Gestionale Garage

Lisa Innocenti Uccini
Giulia Paone

UNIVERSITÀ
DEGLI STUDI
FIRENZE

Febbraio 2022

Contents

1 Analisi

1.1 Statement

Realizzazione di un sitema software per la gestione di un garage coperto.
Il garage è caratterizzato da un numero finito di posti, suddivisi in base alla tipologia di veicolo che ospitano: autoveicolo, maxi(ad esempio camper o furgoni) o motoveicolo. A loro volta i posti auto sono riservati, in parte, alle categorie: donne incinte, disabili e auto elettriche.
Il garage in questione prevede un piano tariffario orario e giornaliero, con un prezzo diverso per la prima ora. L'importo dipende, oltre che dal tempo di permanenza, anche dalla tipologia di veicolo. La classificazione è la seguente: auto piccola, auto media, auto grande, lusso, auto elettrica, maxi e moto.
Un esempio di piano tariffario per una determinata tipologia di veicolo potrebbe essere:
Auto piccola: *prima ora* €2,00; *frazione successiva* €3,00; *giornaliero* €25,00

Ogni utente ha, inoltre, la possibilità di poter fare un abbonamento mensile o annuale. In tal caso, all'utente in questione, sarà sempre riservato un posto assegnatogli al momento della definizione e del pagamento dell'abbonamento. Il garage sarà caratterizzato da un numero finito di abbonamenti che potranno essere attivati contemporaneamente, ovvero da un numero finito di posteggi che potranno essere riservati.

Si prevedono due tipi di utente che possono utilizzare il sistema, un utente "User" che posteggia il veicolo e un utente "Gestore" che gestisce ed eventualmente usufruisce del garage.
Alla prima tipologia di utente è consentito di:

- Accedere al garage inserendo la targa del veicolo e la sua tipologia. In base ai dati inseriti, il software verifica la disponibilità ed eventualmente attribuisce un posteggio, altrimenti rifiuta la richiesta di ingresso. Inoltre apprende autonomamente data e ora di arrivo.

- Uscire dal garage reinserendo la targa, visualizzare l'importo generato dal software ed effettuare il pagamento. A seguito di questa operazione il software rende nuovamente disponibile il posto in questione.

- Creare un abbonamento mensile o annuale inserendo i dati del veicolo (targa e tipologia) a cui verrà assegnato il posto riservato. In base ai dati inseriti, il software verifica la disponibilità ed eventualmente attribuisce un posteggio, altrimenti rifiuta la richiesta di abbonamento. Inoltre apprende autonomamente la data di inizio dell'abbonamento in questione e calcola la data di fine.

All'utente "Gestore" è consentito effettuare tutte le operazioni dell' "User" oltre che:

- Stabilire e modificare sia il numero totale di posteggi che quello associato ad ogni categoria di veicolo.

- Stabilire e modificare i piani tariffari

Per poter effettuare queste operazioni aggiuntive deve attraversare una fase di login, accedendo al sistema inserendo il proprio nome utente e password che possono essere modificati unicamente dal gestore stesso.

1.2 Vincoli

- Per effettuare il login il gestore del Garage deve inserire username e password che non devono essere vuoti;
- La password deve essere inserita correttamente, altrimenti verrà visualizzato un messaggio di errore;
- L'username e la password dell'utente possono contenere solamente numeri, lettere e alcuni caratteri speciali ($%&!?,_;:°#). Possono contenere al massimo 16 caratteri e devono contenerne minimo 4;
- La targa di un autoveicolo è definita nel seguente modo: 2 lettere, 3 numeri, 2 lettere.
- La targa dei motoveicoli è definita nel seguente modo: 2 lettere, 5 numeri.
- Il numero di posti auto associati ad ogni tipologia di veicolo e i vari piani tariffari possono essere modificati unicamente dal gestore;
- Il numero dei posti per ogni tipologia di veicolo deve essere almeno 1;
- Il valore di ogni tariffa deve essere maggiore di 0;
- Ogni piano tariffario deve essere definito in modo che le tariffe corrispondenti a tempi di permanenza più lunghi (compresi gli abbonamenti) devono essere maggiori delle tariffe corrispondenti a tempi di permanenza più brevi. Ad esempio il valore della tariffa per un ora deve essere minore del valore della tariffa per un giorno.

1.3 Use case diagram

Di seguito è riportato lo use case diagram relativo ai casi d'uso individuati sulla base dello statement. Gli stereotipi vengono utilizzati coi seguenti significati: <<include>> indica un'operazione atomica facente parte di un'azione più generale, <<invoke>> specifica un'azione svolta solo in determinate circostanze, <<extend>> indica un'azione svolta solo in certe circostanze che sia anche parte di un caso più complesso.

Notare che, nel secondo schema (Figura 2), la generalizzazione indicata tra gestore e User rappresenta il fatto che il gestore è anche User, pertanto Gestore ha gli stessi use case di un User e in più ne ha di propri.
A scopo illustrativo è stato deciso di inserire uno use case diagram per ogni attore e di inserire la generalizzazione in modo da non replicare gli stessi casi d'uso della figura precedente (Figura 1).

