
Programmazione III e Laboratorio di Programmazione 3

Proff: A.Ciaramella e R.Montella

Anno accademico 2020/2021



Progetto d'Esame



Taxi Finder

A cura di Roberto Vecchio

Indice:

1.0 - Scoperta dei requisiti	3
1.1 - Il problema	3
1.2 - Scenari	3
1.3 - Requisiti funzionali e non funzionali	5
1.4 - Diagramma dei casi d'uso	7
2.0 Class Diagram.....	8
2.1 MVC.....	8
2.2 Diagramma delle classi	11
2.3 Package View e FXML.....	12
2.4 Package Controller.....	15
2.5 Package Model.....	16
2.6 Singleton e persistenza dei dati	21
2.7 Observer pattern	22
2.8 Strategy pattern.....	23
3.0 Il problema dello Shortest Path.....	26
3.1 Shortest Path.....	26
3.2 Grafo.....	26
3.3 Descrizione algoritmo	27
4.0 Principi SOLID	31
5.0 Manuale Utente.....	32
5.1 Mappa di default	32
5.2 Interfaccia utente.....	34

1.0 - Scoperta dei requisiti

1.1 - Il problema

Si vuole sviluppare un'applicazione per la gestione di Taxi. Si suppone di avere una flotta di automobili posizionate in appositi parcheggi distribuiti in una città. Un cliente può prenotare un Taxi (Via sms o e-mail) e attendere presso una opportuna postazione non necessariamente corrispondente a quella dove sostano i taxi. Il taxi che deve effettuare la corsa viene scelto tra quelli liberi ed è quello che garantisce il percorso più breve per raggiungere la postazione (usare l'algoritmo di Dijkstra).

Il taxi può scegliere il percorso per raggiungere la destinazione secondo due strategie (usare l'algoritmo di Dijkstra):

- percorso con meno traffico. Su ogni strada sono previsti due sensori che calcolano il tempo impiegato da un'auto per percorrere quel tratto;
- percorso più breve.

Implementare l'applicazione garantendo le opportune interfacce grafiche.

1.2 - Scenari

Scenario 1 - Cliente

Carlo è un designer e oggi si deve recare urgentemente in ufficio per completare alcuni dei lavori rimasti inadempiti, prima che vi sia la scadenza di consegna progetto. Purtroppo la sua auto è fuori uso, per cui decide di prendere un Taxi e cerca opportunamente un'applicazione per prenotare una corsa, trovando **Taxi Finder**.

Si registra velocemente come cliente inserendo opportunamente le sue credenziali. Il processo di registrazione va a buon fine, infatti l'interfaccia gli mostra l'area dove potrà accedere.

Inserendo le credenziali appena registrate, viene rimandato in home dove nota che può effettuare velocemente una prenotazione di un Taxi presso una postazione vicino casa sua.

Poco dopo aver prenotato la corsa, nota che il suo ordine di prenotazione è passato da “in lavorazione” ad “accettato”, per cui si reca alla postazione e prende il taxi pochi minuti dopo.

Scenario 2 - Tassista

Francesco è un tassista, dipendente della società Taxi Finder. Dopo aver sostato per circa trenta minuti in uno dei parcheggi dell'azienda, viene notificato dal sistema; è stato associato ad un nuovo ordine di prenotazione, data la vicinanza del parcheggio con la postazione del cliente.

Francesco potrebbe decidere di raggiungere il cliente attraverso il percorso più breve oppure attraverso il percorso con meno traffico, fornитogli dal sistema.

Visto che si trova in un orario di punta, decide la seconda alternativa, in modo da arrivare il prima possibile verso la postazione richiesta. Pochi minuti dopo, prende in carico il cliente e lo porta a destinazione.

Scenario 3 - Gestore

Alberto è uno dei gestori di Taxi Finder ed il suo compito è quello di monitorare il lavoro dei tassisti, eventualmente assumerli o congedarli, gestire parcheggi e postazioni di attesa quando l'azienda decide di insediarne nuovi.

Il sistema, appena un gestore vi entra, mostra la lista dei tassisti con le corrispettive informazioni ed il loro stato (libero o occupato), dipendentemente da ciò che stanno facendo in quel momento; inoltre i gestori possono guardare la lista dei parcheggi e postazioni con corrispettive informazioni annesse.

Oltre a queste informazioni, i gestori possono sapere, attraverso il sistema, dove sta sostando un tassista ed il taxi ad esso associato; inoltre possono vedere quali tassisti sono in un determinato parcheggio e i diversi collegamenti tra un parcheggio / postazione ed un altro.

Scenario 4 - Gestore (Assunzione di un tassista)

Luca ha saputo che Taxi Finder sta cercando del personale. Il soggetto in questione ha già una licenza da tassista e un taxi, che costituiscono di fatto alcuni dei prerequisiti per essere assunti. Invia, dunque, il curriculum presso la loro sede ed attende una risposta.

Il direttore delle risorse umane resta molto colpito da Luca, per cui, di comune accordo, chiede ad Alberto (Gestore - guardare scenario 3) di

procedere con l'inserimento di quest'ultimo nel sistema, così che possa iniziare quanto prima.

Alberto utilizza opportunamente l'interfaccia, inserendo le generalità di Luca ed aggiungendo il taxi che possiede, così da poterli associare nel sistema. Il processo di assunzione va a buon fine e le credenziali (username: codice fiscale e password) gli vengono fornite in forma cartacea in modo che le possa portare sempre con sé.

Inoltre gli viene detto in quale parcheggio si deve recare per aspettare il prossimo cliente.

1.3 - Requisiti funzionali e non funzionali

I seguenti requisiti funzionali (FR) e non funzionali (NFR) devono essere affrontati nel progetto:

FR1 - Creare un profilo: Il cliente deve poter registrare un profilo, capace di memorizzare le informazioni ad esso associate ed effettuarne l'accesso.

FR2 - Ordine di prenotazione: Il cliente deve poter prenotare una corsa presso una determinata postazione di attesa, attraverso una due delle modalità previste da politiche aziendali: SMS o Email.

FR3 - Storico prenotazioni: Il cliente ed il tassista devono poter visualizzare il proprio storico delle prenotazioni, con informazioni annesse.

FR4 - Assumere Tassisti: il gestore deve poter utilizzare il software per assumere nuovi tassisti che possono essere gestiti dal sistema.

FR5 - Congedare Tassisti: Il gestore deve poter utilizzare il software per congedare tassisti, in modo tale che non siano gestiti dal sistema.

FR6 - Aggiungere / rimuovere parcheggi: I gestori devono poter aggiungere o rimuovere parcheggi secondo le disponibilità aziendali, in modo che possano essere gestiti dal sistema.

FR7 - Aggiungere / rimuovere postazioni: Il gestore deve poter aggiungere o rimuovere postazioni di attesa secondo le disponibilità aziendali, in modo che possano essere gestiti dal sistema.

FR8 - Aggiungere / rimuovere collegamenti tra postazioni: Il gestore deve poter aggiungere o rimuovere collegamenti tra postazioni di attesa o parcheggi, in modo che il sistema possa mappare i percorsi più brevi per raggiungere un determinato luogo.

FR9 - Visualizzare Tassisti: Il gestore deve poter visualizzare la lista di tutti i tassisti con informazioni annesse ed il loro stato in modo da capire chi è attualmente libero o occupato con un cliente.

FR10 - Visualizzare Parcheggi: Il gestore deve poter visualizzare la lista di tutti i parcheggi con informazioni annesse.

FR11 - Visualizzare Postazioni: Il gestore deve poter visualizzare la lista delle postazioni con informazioni annesse .

FR12 - Notificare Tassista: Il sistema deve notificare il tassista locato nel parcheggio più vicino quando viene effettuato un ordine di prenotazione.

FR13 - Scegliere Modalità Raggiungimento: Il tassista una volta ricevuto l'ordine deve poter scegliere come raggiungere la postazione, secondo due modalità previste dalle politiche aziendali: Percorso più breve (in km) o percorso con meno traffico (minuti di percorimento). Una volta scelta la modalità, al tassista viene indicato il percorso da seguire.

NFR1 - Usabilità: l'app dovrebbe essere intuitiva da usare e l'interfaccia utente dovrebbe essere semplice da capire, vista la natura gestionale. Tutte le interazioni devono essere completate in meno di tre click.

NFR3 - Affidabilità: Il sistema deve essere sempre performante.

Vincoli aggiuntivi:

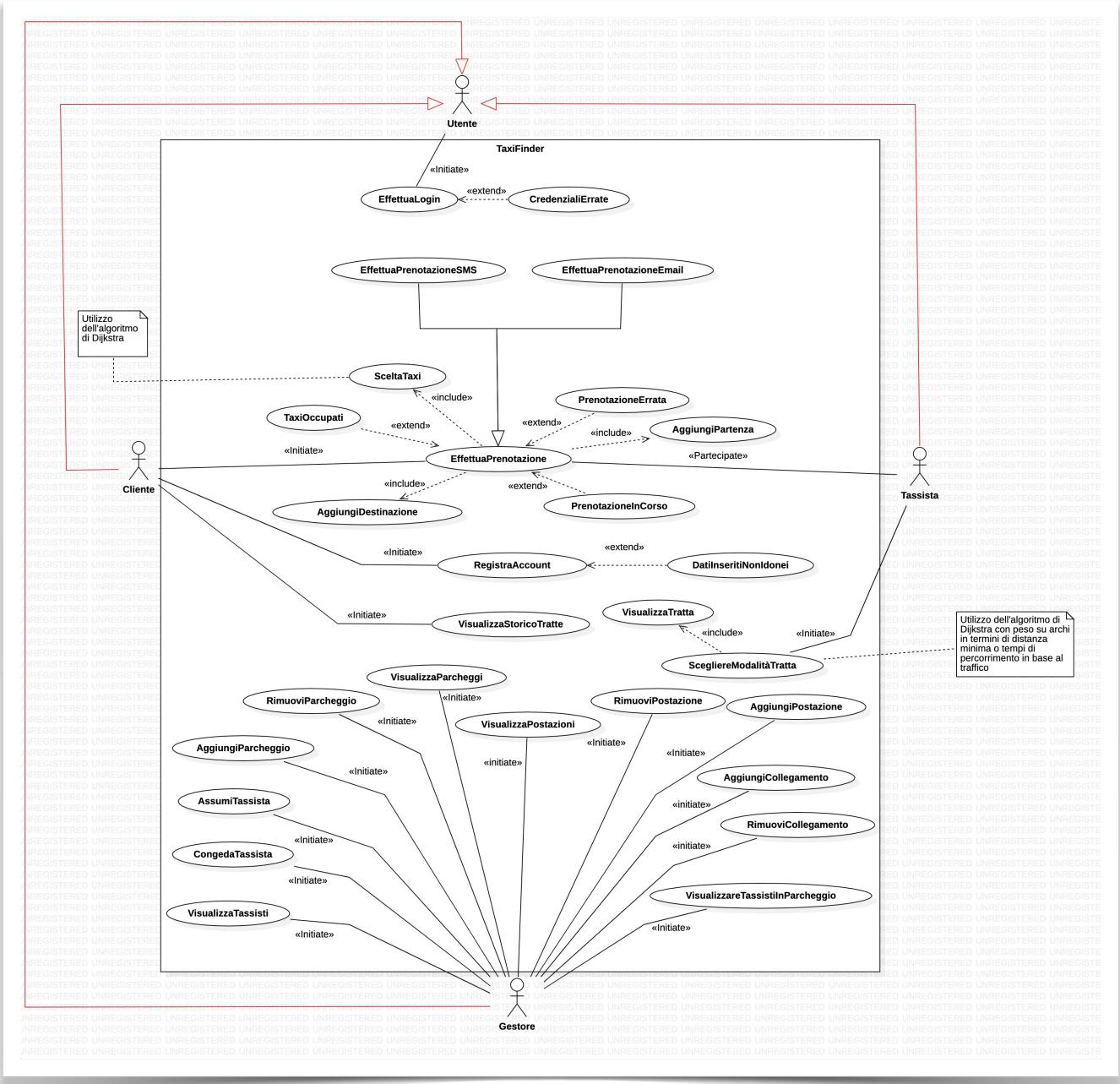
- Il linguaggio utilizzato deve essere **Java**;
- Inserire **sufficienti commenti**;
- Generare documentazione in HTML con tool **javadoc**;
- Utilizzare **annotazioni, eccezioni e file/database**;
- Il rispetto dei principi della programmazione **SOLID** (Single Responsibility Principle (**SRP**)), Open-Closed Principle (**OCP**), Liskov Substitution Principle (**LSP**), Interface Segregation Principle (**ISP**), Dependency Inversion Principle (**DIP**)
- Utilizzare almeno **due pattern** tra i design pattern noti;
- Utilizzare **l'algoritmo di Dijkstra** per capire qual'è il tassista locato al parcheggio più vicino, qual'è il percorso più breve e con meno traffico.

1.4 - Diagramma dei casi d'uso

Dopo aver opportunamente analizzato la traccia fornita, aver espletato alcuni scenari ed analizzato i requisiti funzionali e non, con vincoli annessi, si evince la natura gestionale del sistema da sviluppare, dove risaltano tre attori che interagiscono con il software:

Gestore	Si occupa di gestire e monitorare le entità che interagiscono nel sistema in questione
Tassista	Si occupa di prendere in carico delle prenotazioni e di attendere opportunamente il cliente successivo presso il parcheggio scelto
Cliente	Figura non aziendale ma che interagisce con il sistema per poter effettuare una prenotazione di una corsa

Quanto è stato elencato nei paragrafi precedenti può essere quindi documentato attraverso l'utilizzo dell'UML (**Unified Modeling Language**), in particolare attraverso il diagramma dei casi d'uso, che mostra le interazioni tra l'ambito della descrizione e le entità a esso esterne. Di seguito l'Use case Diagram di Taxi Finder:



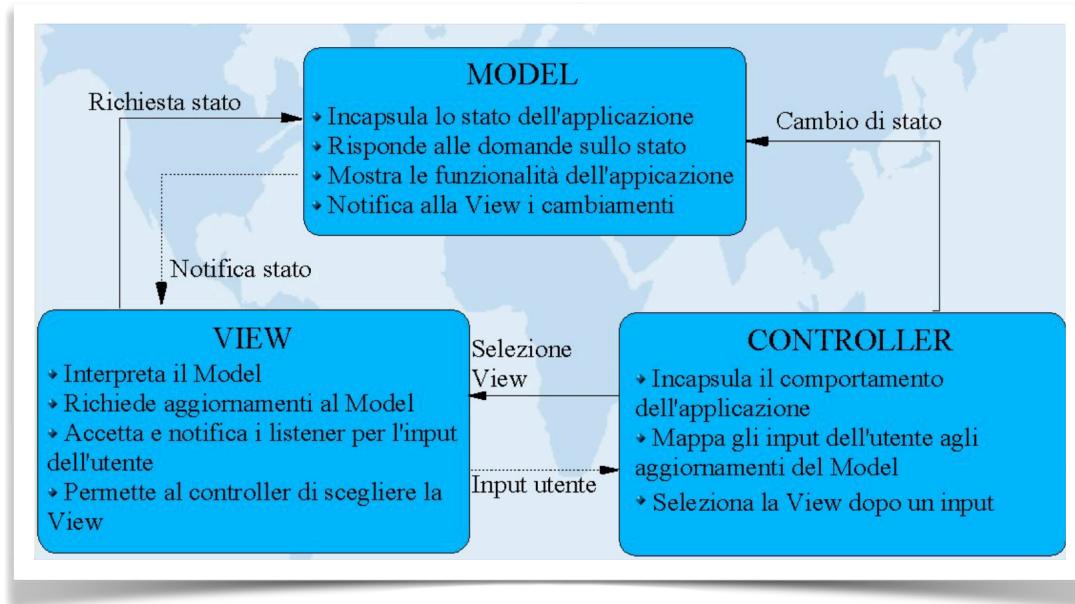
2.0 Class Diagram

2.1 MVC

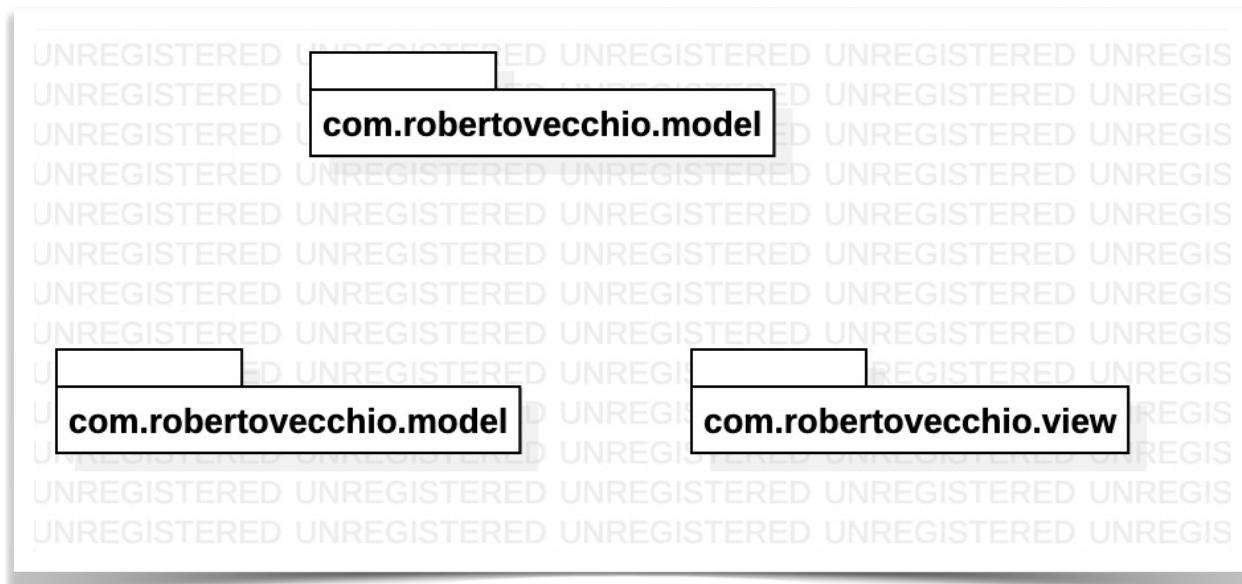
Per questo progetto è stato impostato un pattern architetturale, il quale opera ad un livello diverso e più ampio rispetto ai design pattern. I pattern architetturali esprimono schemi di base per impostare l'organizzazione strutturale di un sistema software.

