Progetto P.I.S.S.I.R

Gruppo 06

Galliera - Marino - Ternullo - Vecchio

August 28, 2024

Contents

List of Figures

Listings

Chapter 1

Specifica

1.1 Dominio Applicativo

Il dominio applicativo va a fornire una panoramica del sistema di gestione di un parcheggio smart dotato di MWbot. Le classi principali sono:

- Utente: rappresenta una persona, cliente o amministratore, che interagisce col sistema.
- Prenotazione: definisce l'occupazione di un determinato posto da parte di un utente per un determinato lasso di tempo. Sono le prenotazioni non ancora concluse.
- Storico: registro delle prenotazioni passate e concluse, sono già state pagate.
- Ricarica: rappresenta una ricarica effettuata da un MWBot durante una prenotazione. Percentuale richiesta indica la percentuale effettiva di ricarica a cui l'auto deve essere portata.
- Costi: rappresenta i costi di parcheggio, ricarica, costo premium e penale.

Tutti i clienti possono arrivare al parcheggio e occupare il posto assegnatoli dal sistema. Solo i clienti Premium possono prenotarne uno preventivamente.
Ad ogni prenotazione verrà associato un pagamento che dipenderà dai costi, dalla durata della sosta e dall'eventuale ricarica, la quale può essere richiesta dal cliente al momento del parcheggio.
L'amministratore gestisce le prenotazioni e può, in qualsiasi momento, modificare i costi del parcheggio.
Sono state create,inoltre, due nuove classi:

- Carta: oggetto che rappresenta la carta associata al cliente (ad ogni cliente può essere associata una sola carta alla volta).
- TimeStamp: oggetto che rappresenta un istante nel tempo.

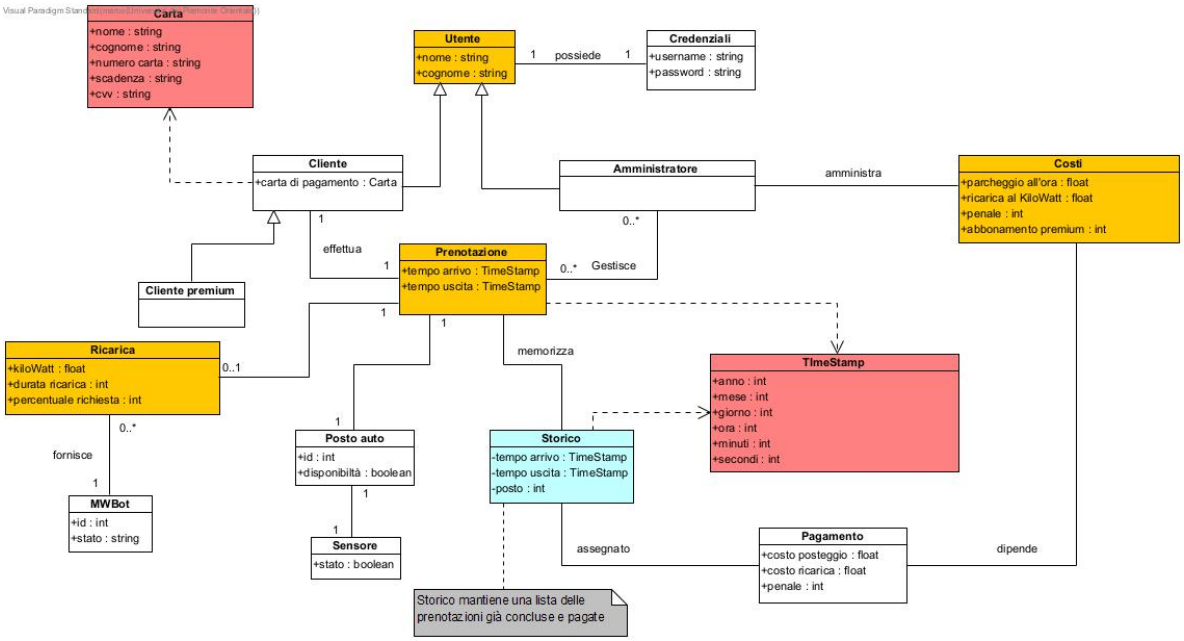

Figure 1.1: Diagramma del dominio applicativo

1.2 Casi d'uso

Diagramma dei casi d'uso

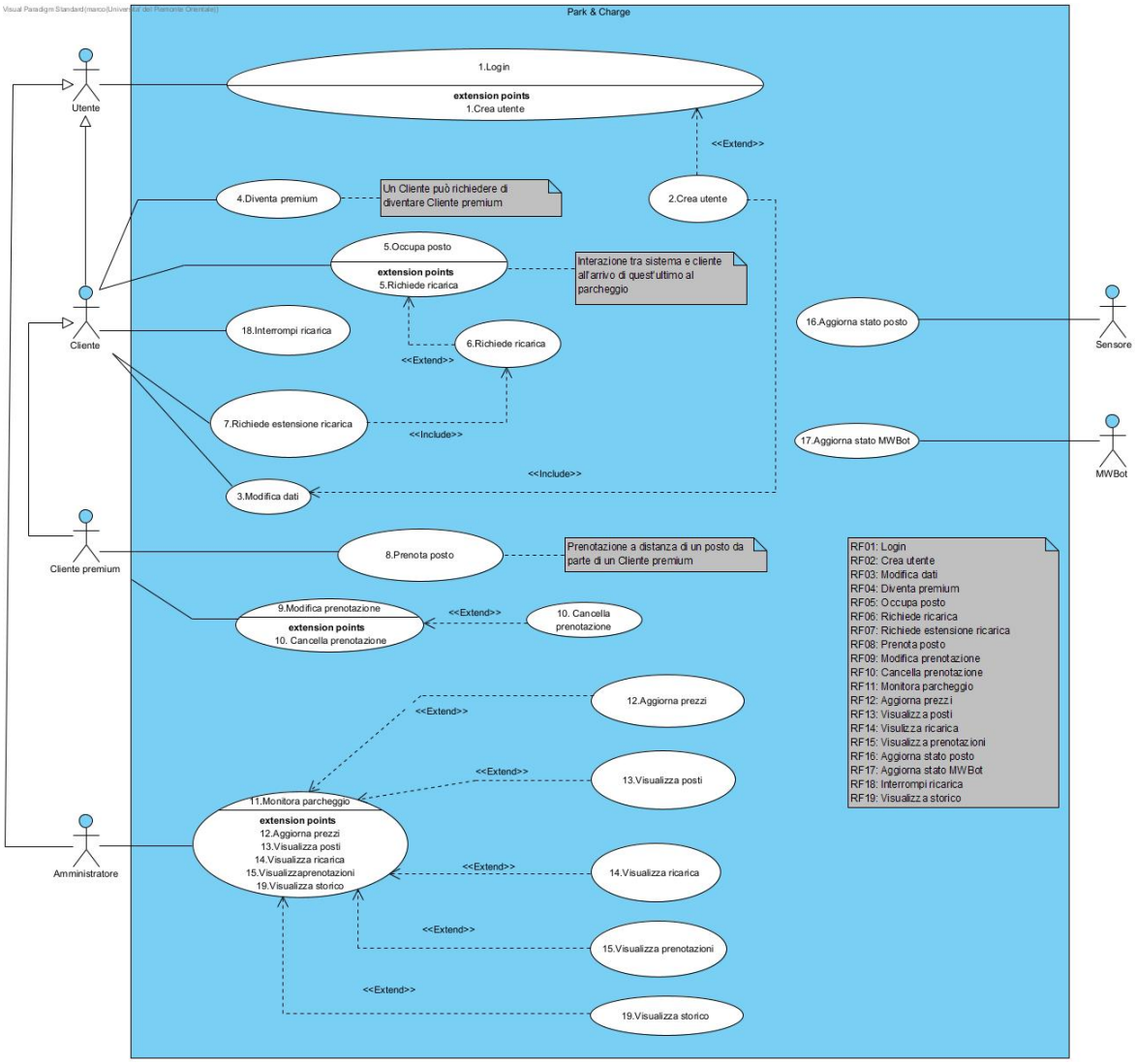

Figure 1.2: Diagramma dei casi d'uso

1.2.1 Login

Un utente inserisce nel form le proprie credenziali (username e password). Se le credenziali sono corrette l'utente accede al sistema nel menù a lui dedicato (menù cliente, menù cliente premium, menù amministratore). Se le credenziali sono errate viene mostrato un messaggio di errore e gli viene ripresentato il form di login.
Dal form di login sarà anche possibile accedere al form di registrazione.

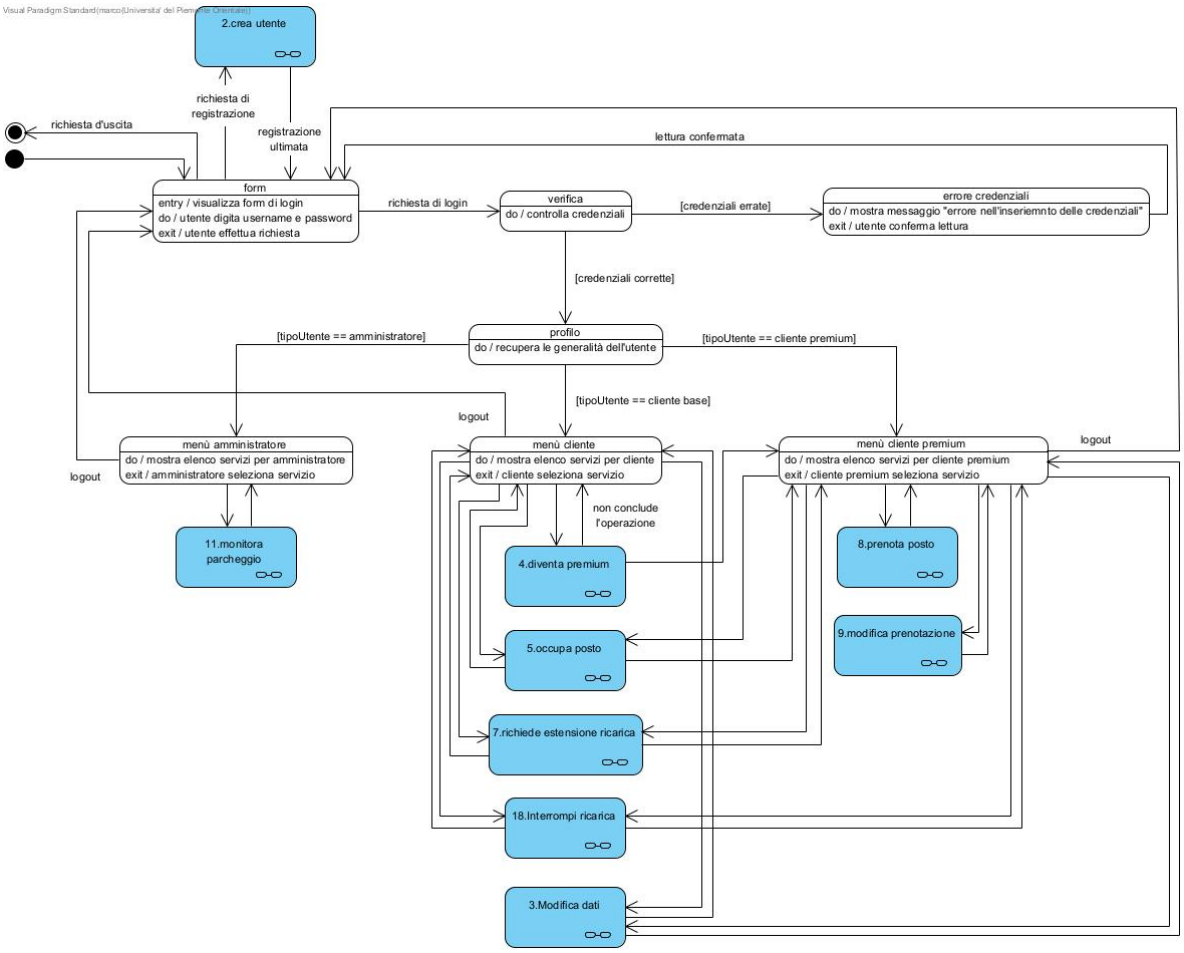

Figure 1.3: RF01 Login

1.2.2 Crea utente

Nel form di registrazione l'utente potrà andare ad inserire i propri dati, quali: Nome, Cognome, Username e Password; successivamente verrà richiesto di inserire i dati di una carta di credito da associare all'account appena creato. Dopo la registrazione l'utente viene riportato al form di login.

1.2.3 Modifica dati

Nel form l'utente potrà andare a modificare i propri dati, quali: Nome, Cognome, Username, Password e dati della carta.

1.2.4 Diventa premium

Un utente base può scegliere di diventare un utente premium con un pagamento una tantum.
Se un utente premium prova a fare l'upgrade, il sistema non glielo permette.

1.2.5 Occupa posto

Un utente, base o premium, che arriva al parcheggio richiede che gli venga assegnato un posto.
Nel caso in cui l'utente, ovviamente premium, abbia prenotato un posto gli verrà sicuramente asseg-

nato, altrimenti solamente se ci sono posti non prenotati nel lasso di tempo scelto.