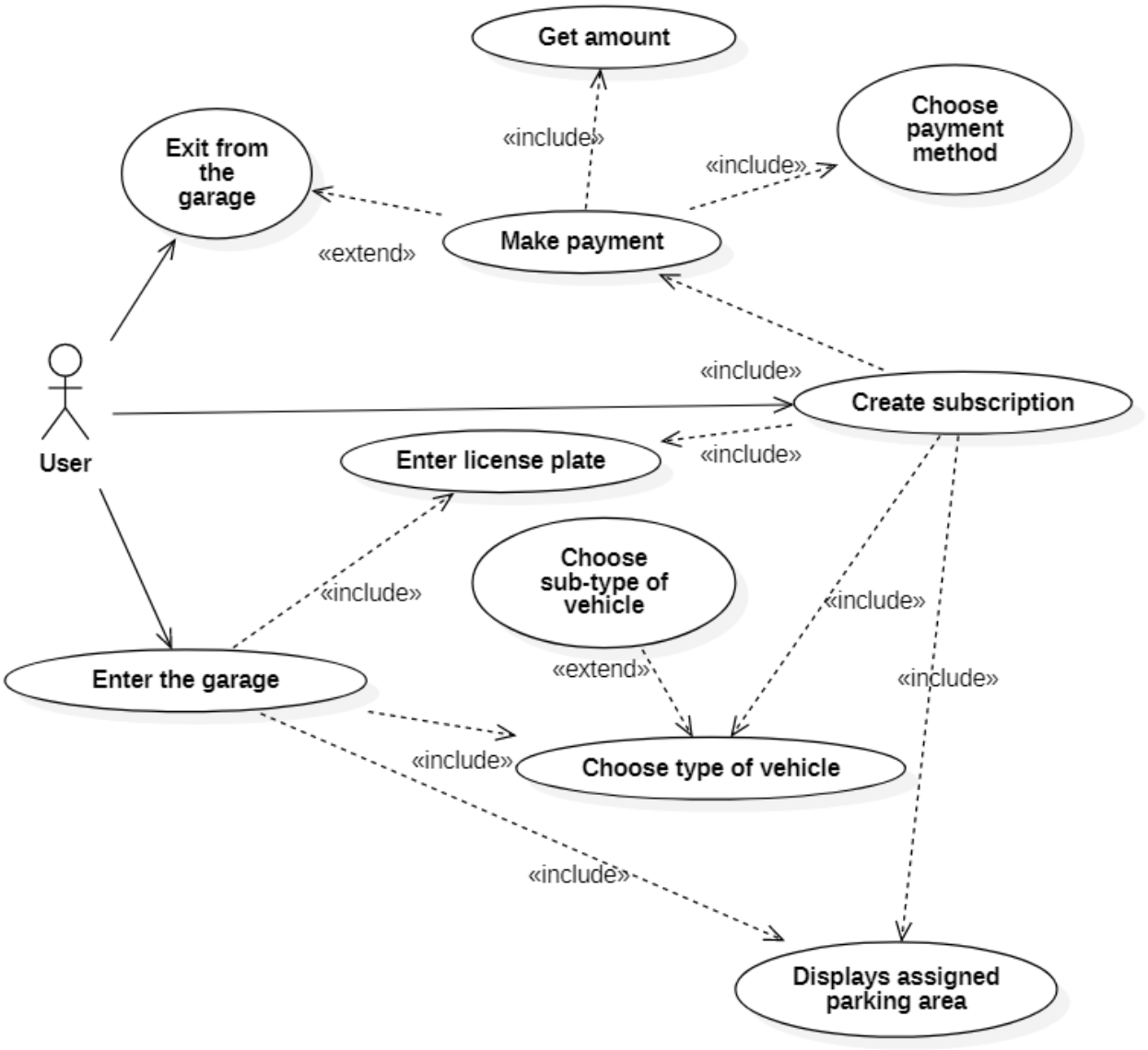


Figure 1: Use case diagram - User

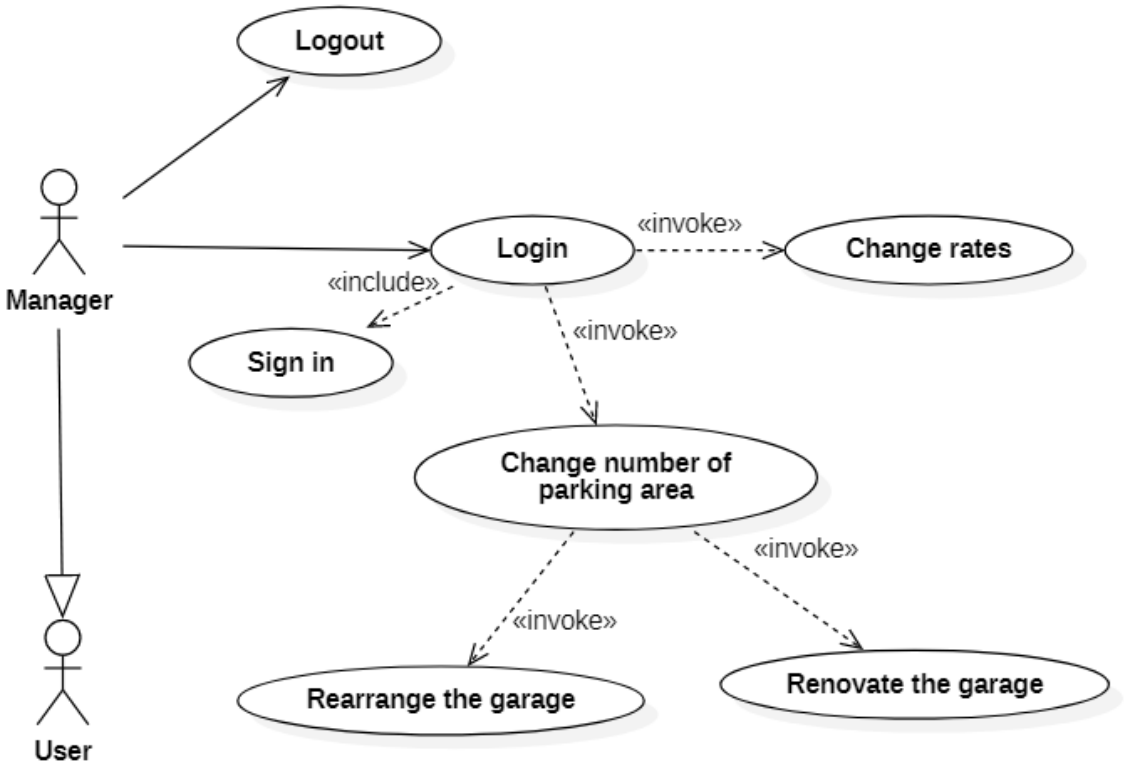


Figure 2: Use case diagram - Manager

1.4 Use case template

Vengono riportati gli use case templates atti a documentare in modo più specifico alcuni casi d'uso individuati nello use case diagram (Figura 1 e 2).
E' stato scelto di riportare i templates solo per gli use case ritenuti più interessanti e rilevanti ai fini del progetto.

Use Case	Signin
Scope	Azioni riservate al gestore
Level	User Goal
Actor	Gestore del garage
Basic Course (Vedi Mock-ups in Figura 4)	1 – L'utente, dalla schermata iniziale, accede alla pagina dedicata al Login; 2 – L'utente inserisce username e password negli appositi campi; 3 – L'utente preme il pulsante di conferma; 4 – Vengono verificati i dati; 5 – Il sistema passa alla schermata riservata da cui potrà eseguire le operazioni di modifica.
Alternative Course	4.1 – Il sistema è offline 4.2 – Le credenziali inserite sono errate 4.2.a – Viene mostrato un messaggio d'errore 4.2.b – L'utente viene riportato alla schermata di inserimento dei dati 4.3 – Almeno uno dei due campi è vuoto 4.3.a – Viene mostrato un messaggio d'errore

Use Case	Modifica piano tariffario
Scope	Azioni riservate al gestore
Level	Function
Actor	Gestore del garage
Basic Course	1 – Il gestore, una volta eseguito il Login, accede alla pagina da cui potrà eseguire l'operazione di modifica; 2 – Il gestore sceglie il tipo di veicolo per il quale modificare il piano tariffario; 3 - Il gestore sceglie il tempo di permanenza a cui cambiare il prezzo tra i seguenti: prima ora, ore successive, giornaliero, abbonamento mensile, abbonamento annuale; 4 - Visualizza una nuova schermata in cui inserisce il nuovo prezzo associato alla tipologia di veicolo e tempo di permanenza scelti in precedenza; 5 - Il gestore preme il pulsante di conferma; 6 - Viene verificato il dato inserito; 7 - Il sistema torna alla schermata riservata al gestore dalla quale può effettuare nuove modifiche;
Alternative Course	6.1 - Il prezzo inserito dal gestore non rispetta i vincoli a esso associati 6.1.a - Viene rifiutata la richiesta di modifica e viene mostrato il messaggio di errore; 6.1.b - Il gestore viene riportato alla schermata di scelta del tipo di veicolo per il quale modificare il piano tariffario;

