siamo riusciti a lavorare in modo asincrono senza incorrere in problematiche bloccanti e senza far confliggere il codice ad ogni commit nel branch 'master'.
Il fatto di non dover risolvere le problematiche dovute ai merge ci ha fatto guadagnare del tempo impiegato per la scrittura di codice più pulito o aggiunta di feature.

Il sistema di controllo di versione distribuito (DVCS) che abbiamo scelto per collaborare e tenere traccia delle modifiche del software è stato Git, utilizzato attraverso l'interfaccia grafica fornita dall'applicazione GitHub Desktop.
Durante il corso dello sviluppo ci siamo confrontati spesso per aggiornarci sui progressi fatti tra le rispettive aree di lavoro e per cercare di risolvere determinati problemi di conflitto che si verificavano durante la fase di progettazione delle interfacce grafiche.
Infatti, per quando riguarda la fase di implementazione della View ci siamo trovati a scrivere in più di un componente sullo stesso file per aggiungere alcune sezioni concettualmente indipendenti ma ciò causava comunque conflitti a livello di DVCS.

3.2 Testing

Per il testing automatizzato abbiamo scelto di utilizzare il framework di unit testing JUnit 5, portando avanti il progetto insieme all'idea di sviluppo guidato da test (Test Driven Development).
Il motivo principale che giustifica tale metodologia di sviluppo è che ad ogni nuovo metodo o feature aggiunta ad una classe è stato possibile verificarne subito il suo funzionamento in modo indipendente.
Infatti, nel caso in cui il test fosse risultato corretto si poteva continuare normalmente con lo sviluppo del programma, ma, soprattutto, se il test fosse fallito era facilmente immaginabile che l'errore dovesse quasi sicuramente riguardare il segmento di codice appena aggiunto e questo ci ha fatto risparmiare molto tempo in fase di debug.
Attraverso i metodi forniti dalla suite di JUnit 5 abbiamo scritto i test per verificare passo-passo se i risultati restituiti da un metodo, una stampa su schermo, oppure una nuova feature, fossero esattamente quelli che ci stavamo aspettando. In alcuni casi è stato scelto di testare anche quei casi che causavano il malfunzionamento del programma e quindi scrivere poi un'apposita classe per catturare l'eccezione malevola.
I test ci sono risultati utili anche nelle fasi finali del progetto, ossia quando abbiamo iniziato a sviluppare le GUI del programma. Infatti, utilizzando i test, ci è stato possibile eseguire una GUI per volta così da poter procedere nello sviluppo e per verificarne il corretto funzionamento in modo molto più rapido rispetto a lanciare la GUI principale ad ogni prova e poi andare a vedere tutte le GUI "figlie".
I test prodotti li possiamo andare a vedere in

- Test.java

- TestPosto.java
- TestGrafica.java

3.3 Note di sviluppo

In questa sezione verrano mostrate le funzionalità avanzate di Java utilizzate per la realizzazione delle classi implementate.

Arlind Pecmarkaj

- *Interfaccia parametrizzata*: nel lettore da file per permettere di leggere qualsiasi tipo di dato da file per chi implementa il contratto.
- *Optional*: per evitare ove possibile l'uso della keyword null.
- *ArrayList e Collection in generale*: per tenersi la collezione dei parcheggi e degli abbonamenti.
- *Classi di lettura e scrittura da file di java.io*: per permettere il prelievo e la scrittura dei dati
- *Regex*: nella lettura da file, viene prelevata una linea alla volta che per essere divisa per spazi ha bisogno di una espressione RegEx.

Non è presente nessun algoritmo in quanto le classi progettate da me usano già metodi presenti nelle classi base di Java.

Tommaso Petrelli

- *Stream*: utilizzati per generare collezioni specifiche. Utilizzati nella classe GUIParcheggio.
- *Lambda Expressions*: utilizzati per definire i filtri di ricerca degli stream. Utilizzate nella classe GUIParcheggio.
- *Regex*: utilizzata un'espressione regolare per verificare che l'utente specifichi una percentuale di ricarica dell'auto elettrica valida. Con questa RegEx si vuole controllare quindi se sia stato inserito correttamente un valore numerico intero prima di effettuare la ricarica del veicolo. Troviamo l'uso di RegEx nella classe GUIRicaricaAuto.
- *Optional*: utilizzati in quei campi o risultati che non sempre garantivano la presenza di un valore. Troviamo l'uso di Optional nella classa AbstractPosto.

Per quanto riguarda gli algoritmi si è 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 ricaricaVeicolo() nella classe ColonnaSupercharger.
Prima di spiegare i passi dell'algoritmo si avvisa che si assume che i veicoli elettrici abbiano la quantità di carburante espressa in kWh (Chilowattora), per cui dobbiamo effettuare delle conversioni in termini di percentuale (%) per migliorare la UX (User Experience). Inoltre si verifica la validità dei parametri passati all'algoritmo considerando la classe IllegalChargerException.
Passi dell'algoritmo:

1. Acquisisco valore che indica la percentuale da raggiungere (es: 70%).
2. Calcolo il delta necessario per completare la ricarica conoscendo la percentuale attuale (es: se attuale = 10% allora delta = 60%).
3. Converto il delta in termini di kWh (es: se 100% = 100 kWh allora 60% = 60kWh).
4. Calcolo il tempo di ricarica dividendo il delta in kWh per la potenza del Supercharger (es: valore ottenuto 0.5 allora tempo = 30 minuti).
5. Ricarico il veicolo.

Leonardo Bigelli

- *Stream*: fondamentali per la ricerca nelle collezioni e utilizzati in ParcheggioImpl.java e GUIParcheggio.java.
- *Lambda Expressions*: per i filtri di ricerca degli stream e utilizzati in ParcheggioImpl.java e GUIParcheggio.java.
- *Optional*: sfruttati per la presenza o meno di posti, veicoli e monopattini disponibili e per evitare la keyword null.
- *Classi generiche*: per gestire diversi tipi di valori di ritorno, utilizzate in Sensore.java

L'unico algoritmo degno di nota è utilizzato nel metodo 'filtraAggiungi()' della classe ParcheggioImpl. Il metodo preso in questione permette l'inserimento di un nuovo veicolo parcheggiandolo nel posto più opportuno. Viene utilizzata una strategia di filtraggio (Pattern Strategy), passata come parametro. Nel caso di un veicolo con alimentazione elettrica, l'algoritmo darà priorità nel parcheggiare l'auto in questione in un posto elettrico. Se non

fossero disponibili, in quanto tutti occupati oppure data la mancanza dalla tipologia del parcheggio, il veicolo verrà parcheggiato nel primo posto per le auto disponibile. Questo metodo effettua tutti i vari controlli per verificare la possibilità di parcheggiare o meno un determinato veicolo (es. alimentazione a metano nel caso di un parcheggio sotterraneo, l'altezza del veicolo).

Martin Berardi e Bogdan Andrei Dragne
Per la natura delle classi sviluppate non sono stati usati meccanismi avanzati di Java.