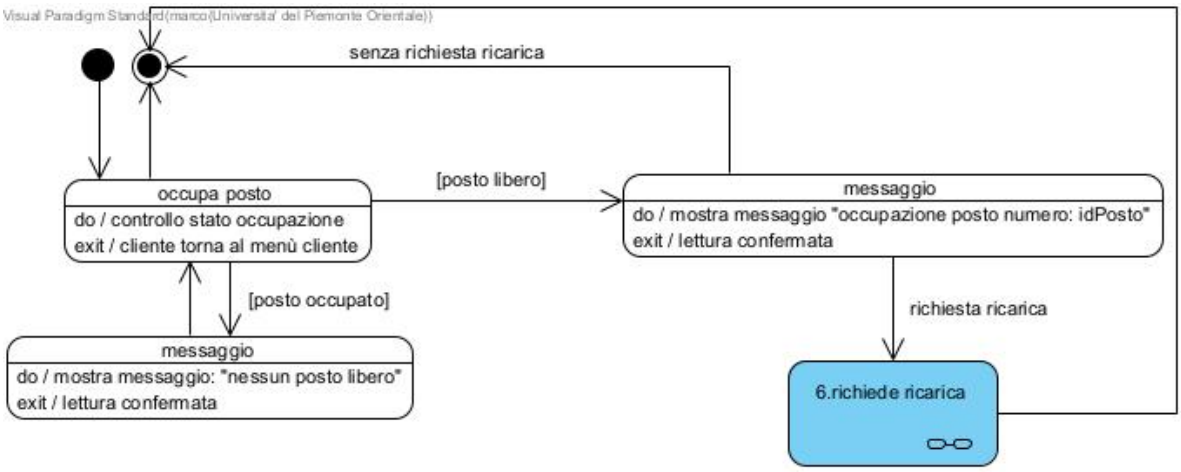

Figure 1.4: RF05 Occupa posto

1.2.6 Richiede ricarica

Un utente, base o premium, che arriva al parcheggio può richiedere che gli venga fornita una ricarica, richiedendo che essa venga fornita fino al raggiungimento di una determinata percentuale di carica della vettura. A quel punto il sistema darà all'utente una previsione del tempo richiesto dal MWBot, in base alla disponibilità, per ricaricare la vettura alla percentuale di carica desiderata.
Nel caso in cui la ricarica non dovesse essere soddisfacibile entro il tempo di sosta il sistema mostrerà un messaggio di avviso. Al termine della ricarica l'utente riceve una notifica.
La policy di scheduling delle ricariche è **EDF**, *Earliest Deadline First*. Questa politica garantisce il rispetto della deadline prefissata (il timestamp di fine posteggio) e massimizza il numero di job (ricariche) soddisfabili. L'algoritmo è preemptive (una ricarica iniziata non deve essere per forza conclusa direttamente) e dinamico (le ricariche vengono prese in carico in modo dinamico se soddisfacibili).

1.2.7 Richiede estensione ricarica

Un utente che ha richiesto la carica del proprio veicolo può, al termine di questa, può richiederne un'estensione.

1.2.8 Prenota posto

Un utente premium può prenotare un posto per un determinato lasso di tempo tramite un form: il sistema accetterà la richiesta solo se ci sono posti liberi in quel lasso di tempo.
Nel caso di successo il sistema mostrerà un messaggio di successo senza specificare il posto assegnatogli, altrimenti mostra un messaggio di errore e riporta l'utente nel form iniziale.

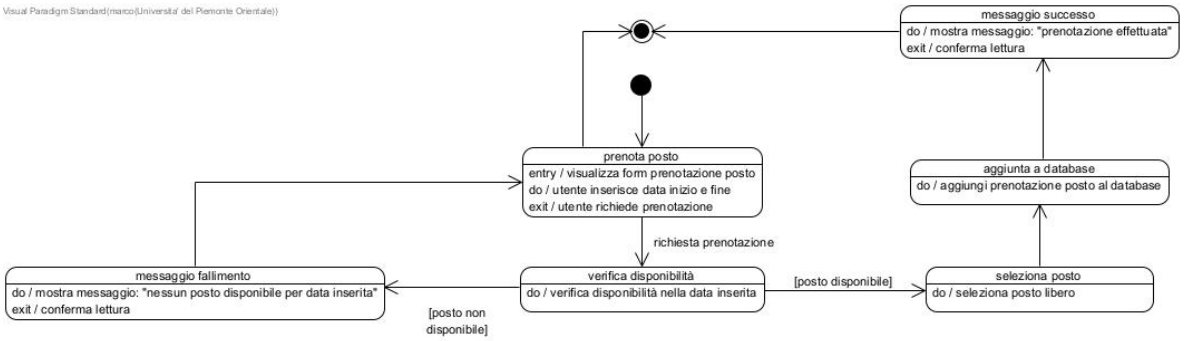

Figure 1.5: RF08 Prenota posto

1.2.9 Modifica Prenotazione

L'utente premium può modificare la prenotazione effettuata prima dell'ora di inizio.

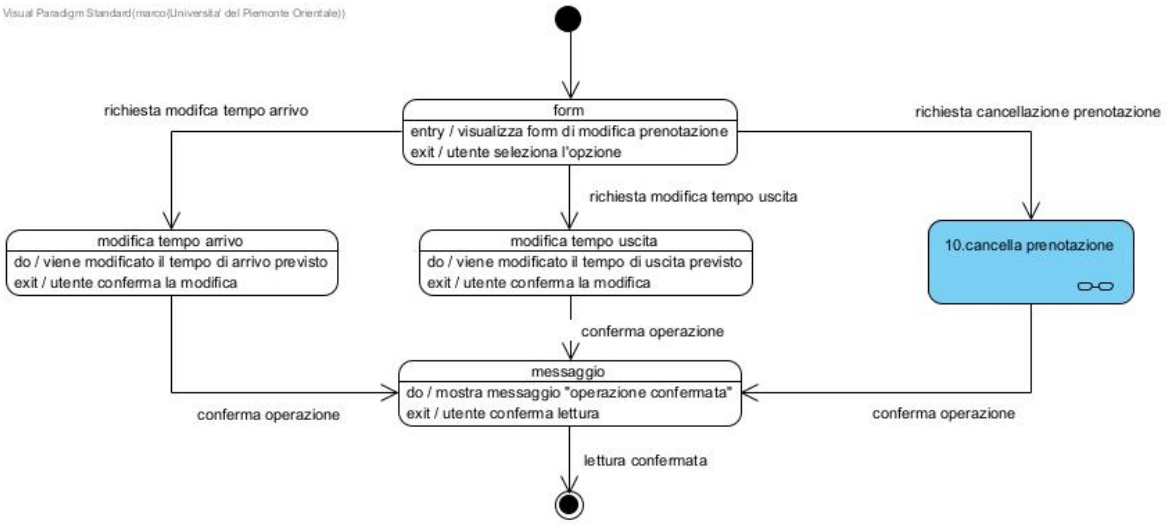

Figure 1.6: RF09 Modifica prenotazione

1.2.10 Cancella prenotazione

Un utente premium, prima dell'ora di inizio della prenotazione effettuata, può cancellare tale prenotazione senza penale. Il posto che gli era stato assegnato ritorna libero nel lasso di tempo della prenotazione.

1.2.11 Monitora parcheggio

L'Amministratore può, attraverso un apposito form, accedere agli altri form di aggiornamento prezzi, visualizzazione posti, visualizzazione ricariche e le prenotazioni effettuate da utenti premium.

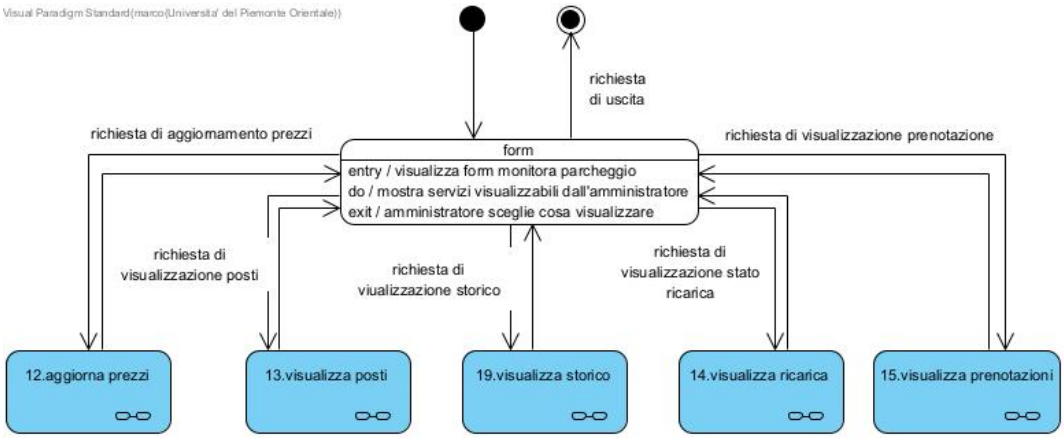

Figure 1.7: RF11 Monitora Parcheggio

1.2.12 Aggiorna prezzo

L'Amministratore può, attraverso un apposito form, andare a modificare i costi: del parcheggio all'ora, della ricarica al KiloWatt, della penale e il costo per l'abbonamento al servizio Premium.

1.2.13 Visualizza posti

L'Amministratore è in grado di visualizzare tutti i posti del parcheggio e le loro caratteristiche: se sono prenotati, occupati o liberi.

1.2.14 Visualizza ricarica

L'Amministratore è in grado di visualizzare le ricariche in corso e quelle prenotate, può anche vedere dove sono gli MWBot e cosa stanno facendo in quel momento.

1.2.15 Visualizza prenotazioni

L'Amministratore è in grado di visualizzare le prenotazioni attuali e future degli utenti.

1.2.16 Aggiorna stato posto

Il sensore aggiorna periodicamente lo stato del posto che controlla.

1.2.17 Aggiorna stato MWBot

L'MWBot aggiorna il proprio stato.

1.2.18 Interrompi ricarica

Un utente può richiedere che la ricarica in corso venga istantaneamente interrotta. È obbligatorio interrompere la ricarica in corso per spostare il veicolo.

1.2.19 Visualizza storico

L'Amministratore può visualizzare lo storico dell'occupazione dei posti, delle ricariche e i pagamenti avvenuti.

1.3 Requisiti funzionali e Non funzionali

REQUISITI FUNZIONALI	ID	GRUPPO
login	RF01	1/2
crea utente	RF02	1
modifica dati	RF03	1
diventa premium	RF04	1
occupa posto	RF05	1
richiede ricarica	RF06	2
richiede estensione ricarica	RF07	2
prenota posto	RF08	1
modifica prenotazione	RF09	1
cancella prenotazione	RF10	1
monitora parcheggio	RF11	2
aggiorna prezzi	RF12	1/2
visualizza posti	RF13	2
visualizza ricariche	RF14	2
visualizza prenotazioni	RF15	2
aggiorna stato posti	RF16	1
aggiorna stato MWBot	RF17	2
interrompi ricarica	RF18	2
visualizza storico	RF19	2

GRUPPO	COMPONENTI
1	Marino - Vecchio
2	Galliera - Ternullo

Figure 1.8: Requisiti Funzionali

ID	DESCRIZIONE
RNF01	Specifica e progettazione saranno definite tramite diagrammi UML e linguaggio naturale.
RNF02	Il sistema sarà implementato in Java.
RFN03	L'interfaccia utente sarà realizzata tramite Java Swing.
RFN04	I database saranno implementati usando SQLite e Java DataBase Connectivity.
RFN05	Verrà utilizzato SQLiteBrowser per visulazzare e modificare i contenuti dei database.
RFN06	Verrà utilizzato Visual Paradigm per disegnare i diagrammi UML e le bozze dell'interfaccia utente.
RFN07	Verrà utilizzato Gradle per compilare, eseguire e testare il codice.
RFN08	Verrà utilizzato Git per condividere documentazione e codice tra gli sviluppatori.
RFN09	Verrà utilizzato GitLab come repository on-line.
RFN10	Verrà utilizzato Swagger OpenAPI per la documentazione dell'Api REST.
RFN11	Le date saranno memorizzate nel pattern: YYYY-MM-DD HH-MI-SS.
RFN12	L'username deve contenere almeno 4 caratteri.
RFM13	La password deve contenere almeno 6 caratteri di cui almeno una cifra.
RFM14	Verrà utilizzato MQTT per la comunicazione con il sottoinsieme IoT
RFM15	Il punto di contatto tra Frontend e Backend sarà dato da una API REST
RFM16	Gli emulatori dei sensori, MwBot e device sarannoi realizzati tramite script Python
RFM17	Il Broker Mqtt utilizzato sarà Mosquitto

Figure 1.9: Requisiti Non Funzionali

Chapter 2

Progettazione

2.1 Architettura del sistema

I componenti principali dell'architettura sono:

- Frontend: creazione di una desktop app che opera da punto di accesso al sistema, che contatta il backend tramite una API REST. Permette alle varie categorie di utenti (Clienti, Clienti Premium, Amministratore) di accedere a funzionalità diverse.
- Backend: permette agli utenti di accedere ai servizi del parcheggio e di ricarica. Gestisce lo scheduling delle ricariche con algoritmo EDF. Comunica attraverso MQTT col MQTT Broker per il controllo degli MWBot, l'accesso dati dei sensori e per inviare notifiche a utenti(quando viene conclusa la ricarica e a sosta conclusa). Si interfaccia inoltre agli utenti (Cliente e Amministartore) attraverso una REST API dove espone varie funzionalità tra cui prenotazione del posto, prenotazione ricarica e monitoraggio da parte dell'amministratore.
- Gestore IoT: permette al sistema di comunicare con MWBot e i sensori del parcheggio tramite un message broker (mosquitto).
- Database: permette al sistema di tenere delle prenotazioni, dei posti occupati, dei pagamenti e degli utenti.