Use Case	Accedi al parcheggio
Scope	Azioni permesse al cliente
Level	User Goal
Actor	User
Basic Course (Vedi Mock-ups in Figura 5)	1 – L'utente, dalla schermata iniziale, accede alla pagina dedicata all'inserimento delle informazioni riguardanti il proprio veicolo; 2 – L'utente inserisce la targa e preme il pulsante di conferma; 3 – Viene verificato il dato inserito; 4 - Si visualizza una pagina in cui all'utente è chiesto di scegliere se creare un nuovo abbonamento o accedere al garage 5 – L'utente visualizza una nuova schermata nella quale inserisce il tipo di veicolo (maxi, moto, auto); 5.a - L'utente è dotato di un autoveicolo, quindi dovrà specificare la sottocategoria (disabile, donna incinta, auto elettrica, nessuna delle precedenti); 6 - L'utente conferma le scelte fatte; 7 - Viene verificata la disponibilità all'interno del garage per il tipo di veicolo richiesto; 8 - L'applicazione mostra all'utente il numero del posto assegnato;
Alternative Course	3.1 - La targa non rispetta i vincoli (Figura 6) 3.1.a - Viene mostrato il messaggio di errore 3.1.b - L'utente viene riportato alla schemata di inserimento della targa 3.2 - La targa è associata ad un abbonamento; 3.2.a - Viene verificato se l'abbonamento è scaduto o meno; 3.2.b - Se non è scaduto l'applicazione mostra all'utente il posto riservato associato all'abbonamento; 3.2.c - Se è scaduto si procede con l'accesso normale al garage; 7.2 - Se non vi è disponibilità di posti viene visualizzato un messaggio di notifica del problema; (Figura 7) 7.2.a - L'utente viene rimandato alla schermata iniziale;

Use Case	Crea abbonamento
Scope	Azioni permesse al cliente
Level	User Goal
Actor	User
Basic Course	1 – L'utente, dalla schermata iniziale, accede alla pagina dedicata all'inserimento delle informazioni riguardanti il proprio veicolo; 2 – L'utente inserisce la targa e preme il pulsante di conferma; 3 – Viene verificato il dato inserito; 4 - Si visualizza una pagina in cui all'utente è chiesto di scegliere se creare un nuovo abbonamento o accedere al garage 5 – L'utente visualizza una nuova schermata nella quale inserisce il tipo di veicolo (maxi, moto, auto); 5.a - L'utente è dotato di un autoveicolo, quindi dovrà specificare la sottocategoria (disabile, donna incinta, auto elettrica, nessuna delle precedenti); 6 - L'utente conferma le scelte fatte; 7 - L'utente sceglie la tipologia di abbonamento che desidera: mensile o annuale; 8 - Viene verificata la disponibilità all'interno del garage per il tipo di veicolo richiesto; 9- L'applicazione mostra all'utente il numero del posto assegnato;
Alternative Course	3.1 - La targa non rispetta i vincoli (Figura 6) 3.1.a - Viene mostrato il messaggio di errore 3.1.b - L'utente viene riportato alla schemata di inserimento della targa 3.2 - La targa è associata ad un abbonamento; 3.2.a - Viene verificato se l'abbonamento è scaduto o meno; 3.2.b - Se non è scaduto l'applicazione mostra all'utente il posto riservato associato all'abbonamento; 8.1 - Se gli abbonamenti disponibili per la tipologia di veicolo richiesto sono esauriti viene visualizzato un messaggio di errore; 8.1.a - L'utente viene rimandato alla schermata iniziale;

1.5 Mock-ups

Per alcuni use case analizzati nella sezione precedente, sono state create delle rappresentazioni grafiche di come si immagina che il sistema operi una volta realizzato completamente.

Figure 3: Schermata iniziale

Figure 4: SignIn - Accesso area riservata al manager

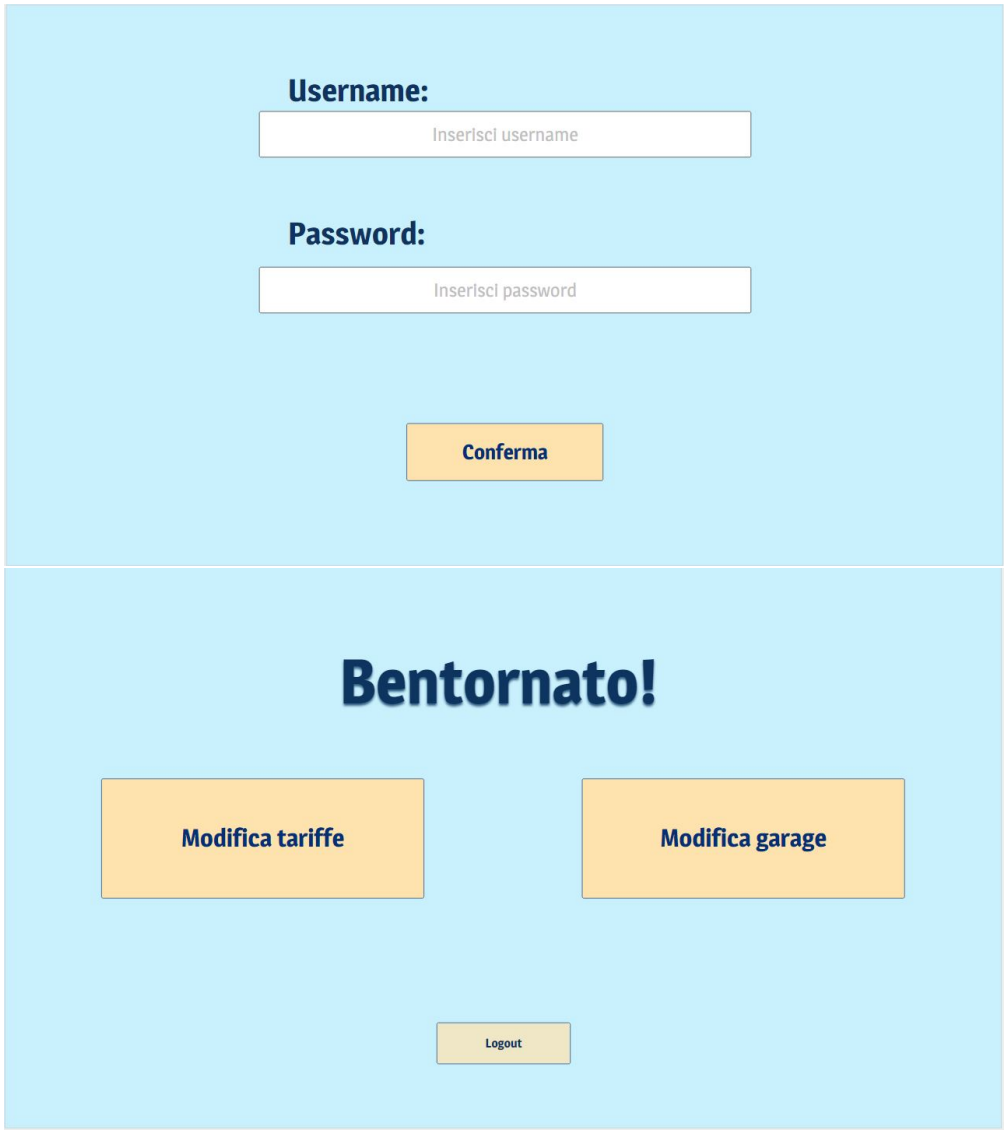
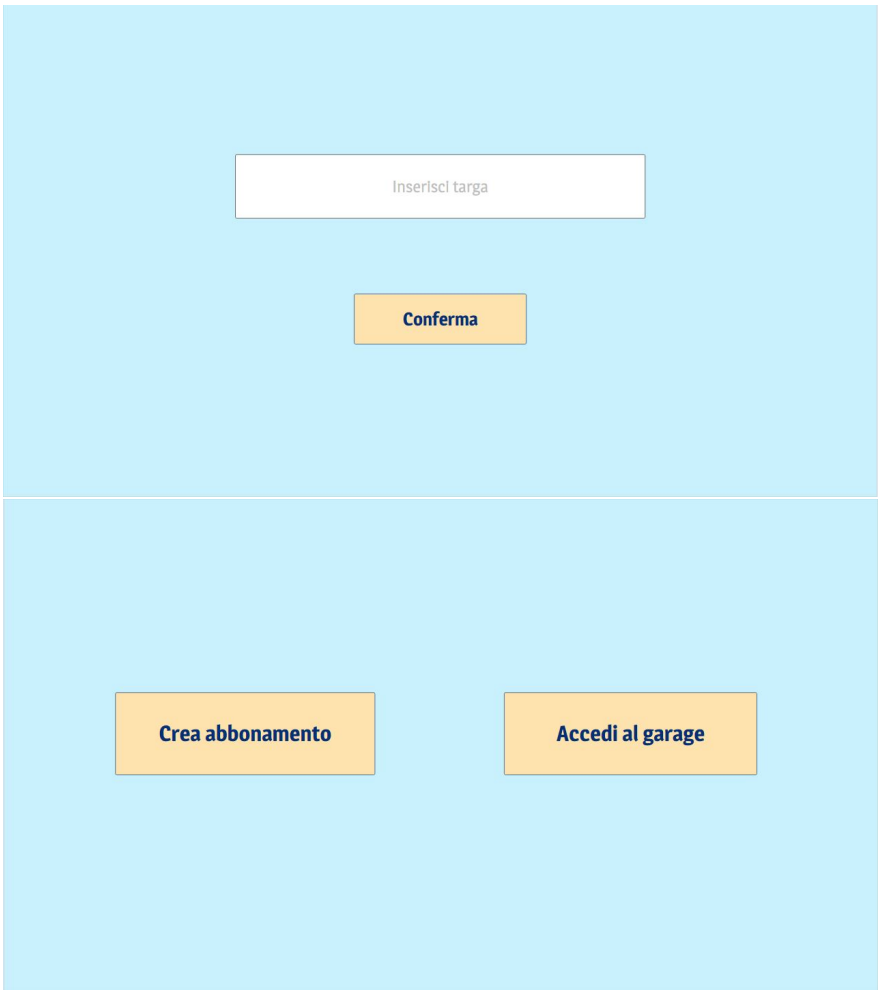