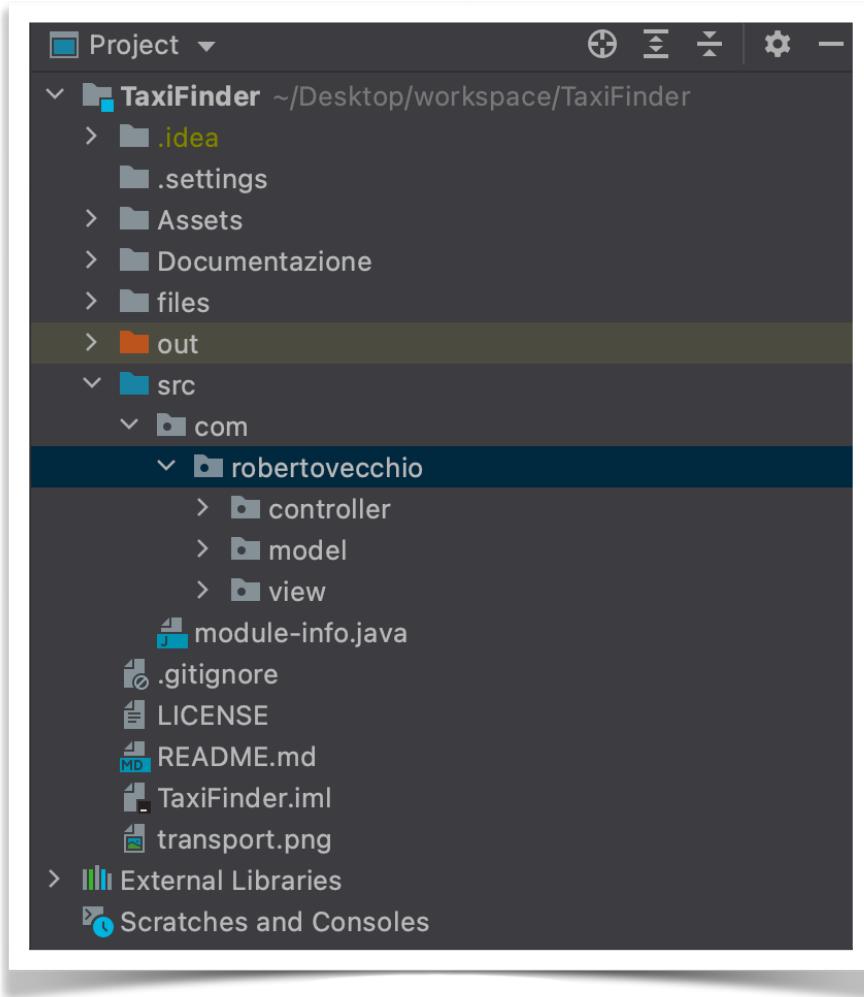
È stato implementato il Model-View-Controller (**MVC**) il quale è un pattern architetturale che ha lo scopo di separare i componenti software di un'applicazione in:

- Funzionalità di business (**model**);
 - Logica di presentazione (**view**)
 - Controllo che utilizzano tali funzionalità(**controller**)



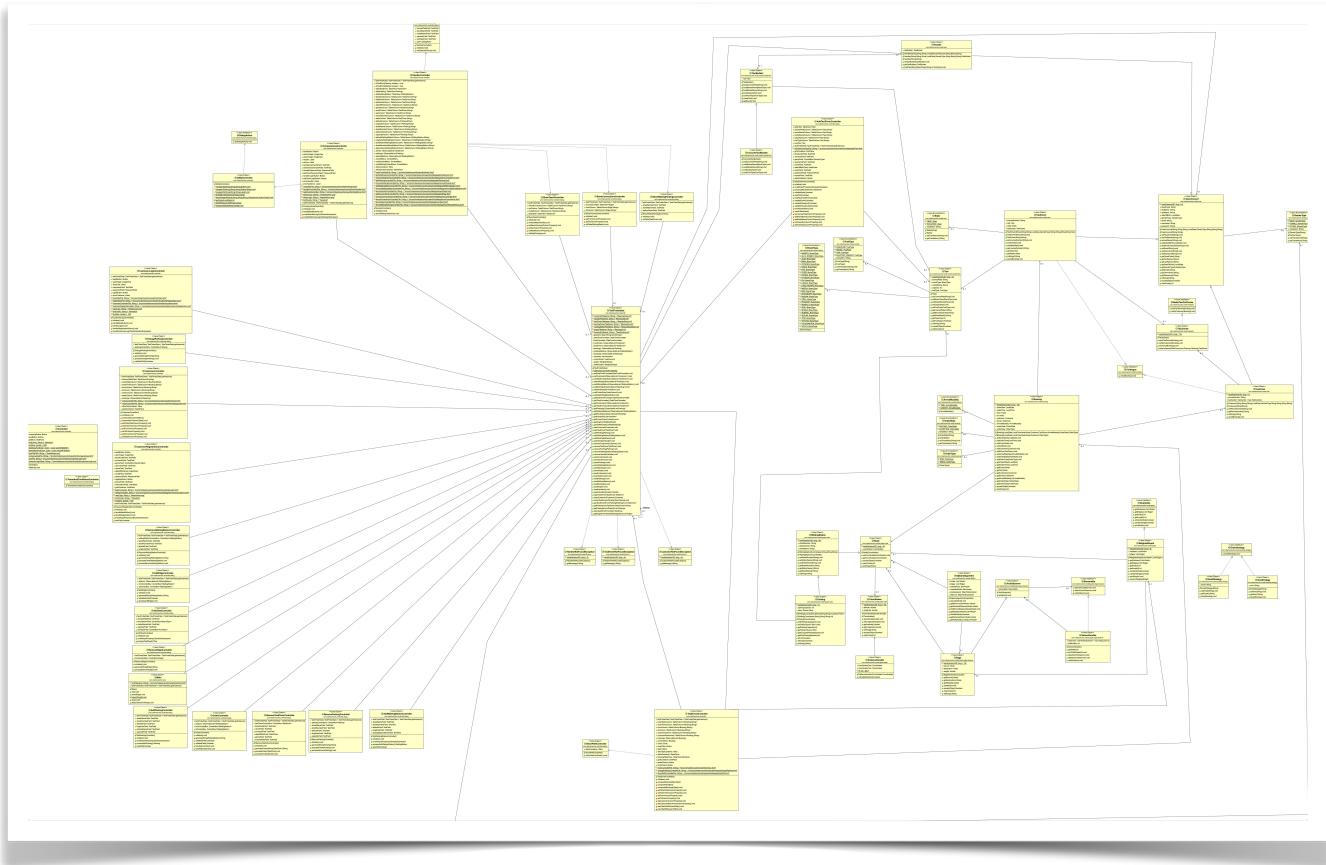
Tale pattern architetturale è stato implementato nel progetto di Taxi Finder in quanto funge da linea guida per i principi SOLID ed in particolare per il Single Responsibility Principle (SRP). Infatti ogni elemento del programma gode di una singola responsabilità e tale viene interamente encapsulata dall'elemento stesso. Di seguito alcune immagini che mostrano la sua implementazione:





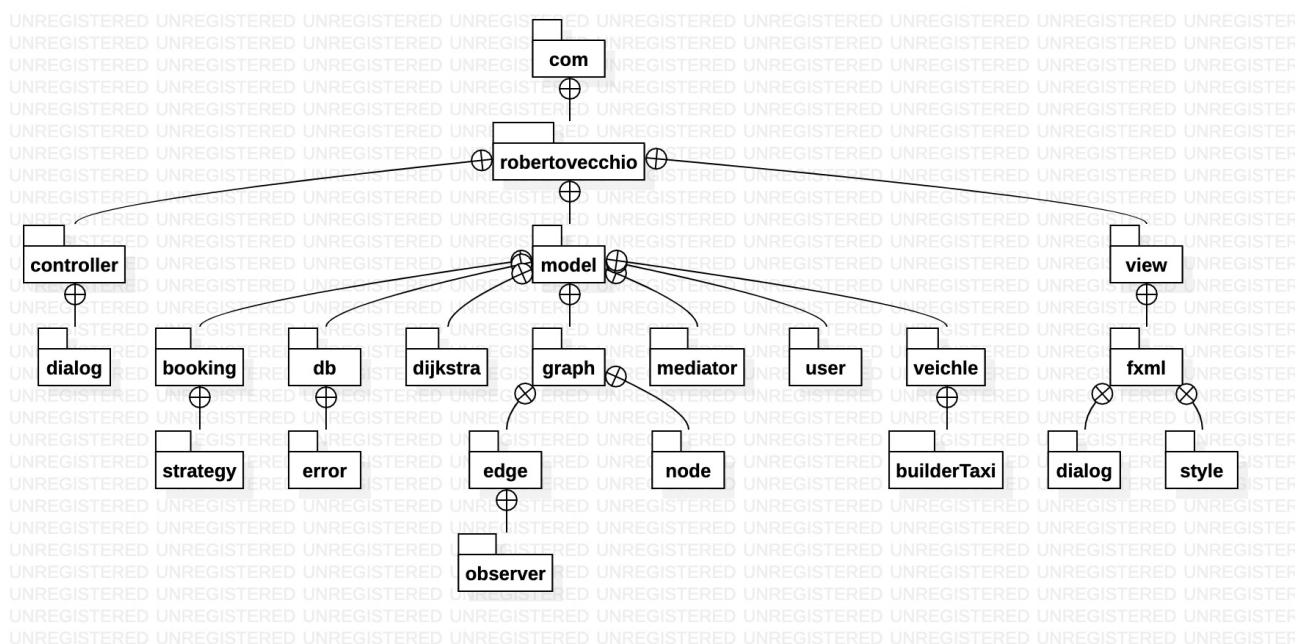
2.2 Diagramma delle classi

Il sistema software Taxi Finder presenta il seguente diagramma delle classi:



N.b è possibile visualizzare l'immagine dettagliatamente nella folder di progetto.

Le classi implementate sono state suddivise in package per migliorare la leggibilità della struttura del progetto e per evitare codice ridondante. I package sono organizzati secondo la seguente struttura :



2.3 Package View e FXML

Vista la natura gestionale di Taxi Finder si è deciso di sviluppare il sistema software come programma standalone con supporto grafico. In particolare, si è deciso di utilizzare la libreria grafica JavaFX, in quanto consente di sviluppare applicazioni dotate di un'interfaccia utente costruita attraverso un insieme di componenti UI.

Un altro motivo per cui è stato deciso l'utilizzo di JavaFx risiede nel fatto che facilita e sprona all'utilizzo del pattern architetturale MVC.

Attraverso la libreria in questione, è stato possibile implementare interfacce grafiche attraverso l'utilizzo di file dotati di estensione **FXML**, i quali sfruttano il linguaggio xml per abbreviare i tempi di sviluppo dell' UI.

Di seguito possiamo vedere come è stata sviluppata l'interfaccia di login del cliente:

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.layout.*?>
<?import javafx.scene.controlToolBar?>
<?import javafx.scene.control.Button?>
<?import javafx.scene.control.Tooltip?>
<?import javafx.scene.image.ImageView?>
<?import javafx.scene.image.Image?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.control.TextField?>
<?import javafx.scene.control.PasswordField?>
<?import javafx.scene.text.Font?>
<BorderPane xmlns="http://javafx.com/javafx"
             xmlns:fx="http://javafx.com.fxml"
             fx:controller="com.robertoveccchio.controller.CustomerLoginController">

    <top>
        <ToolBar>
            <HBox>
                <Button text="Indietro" onAction="#handleBackButton" fx:id="backButton">
                    <tooltip>
                        <Tooltip text="Torna indietro"/>
                    </tooltip>
                    <graphic>
                        <ImageView>
                            <Image url="@/toolbarButtonGraphics/navigation/Back16.gif"/>
                        </ImageView>
                    </graphic>
                </Button>
            </HBox>
        </ToolBar>
    </top>
    <center>
        <VBox fx:id="vboxUser" alignment="CENTER">
            <GridPane alignment="CENTER" vgap="15" hgap="10">

                <ImageView fx:id="userImage"
                           GridPane.rowIndex="0"
                           GridPane.columnIndex="0"
                           GridPane.columnSpan="2"
                           GridPane.halignment="CENTER"/>
            </GridPane>
        </VBox>
    </center>
</BorderPane>
```

```

<Label text="Login Utente"
       GridPane.rowIndex="1"
       GridPane.columnIndex="0"
       GridPane.columnSpan="2"
       GridPane.halignment="CENTER" textFill="#494949">
    <font>
        <Font name="Helvetica Bold" size="20"/>
    </font>
</Label>

<Label text="Username:"
       GridPane.rowIndex="2"
       GridPane.columnIndex="0"/>

<TextField promptText="es. Roberto"
            GridPane.rowIndex="2"
            fx:id="usernameField"
            GridPane.columnIndex="1"/>

<Label text="Password:"
       GridPane.rowIndex="3"
       GridPane.columnIndex="0"/>

<PasswordField promptText="es. 123"
                GridPane.rowIndex="3"
                fx:id="passwordField"
                GridPane.columnIndex="1"/>

<Button text="Login"
        GridPane.rowIndex="4"
        GridPane.columnIndex="0"
        GridPane.columnSpan="2"
        GridPane.halignment="CENTER"
        fx:id="loginButton"
        onAction="#handleLogin"
        prefWidth="240">
    <graphic>
        <Image>
            <Image url="@/toolbarButtonGraphics/general/Open24.gif"/>
        </Image>
    </graphic>
</Button>

<Label text="Credenziali Errate"
       GridPane.rowIndex="5"
       GridPane.columnIndex="0"
       GridPane.columnSpan="2"
       textFill="#E04F5F"
       visible="false"
       fx:id="errorCustomer"
       GridPane.halignment="CENTER">
    <font>
        <Font name="helvetica bold" size="15"/>
    </font>
</Label>

</GridPane>
</VBox>
</center>
</BorderPane>

```

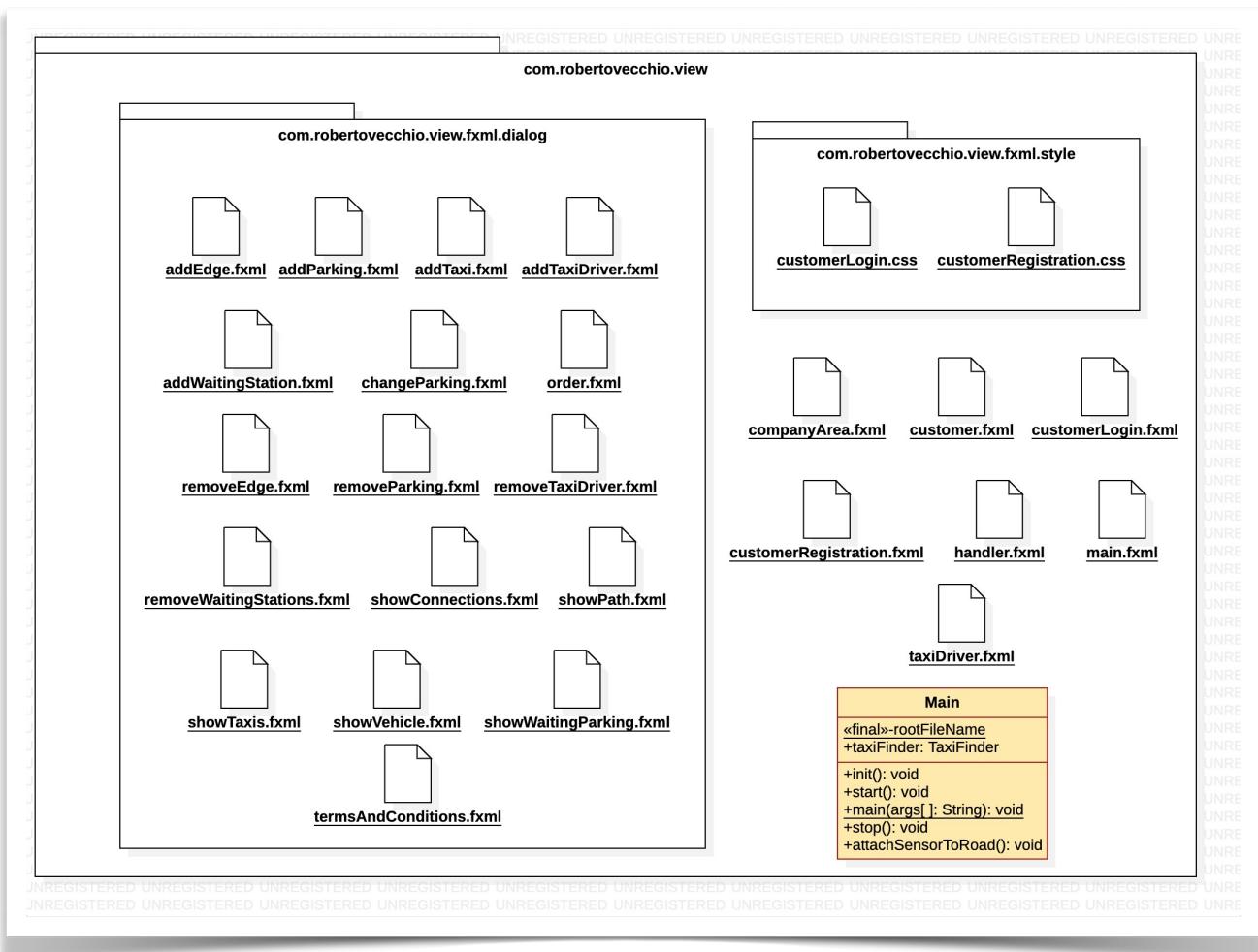
Tale view, prima di poter essere visualizzata a schermo, sfrutta la classe **FXMLLoader**, la quale è responsabile di caricare un file sorgente **FXML**, effettuarne il parsing e ritornare un oggetto di tipo grafo, come possiamo notare dal seguente screen tratto dal progetto:

```
Stage stage = (Stage)button.getScene().getWindow();

FXMLLoader loader = new FXMLLoader();
try {
    Parent root = loader.load(new FileInputStream(fileName));

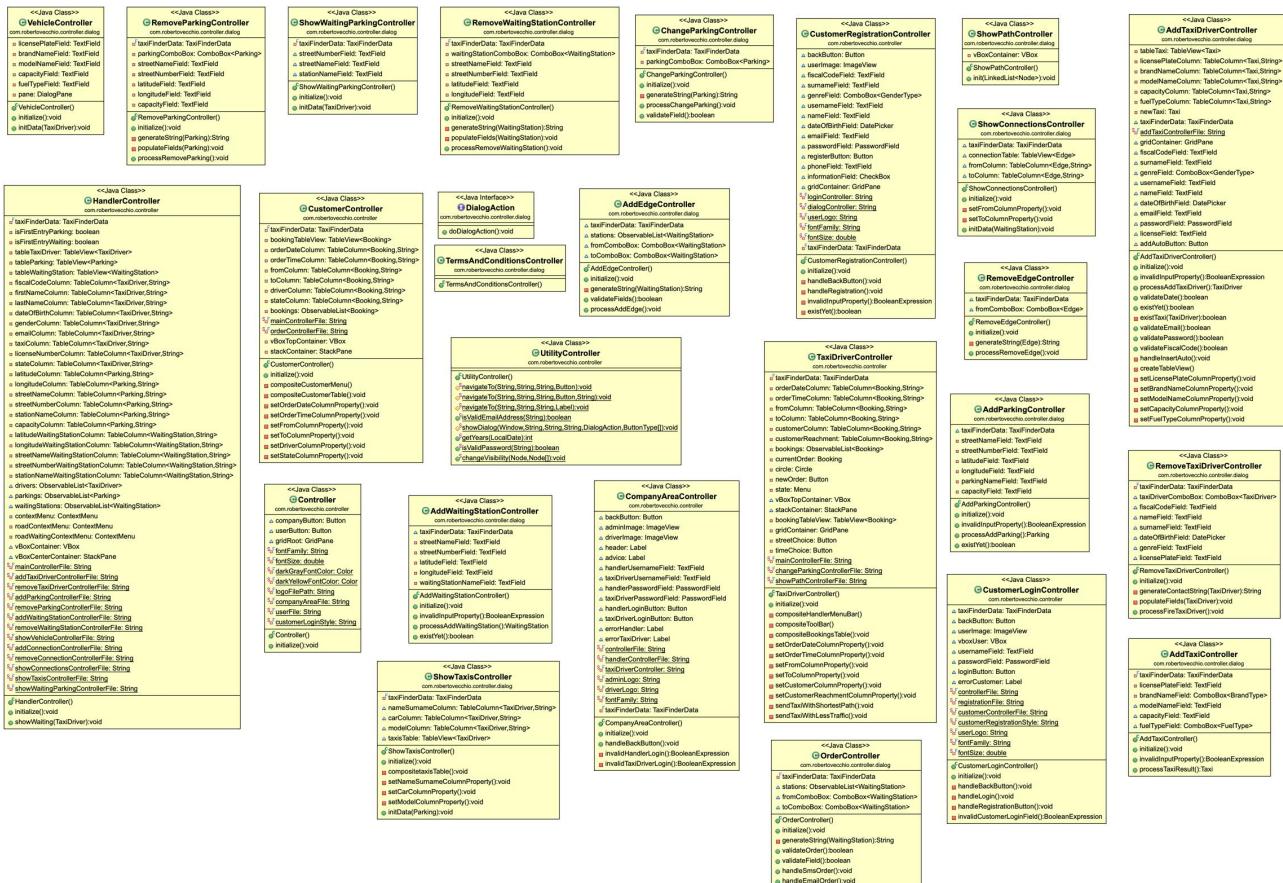
    stage.setTitle(title);
    stage.setScene().setRoot(root);
    stage.setScene().getStylesheets().add(stylesheets);
} catch (IOException e){
    System.out.println(error);
}
```

Il package View rappresenta quindi di fatto una raccolta di tutti i file fxml che vanno a definire le diverse componenti UI ed informazioni ad essi annesse, come nel caso della personalizzazione dell'interfaccia la quale richiede il supporto di fogli di stile (.css). Di seguito un diagramma che chiarisce la struttura e le componenti grafiche di Taxi Finder:



2.4 Package Controller

Il package controller di Taxi Finder contiene classi e sotto-package che si occupano di incapsulare il comportamento dell'applicazione, mappare gli input dell'utente agli aggiornamenti del model e di selezionare una view dopo l'input. Di seguito ne possiamo apprezzare il diagramma delle classi:



In particolare nel nostro sistema troviamo due tipi di controller:

- Controller contenuti direttamente nel package **com.robertovecchio.controller** che rappresentano di fatto coloro che gestiscono e interagiscono con le view principali;
 - Controller contenuti nel package **com.robertovecchio.controller.dialog** che rappresentano di fatto coloro che gestiscono e interagiscono con le finestre secondarie di tipo dialog che si aprono parallelamente a quelle principali.

Entrambi sono caratterizzati da un metodo `initialize()`, che se opportunamente marcato con annotazione `@FXML` servirà da inizializzatore quando il controller viene richiamato per la prima volta. Questo metodo viene spesso sfruttato nel progetto per popolare `TableView` ed elementi grafici annessi ad una view:

```

/* Questo metodo inizializza la view a cui è collegato il controller corrente */
 */
@FXML
public void initialize(){
    /* Inizializziamo una nuova imageview */
    ImageView logoImage = new ImageView();

    try {
        /* Inizializziamo il logo */
        Image logo = new Image(new FileInputStream(LogoFilePath));
        logoImage.setImage(logo); // impostiamo l'immagine all'imageview
        logoImage.setFitHeight(100); // impostiamo altezza
        logoImage.setPreserveRatio(true); // impostiamo la conservazione delle proporzioni originali
    } catch (FileNotFoundException e){
        /* Catturiamo l'errore */
        System.out.println("File Non trovato");
    }

    /* Inizializziamo un textFlow */
    TextFlow textFlow = new TextFlow();

    /* Creiamo la prima parte dell'intestazione */
    Text headerTextFirstPart = new Text(s: "T");
    headerTextFirstPart.setFont(Font.font(fontFamily, FontWeight.BOLD, fontSize));
    headerTextFirstPart.setFill(darkYellowFontColor);

    /* Creiamo la seconda parte dell'intestazione */
    Text headerTextSecondPart = new Text(s: "x1");
    headerTextSecondPart.setFont(Font.font(fontFamily, FontWeight.NORMAL, fontSize));
    headerTextSecondPart.setFill(darkGrayFontColor);

    /* Creiamo la terza parte dell'intestazione */
    Text headerTextThirdPart = new Text(s: "=");
    headerTextThirdPart.setFont(Font.font(fontFamily, FontWeight.BOLD, fontSize));
    headerTextThirdPart.setFill(darkYellowFontColor);
}

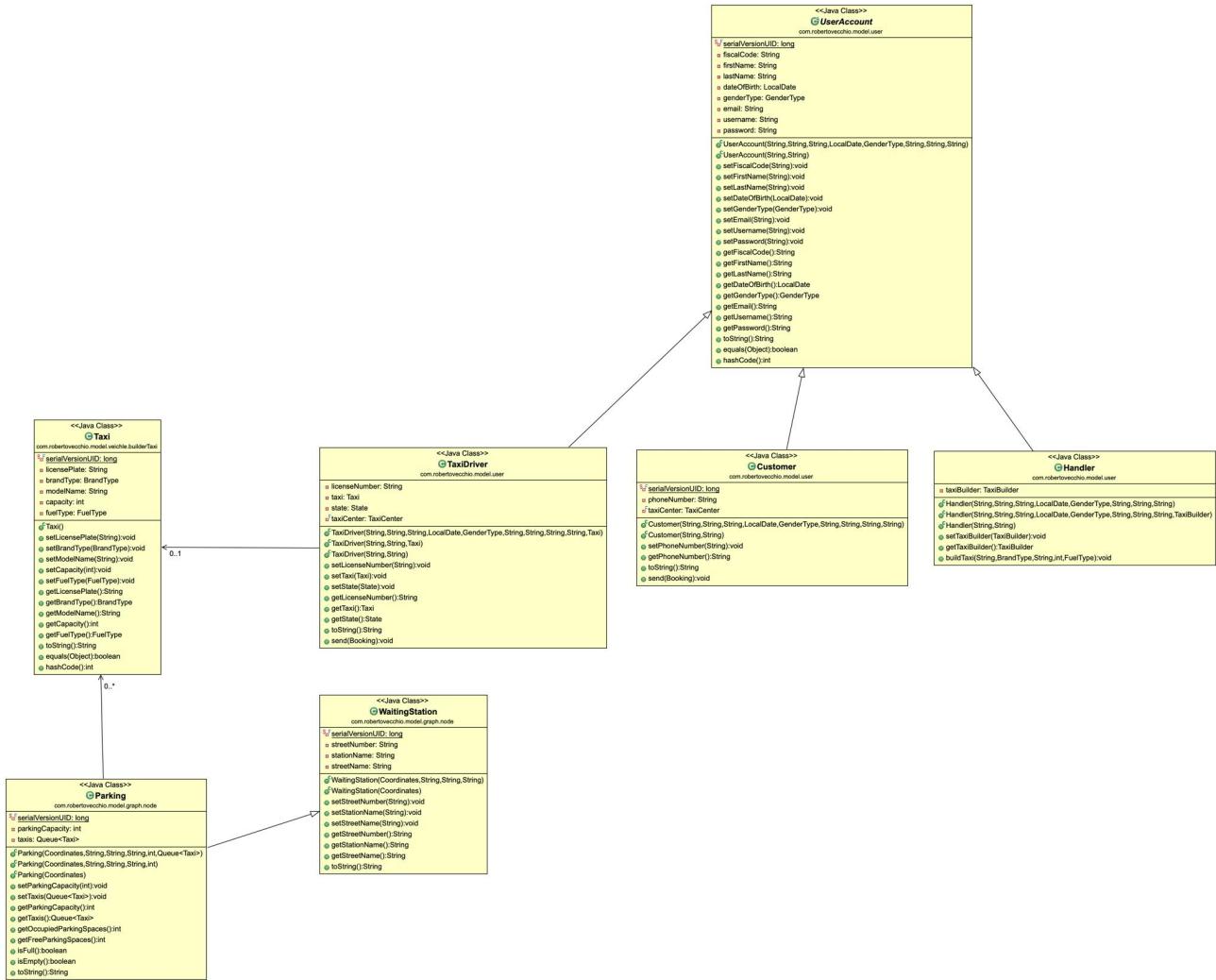
```

2.5 Package Model

Il package model ha lo scopo di raggruppare tutte le astrazioni di tipo entity che definiscono le entità presenti nel sistema o che interagiscono con esso. Al suo interno, infatti, organizzati in diversi sotto-package troviamo le classi:

- UserAccount (utente generico)
- Handler (gestore)
- TaxiDriver (tassista)
- Taxi
- Customer (cliente)
- Parking (parcheggi)
- WaitingStations (postazioni di attesa per il cliente)

Le classi servono a definire le caratteristiche comuni a tutti i clienti, ai tassisti, ai gestori, alle prenotazioni, ai parcheggi, alle aree di attesa ed ai taxi:



Le variabili d'istanza delle classi in questione sono tutte dichiarate di tipo privato con metodi tipo getter e setter annessi per rispettare l'incapsulamento. Ai metodi pubblici si aggiungono i costruttori che hanno la funzione di istanziare una classe quando il metodo costruttore viene richiamato, passando opportunamente i parametri di input prestabiliti.

Inoltre queste classi sfruttano alcuni metodi ereditati da Object, come:

- `toString`
- `equals`
- `hashCode`

Questi sono sfruttati per implementarne una loro variante in modo tale che possano fungere da metodi di utility quando si è concitati nello sviluppo, oppure sfruttati da alcuni metodi del Framework Collections come contains().

Di seguito ne mostriamo l'implementazione:

```
public class Customer extends UserAccount implements Colleague {

    /**
     * Numero seriale utile ai fini della memorizzazione
     */
    @Serial
    private final static long serialVersionUID = 1L;

    //=====
    // Variabili d'istanza
    //=====

    /**Numero telefonico utente*/
    private String phoneNumber;
    /**
     * Centro taxi con cui comunicherà il cliente
     */
    private final TaxiCenter taxiCenter = new TaxiCenter();

    //=====
    // Costruttori
    //=====

    /**
     * Costruttore di un Gestore
     * @param fiscalCode codice fiscale utente
     * @param firstName nome utente
     * @param lastName cognome utente
     * @param dateOfBirth data di nascita utente
     * @param genderType genere sessuale utente
     * @param email email utente
     * @param username username utente
     * @param password password utente
     * @param phoneNumber numero telefonico utente
     * @see LocalDate
     * @see GenderType
     */
    public Customer(String fiscalCode, String firstName,
                    String lastName, LocalDate dateOfBirth,
                    GenderType genderType, String email,
                    String username, String password,
                    String phoneNumber) {
        // Richiamo il costruttore della classe astratta UserAccount
        super(fiscalCode, firstName, lastName, dateOfBirth, genderType, email, username,
              password);

        // inizializzazione delle variabili d'istanza
        this.phoneNumber = phoneNumber;
    }

    /**
     * @param username username utente
     * @param password password utente
     * @see LocalDate
     * @see GenderType
     */
}
```

```

public Customer(String username, String password) {
    // Richiamo il costruttore della classe astratta UserAccount
    super(username, password);
}

//=====
//          Setter
//=====

/**
 * Setter password utente
 * @param phoneNumber Password utente
 */
public void setPhoneNumber(String phoneNumber){
    this.phoneNumber = phoneNumber;
}

//=====
//          Getter
//=====

/**
 * Getter numero di telefono
 * @return Numero di telefono utente
 */
public String getPhoneNumber(){
    return this.phoneNumber;
}

//=====
//          Metodi Sovrascritti
//=====

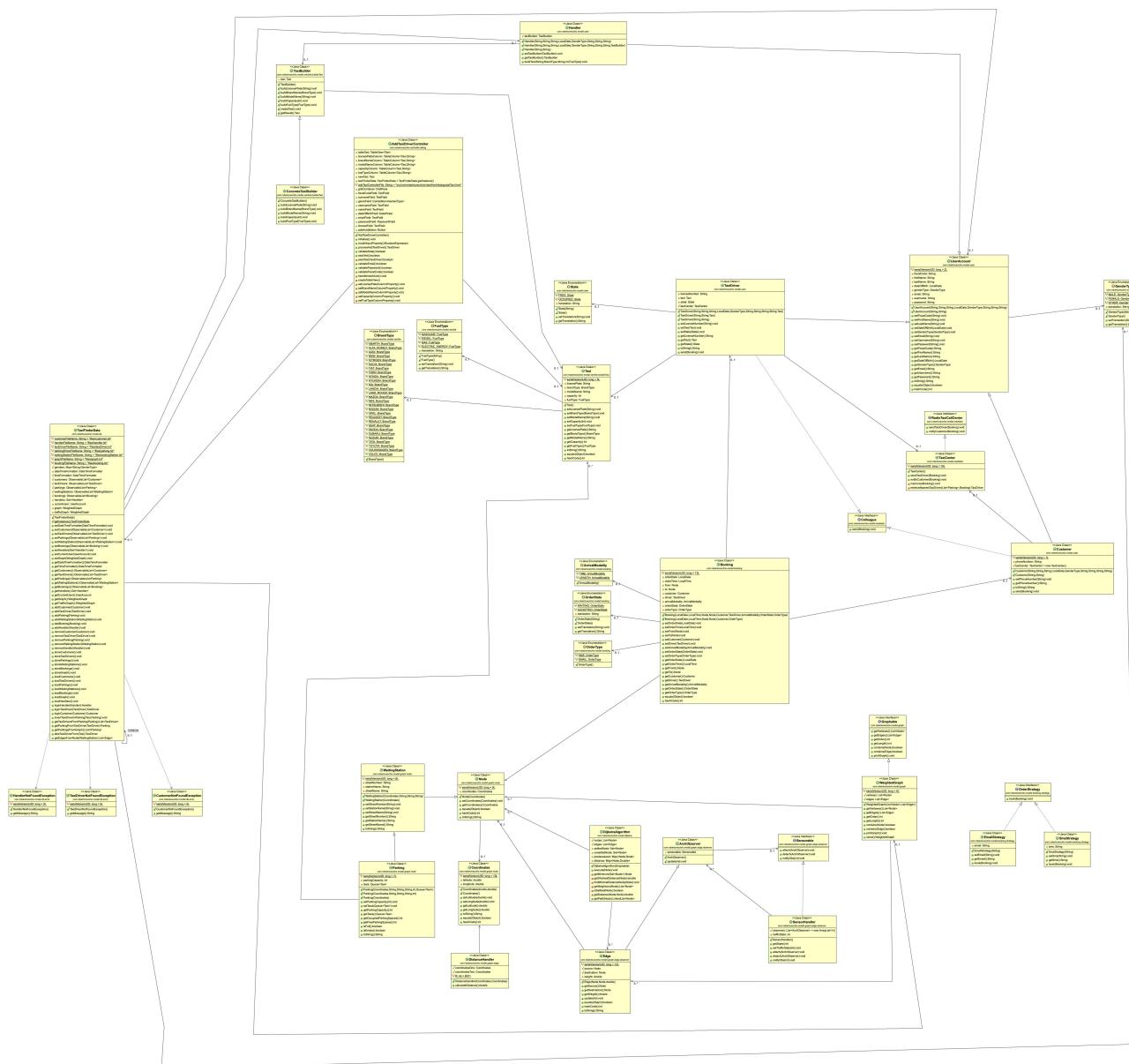
/**
 * Metodo che restituisce una stringa dell'oggetto corrente
 * @return Ritorna l'oggetto sotto forma di stringa
 */
@Override
public String toString() {
    return super.toString() + "\nCustomer{" +
        "phoneNumber='" + phoneNumber + '\'' +
        '}';
}