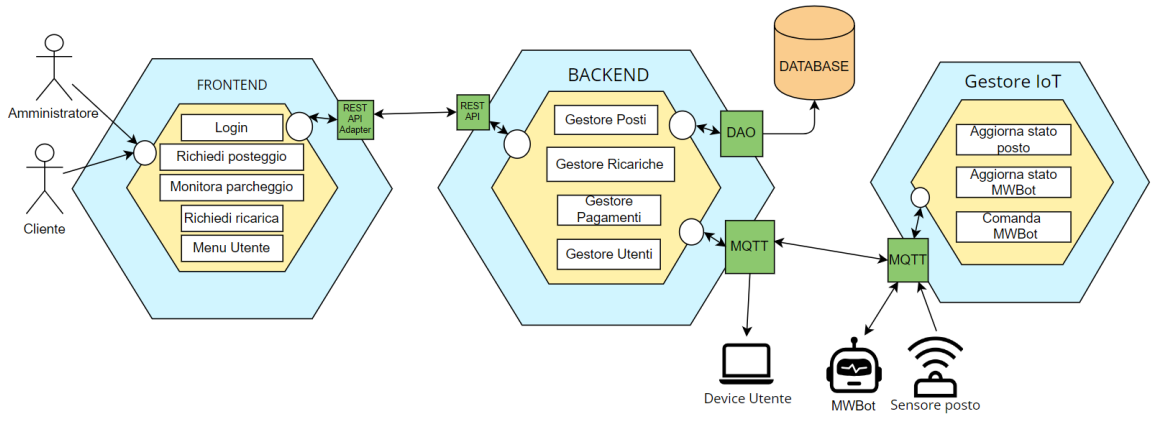

Figure 2.1: Architettura a microservizi

La rappresentazione di tale architettura nel diagramma delle classi è la seguente:

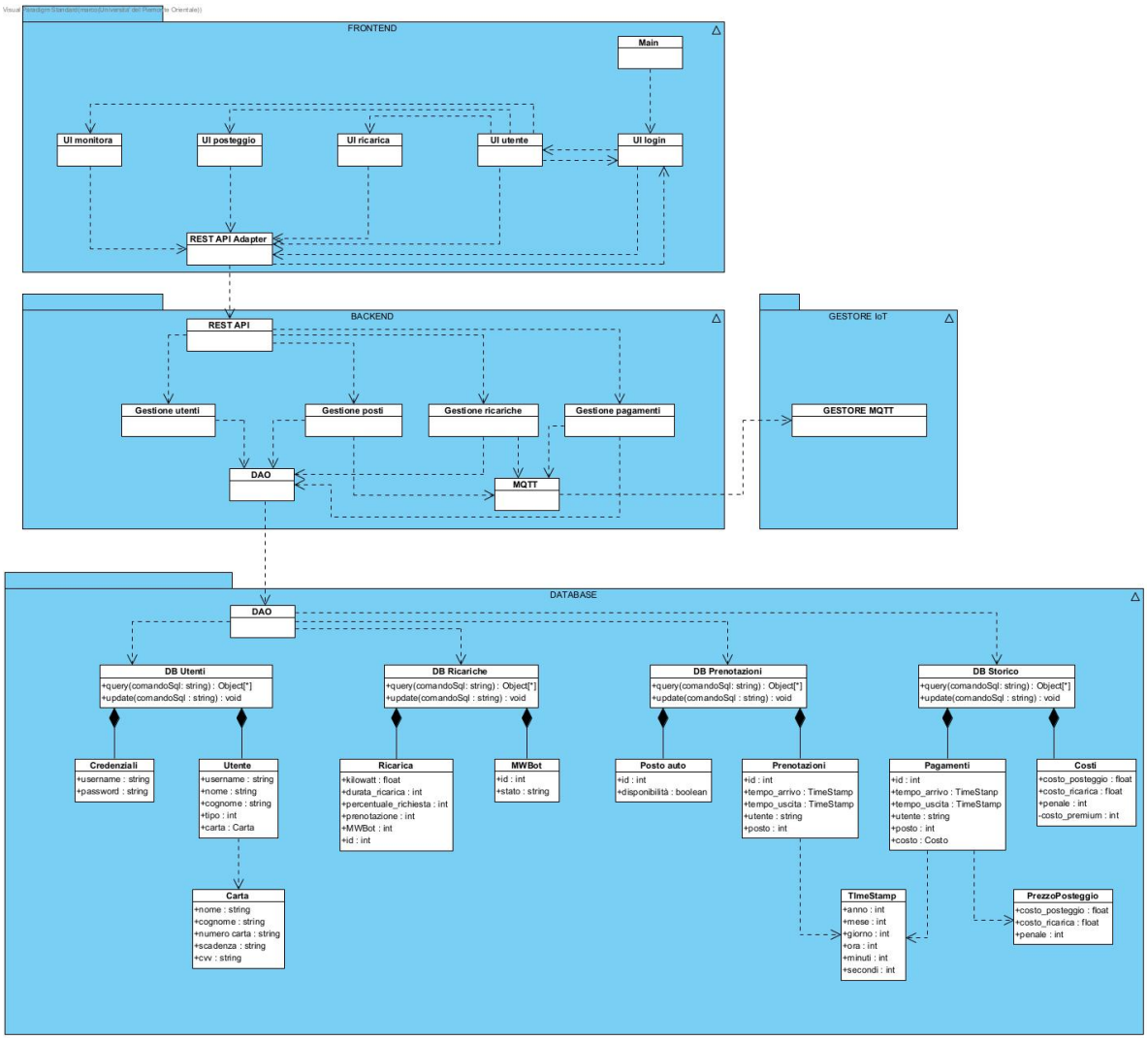

Figure 2.2: Diagramma delle classi

2.2 API REST

Abbiamo sviluppato la documentazione dell'API REST tramite l'editor Swagger ed è la seguente:

- link documentazione API rest[1]
- link codice yaml[2]

[1]https://nicomarino.altervista.org/RestAPI.html
[2]https://app.swaggerhub.com/apis/MARCOVECCHIO/ParkCharge/1.0.0

2.3 MQTT TOPICS

2.3.1 ParkCharge/StatoPosti/ID_Posto

Il GestoreIoT aggiorna lo stato del singolo posto definito da ID_Posto.
Il formato di data è:

```
{
    "stato" :String
}
```

Listing 2.1: ParkCharge/StatoPosti/ID_Posto

dove il campo stato ∈ {"libero", "occupato"}.
Permessi: GestoreIoT Write, BackEnd Read.

2.3.2 ParkCharge/StatoSensore/ID_Posto

Il sensore aggiorna lo stato del posto che monitora definito da ID_Posto.
Il formato di data è:

```
{
    "stato" :String
}
```

Listing 2.2: ParkCharge/StatoSensore/ID_Posto

dove il campo stato ∈ {"libero", "occupato"}.
Permessi: Sensore Write, GestoreIoT Read.

2.3.3 ParkCharge/StatoMWBot/ID_MWBot

Ogni MWBot comunica il proprio stato.

```
{
    "statoCarica" :String,
    "posizione" :Integer,
    "percentualeRicarica" :Integer
    "KW_Emessi" :Float
}
```

Listing 2.3: ParkCharge/StatoMWBot/ID_MWBot

dove il campo statoCarica ∈ {"caricando", "finito"}.
Permessi: MWBot Write, GestoreIoT Read.

2.3.4 ParkCharge/StatoRicariche/ID_MWBot

Il GestoreIoT comunica lo stato del singolo MWBot.

```
{
    "statoCarica" :String,
    "posizione" :Integer,
    "percentualeRicarica" :Integer
    "KW_Emessi" :Float
}
```

Listing 2.4: ParkCharge/StatoRicariche/ID_MWBot

dove il campo statoCarica ∈ {"caricando", "finito"}.
Permessi: GestoreIoT Write, BackEnd Read.

2.3.5 ParkCharge/RichiediRicarica/ID_MWBot

Il sistema comunica il prossimo comando per l'MWBot al GestoreIoT.

```
{
    "target" :Integer,
    "percentualeRicarica" :Integer
}
```

Listing 2.5: ParkCharge/RichiediRicarica/ID_MWBot

Permessi: BackEnd Write, GestoreIoT Read.

2.3.6 ParkCharge/EseguiRicarica/ID_MWBot

Il GestoreIoT comunica il prossimo comando all'MWBot.

```
{
    "target" :Integer,
    "percentualeRicarica" :Integer
}
```

Listing 2.6: ParkCharge/EseguiRicarica/ID_MWBot

Permessi: GestoreIoT Write, MWBot Read.

2.3.7 ParkCharge/Notifiche/SostaConclusa/ID_Utente

Il BackEnd comunica al dispositivo dell'utente il periodo finale di sosta, ed eventuale ricarica, con il costo pagato. Un utente può fare subscribe a tutto con ParkCharge/Notifiche/#/ID_Utente/

```
{
    "tempoSosta" :Integer,
    "costoSosta" :Integer,
    "kilowattUsati" :Integer,
    "costoRicarica" :Integer
}
```

Listing 2.7: ParkCharge/Notifiche/SostaConclusa/ID_Utente

Permessi: BackEnd Write, UserDevice Read.

2.3.8 ParkCharge/Notifiche/RicaricaConclusa/ID_Utente

Il BackEnd comunica al dispositivo dell'utente che la ricarica è stata conclusa, dando la possibilità di richiedere un'ulteriore ricarica.

```
{
    "kilowattUsati" :Integer,
    "costoRicarica" :Integer
}
```

Listing 2.8: ParkCharge/Notifiche/RicaricaConclusa/ID_Utente

Permessi: BackEnd Write, UserDevice Read.

2.3.9 ParkCharge/Pagamento/ID_Utente

Il dispositivo dell'utente comunica al BackEnd che l'utente ha accettato il pagamento.

```
{
    "username" :String,
    "totale" :Float,
    "status" :String
}
```

Listing 2.9: ParkCharge/Notifiche/RicaricaConclusa/ID_Utente

dove il campo status $\in$ {"pagamento_effettuato"}. Permessi: BackEnd Read, UserDevice Write.