Figure 5: Accesso al garage dello User



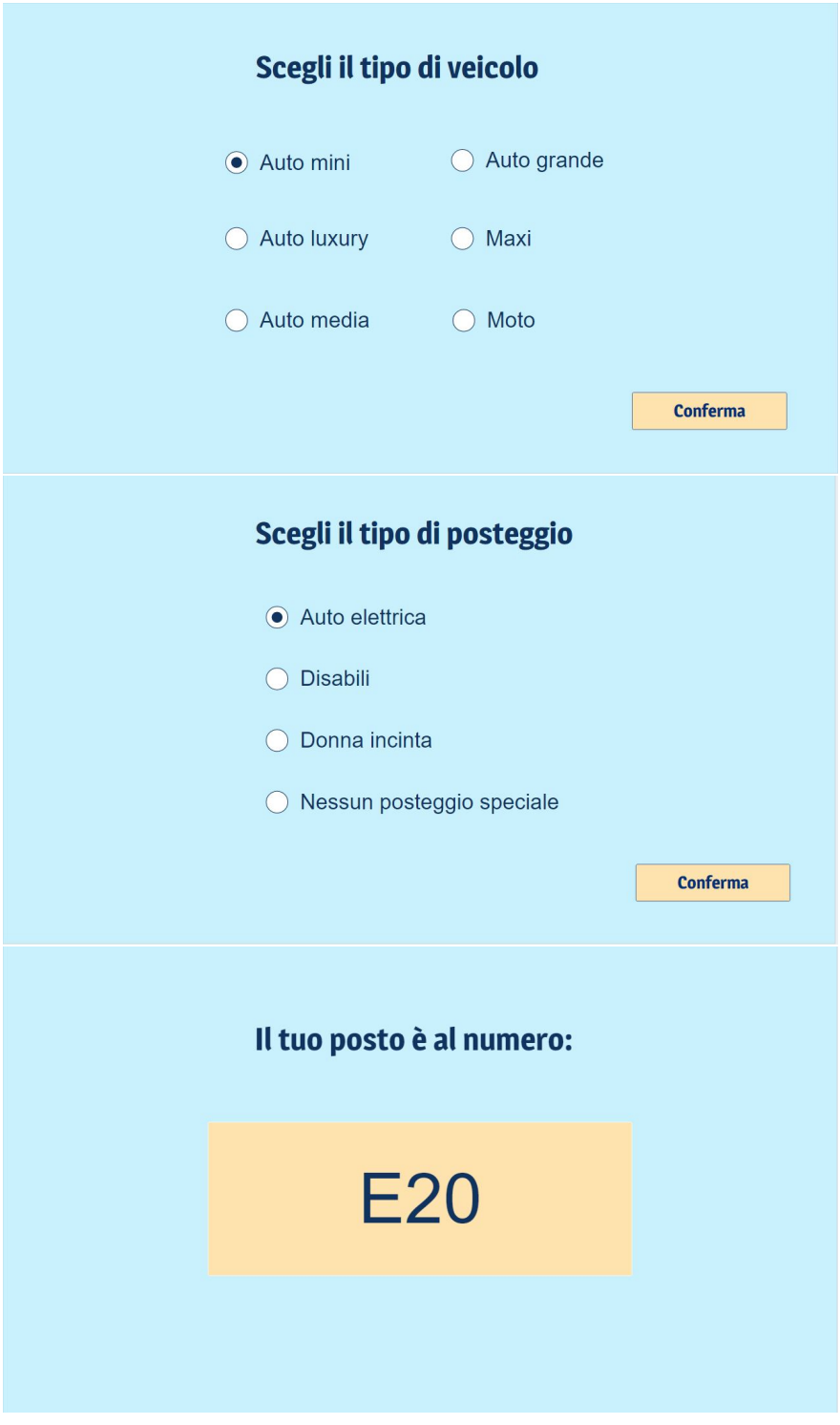

Scegli il tipo di veicolo
Auto mini
Auto grande
Auto luxury
Maxi
Auto media
Moto
Conferma
Scegli il tipo di posteggio
Auto elettrica
Disabili
Donna incinta
Nessun posteggio speciale
Conferma
Il tuo posto è al numero:
E20