/**
 * Metodo che sfrutta il mediator del mediator pattern per comunicare l'aggiornamento
 * della prenotazione con
 * i parametri scelti dal tassista.
 * @param booking Prenotazione che ha bisogno di essere aggiornata e comunicata
 */
@Override
public void send(Booking booking) {
    taxiCenter.sendTaxiDriver(booking);
}
}

```

Il diagramma delle classi del model nella sua interezza è più complesso in quanto fa utilizzo dei design pattern utili per la programmazione SOLID.

Di seguito ne mostriamo il diagramma:



N.b è possibile visualizzare l'immagine dettagliatamente nella folder di progetto.

2.6 Singleton e persistenza dei dati

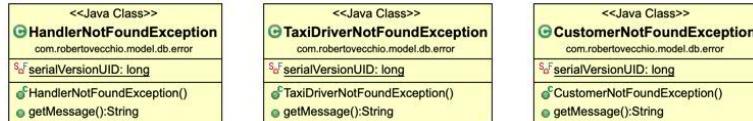
La persistenza dei dati è interamente gestita dalla classe TaxiFinderData, la quale ha lo scopo di interfacciarsi con i file posti nella cartella files locata all'interno della root di progetto.



Grazie ai diversi metodi presenti nella classe, possiamo effettuare tutte le operazioni richieste, come ad esempio effettuare l'accesso, la registrazione, effettuare un'assunzione ecc..

A questa classe è stato applicato il Pattern Creazionale **Singleton**, versione eager. In questo modo si garantisce che la classe abbia una sola istanza fornendo un punto globale di accesso ad essa.

Questa classe è anche in grado di lanciare errori personalizzati quando un utente di tipo tassista, cliente o gestore non viene trovato. Di seguito se ne riporta il diagramma:



Queste sono state sfruttate attraverso la parola chiave `throws`, la quale permette la propagazione dell'errore; infatti gli errori sono stati gestiti nel seguente modo per rendere visibili messaggi di errore che altrimenti sarebbero rimasti invisibili:

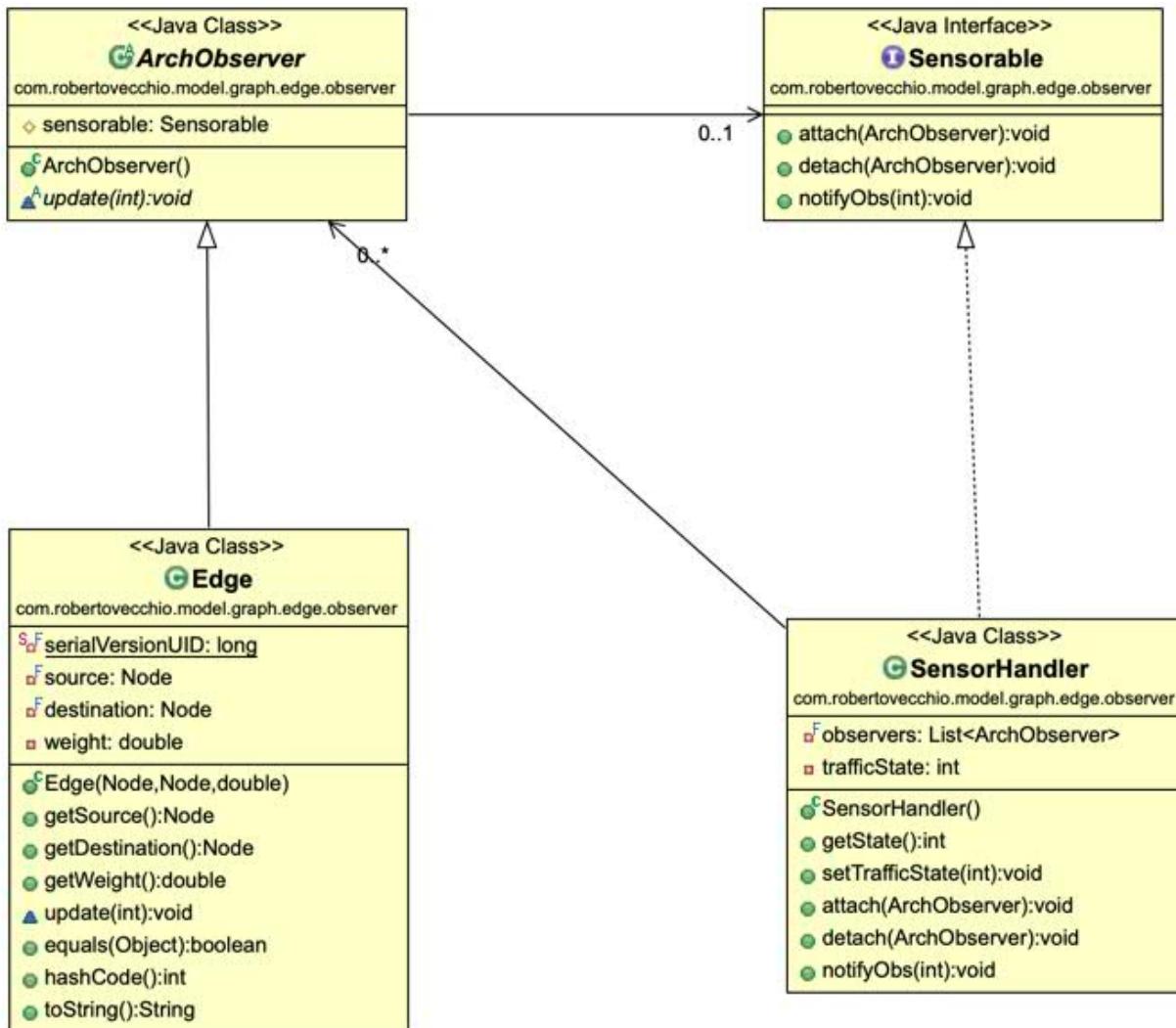
```
try {
    /* Inizializziamo l'handler con i dati di login */
    Handler handler = taxiFinderData.loginHandler(new Handler(username,password));
    /* Carichiamo le liste inerenti all'admin */
    taxiFinderData.setCurrentUser(handler);

    /* Cambiamo Scene */
    UtilityController.navigateTo(handlerControllerFile,
        String.format("%s %s - %s", handler.getFirstName(),
            handler.getLastName(),
            handler.getUsername()),
        error: "Errore di navigazione, alcuni file non sono stati trovati",
        handlerLoginButton);
} catch (HandlerNotFoundException e){
    /* Stampiamo l'errore */
    System.out.println(e.getMessage());

    /* Mostriamo l'errore ad interfaccia */
    errorHandler.setVisible(true);
}
```

2.7 Observer pattern

Il package **com.robertovecchio.model.graph.edge.observer** contiene le classi relative all'observer pattern:



ArchObserver: rappresenta l'Observable dell'observer pattern ed esso ha lo scopo di esporre l'interfaccia che consente agli osservatori di iscriversi e cancellarsi, mantenendo una reference a tutti gli osservatori iscritti.

Edge: rappresenta il ConcreteObservable dell'observer pattern ed esso mantiene lo stato del soggetto osservato e notifica gli osservatori in caso di cambio di stato.

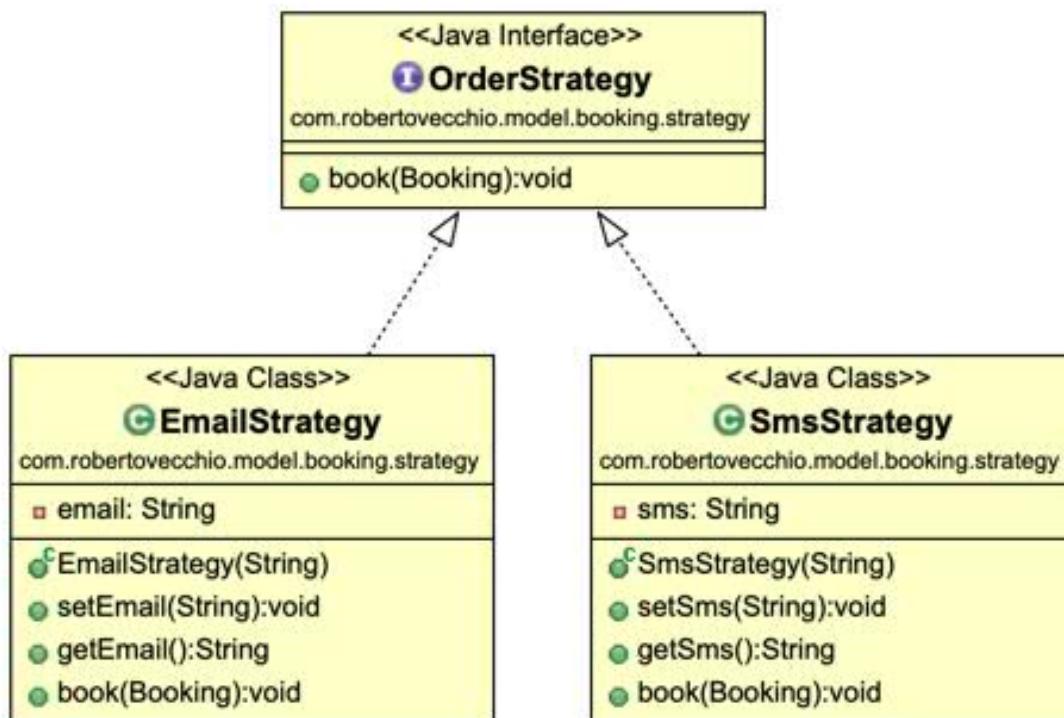
Sensorable: rappresenta L'Observer dell'observer pattern ed esso espone l'interfaccia che consente di aggiornare gli osservatori in caso di cambio di stato del soggetto osservato.

SensorHandler: rappresenta il Concrete Observer dell'observer pattern ed esso implementa l'interfaccia dell'observer definendo il comportamento in caso di cambio di stato del soggetto osservato.

Il pattern in questione è stato utilizzato per aggiornare in tempo reale il tempo di percorimento delle strade in termini di minuti, in modo tale da simulare il traffico ed il peso dell'arco tra un nodo e l'altro nel caso in cui il tassista avesse scelto di raggiungere il cliente nel minor tempo possibile.

2.8 Strategy pattern

Il package **com.robertovecchio.model.booking.strategy** contiene le classi relative allo strategy pattern:



OrderStrategy: rappresenta l'interfaccia Strategy dello strategy pattern ed essa dichiara un'interfaccia che verrà invocata dal Context in base all'algoritmo prescelto.

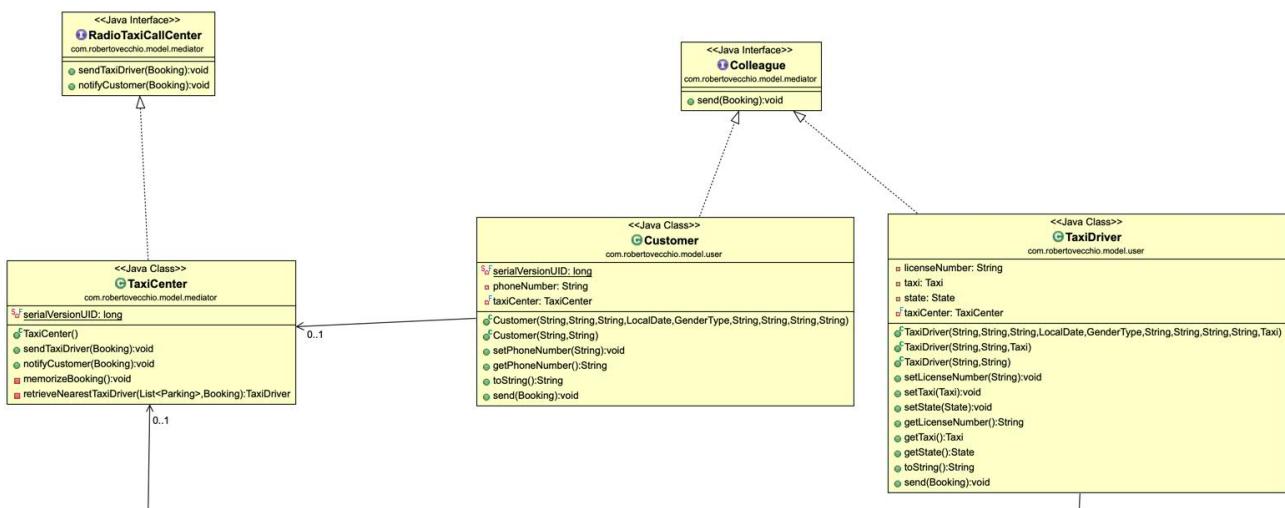
EmailStrategy: rappresenta la ConcreteStrategyA dello strategy pattern ed essa effettua l'overwrite del metodo di OrderStrategy al fine di ritornare l'implementazione dell'algoritmo. In particolare, provvede ad effettuare la prenotazione di una corsa tramite email.

SmsStrategy: rappresenta la ConcreteStrategyB dello strategy pattern ed essa effettua l'overwrite del metodo di OrderStrategy al fine di ritornare l'implementazione dell'algoritmo. In particolare, provvede ad effettuare la prenotazione di una corsa tramite email.

Si ricordi che lo scopo dello Strategy pattern (pattern comportamentale) è quello di definire una famiglia di algoritmi, incapsularli e renderli intercambiabili, dove l'algoritmo cambia indipendentemente dai client che lo usano. In particolare lo si usa quando si ha la necessità di modificare dinamicamente gli algoritmi utilizzati da un'applicazione. Data una definizione formale, si può ben comprendere che lo strategy pattern è stato implementato per definire una famiglia di algoritmi per la prenotazione, incapsolandoli e rendendoli intercambiabili.

2.9 Mediator pattern

Il package **com.robertoveccchio.model.mediator** contiene le classi relative al mediator pattern:



RadioTaxiCallCenter: rappresenta l'interfaccia Mediator del mediator pattern, il quale definisce un'interfaccia per comunicare con i Colleague.

TaxiCenter: rappresenta il ConcreteMediator del mediator pattern ed esso mantiene la lista dei colleghi e implementa lo scambio di messaggi tra loro.

Colleague: rappresenta l'interfaccia Colleague del mediator pattern e definisce l'interfaccia di tutti i Colleague.

Customer e TaxiDriver: rappresentano i concreteColleague del mediator pattern, si occupano di implementare i singoli colleghi e le modalità di comunicazione con il mediator.

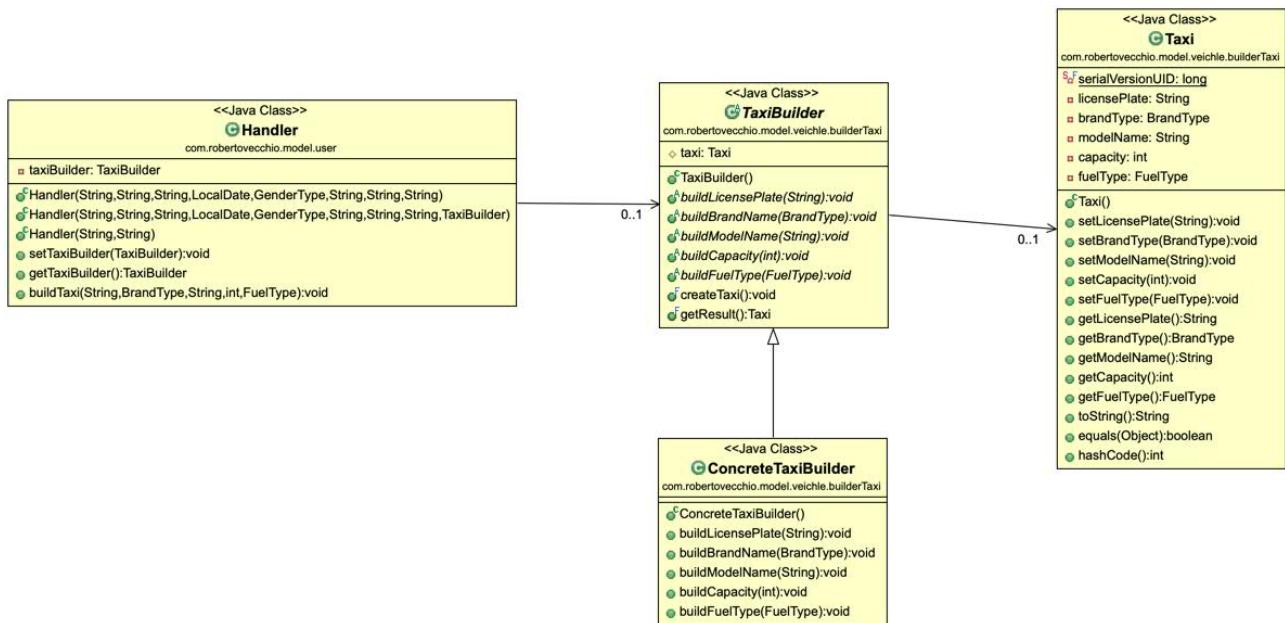
Si ricordi che lo scopo del mediator è quello di definire un oggetto che incapsula il meccanismo di interazione di oggetti, consentendo il loro disaccoppiamento in modo da variare facilmente le interazioni tra di loro.

Proprio in base alla definizione formale, si può capire che la scelta del mediator è stata quasi una scelta obbligatoria, per disaccoppiare l'interazione tra un cliente ed un tassista. Infatti, quando si prenota realmente

un taxi non si conosce quale sarà il taxi più vicino ma si comunica con un operatore che richiede a quest'ultimo di recarsi verso la postazione del cliente. Con questo pattern abbiamo quindi, di fatto, simulato la realtà sostituendo alla figura dell'operatore una classe che astrae il suo concetto e simula il suo comportamento, ovvero la classe **TaxiCenter** la quale funzionerà da mediatore.

2.10 Builder pattern

Il package **com.robertovecchio.model.vehicle.builderTaxi** contiene le classi relative al builder pattern:



Handler: rappresenta il director del Builder pattern ed esso ha lo scopo di costruire un oggetto partendo dall'interfaccia Builder.

TaxiBuilder: rappresenta il Builder del Builder pattern ed esso specifica una interfaccia atta alla creazione del product.

ConcreteTaxiBuilder: rappresenta il ConcreteBuilder del Builder Pattern ed esso costruisce il Product in base ai metodi definiti nel Builder.

Taxi: rappresenta il Product del Builder pattern e di fatto rappresenta l'oggetto complesso da costruire.

Si rammenti che lo scopo del **Builder** è di separare la costruzione di un oggetto complesso dalla sua rappresentazione in modo tale che lo stesso processo di costruzione può creare differenti rappresentazioni.

Si è scelto quindi utilizzato il Builder per costruire un oggetto complesso come il taxi in modo da fornirne differenti rappresentazioni.

3.0 Il problema dello Shortest Path

3.1 Shortest Path

Nella traccia è stato specificato di dover utilizzare l'algoritmo Shortest Path di Dijkstra. Trovare il percorso più breve (dall'inglese Shortest path) in una rete è un problema comunemente riscontrato, infatti ne abbiamo l'esempio concreto con un'azienda che deve gestire una flotta di Taxi.

Però prima di analizzare nel dettaglio la sua implementazione, è necessario fare una breve regressione alla definizione di grafo in quanto è stata utile per astrarre il suo concetto ricavandone delle classi.

3.2 Grafo

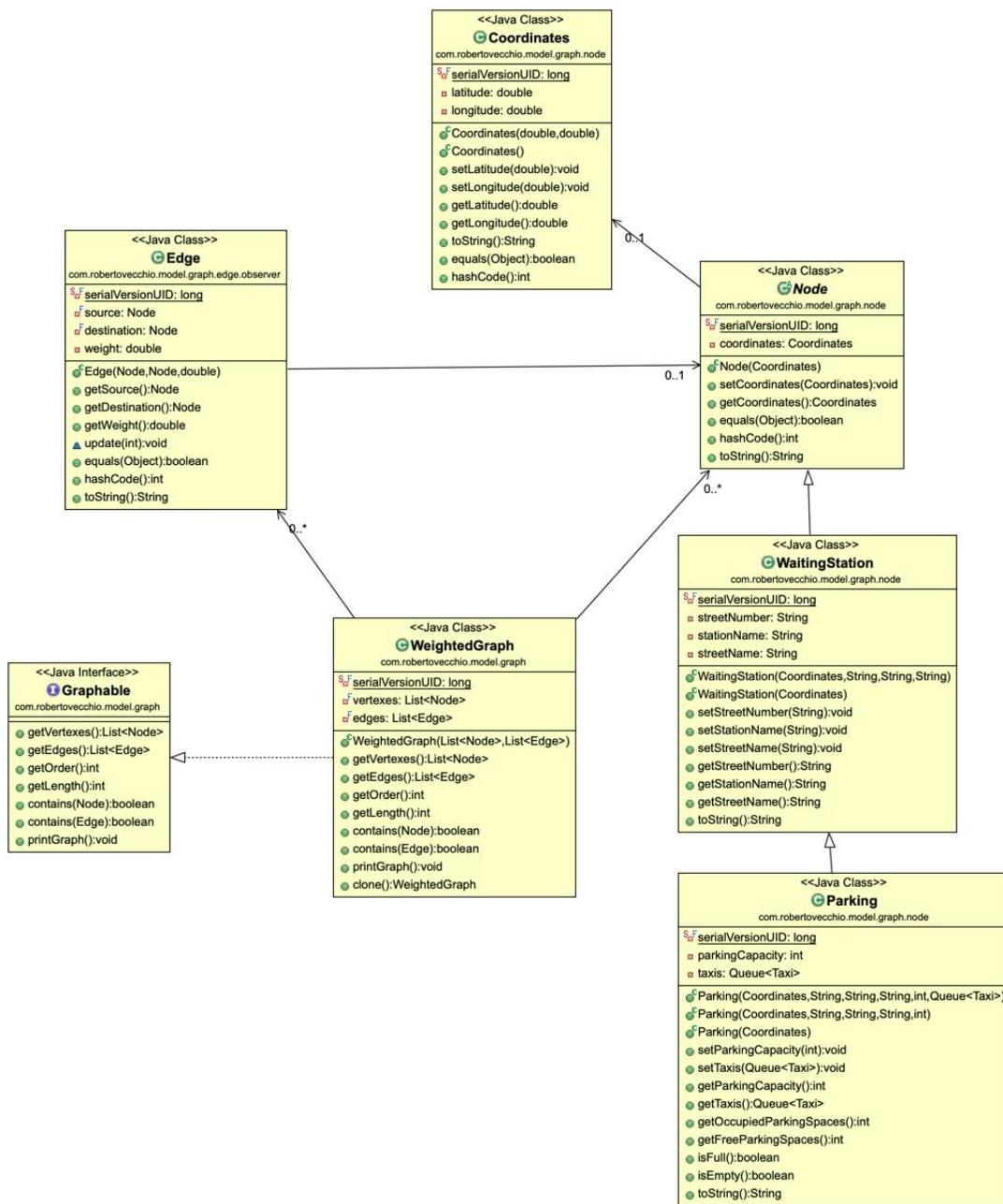
Un grafo è composto da **nodi** e **linee direzionali** che definiscono una connessione tra un nodo e l'altro. Un nodo è una posizione discreta in un grafo e le linee possono essere direzionali o non direzionali ed avere un peso associato.

La descrizione matematica per un grafo è $G = \{V, E\}$, che significa che il grafo è definito da un set di vertici ed una collezione di linee.

L'ordine del grafo è il numero di nodi, la grandezza del grafo è il numero di linee.

3.3 Descrizione algoritmo

L'algoritmo di Dijkstra trova lo Shortest path da una sorgente su tutte le destinazioni in un grafo direzionale. Date queste breve informazioni, si può quindi mostrare un diagramma delle classi che riassume il funzionamento di Taxi Finder relativamente al grafo per la gestione del problema dello Shortest Path:



Nel Sistema Taxi Finder le postazioni di attesa ed i parcheggi sono nodi, i quali hanno delle coordinate espresse in latitudine e longitudine, in modo tale da determinare, nel caso in cui il tassista scelga il percorso più breve, il vettore distanza tra i due nodi.

Questa distanza viene gestita separatamente da una classe indipendente, la quale è stata creata per non violare il principio di singola responsabilità:

```

public class DistanceHandler {

    private final Coordinates coordinatesOne;
    private final Coordinates coordinatesTwo;

    private static final int R = 6371;

    public DistanceHandler(Coordinates coordinatesOne, Coordinates coordinatesTwo){
        this.coordinatesOne = coordinatesOne;
        this.coordinatesTwo = coordinatesTwo;
    }

    public double calculateDistance(){
        double latAlfa, latBeta;
        double lonAlfa, lonBeta;
        double fi;
        double p, d;

        /* Convertiamo i gradi in radians */
        latAlfa = Math.PI * this.coordinatesOne.getLatitude() / 180;
        latBeta = Math.PI * this.coordinatesTwo.getLatitude() / 180;
        lonAlfa = Math.PI * this.coordinatesOne.getLongitude() / 180;
        lonBeta = Math.PI * this.coordinatesTwo.getLongitude() / 180;

        /* Calcola l'angolo compreso fi */
        fi = Math.abs(lonAlfa - lonBeta);

        /* Calcola il terzo lato del triangolo sferico */
        p = Math.acos(Math.sin(latBeta) * Math.sin(latAlfa) +
                     Math.cos(latBeta) * Math.cos(latAlfa) * Math.cos(fi));

        /* Calcola la distanza sulla superficie terrestre R = ~6371 km */
        d = p * R;
        return d;
    }
}

```

Inoltre come si evince dalla definizione di grafo, esso è un insieme di nodi e collegamenti, per cui questi ultimi, nel diagramma delle classi, hanno una dipendenza.

Infine è stata creata l'interfaccia **Graphable** per rispettare il principio di inversione delle dipendenze. In questo modo una classe non dovrà dipendere dall'implementazione del grafo.

Passando invece a descrivere l'algoritmo, si può affermare che l'idea di Dijkstra è semplice in quanto partiziona i nodi in due differenti set:

- **settled**
- **unsettled**

Inizialmente tutti i nodi sono di tipo unsettled perché devono essere ancora valutati. Un nodo si muove nei settled set se uno Shortest path dalla sorgente a questo nodo è stato trovato.

Inizialmente la distanza di ogni nodo dalla sorgente è impostata ad un numero molto alto.

Inoltre, all'inizio, solo la sorgente è nel set dei nodi unsettled. L'algoritmo esegue fin tanto che gli unsettled Node sono vuoti. In ogni iterazione viene selezionato il nodo con distanza minore dalla sorgente al di fuori degli unsettled. Viene letto ogni nodo che va verso l'esterno della sorgente e valuta per ogni nodo di destinazione, negli edge in cui non sono ancora settled, se la destinazione conosciuta dalla sorgente a questo nodo può essere ridotta utilizzando quell'edge.

Se ciò risultasse vero, allora la distanza viene aggiornata e il nodo viene aggiunto ai nodi che hanno bisogno di essere valutati. Di seguito lo pseudocodice dell'algoritmo:

Foreach node set distance[node] = High

settledNodes = empty

unsettledNodes = empty

Add sourceNode to UnsettledNode

distance[sourceNode] = 0

while(unsettledNode is not empty){

 evaluationNode = getNodeWithLowestDistance(unsettledNode)

 remove evaluationNode from unsettledNode

 add evaluationNode to settledNode

 evaluateNeighbors(evaluationNode)

}

getNodeWithLowestDistance(unsettledNode){

 find the node with the lowest distance and return it

}

evaluateNeighbors(evaluationNode){

 foreach destinationNode that can be reached via an edge from
 evaluationNode AND is not in settledNodes{

 edgeDistance = getDistance(edge(evaluationNode, destinationNode))

 newDistance = distance [evaluationNode] + edgeDistance

 if(distance[destinationNode] > new distance){

 distance[destinationNode] = newDistance

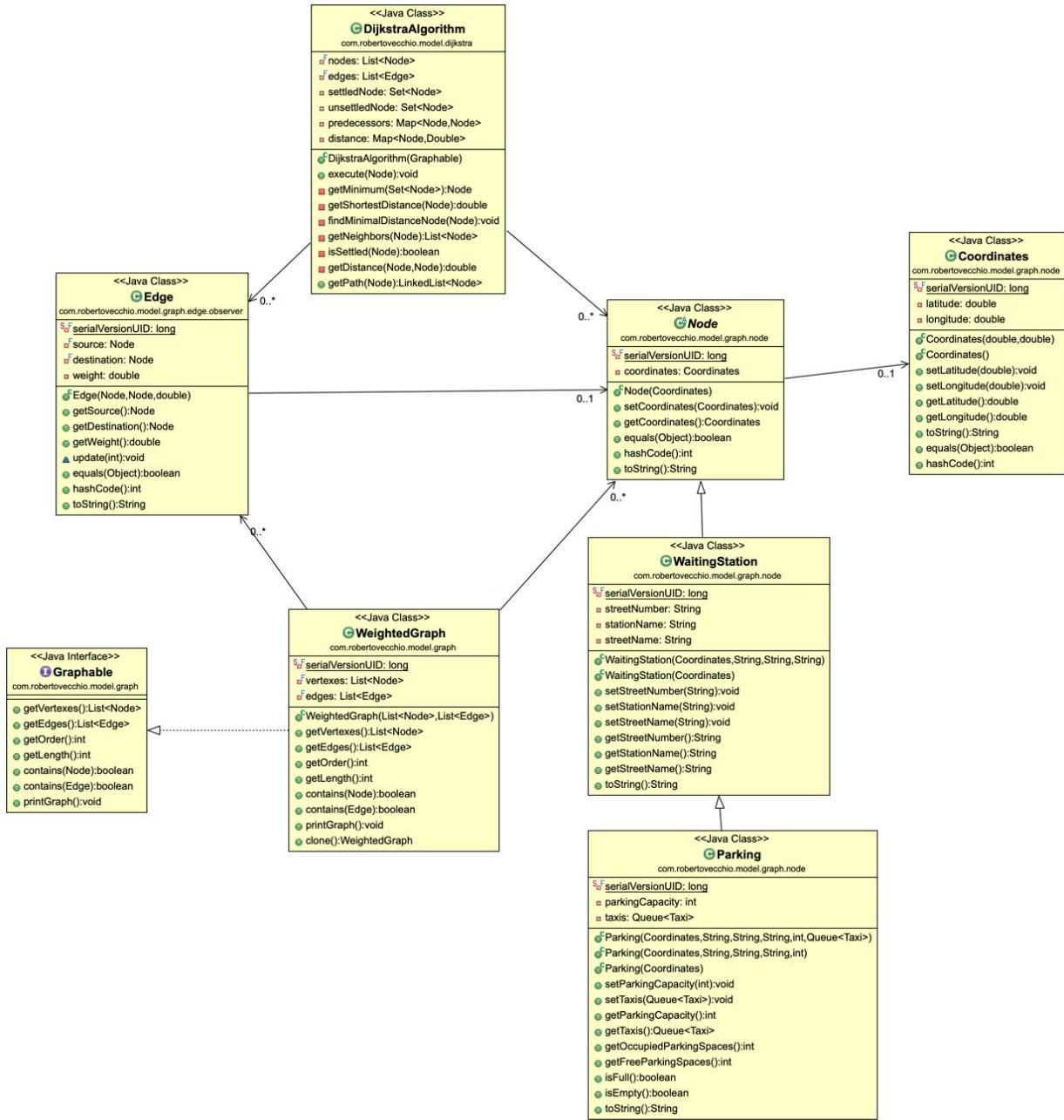
 add destinationNode to unsettledNodes

 }

}

}

Sulla base di questa premessa è stata sviluppata una classe **DijkstraAlgorithm**, che per il principio di singola responsabilità si occupa di eseguire l'algoritmo di Dijkstra (Shortest Path) sulla base del grafo che viene passato come parametro di input al costruttore. Di seguito il relativo diagramma delle classi:



Dijkstra lavora sui pesi degli archi tra nodi, per cui rimane indipendente dal valore che vogliamo valutare; se servisse il percorso più breve, basterebbe impostare il peso degli archi attraverso la classe **distanceHandler**, altrimenti, si potrà sfruttare l'observer sviluppato per le tratte per recuperare il valore in termini di minuti per il corrispettivo percorimento.

4.0 Principi SOLID

I principi **SOLID** sono cinque e definiscono lo sviluppo di un buon software estendibile e manutenibile, in modo tale da evitare la ridondanza del codice e complessità inutili. I cinque Principi sono:

1. Single Responsibility Principle (**SRP**): ogni classe, metodo e variabile deve aver una sola responsabilità;
2. Open Closed Principle (**OCP**): ogni classe deve essere chiusa alla modifica, ma aperta all'estensione;
3. Liskov's Substitution Principle (**LAP**): dobbiamo essere sicuri che le nuove classi estendano le loro classi base senza cambiare il comportamento;
4. Interface Segregation Principle (**ISP**): è preferibile avere molte interfacce specifiche e piccole, anziché poche, generali e grandi;
5. Dependency Inversion Principle (**DIP**): le classi di alto livello (per incapsulare la logica complessa) non devo dipendere pesantemente dalle classi di basso livello (per operazioni primarie).

Nel sistema preposto :

- Il Primo principio viene rispettato, non solo grazie alla progettazione fatta antecedentemente allo sviluppo, ma anche grazie all'implementazione del pattern architettonale **MVC** che agevola il rispetto della regola;
- Il secondo principio viene rispettato infatti nel caso dei nodi, possiamo estendere la classe Node per aggiungere altre tipologie di nodi, senza che la classe stessa venga modificata;
- Il terzo principio viene rispettato in quanto ci sono classi che sono completamente sostituibili alle classi padre come nel caso di Parking con WaitingStations o con Customer/Handler/TaxiDriver con UserAccount;
- Il quarto principio viene rispettato in quanto sono stati utilizzati diversi pattern che hanno previsto l'utilizzo di interfacce specifiche e piccole e che vengono implementate solo da chi li utilizza;
- Il quinto principio viene rispettato in quanto ci sono diverse classi che dipendono dalle astrazioni e non dalle corrispondenti implementazioni.

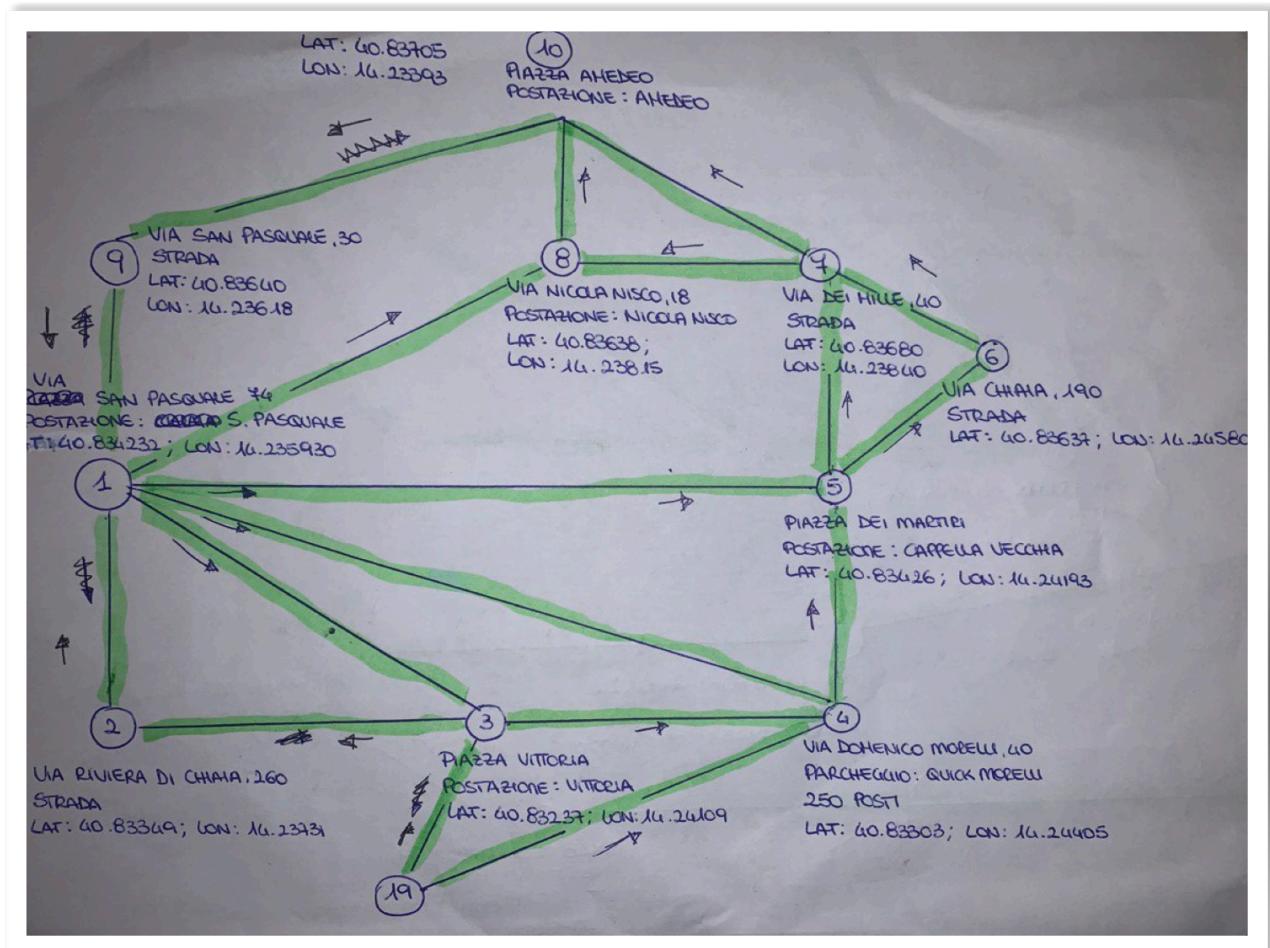
Il tutto è stato possibile soprattutto grazie all'utilizzo di Design pattern che fungono da linea guida per il rispetto dei principi SOLID.

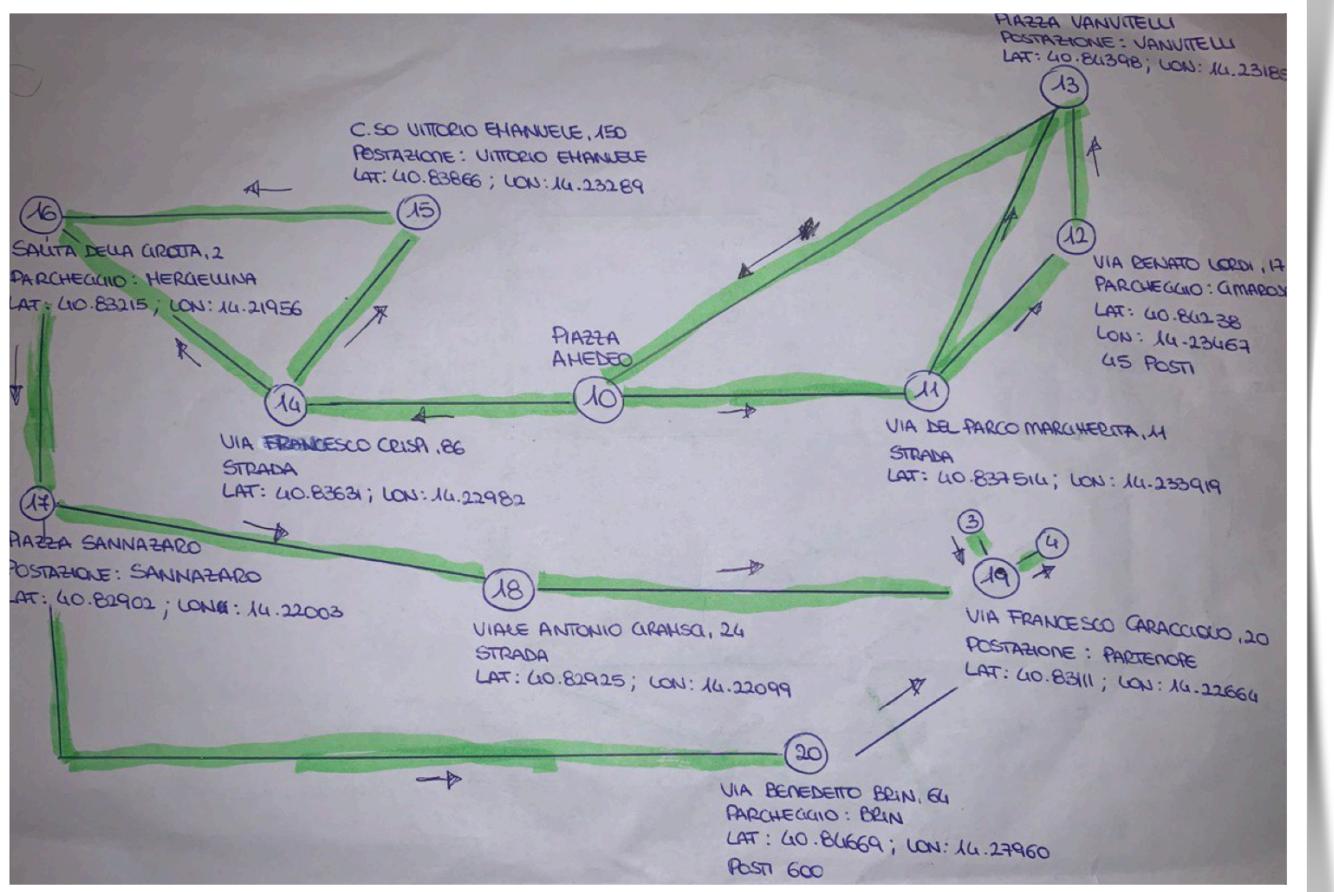
5.0 Manuale Utente

5.1 Mappa di default

Il sistema software Taxi Finder viene fornito agli utilizzatori con una mappa (grafo) di default in modo tale che possano testarne l'utilizzo senza dover inserire via software tutti nodi e collegamenti reali / casuali. Tale persistenza è volta al semplice test del software senza però esserne vincolante.

Di seguito un'immagine che aiuta ad orientarsi sui diversi nodi ed archi già esistenti:



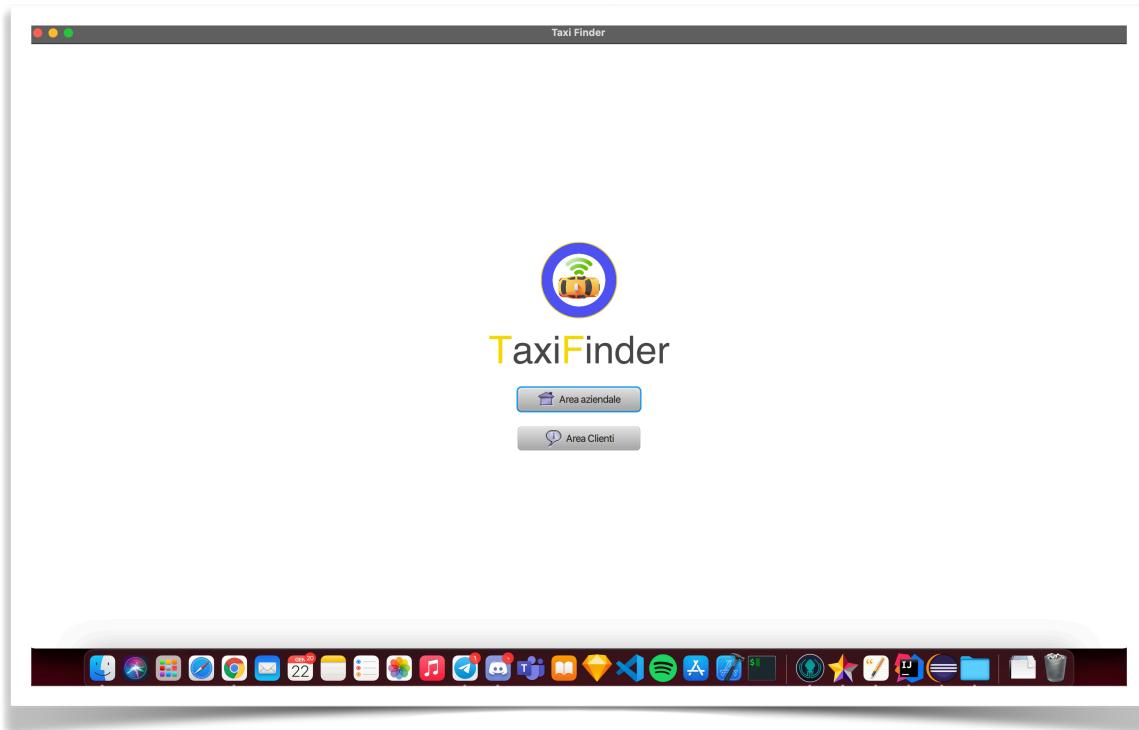


Inoltre va fatta una breve premessa, il grafo su cui è monitorato il traffico viene inizializzato all'inizio del programma; per poter testare il suo utilizzo è quindi necessario ravviare il sistema software in questione.

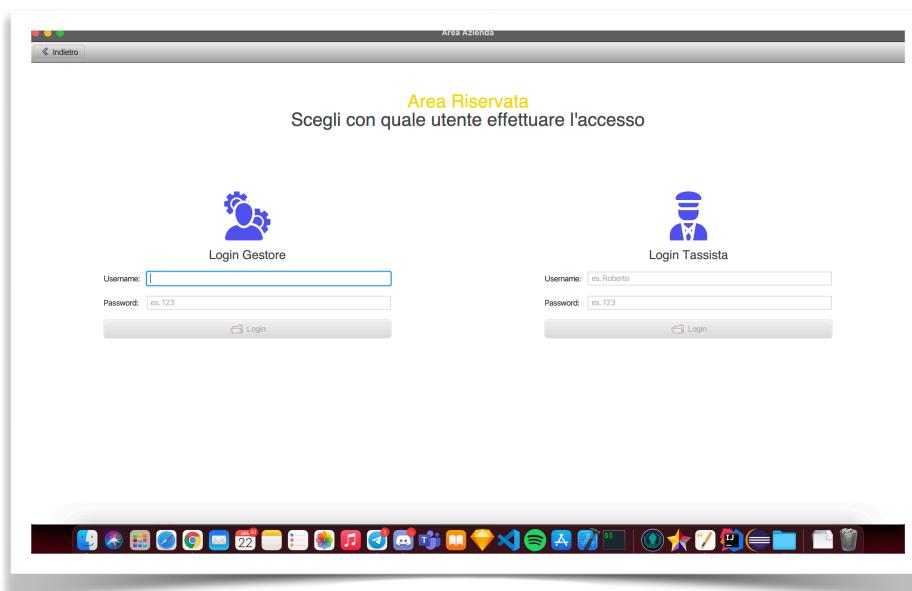
5.2 Interfaccia utente

Eseguendo Taxi Finder ci troveremo davanti alla home page, dove ci verrà chiesto di entrare in una delle due sezioni:

- Area aziendale
- Area utente



Entrando nell'area utente potremo accedere come gestore (admin) oppure come tassista:



Possiamo effettuare l'accesso ad admin con le seguenti credenziali:

- Username: admin
- Password: admin

Una volta inserite le credenziali ci troveremo davanti questa schermata:

The screenshot shows a Mac OS X desktop with a window titled "Roberto Vecchio - admin". The window contains a table of driver information:

Codice Fiscale	Nome	Cognome	Data di nascita	Genere	Email	Targa Taxi	Nr. Licenza	Stato
VCCRR193R06F839A	Roberto	Vecchio	11-01-2002	UOMO	robertovecchio@taxifinde...	SE456WX	LXFNFKN384DFNFSD	Libero
SPSVLR75E44F839R	Valeria	Esposito	19-01-1995	DONNA	valeriaesposito@taxifinde...	TE567TG	45346532	Libero
PSCFNN65M13F839C	Fernando	Pascariello	13-01-1988	UOMO	femandopascariello@taxi...	LA345PE	98837894657	Libero
RSSMRA80A01F839W	Mario	Rossi	09-01-1998	UOMO	mariorossi@taxifinder.com	FG567KL	243494U2	Libero

The status column shows all drivers as "Libero" (Free). At the bottom of the window is a toolbar with various icons.

Dove possiamo vedere la lista dei tassisti insieme al loro stato lavorativo attuale; infatti se il loro stato è impostato su occupato, vuol dire che in quel momento sono occupati.

Cliccando sul menu tassista potremo visualizzare dei sottomenu:

- Assumi tassista
- Congeda tassista

Se si clicca sul primo ci troveremo nella seguente sezione:

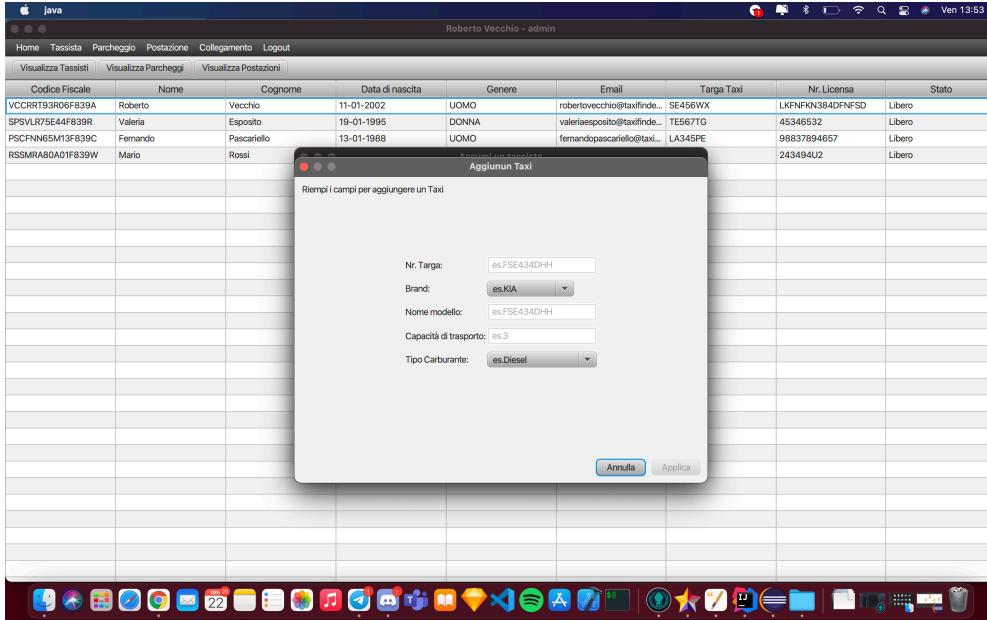
The screenshot shows the same Mac OS X desktop with the "Assumi un tassista" dialog box open over the main driver list window. The dialog box contains fields for entering new driver information:

Riempি i campi per aggiungere un Tassista da assumere	
Codice Fiscale: <input type="text" value="es: VCCRR1..."/>	Nome: <input type="text" value="es: Roberto"/>
Cognome: <input type="text" value="es: Vecchio"/>	Data di nascita: <input type="text" value="es: 01/01/1993"/>
Genero: <input type="select" value="es: Uomo"/>	Email: <input type="text" value="@taxifinder.com"/>
Username: <input type="text" value="es: 123"/>	Password: <input type="text" value="es: 123"/>
Nr. Licenza: <input type="text" value="es: T23"/>	

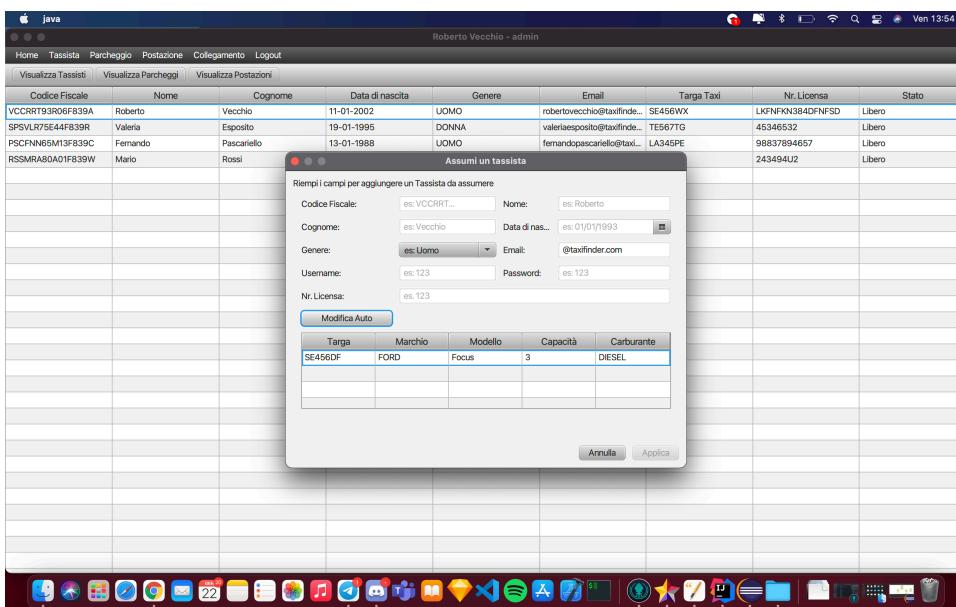
At the bottom of the dialog box are two buttons: "Annulla" and "Applica".

Il bottone adibito a processare l'operazione in questione sarà disabilitato fintanto che tutti i campi non saranno riempiti opportunamente con valori validi, infatti sono previsti controlli su codice fiscale, età e password, altrimenti verrà mostrato a video un messaggio di errore relativo al campo in questione.

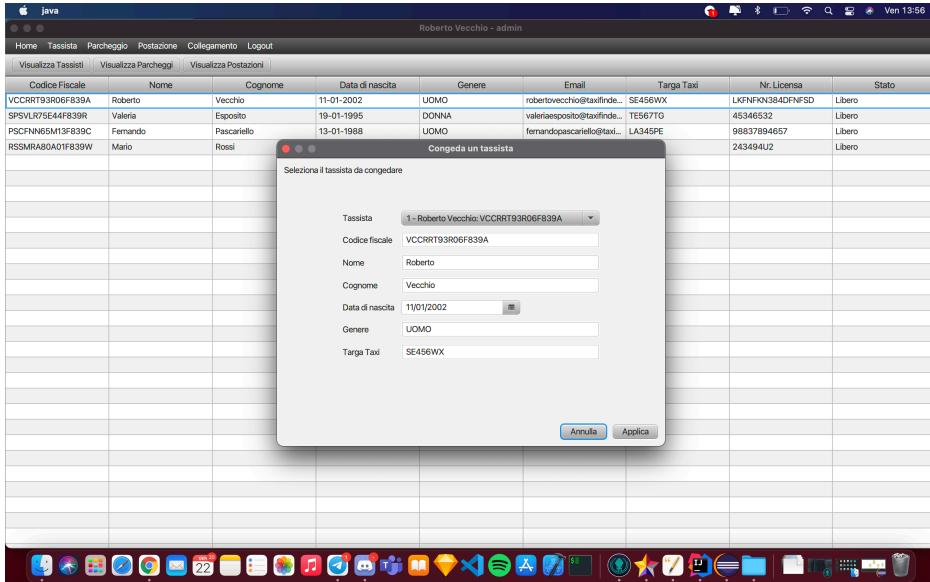
Cliccando su inserisci auto verrà aperta una view adibita all'inserito di una nuova auto:



Una volta inseriti i campi, premendo su applica, ritorneremo nella schermata precedente, la quale sarà popolata dinamicamente in base all'auto scelta:



Premendo invece su congeda tassista avremo la seguente schermata:

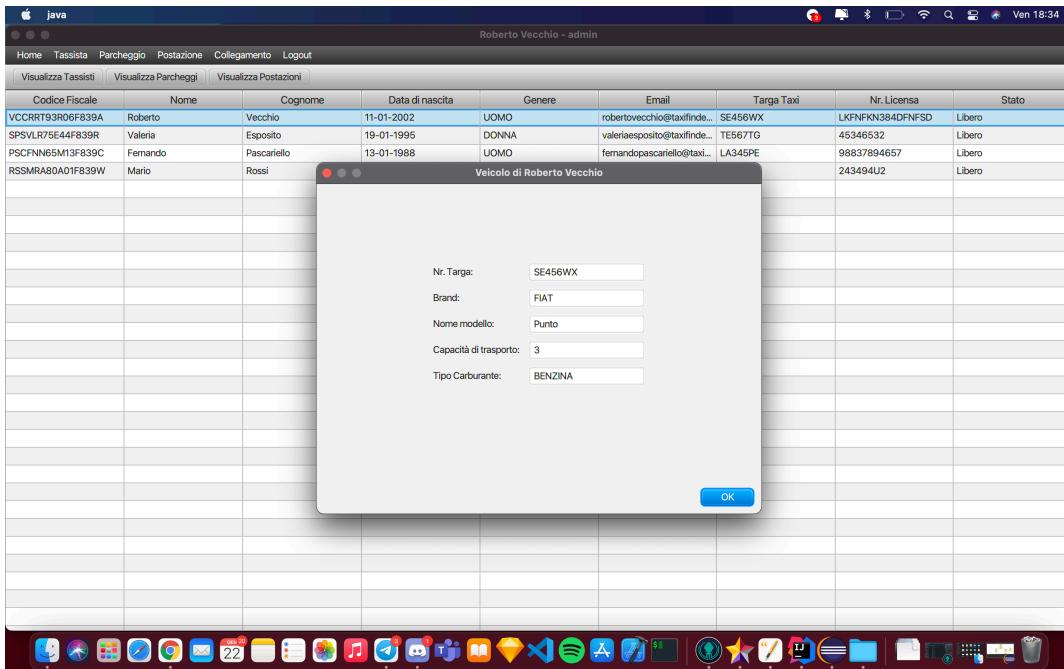


Dove attraverso il menu a tendina possiamo comodamente scegliere chi congedare, riempendo dinamicamente le textfield presenti al di sotto.

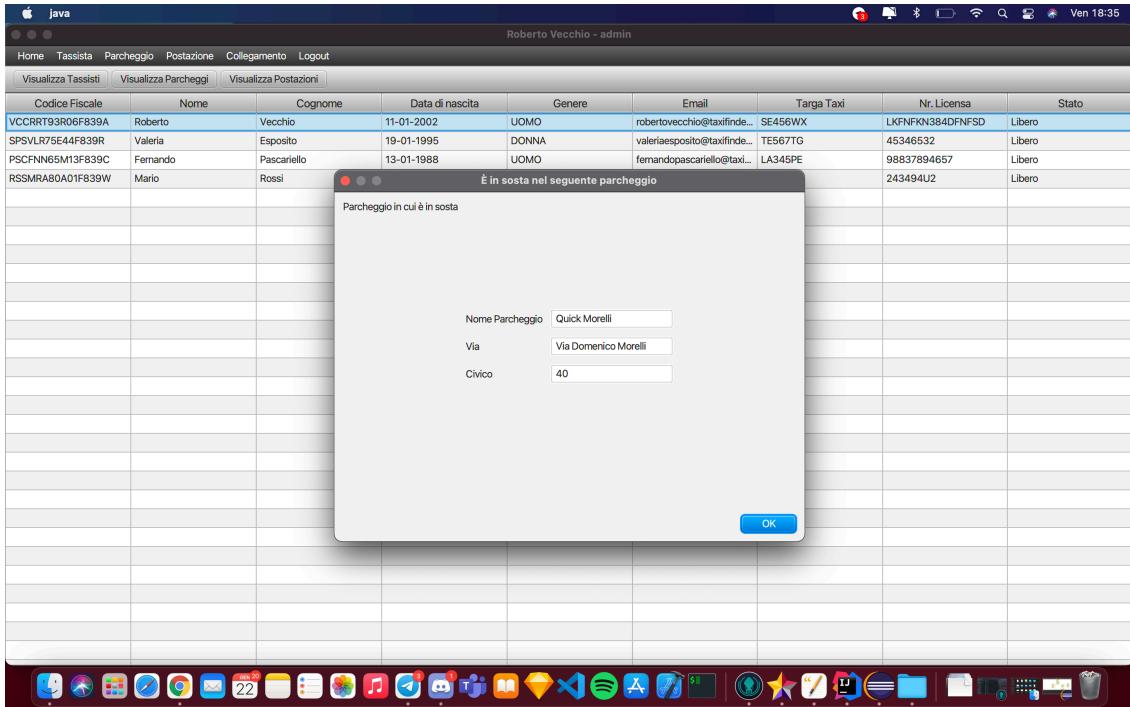
Cliccando con il tasto destro su uno degli elementi mostrati avremo le seguenti opzioni:

- Mostra veicolo;
- Mostra dove sosta;

Cliccando su mostra veicolo, possiamo osservare i dettagli del veicolo associati al tassista:



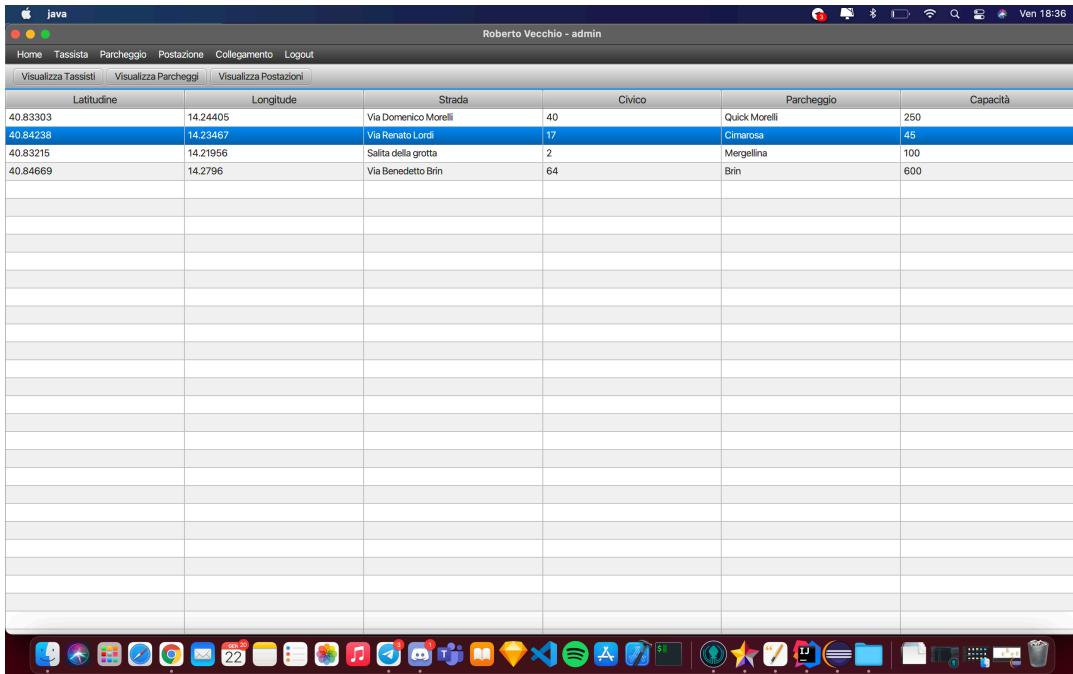
Mentre cliccando su “mostra dove sosta” possiamo vedere dove sosta il tassista:



Come si può notare sulla toolbar sono presenti tre bottoni:

- Visualizza tassisti;
- Visualizza parcheggi;
- Visualizza Postazioni;

Cliccando su visualizza parcheggi possiamo visualizzare la lista dei parcheggi:



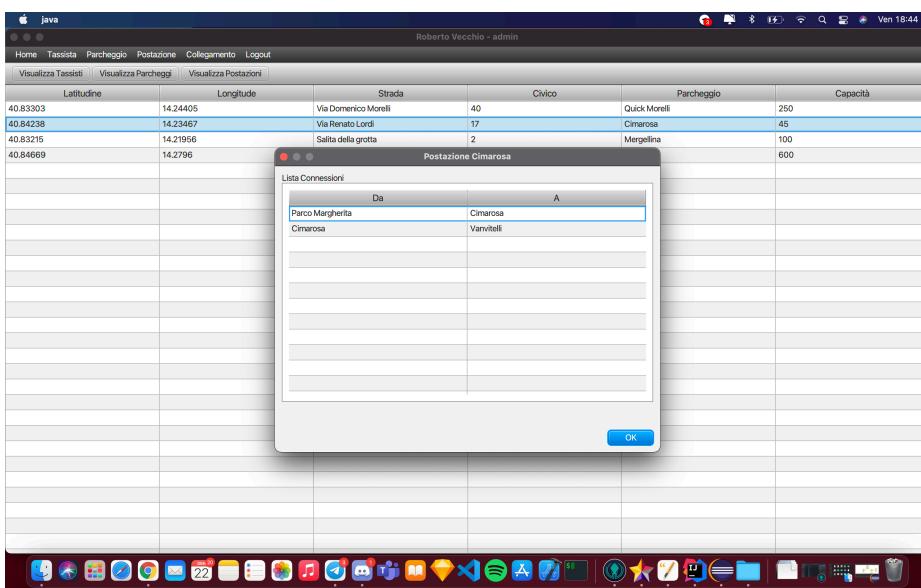
A screenshot of a Java application window titled "Roberto Vecchio - admin". The window contains a table with columns: Latitude, Longitude, Strada, Civico, Parcheggio, and Capacità. The data is as follows:

Latitude	Longitude	Strada	Civico	Parcheggio	Capacità
40.83303	14.24405	Via Domenico Morelli	40	Quick Morelli	250
40.84238	14.23467	Via Renato Lordi	17	Cimarsosa	45
40.83215	14.21956	Salita della grotta	2	Mergellina	100