2.4 Diagramma dei componenti

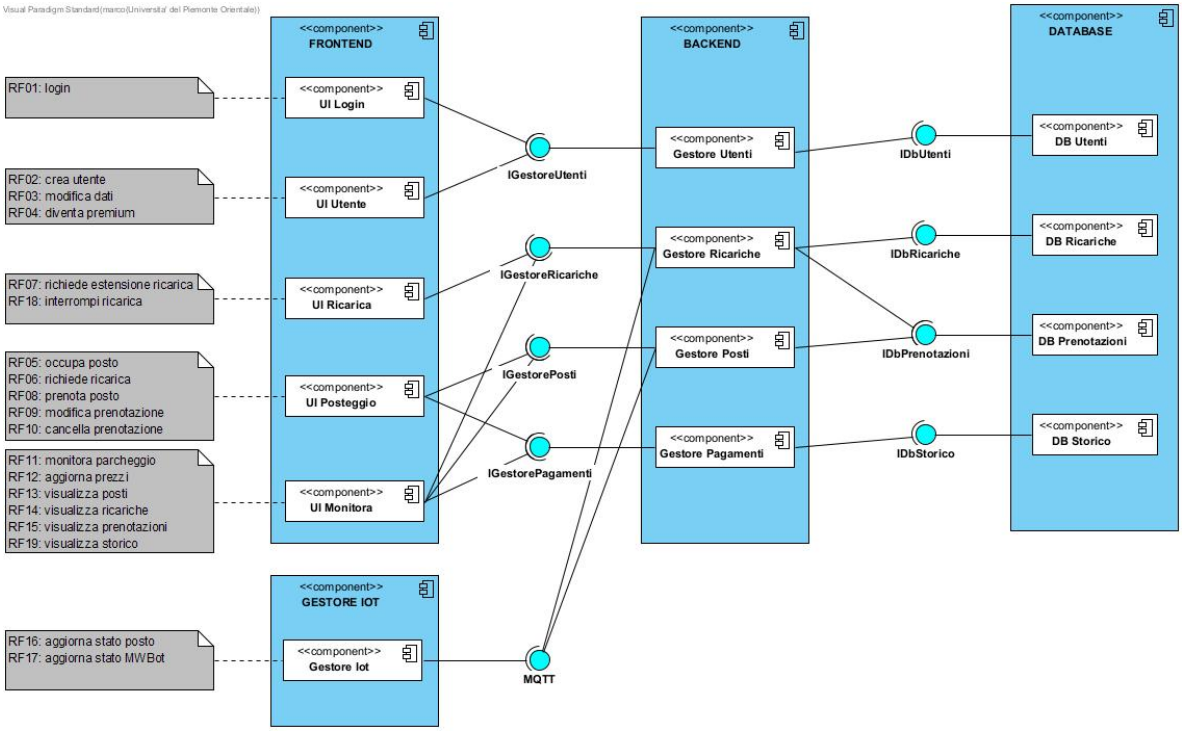

Figure 2.3: Diagramma interazione tra componenti

2.5 Diagrammi di sequenza

2.5.1 Sequenza Login

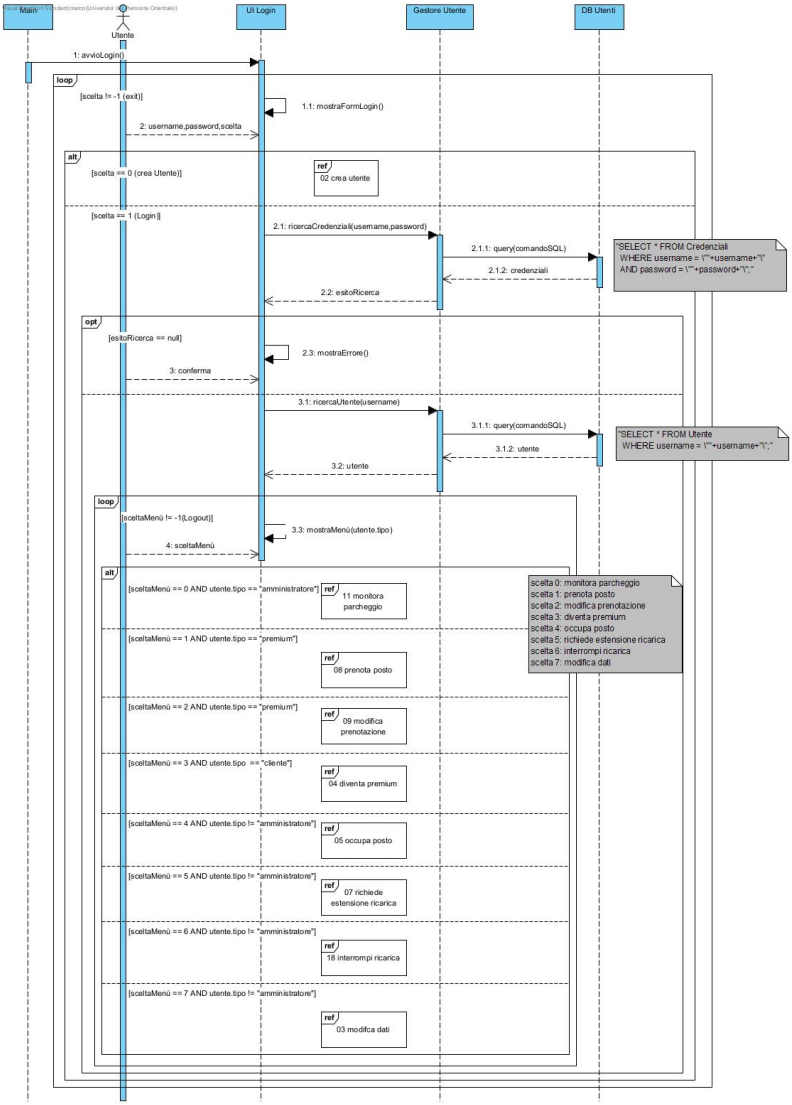

Figure 2.4: Diagramma di sequenza RF01 Login

2.5.2 Sequenza Richiede ricarica

Un utente può richiedere una ricarica solo nel caso in cui stia occupando un posto e non ne abbia già richiesta un'altra. Se entrambe le condizioni sono verificate, l'utente può richiedere una percentuale da ricaricare. Dopodiché lo scheduling EDF per accettare o meno la ricarica ha bisogno dell'orario di fine della prenotazione (query al DB Prenotazioni) e le ricariche già in coda (query al DB Ricariche). Se la ricarica è accettabile viene inserita nella coda delle ricariche e mostrato un messaggio di conferma all'utente.