Figure 6: Errore - targa inserita non correttamente

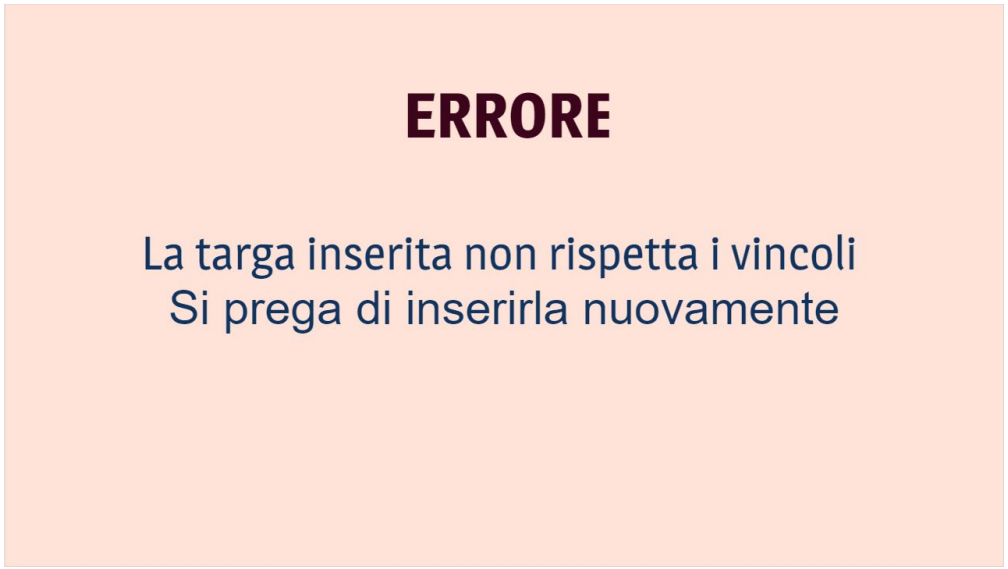
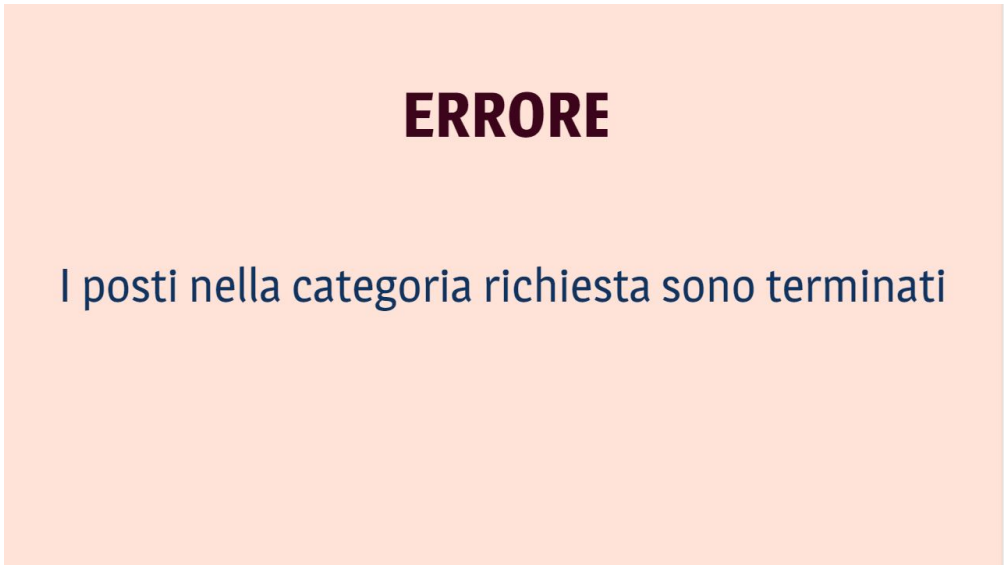


Figure 7: Errore - garage pieno



2 Progettazione

2.1 Page Navigation Diagram

Di seguito è riportato come si immagina avvenga la navigazione tra le diverse pagine dell'interfaccia grafica, che poi però non sarà implementata realmente in questo progetto.

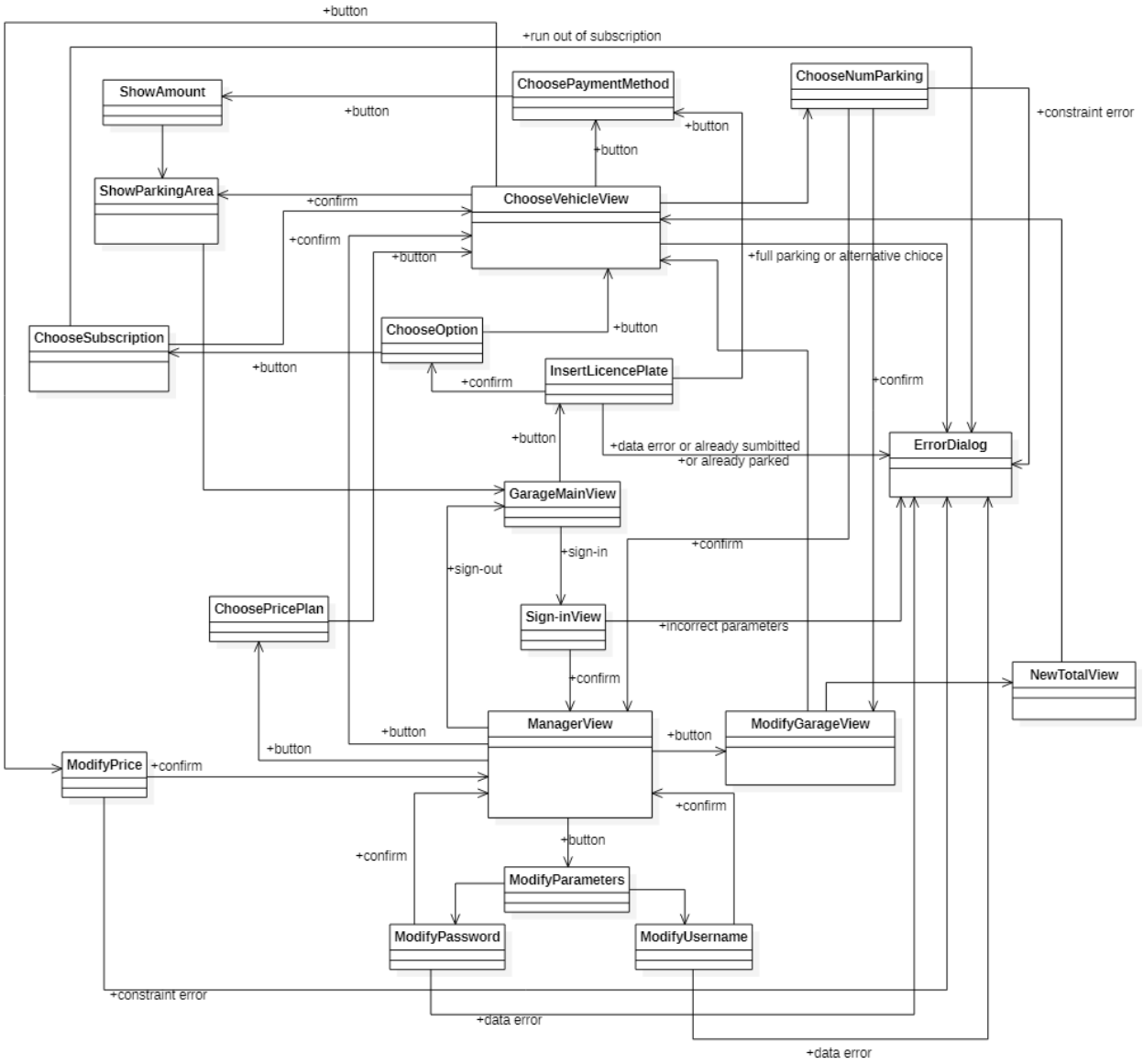


Figure 8: Page navigation diagram

2.2 Class Diagram

Il class diagram riportato di seguito è al livello di implementation, ciò significa che un oggetto nel diagramma corrisponde ad un oggetto nel linguaggio OO.