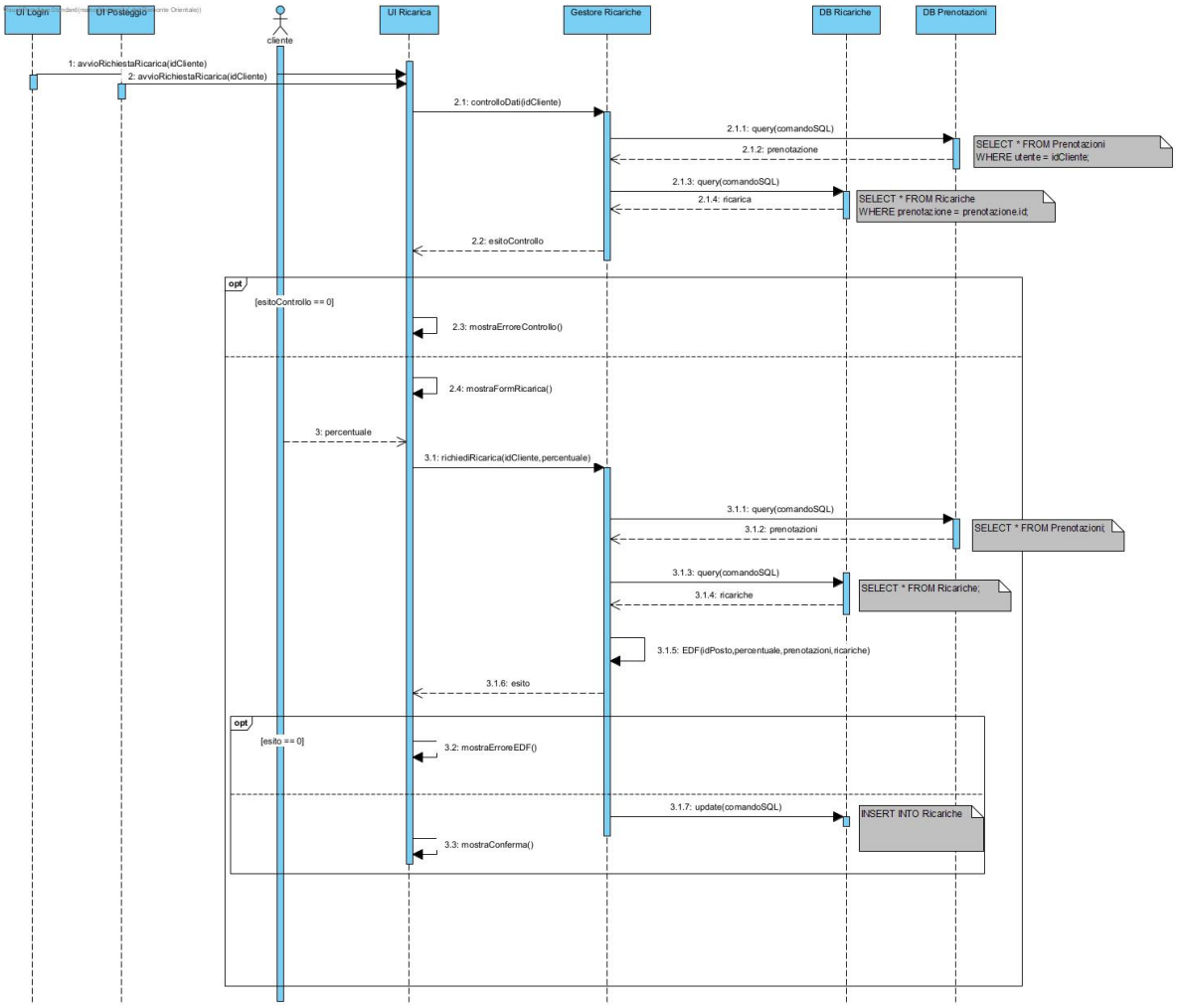

Figure 2.5: Diagramma di sequenza RF06 Richiede ricarica

2.5.3 Sequenza Prenota posto

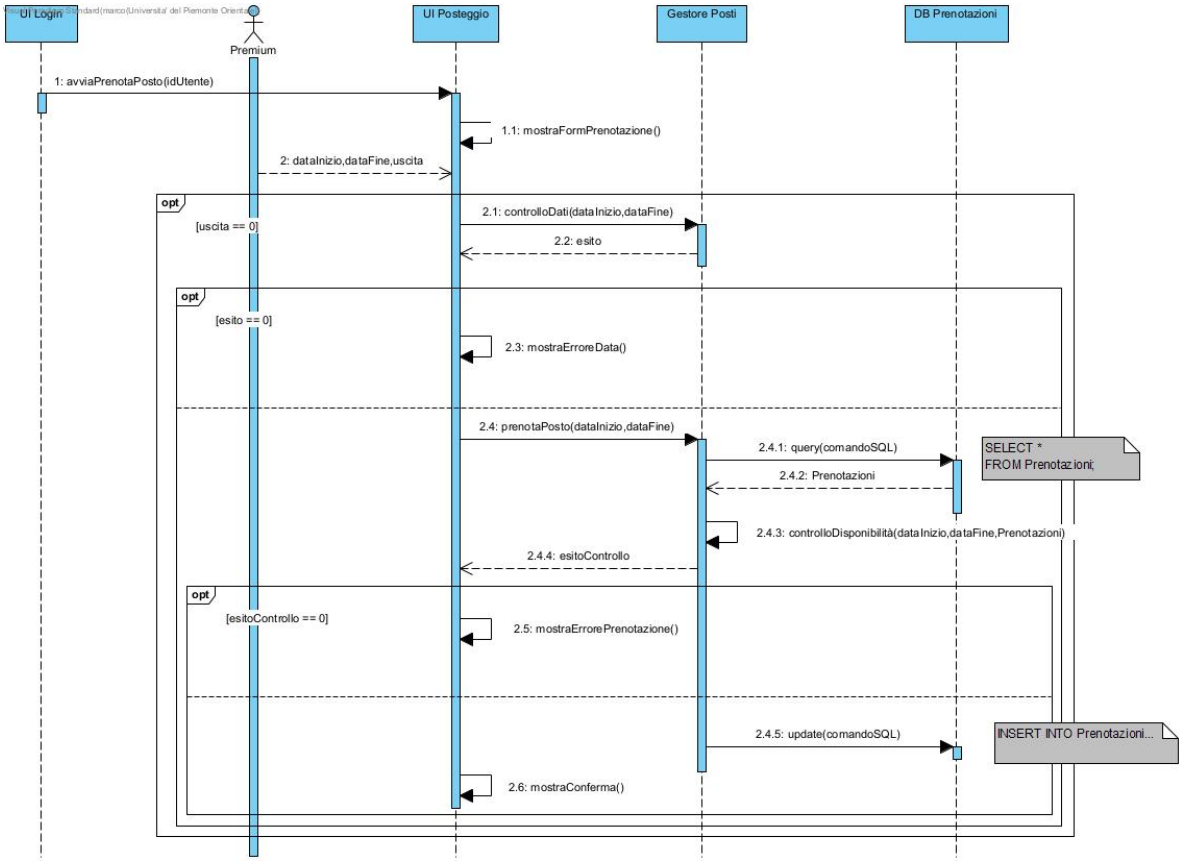

Figure 2.6: Diagramma di sequenza RF08 Prenota posto

2.5.4 Sequenza Monitora parcheggio

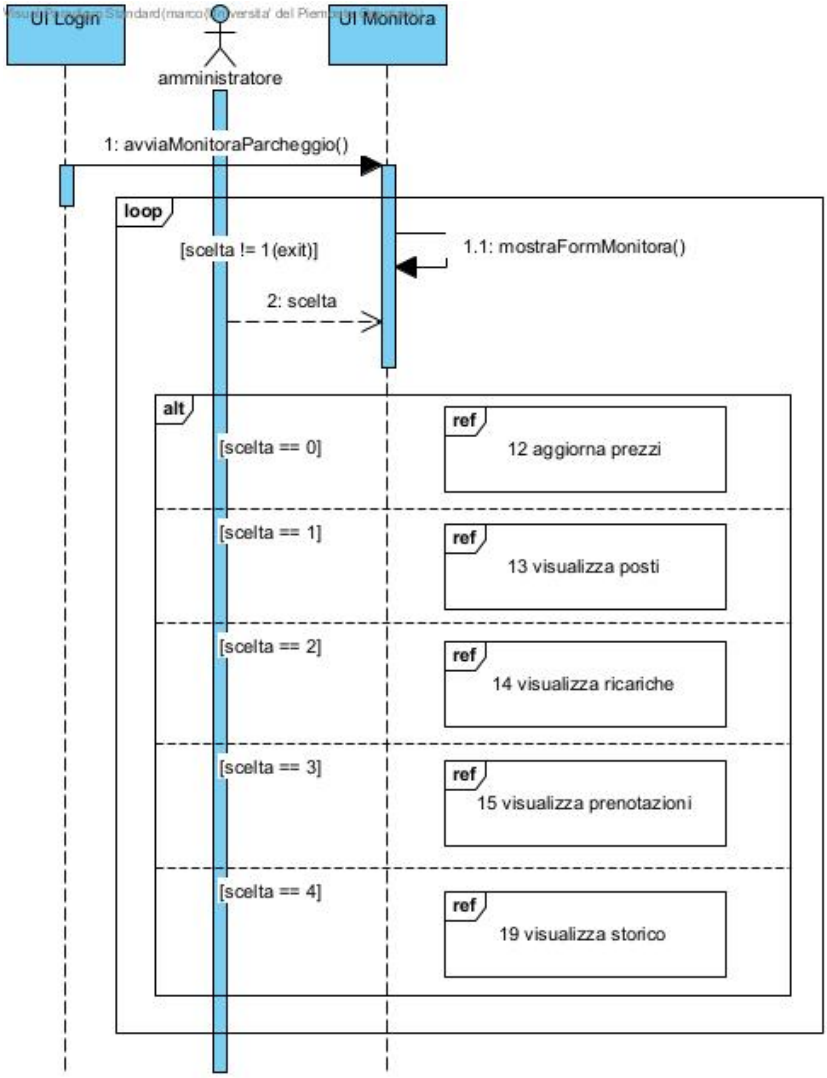

Figure 2.7: Diagramma di sequenza RF11 Monitora parcheggio

2.5.5 Sequenza Visualizza storico

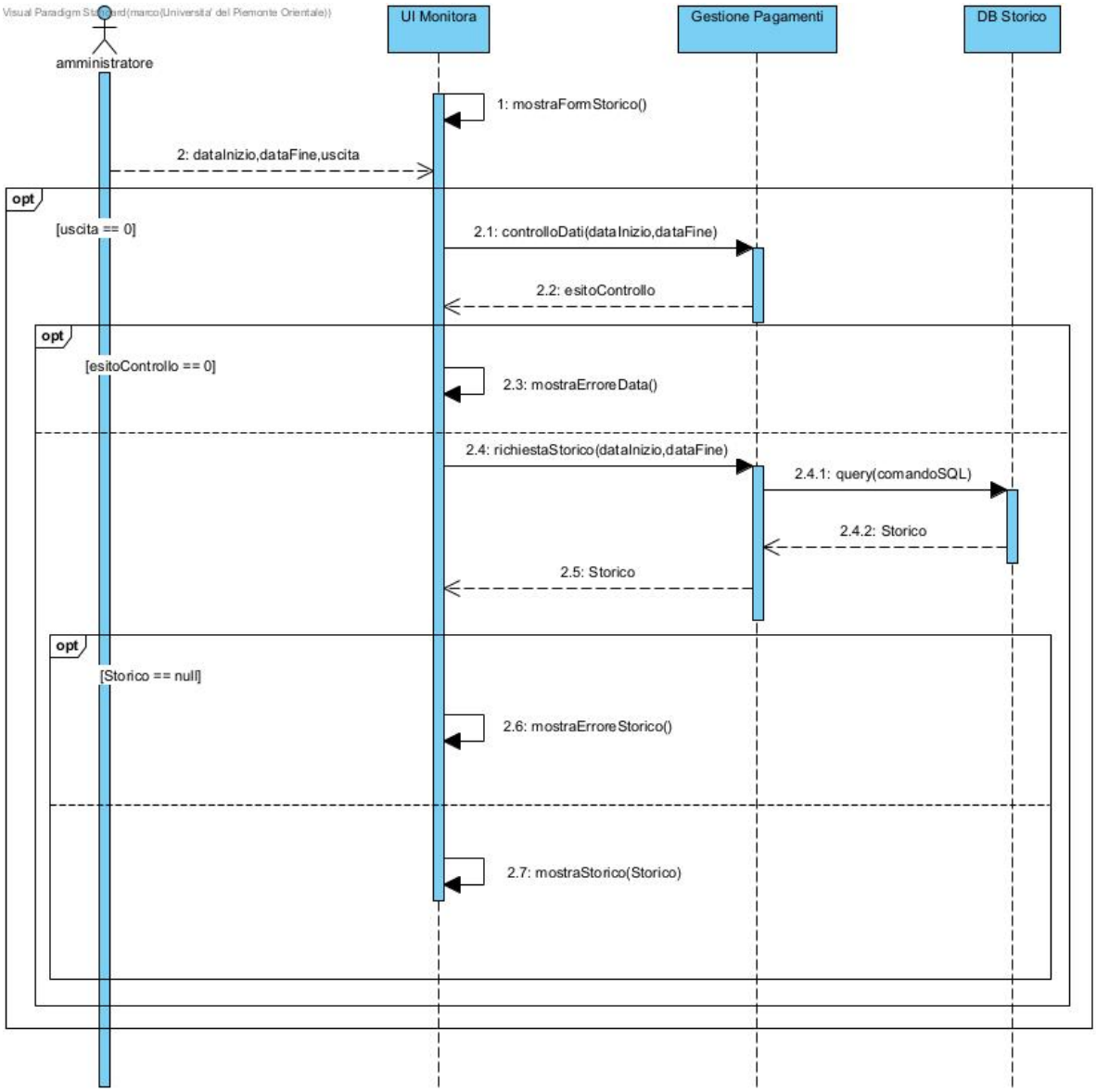

Figure 2.8: Diagramma di sequenza RF19 Visualizza storico

2.6 Interazioni IoT (alto livello)

L'obbiettivo di questa sezione è descrivere l'interazione, le azioni e gli eventi che scaturiscono dai dispositivi IoT. Gli eventi da tenere in considerazione sono:

1. Il sensore di un posto registra l'occupazione del posto.
2. Il sensore di un posto registra la liberazione del posto.
3. L'MWBot inizia la ricarica assegnatagli.

4. L'MWBot ricarica una certa percentuale.
5. L'MWBot completa la ricarica assegnatagli.

Il sensore di un posto registra l'occupazione del posto

- **Prerequisiti**: il posto monitorato da un sensore è libero (non occupato da un veicolo).
- **Evento**: un veicolo occupa il posto.
- **Azioni**: il sensore invia un messaggio per avvisare il BackEnd dell'occupazione del posto. Nel database il posto è etichettato come occupato.

Il sensore di un posto registra la liberazione del posto

- **Prerequisiti**: il posto monitorato da un sensore è occupato (ovviamente da un veicolo con una prenotazione attiva).
- **Evento**: il veicolo libera il posto.
- **Azioni**: il sensore invia un messaggio per avvisare il BackEnd della liberazione del posto. Nel database il posto è etichettato come libero. La prenotazione che occupava quel posto viene terminata e spostata nello storico, il pagamento è richiesto.

L'MWBot inizia la ricarica assegnatagli

- **Prerequisiti**: l'MWBot riceve dal BackEnd l'ordine di effettuare una ricarica in un certo posto.
- **Evento**: l'MWBot inizia tale ricarica.
- **Azioni**: l'MWBot invia un messaggio al BackEnd per informarlo dell'avvenuto inizio della ricarica, il BackEnd modifica lo stato a database dell'MWBot a occupato.

L'MWBot conclude la ricarica assegnatagli

- **Prerequisiti**: l'MWBot stava ricaricanto un veicolo.
- **Evento**: l'MWBot completa tale ricarica (è a conoscenza di quanto erogare).
- **Azioni**: l'MWBot invia un messaggio al BackEnd per informarlo dell'avvenuta conclusione della ricarica. Il BackEnd modifica lo stato a database dell'MWBot a libero. Se ci sono altre ricariche in coda ne seleziona una (tramite EDF) e la assegna come per *L'MWBot inizia la ricarica assegnatagli*.

L'MWBot ricarica una certa percentuale : per mantenere sincronia tra MWBot e BackEnd, l'MWBot aggiorna il BackEnd a ogni 1% ricaricato.

- **Prerequisiti**: l'MWBot sta ricaricanto un veicolo.
- **Evento**: l'MWBot eroga un 1% alla batteria del veicolo (il veicolo passa per esempio dal 45% al 46%).
- **Azioni**: l'MWBot invia un messaggio al BackEnd per informarlo dell'avvenuta erogazione fino a una certa percentuale. Il BackEnd salva nel database la percentuale erogata per quella ricarica. L'algoritmo di scheduling EDF è preemptive, il BackEnd interroga quindi EDF per determinare il posto da ricaricare: potrebbe essere lo stesso (l'MWBot continua la ricarica) o no (l'MWBot si sposta). In entrambi i casi il BackEnd comunica il nuovo posto da ricaricare all'MWBot (se è già in quello corretto continua, altrimenti si sposta).

Chapter 3

Implementazione

Abbiamo sviluppato l'implementazione del progetto suddividendo il sistema in tre microservizi: frontend, backend e la gestione IoT. Il cervello del sistema è nel backend, il quale gestisce la logica, interazione con il database, sheduling delle ricariche, le prenotazioni, viene definita la REST API e comunicazione con il gestore IoT tramite MQTT. Il frontend fornisce l'interfaccia utente sviluppata con Java Swing e va ad implementare le chiamate all'API REST. In fine il gestore IoT viene utilizzato per la comunicazione (tramite MQTT) con i sensori dei posti e l'MWBot.
La divisione dei compiti è stata fatta tramite requisiti funzionali, cercando di distribuirli nel modo più equo possibile. Alcuni requisiti sono stati svolti assieme.

Login Le principali classe coinvolte sono:

- **Main (Frontend)**: all'interno di questa classe troviamo il metoto *main* tramite il cui viene richiamato il metodo *avvioLogin* che servirà per avviare l'interfaccia grafica all'interno di **UiLogin**.
- **UiLogin (Frontend)**: Contiene il metodo *avviaLogin*, che avvia il metodo *mostraFormLogin* tramite cui l'utente può registrasi oppure inserire le credenziali per il login. Nel primo caso verrà richiamato *avviaCreaUtente* mentre nel secondo verrà richiamato *ricercaCredenziali* che farà una richiesta HTTP GET per verificare la correttezza di username e password. Nel caso si abbia riscontro positivo (200 OK) allora verrà richiamato il metodo *ricercaUtente* che farà un'altra richiesta HTTP GET per ottenere i dati dell'utente e verrà utilizzato il metodo *mostraMenu* che serve per mostra il menù adatto al tipo di utente (base, premium o amministratore).
- **RestApi (Backend)**: Questa classe gestisce le richieste HTTP GET relative al login. La prima di queste servirà per verificare la correttezza delle credenziali e quindi dopo il parsing del body viene chiamato il metodo *getCredenziali* nella classe *GestoreUtenti* e il risultato viene assegnato alla variabile *credenziali*. Il risultato viene quindi valutato: se lo status di *credenziali* è 200, l'operazione è riuscita e l'oggetto credenziali viene restituito; se lo status è 404, viene restituito un messaggio di errore dove viene detto che le credenziali dell'utente non sono state trovate. Ecco un esempio di richiesta:

```
GET /api/v1.0/credenziali/{username}/{password} HTTP/1.1
Host: localhost:4568
[
    {
        "username": "mrossi",
        "passeord": "password123"
    }
]
```

Listing 3.1: GET /credenziali/username/password

La seconda invece servirà per le informazioni dell'utente e quindi dopo il parsing del body viene chiamato il metodo *getUtente* nella classe *GestoreUtenti* e il risultato viene assegnato alla variabile *utente*. Il risultato viene quindi valutato: se lo status di *utente* è 200, l'operazione è riuscita e l'oggetto utente viene restituito; se lo status è 404, viene restituito un messaggio di errore dove viene detto che l'utente non è stato trovato. Ecco un esempio di richiesta:

```
GET /api/v1.0/utenti/{username} HTTP/1.1
Host: localhost:4568
[
    {
        "username": "mrossi",
        "nome": "Mario",
        "cognome": "Rossi",
        "tipo": "1",
        "carta": "1234567812345678"
    }
]
```

Listing 3.2: GET /utenti/username

- **GestoreUtenti(Backend)**: Questa classe contiene il metodo *getCredenziali* e *getUtente*, che ritornano rispettivamente le credenziali e l'utente dati username e password. Nel caso non is ottenga risultato dalle query verrà ritornato null.

Aggiorna prezzi Le principali classi coninvolte sono:

- **UiLogin (FrontEnd)**: Nel caso l'utente sia un amministratore può, tramite un menù, selezionare la voce per monitorare il parcheggio, si apre quindi un form dove l'amministratore può scegliere di modificare i preggi, se lo fa viene avviato il metodo *avviaModificaPrezzi* all'interno dell' **UiMonitora**
- **UiMonitora (FrontEnd)**: contiene il metodo avviaModificaPrezzi all'interno del quale viene chiesto all'amministratore di inserire i nuovi valori per i prezzi, se conferma l'operazione viene effettuata una richesta HTTP PUT. Se la risposta della richiesta è 200 viene restituito il messaggio di conferma, altrimenti di errore.
- **RestApi(Backend)**: La classe contiene una richiesta HTTP di tipo PUT, viene richiamato il metodo *aggiornaPrezzi* all'interno della classe *GestorePagamenti* e il risultato sarà assegnato alla variabile *update*. Viene quindi valutato il risultato: se *update* è true allora vuol dire che la

richiesta di modifica è andata a buon fine e quindi viene ritornato i costi modificati, se invece e false viene restituita la stringa "Prezzi non aggiornati". Ecco un esempio di richiesta:

```
PUT /api/v1.0/costo HTTP/1.1
Host: localhost:4568
Content-Type: application/json
{
  "costo_posteggio": 0,
  "costo_ricarica": 0,
  "penale": 0,
  "costo_premium": 0
}
```

Listing 3.3: PUT /costo

3.1 Gruppo 1

I requisiti funzionali a noi assegnati sono stati:

1. Crea utente
2. Modifica dati
3. Diventa premium
4. Occupa posto
5. Prenota posto
6. Modifica prenotazione
7. Cancella prenotazione
8. Aggiorna stato posto

Creazione Utente Le principali classi coinvolte sono:

- **UiLogin (FrontEnd)**: Se l'utente non ha ancora un account, può selezionare l'opzione "Crea Utente" dal menù, che invoca il metodo *avviaCreaUtente* all'interno della classe *UiUtente*.
- **UiUtente (FrontEnd)**: Contiene il metodo *avviaCreaUtente*, che avvia il processo di creazione dell'utente tramite una richiesta HTTP POST. Questo metodo richiama *mostraFormRegistrazione*, dove l'utente inserisce i propri dati. Successivamente, il metodo *controlloFormato* verifica la correttezza dei campi. I valori inseriti sono assegnati ai campi dell'oggetto utente, che viene poi convertito in una stringa JSON inclusa nel body della richiesta HTTP POST. Se la risposta è 201, l'utente viene restituito; altrimenti, viene visualizzato un messaggio di errore.
- **RestApi (Backend)**: Questa classe gestisce la richiesta HTTP POST. Dopo il parsing del corpo della richiesta, viene chiamato il metodo *creaUtenti* nella classe *GestoreUtenti* e il risultato viene assegnato alla variabile *response*. Il risultato viene quindi valutato: se lo status di *response* è 201, l'operazione è riuscita e l'oggetto utente viene restituito; se lo status è 400, viene restituito un messaggio di errore. Ecco un esempio di richiesta:

```
POST /api/v1.0/utenti HTTP/1.1
Host: localhost:4568
Content-Type: application/json
{
  "carta" :"carta",
  "tipo" :0,
  "password" :"password",
  "cognome" :"cognome",
  "nome" :"nome",
  "username" :"username"
}
```

Listing 3.4: POST /utenti

- **GestoreUtenti (Backend)**: Questa classe contiene il metodo *creaUtenti*, che riceve come argomenti l'utente e le credenziali. Controlla che non esista un altro utente con lo stesso username e, in caso contrario, esegue le query necessarie per creare l'utente e le relative credenziali.

Modifica dati Le classi principali sono:

- **UiLogin (FrontEnd)**: Nel caso l'utente sia un cliente (sia base che premium) può, tramite un menù, selezionare la voce modifica dei dati e richiamare il metodo *avviaModificaDati* all'interno della classe *UiUtente*.
- **UiUtente (FrontEnd)**: Contiene il metodo *avviaModificaDati*, il quale va ad avviare il processo di modifica dei dati tramite una richiesta HTTP PUT. Viene richiamato il metodo *mostraFormModificaDati* tramite cui l'utente inserisce i nuovi dati e può scegliere se modificare i dati o meno, se decide di modificarli viene richiamata la funzione *controlloFormato* che effettua il controllo sui tre campi modificabili (nome, cognome e carta). I campi all'interno dell'oggetto Utente vengono quindi aggiornati con i nuovi valori, l'oggetto viene poi convertito in una stringa JSON, la quale farà parte del body della richiesta HTTP PUT. Se la risposta della richiesta è 200 viene restituito l'utente, altrimenti un messaggio di errore.
- **RestApi (Backend)**: La classe contiene una richiesta HTTP di tipo PUT. All'interno della classe dopo il parsing del Body della richiesta e l'estrazione del parametro ':username' dall' URL viene richiamato il metodo *modificaDatiUtente* all'interno della classe *GestoreUtenti* e il risultato assegnato alla variabile response. Viene, quindi, valutato il risultato: se lo status di response è '200' vuol dire che la richiesta è andata a buon fine e viene restituito l'oggetto utente, se invece lo status è 404 viene restituita la stringa 'Utente non trovato'.

```
PUT /api/v1.0/utenti/{username} HTTP/1.1
Host: localhost:4568
Content-Type: application/json
{
  "nome": "string",
  "cognome": "string",
  "carta": "string"
}
```

Listing 3.5: PUT /utenti/username

- **GestoreUtenti (Backend)**: Contiene il metodo *modificaDatiUtente* che ha come argomenti l'username sotto forma di stringa e l'oggetto utente di cui fa a effettuare l'aggiornamento dei dati. La query viene costruita tramite uno StringBuilder. Dato che ci sono più campi da modificare (nome, cognome e carta) viene dichiarata una variabile booleana primoCampo inizializzata a false, che serve per tenere conto di quale sia la prima delle tre ed inserire le virgole necessarie all'interno della query. Viene quindi eseguita la query e restituito il risultato di quest'ultima.

Diventa Premium Le classi principali sono:

- **UiLogin (FrontEnd)**: Nel caso l'utente sia un cliente base può, tramite un menù, selezionare la voce per diventare un utente premium e richiamare il metodo. *avviaDiventaPremium* all'interno della classe *UiUtente*.
- **UiUtente (FrontEnd)**: Contiene il metodo *avviaDiventaPremium* che mostra un form per la scelta, se l'utente decide di proseguire viene avviato il processo per diventare cliente premium tramite una richiesta HTTP PUT. Se la risposta della richiesta è 200 viene mostrato un messaggio contenente il costo addebitato e restituito l'utente aggiornato, altrimenti un messaggio di errore.
- **RestApi (Backend)**: La classe contiene una richiesta HTTP di tipo PUT. All'interno della classe dopo il parsing del Body della richiesta e l'estrazione del parametro ':username' dall' URL viene richiamato il metodo *diventaPremium* all'interno della classe *GestoreUtenti* e il risultato assegnato alla variabile response. Viene, quindi, valutato il risultato: se lo status di response è '200' vuol dire che la richiesta è andata a buon fine viene restituito il costo del passaggio a premium tramite il metodo *getcostoPremium* all'interno della classe GestorePagamenti, se invece lo status è 404 viene restituita la stringa 'Utente non trovato'.

```
PUT /api/v1.0/utenti/tipo/{username} HTTP/1.1
Host: localhost:4568
Content-Type: application/json
{
  "costo_premium" :0
}
```

Listing 3.6: PUT /utenti/tipo/username

- **GestoreUtenti (Backend)**: Contiene il metodo *diventaPremium* che ha come argomenti l'username sotto forma di stringa. Viene quindi eseguita la query per l'aggiornamento del tipo dell'utente e restituito il risultato di quest'ultima.
- **GestorePagamenti (Backend)**: contiene il metodo *getoCostoPremium* in cui viene eseguita la query per ottenere dal database il costo del diventare premium, la risposta viene poi restituita sotto forma di Integer.

Occupa posto Le classi principali sono:

- **UiLogin (FrontEnd)**: Nel caso l'utente sia un cliente (sia base che premium) può, tramite un menù, selezionare la voce occupa un posto per richiamare il metodo *avviaOccupaPosto*.
- **UiPosteggio (FrontEnd)**: Contiene il metodo *avviaOccupaPosto* all'interno del quale, per prima cosa, viene controllato il fatto che l'utente non abbia già una prenotazioni attiva, nel caso ci fosse viene mostrato il posto a lui dedicato (se in ritardo di più di 30 minuti sul tempo di arrivo varrà applicata la penale) mentre nel caso non ne abbia viene richiamato metodo *mostraFormOccupaPosto*, il quale permette all'utente di inserire il tempo di uscita della prenotazione. Viene

quindi effettuato un controllo sul formato delle date tramite il metodo *controlloFormatoTempo*. I campi all'interno dell'oggetto Prenotazione vengono quindi aggiornati con i nuovi valori, l'oggetto viene poi convertito in una stringa JSON, la quale farà parte della richiesta HTTP POST. Se la risposta della richiesta è 201 viene restituito un messaggio di conferma e la prenotazione, altrimenti un messaggio di errore.

- **RestApi (Backend)**: La classe contiene una richiesta HTTP di tipo POST. All'interno della classe dopo il parsing del Body della richiesta e l'estrazione del parametro ':username' dall' URL viene chiamato il metodo *creaPrenotazione* all'interno della classe *GestorePosti* e il risultato assegnato alla variabile nuovaPrenotazione. Viene, quindi, valutato il risultato: se la variabile è diversa da null vuol dire che la richiesta è andata a buon fine viene restituita la prenotazione, se invece è uguale a null viene restituita la stringa 'Nessun posto disponibile nel periodo richiesto'.

```
POST /api/v1.0/prenotazioni/{username} HTTP/1.1
Host: localhost:4568
Content-Type: application/json
{
  "utente" :"utente",
  "posto" :6,
  "tempo_arrivo" :"2024-01-30 15:34:22",
  "id" :0,
  "tempo_uscita" :"2024-01-30 15:34:22"
}
```

Listing 3.7: POST /prenotazioni/username

- **GestorePosti (Backend)**: Contiene il metodo *creaPrenotazione* che ha come argomenti l'oggetto Prenotazione, il tipo dell'utente e la provenienza. Dato che la provenienza è 'occupa' si entra nel primo if. Vengono quindi ottenuti tutti gli id dei posti e tutte le prenotazioni, vi è quindi un controllo della disponibilità di un posto tramite il metodo *verificaDisponibilita*. Se c'è disponibilità viene effettuata una query sul database per inserire la nuova prenotazione, che viene poi restituita.

Prenota posto Le classi principali sono:

- **UiLogin (FrontEnd)**: Nel caso l'utente sia un cliente premium può, tramite un menù, selezionare la voce per prenotare un posto e far partire il metodo.

- **UiPosteggio (FrontEnd)**: Contiene il metodo *avviaPrenotaPosto* all'interno del quale, per prima cosa, viene controllato il fatto che l'utente non abbia già una prenotazioni attiva, nel caso ci fosse viene mostrato il posto a lui dedicato mentre nel caso non ne abbia viene avviato il meteodo *mostraFormPrenotaposto*, il quale permette all'utente di inserire il tempo di arrivo e di uscita della prenotazione. Viene quindi effettuato un controllo sul formato delle date tramite il metodo *controlloFormatoTempi*. I campi all'interno dell'oggetto Prenotazione vengono quindi aggiornati con i nuovi valori, l'oggetto viene poi convertito in una stringa JSON, la quale farà parte della richiesta HTTP POST. Se la risposta della richiesta è 201 vengono restituiti un messaggio di conferma e la prenotazione, altrimenti un messaggio di errore.

- **RestApi (Backend)**: La classe contiene una richiesta HTTP di tipo POST. All'interno della classe dopo il parsing del Body della richiesta e l'estrazione del parametro ':username' dall' URL viene chiamato il metodo *creaPrenotazione* all'interno della classe *GestorePosti* e il risultato assegnato alla variabile nuovaPrenotazione. Viene, quindi, valutato il risultato: se la variabile è

diversa da null vuol dire che la richiesta è andata a buon fine viene restituita la prenotazione, se invece è uguale a null viene restituita la stringa 'Nessun posto disponibile nel periodo richiesto'.

```
POST /api/v1.0/prenotazioni/premium/{username} HTTP/1.1
Host: localhost:4568
Content-Type: application/json
{
  "utente" :"utente",
  "posto" :6,
  "tempo_arrivo" :"2024-01-30 15:34:22",
  "id" :0,
  "tempo_uscita" :"2024-01-30 15:34:22"
}
```

Listing 3.8: POST /prenotazioni/premium/username

- **GestorePosti (Backend)**: Contiene il metodo *creaPrenotazione* che ha come argomenti l'oggetto Prenotazione, il tipo dell'utente e la provenienza. Dato che la provenienza è 'prenota' non si entra nel primo if. Vengono quindi ottenuti tutti gli id dei posti e tutte le prenotazioni, vi è quindi un controllo della disponibilità di un posto tramite il metodo *verificaDisponibilita*. Se c'è disponibilità viene effettuata una query sul database per inserire la nuova prenotazione, che viene poi restituita.

Modifica prenotazione Le classi principali sono:

- **UiLogin (FrontEnd)**: Nel caso l'utente sia un cliente premium può, tramite un menù, selezionare la voce per modificare una prenotazione e far partire il metodo *avviaModificaPrenotazione*.
- **UiPosteggio (FrontEnd)**: Contiene il metodo *avviaModificaPrenotazione* all'interno del quale, per prima cosa, controllato il fatto che l'utente abbia una prenotazione attiva. Nel caso ci fosse una prenotazione da modifica viene richiamato il metodo *mostraFormModificaPrenotazione*, il quale permette all'utente di inserire il tempo di arrivo e di uscita della prenotazione. Viene quindi effettuato un controllo sul formato delle date tramite il metodo *controlloFormatoTempi*. I campi all'interno dell'oggetto Prenotazione vengono quindi aggiornati con i nuovi valori, l'oggetto viene poi convertito in una stringa JSON, la quale farà parte della richiesta HTTP PUT. Se la risposta della richiesta è 200 vengono restituiti un messaggio di conferma e la prenotazione, altrimenti un messaggio di errore.
- **RestApi (Backend)**: La classe contiene una richiesta HTTP di tipo PUT. All'interno della classe dopo il parsing del Body della richiesta e l'estrazione del parametro ':id' dall' URL viene chiamato il metodo *modificaPrenotazione* all'interno della classe *GestorePosti* e il risultato assegnato alla variabile prenotazione. iene, quindi, valutato il risultato: se la variabile è diversa da null vuol dire che la richiesta è andata a buon fine viene restituita la prenotazione, se invece è uguale a null viene restituita la stringa 'Nessun posto disponibile nel periodo richiesto'.

```
PUT /api/v1.0/prenotazioni/modifica/{id} HTTP/1.1
Host: localhost:4568
Content-Type: application/json
{
  "utente" :"utente",
  "posto" :6,
  "tempo_arrivo" :"2024-01-30 15:34:22",
  "id" :0,
  "tempo_uscita" :"2024-01-30 15:34:22"
}
```

Listing 3.9: PUT /prenotazioni/modifica/id

- **GestorePosti (Backend)**: Contiene il metodo *modifiaPrenotaazione* che ha come argomenti l'oggetto nuovaPrenotazione e vecchiaPrenotazione. Ottiene tutti gli ID dei posti auto e le prenotazioni (da cui viene tolta la vecchia prenotazione), vi è quindi un controllo della disponibilità di un posto tramite il metodo *verificaDisponibilita*. Se c'è disponibilità viene effettuata una query sul database per inserire la nuova prenotazione, che viene poi restituita.