Anche se, per non appesantirlo troppo, nel diagramma non sono state inserite tutte le classi e i metodi implementati nel programma (ad es. i metodi di get e set) ma solo quelli più rilevanti. Per realizzare il programma è stato usato il design pattern MVC pertanto le classi sono state suddivise in 3 package:

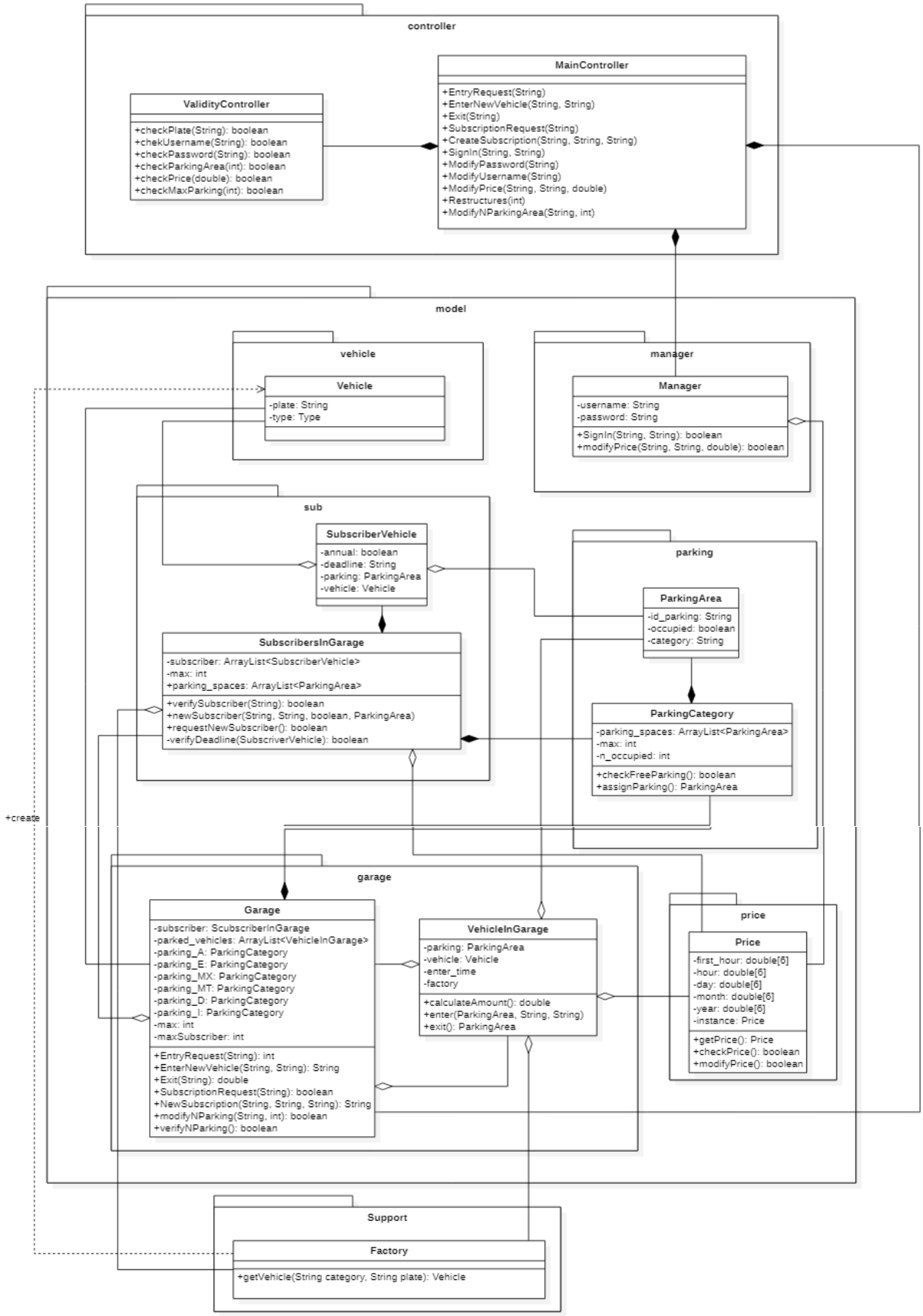
- model: contiene le classi che descrivono il domain model;
- view: contiene le classi che fanno da interfaccia verso l'utente;
- controller: contiene le classi che agiscono come controller, cioè che gestiscono la comunicazione tra Model e View e li aggiornano in base agli input;

E' inoltre presente un ulteriore package Support che contiene le classi che fanno da supporto per l'applicazione, come ad esempio quelle utilizzate per implementare i vari design pattern. Abbiamo deciso di organizzare il modello in packages in modo finalizzato ad arrivare ad una architettura a microservizi. Per fare ciò il package Model è stato a sua volta suddiviso in 6 package, ognuno dei quali gestisce un servizio:

- vehicle: contiene la classe che rappresenta le caratteristiche di un veicolo;
- parking: contiene una classe di descrizione del singolo posteggio e una classe dedicata alla gestione di un gruppo di parcheggi associati ad una stessa categoria;
- manager: permette una rappresentazione separata del gestore del parcheggio;
- subscription: contiene una classe di descrizione del singolo abbonamento associato ad un veicolo e una classe dedicata alla gestione dell'insieme di abbonamenti;
- garage: contiene una classe di descrizione del singolo veicolo in garage e la classe che rappresenta l'intero garage in questione;
- price: contiene la classe che tiene traccia delle tariffe associate ad ogni categoria di veicolo

Nel diagramma vengono mostrati solo i package di Model e Controller in quanto le View sono state già mostrate e i design pattern utilizzati verranno discussi in seguito.

Figure 9: Class Diagram - package Controller e Model



3 Implementazione

3.1 Classi

Partendo dal class diagram (Figure 8 e 9) e dallo use case diagram (Figure 1 e 2) sono state individuati i package e le classi da implementare in Java per realizzare il programma, ognuno con le proprie responsabilità.
Viene fornita una breve descrizione delle classi più significative e dei principali metodi implementati.

3.1.1 View

Le classi contenute nel package view rappresentano le interfacce del sistema. Esse contengono dei metodi che simulano l'interazione con l'utente, non essendo stata implementata una GUI. Per ogni azione eseguita la view comunica col Controller secondo le modalità del design pattern Observer. Ogni view concreta estende la classe astratta View e l'interfaccia Observable.

La classe Observable contiene la definizione dei metodi che instaurano e chiudono la comunicazione con il Controller e del metodo di notifica notifyActionPerformed() che, una volta implementato, notifica l'azione eseguita in modalità pull. Secondo questa modalità i dati non vengono direttamente forniti al controller, ma ci si limita a far sapere a quest'ultimo che è stata eseguita un'azione e si passa come parametro una stringa che rappresenta l'azione eseguita. Sarà il controller che, in base al parametro ricevuto, si occuperà di recuperare i dati e le funzioni di cui a bisogno.