Cancella prenotazione Le classi principali sono:

- **UiLogin (FrontEnd)**: Nel caso l'utente sia un cliente premium può, tramite un menù selezionare la voce per modificare la prenotazione, si apre quindi un form dove l'utente può scegliere di eliminare la prenotazione, se lo fa viene avviato il metodo *avviaCancellaPrenotazione* all'interno dell' UiPosteggio.
- **UiPosteggio (FrontEnd)**: Contiene il metodo *avviaCancellaPrenotazione* all'interno del quale viene chiesto all'utente di confermare la cancellazione, se l'utente conferma parte allora una richiesta HTTP DELETE. Se la risposta della richiesta è 204 viene restituito un messaggio di conferma, altrimenti un messaggio di errore.
- **RestApi (Backend)**: La classe contiene una richiesta HTTP di tipo DELETE. viene chiamato il metodo *cancellaPrenotazione* all'interno della classe GestorePosti e il risultato assegnato alla variabile delete.Viene, quindi, valutato il risultato: se lo status è uguale a '204' vuol dire che la richiesta è andata a buon fine, se invece è uguale a '404' viene restituita la stringa 'Errore nell'eliminazione della prenotazione'.

```
DELETE /api/v1.0/prenotazioni/{id} HTTP/1.1
Host: localhost:4568
```

Listing 3.10: DELETE /prenotazioni/id

- **GestorePosti (Backend)**: Contiene il metodo *cancellaPrenotazione* che ha come argomento una stringa che contiene l'id della prenotazione. All'interno del metodo c'e la query per l'eliminazione della prenotazione dal database, viene quindi restituito il risultato della query.

Aggiorna stato posto All'interno del Backend è presente una classe che corrisponde ad un Client Mqtt sottoscritto al topic sul quale viene pubblicato lo stato del sensore di un posto (*ParkCharge/StatoPosti/#*). Il sensore si può trovare in due stati: occupato o libero. Nel primo caso viene inviato un messaggio per avvisare il BackEnd dell'occupazione del posto. Nel database il posto è etichettato come occupato in modo tale che venga utilizzato da altre prenotazioni. Mentre nel secondo caso il

sensore invia un messaggio per avvisare il BackEnd della liberazione del posto. Nel database il posto è etichettato come libero. La prenotazione che occupava quel posto viene terminata e spostata nello storico, il pagamento è richiesto. Viene quindi richiamato il metodo *effettuaPagamento* che si occupa di calcolare il costo della sosta basandosi sulla prenotazione (tempo di sosta ed eventuale penale) e le possibili ricariche (costo della ricarica e i kiloWatt ricaricati). Viene quindi fatta una publish sul topic del device dell'utente (*ParkCharge/Notifiche/SostaConcluse/User_id*) in modo tale arrivi una notifica contente il pagamento. L'utente, una volta consultato la notifica, tramite un bottone paga la sosta in modo tale che venga fatto una publish sul topic relativo al pagamento (*ParkCharge/Pagamento/User_id*). Tutto questo viene gestito da: emulatore sensori posto (*Sensori*), gestore IoT il quale fa da tramite (*GestoreIoT*), il *Backend* e l'user device (*User*).

3.2 Gruppo 2

I requisiti funzionali a noi assegnati sono stati:

1. Richiedi ricarica
2. Richiedi estensione ricarica
3. Interrompi ricarica
4. Monitora parcheggio
5. Visualizza ricariche
6. Visualizza prenotazioni
7. Visualizza storico
8. Aggiorna stato MWBot

Richiedi ricarica Le classi principali sono:

- **UiLogin (FrontEnd)**: Nel caso l'utente sia un cliente può, tramite un menù selezionare la voce per richiedere una ricarica e far partire il metodo *avviaRichiediRicarica* in UiRicarica.
- **UiRicarica (FrontEnd)**: Contiene il metodo *avviaRichiediRicarica*. Effettua una richiesta HTTP GET per ottenere le informazioni sull'utente richiedente(prenotazione e ricarica in corso).

```
GET /api/v1.0/statoUtente?user='{user}' HTTP/1.1
Host: localhost:4568
{
    "utente": "lverdi",
    "occupazione_iniziata": "si",
    "tempo_arrivo": "2024-06-01 09:00:00",
    "caricando": "no"
    "id_prenotazione": 2
}
```

Listing 3.11: GET /statoUtente?user='user'

Se l'utente può richiedere una ricarica (posteggio in corso e nessuna ricarica in corso) gli viene presentato un form dove può selezionare la percentuale da ricaricare. Viene eseguita una richiesta HTTP POST per richiedere la ricarica. Viene ricevuto un messaggio di risposta che determina

l'avvenuta richiesta o un eventuale errore. Viene invocato EDF per determinare che ricarica effettuare. Viene informato l'MWBot tramite Mqtt.

```
POST /api/v1.0/ricariche?user='{user}'&charge_time='{charge_time}' HTTP/1.1
Host: localhost:4568
```

Listing 3.12: POST /ricariche?user='user'&charge_time='charge_time'

- **RestApi (BackEnd)**: contiene due handlers per la richiesta di ricariche. Il primo GET *statoUtente* ritorna un json contenente informazioni utili per determinare se l'utente può richiedere una ricarica (tempo_arrivo, id_ricarica). Il secondo POST *ricariche* accetta la ricarica richiesta se è accettabile. Per capire se è accettabile invoca il metodo statico *isAcceptable* contenuto nella classe **EDF**. Ritorna al frontend l'esito di questo controllo (bad_request, not_acceptable, ok).
- **EDF (BackEnd)**: l'algoritmo di scheduling per determinare se un job (ricarica) è accettabile e determinare che job eseguire in un determinato momento. Il metodo *isAcceptable* riceve come parametri l'id utente, la percentuale richiesta, la lista delle prenotazioni e la lista delle ricariche (a fini di testing può anche ricevere un LocalDateTime). Ritorna true o false.

Richiedi estensione ricarica Per come è strutturato il codice funziona allo stesso modo di **Richiedi ricarica**. Un utente la cui ricarica à conclusa ne potrà richiedere un'altra nello stesso modo.

Interrompi ricarica

- **UiLogin (FrontEnd)**: Nel caso l'utente sia un cliente può, tramite un menù selezionare la voce per richiedere una ricarica e far partire il metodo *avviaInterrompiRicarica* in UiRicarica.
- **UiRIcarica (FrontEnd)**: contiene il metodo *avviaInterrompiRicarica*. Quando invocato esegue una richiesta HTTP GET per ottenere i dati dell'utente e permettere o meno l'interruzione (un utente che non ha una ricarica in corso non potrà interromperla).

```
GET /api/v1.0/statoUtente?user={user} HTTP/1.1
Host: localhost:4568
```

Listing 3.13: GET /statoUtente?user=user

Se l'utente ha una ricarica in corso gli viene presentato un form con la possibilità di interrompere la ricarica tramite una HTTP DELETE.

```
DELETE /api/v1.0/ricariche?id_prenotazione="id_prenotazione" HTTP/1.1
Host: localhost:4568
```

Listing 3.14: DELETE /ricariche?user=id_prenotazione

- **RestApi**: contiene un handler per DELETE *ricariche*. Un parametro della request è l'id della ricarica da eliminare. Se la ricarica con quel id non esiste, esce e ritorna errore, altrimenti invoca *stopRicaricaByPrenotazione* nella classe **GestoreRicariche**.
- **Gestore Ricariche** il metodo *stopRicaricaByPrenotazione* imposta nel database la percentuale richiesta a quanto erogato fino a quel momento. notifica 'utente dell'avvenuta interruzione della ricarica.

Monitora parcheggio

- **UiLogin** Login amministratore, può richiedere l'avvio del *avviaMonitoraParcheggio* nella classe **UiMonitora**.
- **UiMonitora**: presenta all'amministratore un form con le varie scelte possibili.

Visualizza stato posti Le classi principali sono:

- **UiLogin** Login amministratore, può richiedere l'avvio del *avviaMonitoraParcheggio* nella classe **UiMonitora**.
- **UiMonitora**: presenta all'amministratore un form con le varie scelte possibili. Se viene scelto *Visualizza stato posti* viene mostrata una lista dei posti e se sono liberi o occupati. Queste informazioni sono ottenute del backend tramite una HTTP GET.

```
GET /api/v1.0/posti HTTP/1.1
Host: localhost:4568
[
    {
        "id": 1,
        "disponibilita": 0
    },
    {
        "id": 2,
        "disponibilita": 0
    }
]
```

Listing 3.15: GET /posti

- **RestApi (BackEnd)**: contiene l'handler GET */posti* che ritorna un json dello stato attuale dei posti. Per ottenere tale informazione utilizza il metodo *getStatoPosti* della classe **Gestore Posti**.
- **Gestore Posti (BackEnd)**: contiene il metodo *getStatoPosti* che fa una query al dbPrenotazioni. Ritorna le informazioni ottenute.

Visualizza ricariche

- **UiLogin** Login amministratore, può richiedere l'avvio del *avviaMonitoraParcheggio* nella classe **UiMonitora**.
- **UiMonitora**: presenta all'amministratore un form con le varie scelte possibili. Se viene scelto *Visualizza ricariche in corso* viene mostrata una lista delle ricariche in coda. Queste informazioni sono ottenute del backend tramite una HTTP GET.

```
GET /api/v1.0/ricariche HTTP/1.1
Host: localhost:4568
[
    {
        "kilowatt": 50.0,
        "durata_ricarica": 90,
        "percentuale_richiesta": 80,
        "percentuale_erogata": 0,
        "prenotazione": 1,
        "mwbot": 1
    },
    {
        "kilowatt": 60.5,
        "durata_ricarica": 120,
        "percentuale_richiesta": 90,
        "percentuale_erogata": 0,
        "prenotazione": 2,
        "mwbot": 1
    }
]
```

Listing 3.16: GET /ricariche

- **RestApi (BackEnd)**: contiene l'handler GET */ricariche* che ritorna un json dello stato delle ricariche in coda. Per ottenenere tale informazione utilizza il metodo *getRicariche* della classe **Gestore Ricariche**.
- **Gestore Ricariche (BackEnd)**: contiene il metodo *getRicariche* che fa una query al dbRicariche. Ritorna le informazioni ottenute.

Visualizza prenotazioni

- **UiLogin** Login amministratore, può richiedere l'avvio del *avviaMonitoraParcheggio* nella classe **UiMonitora**.
- **UiMonitora**: presenta all'amministratore un form con le varie scelte possibili. Se viene scelto *Visualizza prenotazioni* viene mostrata una lista dei prenotazioni. Queste informazioni sono ottenute del backend tramite una HTTP GET.

```
GET /api/v1.0/prenotazioni HTTP/1.1
Host: localhost:4568
[
    {
        "id": 2,
        "tempo_arrivo": "2024-06-01 09:00:00",
        "tempo_uscita": "2024-06-01 11:00:00",
        "utente": "lverdi",
        "posto": 2
    },
    {
        "id": 3,
        "tempo_arrivo": "2024-06-01 08:30:00",
        "tempo_uscita": "2024-06-01 09:30:00",
        "utente": "gbianchi",
        "posto": 3
    }
]
```

Listing 3.17: GET /prenotazioni

- **RestApi (BackEnd)**: contiene l'handler GET */prenotazioni* che ritorna un json rappresentante le prenotazioni. Per ottenenere tale informazione utilizza il metodo *getPrenotazioni* della classe **Gestore Posti**.
- **Gestore Posti (BackEnd)**: contiene il metodo *getPrenotazioni* che fa una query al dbPrenotazioni. Ritorna le informazioni ottenute.

Visualizza storico

- **UiLogin** Login amministratore, può richiedere l'avvio del *avviaMonitoraParcheggio* nella classe **UiMonitora**.
- **UiMonitora**: presenta all'amministratore un form con le varie scelte possibili. Se viene scelto *Visualizza storico* viene mostrato un filtro per data, una volta inseriti i dati viene mostrata una lista delle prenotazioni concluse nella data selezionata. Queste informazioni sono ottenute del backend tramite una HTTP GET.

```
GET /api/v1.0/storico?year='year'&month='month' HTTP/1.1
Host: localhost:4568
[
    {
        "utente": "mrossi",
        "penale": 50,
        "posto": 1,
        "costo": 1,
        "costo_posteggio": 5.0,
        "tempo_arrivo": "2024-06-01 08:00:00",
        "costo_ricarica": 10.0,
        "ricarica": 1,
        "id": 1,
        "tempo_uscita": "2024-06-01 10:00:00"
    },
    {
        "utente": "lverdi",
        "penale": 45,
        "posto": 2,
        "costo": 2,
        "costo_posteggio": 4.5,
        "tempo_arrivo": "2024-06-01 09:00:00",
        "costo_ricarica": 9.0,
        "ricarica": 2,
        "id": 2,
        "tempo_uscita": "2024-06-01 11:00:00"
    }
]
```

Listing 3.18: GET /storico

- **RestApi (BackEnd)**: contiene l'handler GET */storico* che ritorna un json dello storico. Per ottenenere tale informazione utilizza il metodo *getStorico* della classe **Gestore Pagamenti**.
- **Gestore Pagamenti (BackEnd)**: contiene il metodo *getStorico* che fa una query al dbStorico. Ritorna le informazioni ottenute.