La classe View contiene i metodi displayText() e displayButton(). Il primo viene utilizzato per mostrare messaggi, cioè simula la comparsa di una nuova vista, il secondo simula la presenza di un bottone.

```
@Override
public void displayText() {
    System.out.println("Benvenuti nel garage di GiuLi");
}
```

Figure 10: displayText della GarageMainView

Altri componenti GUI vengono implementati con metodi appositi. Ad esempio i bottoni di conferma di parametri inseriti i quali sono gestiti introducendo dei metodi, che notificano al controller che il bottone è stato premuto, caratterizzati da alcuni parametri che vengono passati. I parametri rappresentano i valori che dovrebbero essere inseriti dall'utente e prelevati dal controller.

```
public void insertPlateConfirm(String plate){
    System.out.println(plate);
    this.plate = plate;
    this.notifyActionPerformed("plate");
}
```

Figure 11: Metodo che simula la pressione del bottone di conferma della targa nella InsertLicencePlateView

3.1.2 Model

Le classi contenute nel package Model contengono la logica e le caratteristiche dei protagonisti del software. Queste sono racchiuse in diversi sotto-package al fine di costruire un sistema orientato a un'architettura a microservizi, in cui le diverse parti comunicano ma rimangono separate, sono indipendendi.

- **Vehicle**: rappresenta le caratteristiche proprie di un singolo veicolo, quali: la targa e la tipologia. Quest'ultima proprietà viene gestita mediante la definizione di un enum, in quanto la tipologia di un veicolo può essere: luxury car (associato all'acronimo LC), maxi car (associato all'acronimo MXC), midi car (associato all'acronimo MDC), mini car (associato all'acronimo MNC), maxi (associato all'acronimo MX), moto (associato all'acronimo MT).

- **VehicleInGarage**: fornisce la descrizione di un veicolo all'interno del garage. Oltre a contenere al suo interno le caratteristiche del veicolo stesso, contiene la descrizione del posteggio in cui è parcheggiato e la data e l'ora di arrivo. Le funzioni presenti all'interno della classe permettono ad un veicolo di entrare, di uscire e permettono al garage di calcolare la tariffa (in base al tempo di permanenza) che l'utente proprietario del veicolo dovrà pagare.

- **ParkingArea**: descrive le informazioni caratterizzanti di un singolo parcheggio, ovvero l'id, se è occupato o meno e la categoria di veicoli che possono occupare il posteggio in questione.

- **ParkingCategory**: gestisce un insieme di posteggi (ParkingArea) associati ad una stessa categoria. Le funzioni contenute al suo interno permettono di verificare se all'interno del garage c'è posto per un determinato tipo di veicolo e permenttono di assegnare eventualmente un posteggio.

- **SuscriberVehicle**: descrive le caratteristiche associate ad un unico abbonameno, ovvero se è annuale o mensile, data di scadenza, posteggio e veicolo al quale è associato.

- **SubscribersGarage**: gestisce l'insieme di veicoli abbonati (subscriberVehicle). Le funzioni contenute al suo interno permettono di verificare se è possibile creare un nuovo abbonamento, di crearlo se ciò è possibile e di eliminare abbonamenti ormai scaduti.

- **Garage**: si occupa della gestione generale dell'interno garange. E' definito da un insieme di ParkingArea, ognuno dei quali gestisce un insieme di parcheggi associati ad una determinata tipologia di veicoli. Inoltre tiene in memoria l'insieme di veicoli dentro al garage e di abbonamenti. Le funzioni contenute nella classe Garage permettono di eseguire la maggior parte dei servizi offerti dal garage stesso: entrata, uscita, creazione di un abbonamento, modifica del numero di parcheggi associati ad una tipologia di veicolo, modifica del numero totale di parcheggi.

- **Manager**: rappresenta il gestore del garage. Contiene gli attributi username e password, ovvero le credenziali di accesso del manager. Quest'ultime sono modificabili dal gestore stesso una volta eseguito l'accesso.

- **Price**: gestisce le tariffe del parcheggio, associate sia al tempo di permanenza che alla tipologia di veicolo. Inoltre contiene un metodo per modificare le tariffe ed uno per validare la correttezza di un'eventuale modifica.

3.1.3 Controller

Il package Controller contiene le classi MainController e ValidityController che si occupano del controllo del sistema.

La classe MainController è il controller principale. Implementa la classe Observer contenente il metodo retrieveData(), tale metodo è il cuore della logica dell'applicazione.
Esso riceve come parametri una vista ed un'azione, tramite il metodo notifyActionPerformed() che lo richiama. In base alla vista in cui ci si trova e all'azione eseguita, il metodo comunica sia col view che col model per svolgere le operazioni conseguenti.
Oltre al metodo appena descritto contiene altri metodi sempre utili per la gestione

La classe ValidyController contiene dei metodi utili alla validazione dei dati. I suoi metodi vengono utilizzati nel MainController per ramificare le risposte alle azioni, in base alla correttezza dei dati inviati.

```
@Override
public void retriveData(View view, String action) {
    if (view.getClass() == GarageMainView.getClass()) {
        if (action.equals("signin")) {
            currentView = new SignInView();
            ((SignInView) currentView).addController(this);
        } else if (action.equals("enter") || action.equals("exit")) {
            if (action.equals("exit")) {
                exit = true;
            }
            currentView = new InsertLicencePlateView();
            ((InsertLicencePlateView) currentView).addController(this);
        }
```

Figure 12: Parte del metodo retriveData() che gestisce un action proveniente dalla vista GarageMainView

3.1.4 Support

Quest'ultimo package contiene le classi ausiliare utilizzate per l'implementazione di alcuni design pattern. In particolare, comprende le classi Observer, Observable e VehicleFactory.

La classe VehicleFactory realizza il FactoryPattern, che verrrà descritto in seguito. Contiene il metodo getVehicle() per la creazione di veicoli concreti.

3.2 Design Patterns

In questa sezione vengono analizzati i design pattern utilizzati per realizzare il sistema, l'obiettivo non è dare una descrizione generale di come funzionano i vari design pattern bensì esporre il motivo per il quale è stato scelto di implementarli nel progetto.

3.2.1 MVC (Model-View-Controller)

Il design pattern MVC è uno dei pattern architetturali più diffusi e nella nostra applicazione la divisione in package rende evidente la sua funzione (Figure 8 e 9).
Viene utilizzato per separare le responsabilità dell'applicazione in 3 parti. Il Model è costituito da quelle classi che rappresentano i soggetti dell'applicazione, gli elementi contententi i dati da memorizzare ed eventualmente modificare. Le viste (View) costituiscono l'interfaccia con l'utente, in genere grafica, nel nostro caso simulata. Infine, il Controller gestisce la logica di comunicazione tra le parti, permettendo così anche una maggiore tutela dei dati. Questo permette di mantenere separato ciò che viene mostrato all'utente (l'interfaccia) dal resto della logica
Nello specifico, all'interno della nostra applicazione l'utente si ritrova ad interfacciarsi con i servizi offerti dal garage senza accedere direttemente, ad esempio, alle tariffe o alla distribuzione dei posteggi. E' il controller che in base alle richieste dell'utente mobilita le classi del model per eseguire determinate operazioni.

3.2.2 Observer

Il design pattern comportamentale observer viene qui implementato per permettere la comunicazione tra view e model. Come già introdotto in precedenza, le classi coinvolte nel nostro codice sono MainController, il quale si comporta come un Observer, ed ogni view concreta, quale ad esempio GarageMainView, che si comporta come un oggetto Observable. I due metodi chiave sono notifyActionPerformed(), contenuto negli oggetti Observable, e retrieveData(), contenuto in MainController, sono alla base della logica di comunicazione. Il primo notifica l'azione eseguita, chiamando il secondo nel suo corpo e passandogli come parametri la vista corrente e l'azione appena eseguita.
E' stato scelto di realizzare la modalità pull poiché in questo modo il MainController può scegliere da solo i dati che gli interessano a seconda dell'azione da effettuare, anche se ciò comporta un maggiore accoppiamento.