Aggiorna stato MWBot All'interno del Backend 'e presente una classe che corrisponde ad un Client Mqtt sottoscritto al topic sul quale viene pubblicato lo stato del MWBot (ParkCharge/StatoRicariche/#). L'MWBot comunica il proprio stato in 3 casi:

- **L'MWBot inizia la ricarica assegnatagli**: in questo caso l'MWBot comunica al Backend che ha ricevuto la richiesta di ricarica. Il campo *KW_Emessi* sarà zero e *statoCarica* sarà "Charging". In questo caso il Backend non reagisce ma rimane in attesa di messaggi futuri.
- **L'MWBot ricarica una certa percentuale** in questo caso l'MWBot comunica al Backend che ha completato di ricaricare l'1% in un certo posto. Il Backend incrementa di uno a database la percentuale erogata alla ricarica in corso. Questo è necessario per due motivi: l'algoritmo di scheduling è preemptive e una ricarica potrebbe essere interrota in qualsiasi momento dall'utente. In questo modo vi è una sincronia continua tra MWBot e database. Dopodiché viene invocato EDF (algoritmo di scheduling) per capire se mantenere l'MWBot sulla ricarica corrente o spostarlo in un posto diverso.

- **L'MWBot completa la ricarica assegnatagli** in in questo caso l'MWBot comunica al Backend che ha completato la ricarica e si trova in uno stato idle. Il campo *KW_Emessi* sarà uguale alla percentale richiesta e *statoCarica* sarà "Finito". Il Backend farà publish di una notifca per l'utente che comunica la conclusione della ricarica e il costo delle ricarica conclusa. Dopodiché viene invocato EDF per comunicare al MWBot il nuovo job che potrebbe anche essere idle se non sono presento ricariche in coda.

3.3 GestoreIoT

Il GestoreIoT è stato progettato per fungere da intermediario tra gli emulatori e il backend. La sua principale responsabilità è quella di leggere e scrivere sui vari topic MQTT, gestendo le comunicazioni tra i diversi componenti del sistema. Dato che il GestoreIoT deve gestire diversi comportamenti in base al topic su cui opera, abbiamo deciso di implementare il **Pattern Strategy**. Questo approccio ci ha permesso di separare la logica specifica per ciascun topic in strategie distinte, rendendo il codice più modulare e facilmente estendibile. I concetti chiave sono:

1. **Context**: è la classe che va ad utilizzare le varie strategie; nel nostro caso si tratta di *GestoreIoT*.
2. **Strategy**: è un interfaccia che definisce un metodo comune per le *concrete strategis* e nel nostro caso sarà la gestione del messaggio (*handleMessage*).
3. **Concrete Strategies**: sono le implementazioni delle strategie; nel nostro caso *SensorStrategy*, *MWBotStrategy* e *CommandStrategy*.

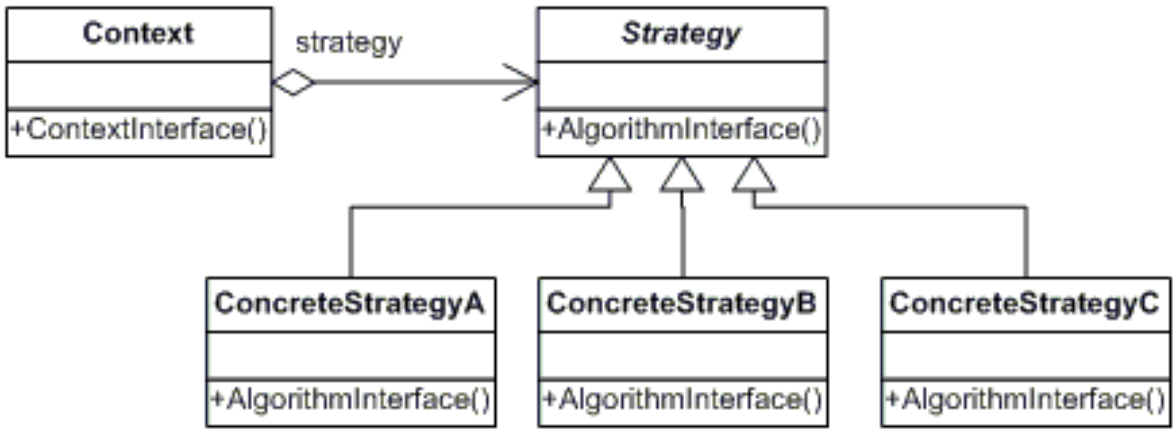

Figure 3.1: Diagramma UML del Pattern Strategy

Un vantaggio è che se in futuro dovessimo supportare nuovi tipi di dispositivi o operazioni, si potranno aggiungere nuove strategie senza modificare il codice esistente. Questo riduce il rischio di introdurre bug e semplifica l'aggiunta di nuove funzionalità.

3.4 Diagramma delle classi completo

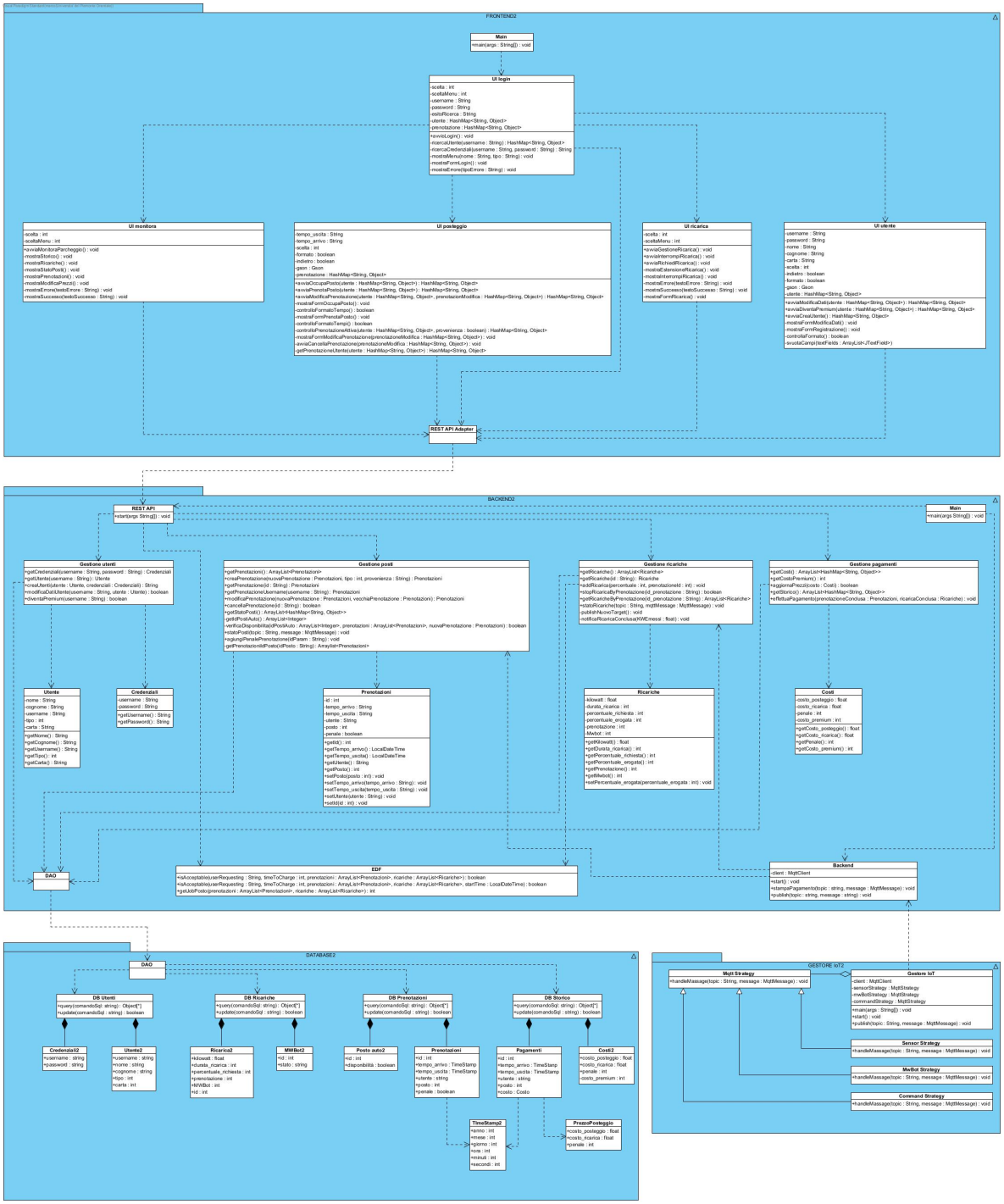

Figure 3.2: Diagramma delle classi completo

Chapter 4

Conclusioni

4.1 Sviluppi Futuri

Il progetto attuale presenta numerose potenzialità per futuri sviluppi e miglioramenti. Una delle principali direzioni verso cui si potrebbe evolvere è la separazione del database dal backend, trasformandolo in un microservizio indipendente. Questa architettura consentirebbe di migliorare la scalabilità e la flessibilità del sistema. In particolare, il database diventerebbe un'entità autonoma con cui i vari emultatori potrebbero interagire direttamente, facilitando così operazioni full duplex. Questo approccio non solo snellirebbe il flusso di dati, ma renderebbe anche più semplice l'integrazione di nuovi componenti nel sistema senza dover intervenire pesantemente sul backend esistente.
Un altro importante sviluppo futuro riguarda il redesign del totem attualmente in uso. Alcune operazioni, come la modifica della prenotazione, potrebbero essere rimosse dall'interfaccia del totem e delegate a una nuova applicazione mobile. Questa app, basata su Android, offrirebbe agli utenti una nuova modalità di interazione con il sistema di parcheggio smart, fungendo da front-end alternativo. Grazie a questa app, gli utenti avrebbero la possibilità di gestire le loro prenotazioni e interagire con il sistema in modo più intuitivo e conveniente, senza dover necessariamente passare attraverso il totem fisico. Questa soluzione non solo migliorerebbe l'esperienza utente, ma consentirebbe anche di ridurre i costi di manutenzione e aggiornamento dei totem stessi.

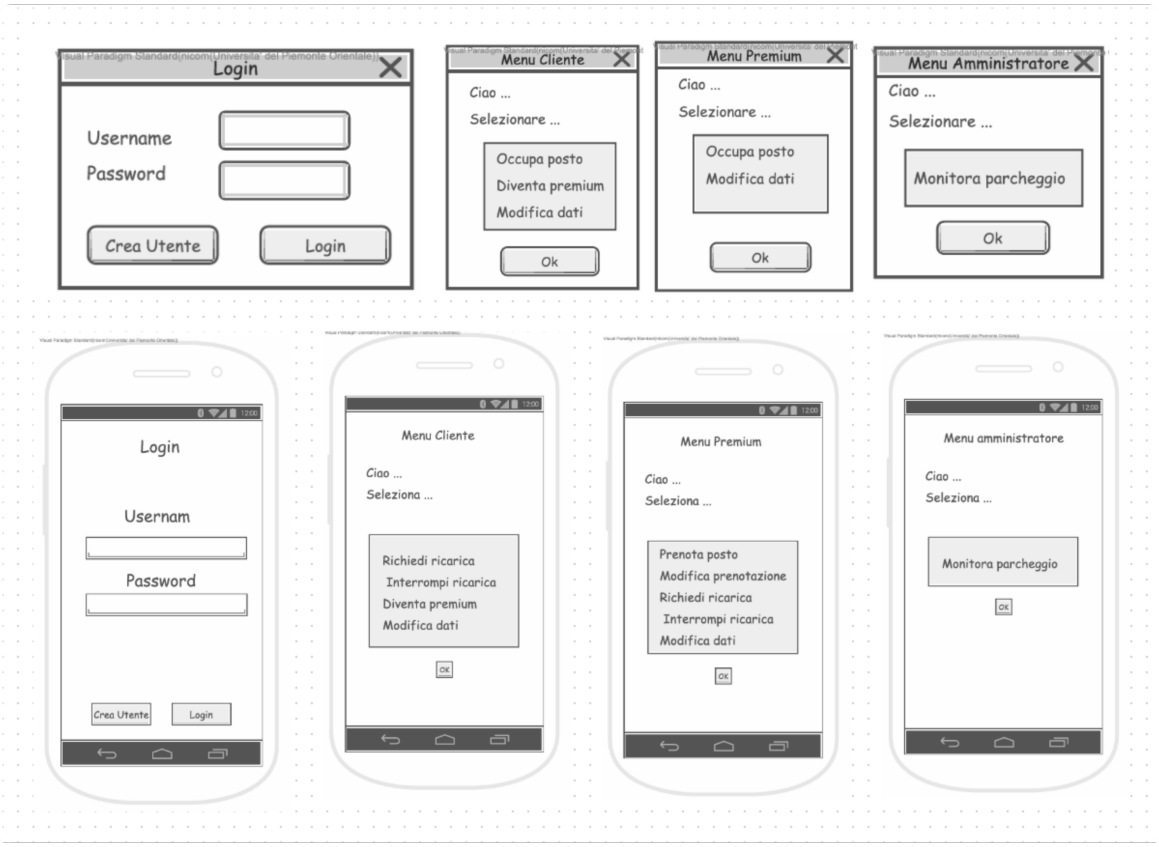

Figure 4.1: Re-Design Totem e Android App