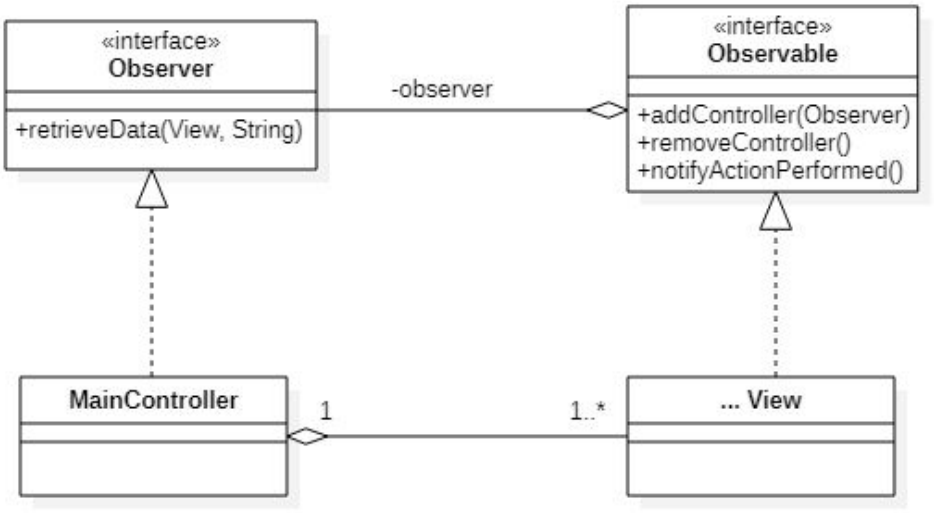


Figure 13: Observer pattern

3.2.3 Factory

Il design pattern factory è di tipo creazionale. In generale, la sua funzione è quella di istanziare oggetti diversi in base alle richieste correnti senza esporne la logica di creazione e fornisce un modo per riferirsi ad essi utilizzando un'interfaccia comune.
Nella nostra applicazione viene implementato nella classe concreteCreate che contiene il metodo getVehicle(). Ogni volta che un nuovo veicolo tenta di accedere al garage, la classe principale Garage richiama il metodo getVehicle passandogli come parametro la tipologia di veicolo. In questo modo il metodo può istanziare il veicolo in questione e restituirlo come parametro di ritorno.

3.2.4 Singleton

Il singleton è un design pattern di tipo creazionale che ha lo scopo di garantire che di una determinata classe venga creata una e una sola istanza. Nella nostra applicazione viene utilizzato per implementare la classe Price. Viene creata un'istanza di Price in tre classi diverse: Manager, VehicleInGarage e SubscribersInGarage. L'uso del Singleton permette di avere un solo oggetto di quel tipo, grazie alla presenta di un attributo statico privato 'istance' e un costruttore statico. Istance contiene il riferimento all'oggetto creato ed ogni volta che viene istanziata la classe il costruttore ritorna tale riferimento. Questo permette la condivizione degli attributi della classe per ogni istanza; in particolare, un'eventuale modifica dei prezzi è disponibile in qualunque parte del codice da cui vi si accede.

3.3 Sintesi del codice

A scopo informativo vengono riportati di seguito alcuni dati relativi al codice:

- N° Packages: 10
- N° Classi: 42
- N° Test effettuati: 20

4 Testing

In questa sezione vengono riportati alcuni test, svolti con la piattaforma di testing JUnit. Si è scelto di suddividere i test in due classi: MainControllerTest e ModelTest.
I test present in MainControllerTest sono test funzionali, orientati alla verifica dei meccanismi di funzionamento dell'MVC nel nostro progetto. Infatti, non avendo implementato interfaccia grafica, abbiamo simulato le azioni e lo scatenarsi di eventi in una determinata fase d'esecuzione. In particolare sono stati testati i casi d'uso descritti negli use case diagram, come ad esempio l'accesso al garage. Di seguito vengono mostrate alcune sezioni di codice.

Preliminarmente ai test veri e propri, contrassegnati dalla direttiva @Test, sono stati utilizzati tre metodi per predisporre l'ambiente. I metodi contrassegnati dalle diretive @BeforeClass ed @AfterClass vengono eseguiti rispettivamente prima e dopo l'intera classe di test. Il primo è un metodo di set up in cui sono istanziati gli oggetti necessari, il secondo è un metodo di tear down. Il metodo contrassegnato da @Before viene eseguito prima di ogni test. Lo abbiamo utilizzato per effettuare un refactory sulla classe principale garage, istanziata precedentemente, così da azzerare le modifiche fatte fino a quel momento dai test precedenti, che altrimenti andrebbero ad inficiare l'esito dei successivi.

```
public class MainControllerTest {

    static Garage garage;
    static Manager manager;
    static MainController controller;

    @BeforeClass
    public static void setUpClass() {
        garage = new Garage(18, 2, 3, 3, 3, 3, 3, 3);
        manager = new Manager("GiuLi", "GiuLi");
    }

    @AfterClass
    public static void TearDownClass() {
        garage = null;
        manager = null;
        controller = null;
    }

    @Before
    public void setUp() {
        garage.refactorGarage();
        GarageMainView gmv = new GarageMainView();
        controller = new MainController(garage, manager, gmv);
        gmv.addController(controller);

    }
```

Figure 14: Test: metodi preliminari ai test

```
@Test
public void testEnterGarageSuccessfull() {
    System.out.println("TEST ENTER GARAGE SUCCESSFULLY");
    String plate = "GI212LU";
    ((GarageMainView) controller.getCurrentView()).enterButtonClicked();
    ((InsertLicencePlateView) controller.getCurrentView()).insertPlateConfirm(plate);
    ((ChooseOptionView) controller.getCurrentView()).enterGarageClicked();
    ((ChooseVehicleView) controller.getCurrentView()).insertCategoryConfirm("auto mini");
    ((ChooseSpecialVehicleView) controller.getCurrentView()).insertSubcategoryConfirm("");

    GarageMainView viewTest = new GarageMainView();
    assertEquals(viewTest.getClass(), controller.getCurrentView().getClass());

    assertTrue(garage.verifyVehicleInGarage(plate));
    System.out.println();
}
```

Figure 15: Test: test ingresso di un veicolo nel garage

I test presenti in ModelTest sono test strutturali di un'unità, orientati a verificare il corretto funzionamento di alcuni metodi. Sono state testate alcune delle funzionalità più rilevanti, come la modifica dell'assegnazione dei posteggi.

```
@Test
public void testModifyMaxParking() {
    int max_old = garage.getMax();

    System.out.println("TEST MODIFY MAX PARKING UNSUCCESSFUL");
    String plate1 = "GP212LU";
    String category1 = "MX";
    String plate2 = "GP212LI";
    String category2 = "MX";
    String parking_area1 = garage.EnterNewVehicle(plate1, category1, "D");
    String parking_area2 = garage.EnterNewVehicle(plate2, category2, "I");

    assertFalse(garage.setMax(2));
    assertFalse(garage.setMax(6));
    assertEquals(max_old, garage.getMax());
}
```

Figure 16: Test: modifica numero massimo di parcheggi senza successo