40.84669	14.2796	Via Benedetto Brin	64	Brin	600

Cliccando con il tasto destro su uno degli elementi mostrati avremo le seguenti opzioni:

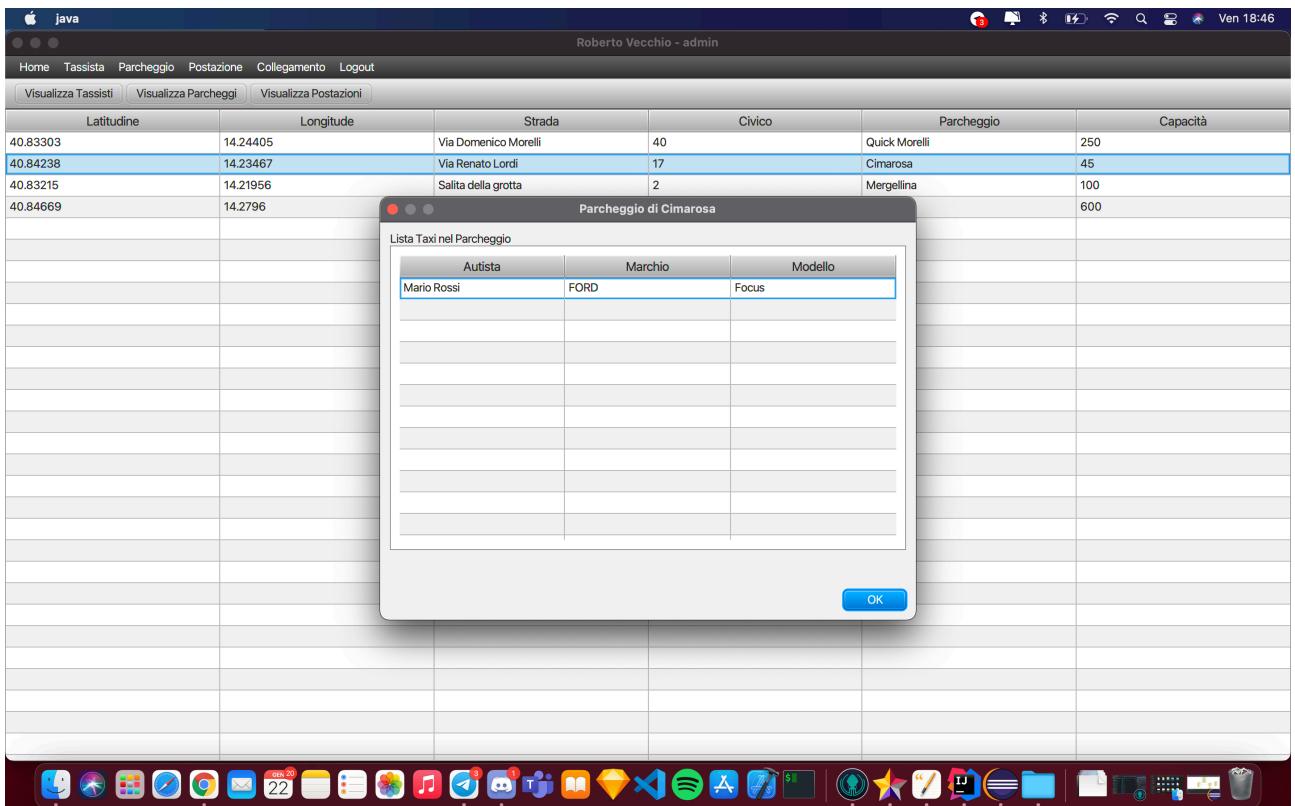
- Mostra collegamenti parcheggio;
- Mostra tassisti parcheggio;

Cliccando su “mostra collegamenti parcheggio” possiamo visualizzare con quali nodi è collegato quel determinato nodo:



A screenshot of the same Java application window, but now with a modal dialog box titled "Postazione Cimarsosa" overlaid. The dialog has a table titled "Lista Connessioni" with columns "Da" and "A". It shows two entries: "Parco Margherita" connected to "Cimarsosa" and "Cimarsosa" connected to "Vanvitelli". At the bottom right of the dialog is a blue "OK" button.

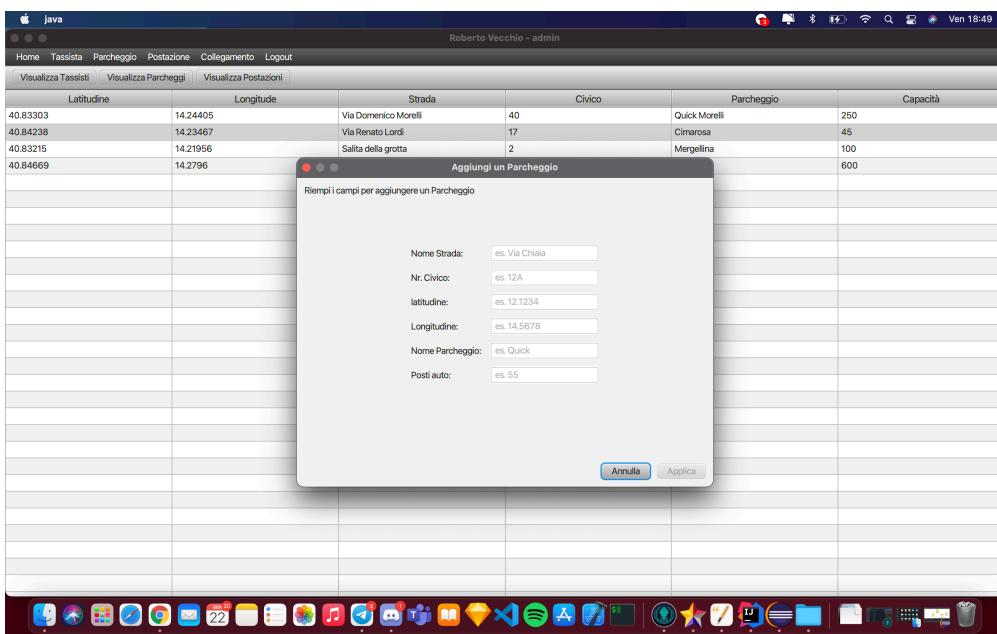
Mentre cliccando su “mostra tassisti parcheggio” possiamo visualizzare quali tassisti sono presenti all’interno di quel determinato parcheggio:



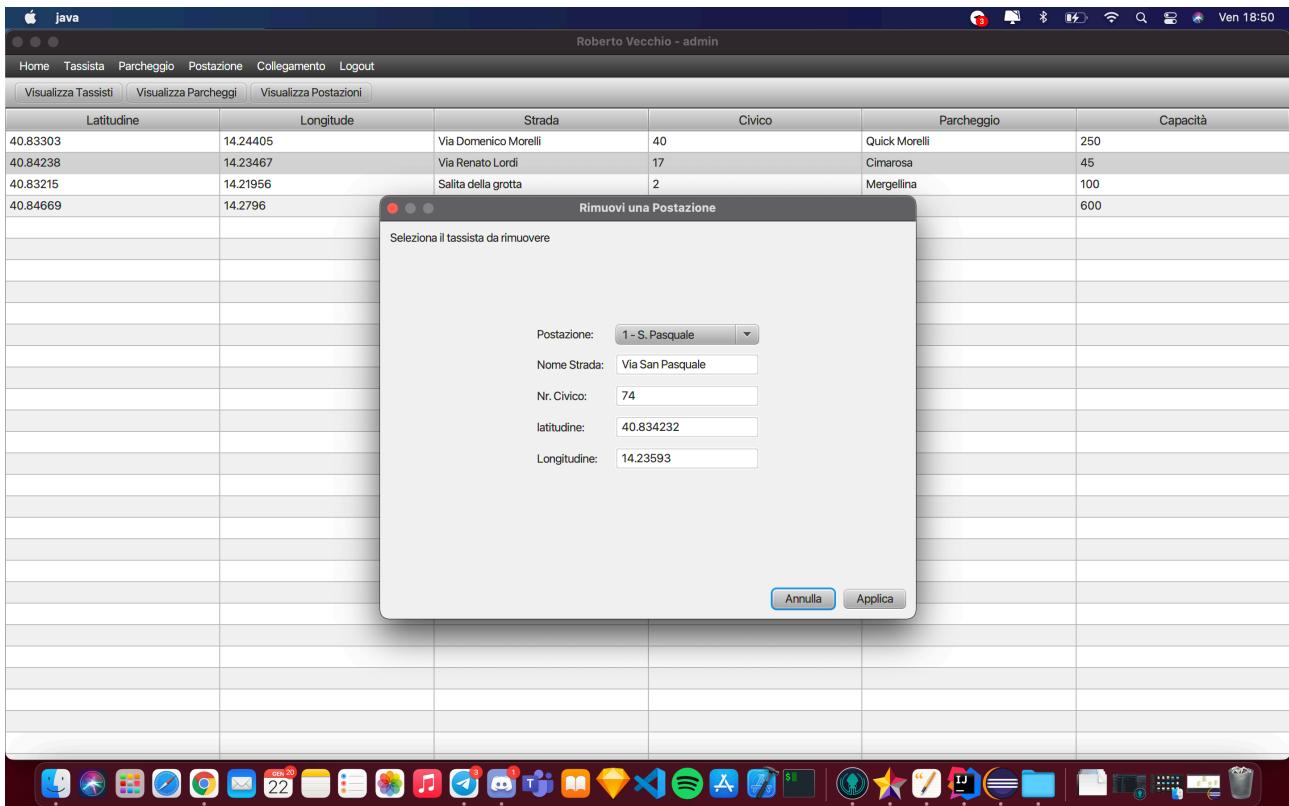
Cliccando sulla voce del menu parcheggio troveremo i seguenti sottomenu:

- Aggiungi parcheggio
- Rimuovi parcheggio

Cliccando su aggiungi parcheggio avremo la seguente schermata:



Cercando invece di rimuovere un parcheggio avremo la seguente:



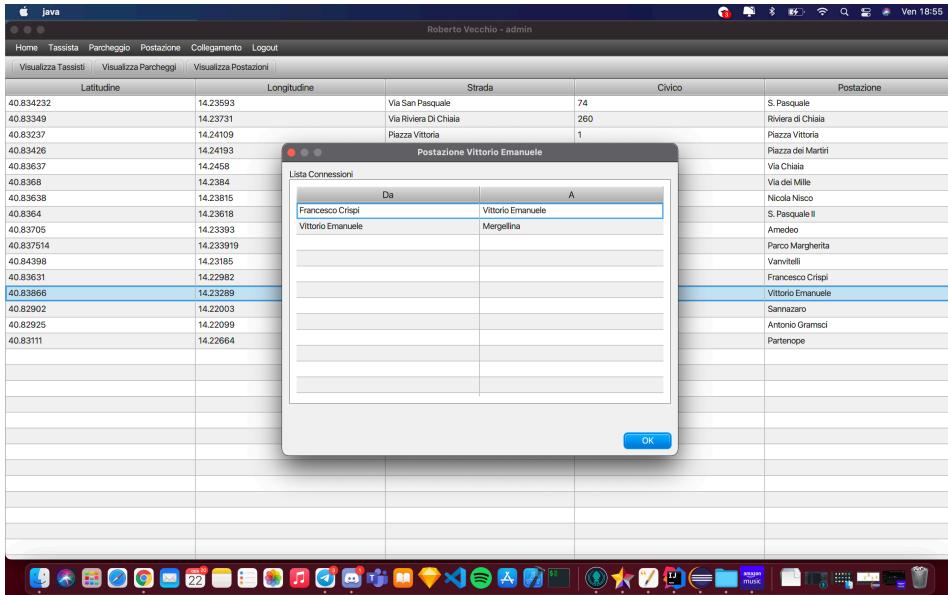
Cambiando sezione cliccando su “Visualizza postazioni” potremo visualizzare la lista postazioni (parcheggi esclusi, a cui viene dedicata una sezione a se):

Latitudine	Longitude	Strada	Civico	Postazione
40.834232	14.23593	Via San Pasquale	74	S. Pasquale
40.83349	14.23731	Via Riviera Di Chiaia	260	Riviera di Chiaia
40.83237	14.24109	Piazza Vittoria	1	Piazza Vittoria
40.83426	14.24193	Piazza Dei Martiri	1	Piazza dei Martiri
40.83637	14.2458	Via Chiaia	190	Via Chiaia
40.8368	14.2384	Via Del Mille	40	Via dei Mille
40.83638	14.23815	Via Nicola Nisco	18	Nicola Nisco
40.8364	14.23618	Via San Pasquale	30	S. Pasquale II
40.83705	14.23393	Piazza Amedeo	1	Amedeo
40.837514	14.233919	Via Del Parco Margherita	11	Parco Margherita
40.84398	14.23185	Piazza Vanvitelli	1	Vanvitelli
40.83631	14.22982	Via Francesco Crispi	86	Francesco Crispi
40.83866	14.23289	Corso Vittorio Emanuele	150	Vittorio Emanuele
40.82902	14.22003	Piazza Sannazzaro	10	Sannazzaro
40.82925	14.22099	Viale Antonio Gramsci	24	Antonio Gramsci
40.83111	14.22664	Via Francesco Caracciolo	20	Partenope

Cliccando con il tasto destro su uno degli elementi mostrati avremo le seguenti opzioni:

- Mostra collegamenti postazione

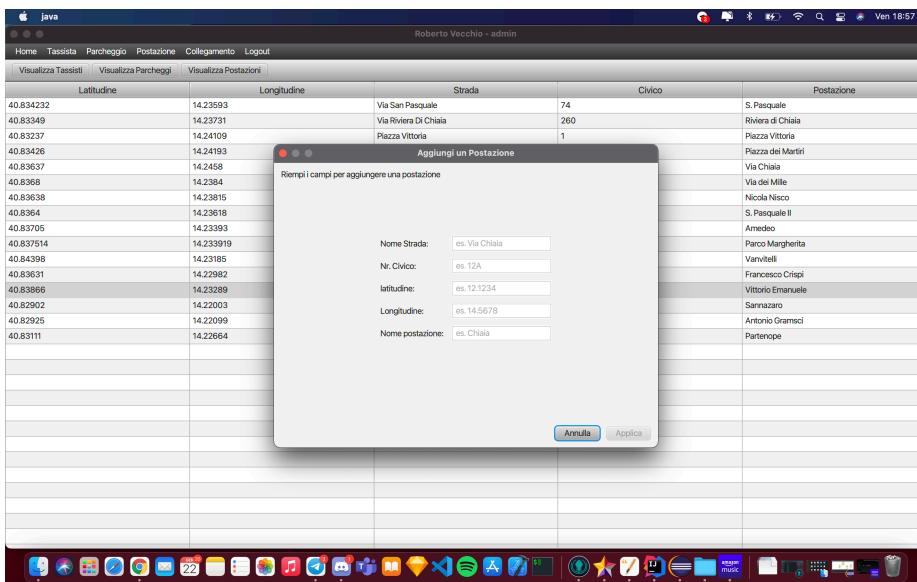
Cliccando su questa voce avremo la possibilità di visualizzare con quali nodi è collegato questo nodo:



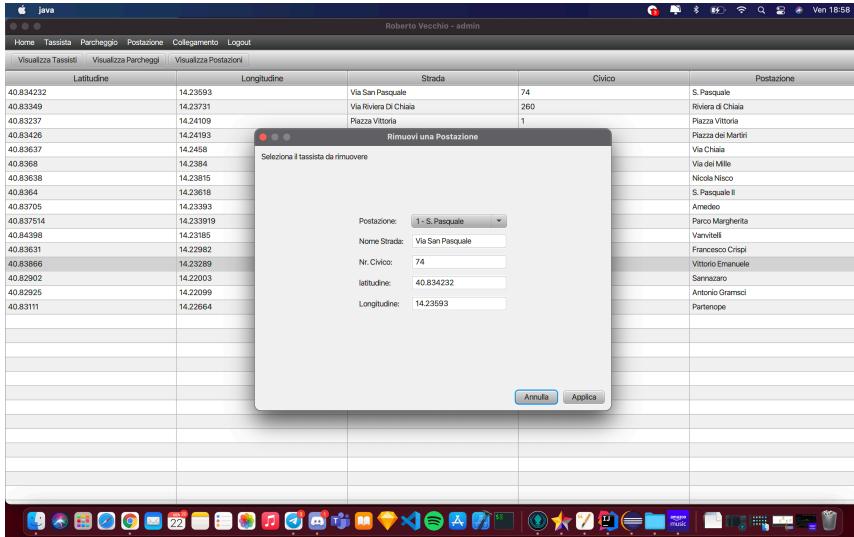
Cliccando sulla voce del menu postazione troveremo i seguenti sottomenu:

- Aggiungi postazione
- Rimuovi postazione

Cliccando su aggiungi postazione avremo la seguente schermata:



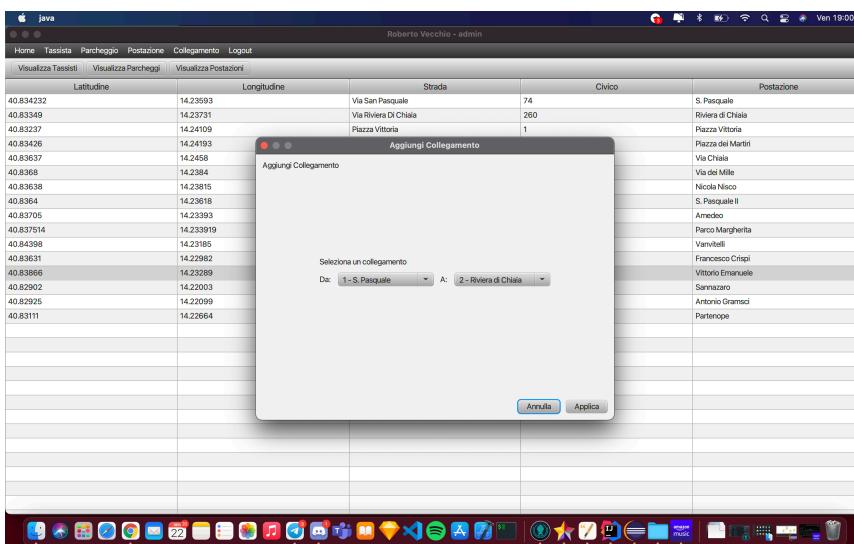
Cercando invece di rimuovere un parcheggio avremo la seguente:



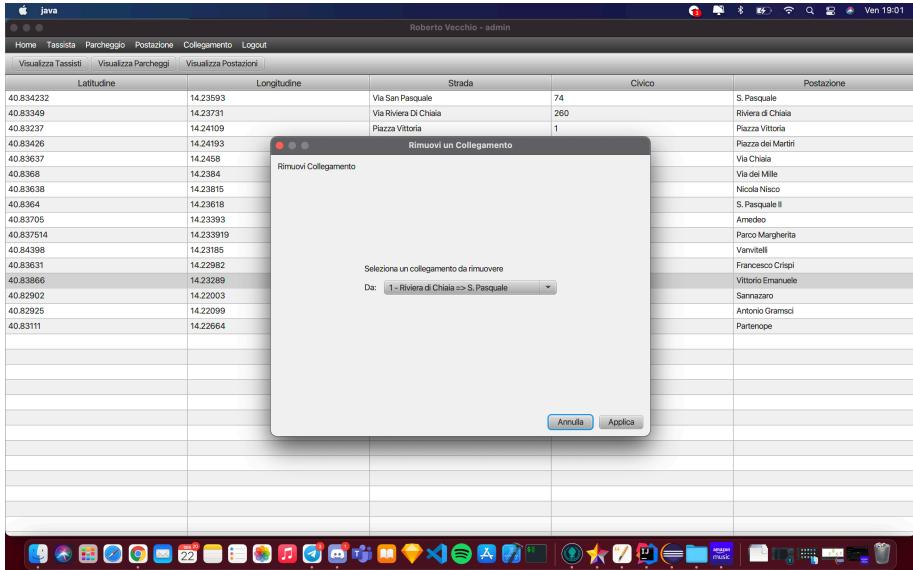
Cliccando sulla voce del menu collegamenti troveremo i seguenti sottomenu:

- Aggiungi collegamento
- Rimuovi collegamento

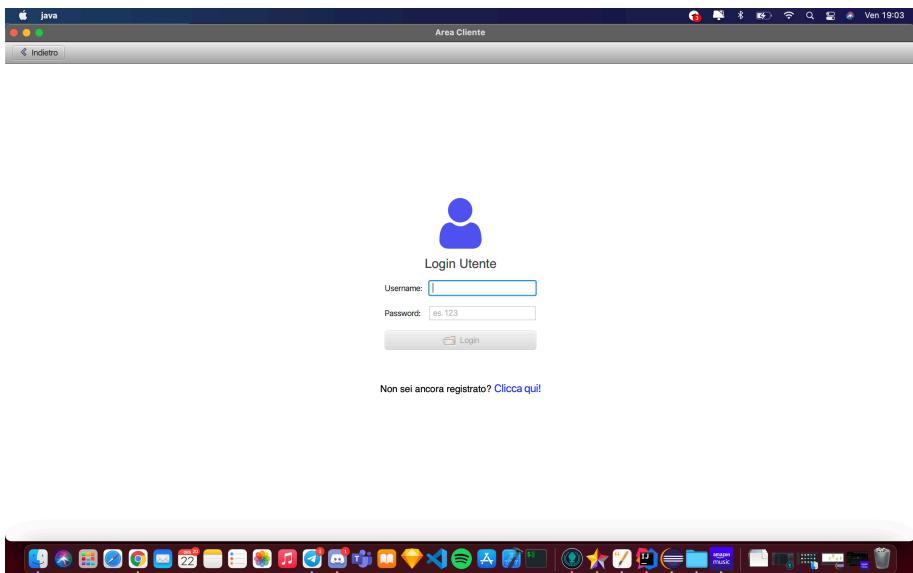
In questa sezione avremo l'occasione di gestire il collegamenti tra nodi. Cliccando su aggiungi collegamento avremo la seguente schermata:



Cercando di rimuoverlo invece avremo la seguente:



Cliccando infine su Logout torneremo alla schermata iniziale. Se provassimo ad entrare nell'area utente avremo davanti la seguente view:



Se volessimo registrare un nuovo utente basterà cliccare sul testo in blu che ha la dicitura “clicca qui”.

Registrazione

Codice Fiscale:

Nome: es: Roberto

Cognome: es: Vecchio

Data di nascita: es: 01/01/1993

Genere: es: Uomo

Email: es: test@gmail.com

Username: es: Roberto93

Password: es: 123

Numero di Telefono: es: 3283844389

Accetta condizioni:

[Qui puoi visualizzare Termini e condizioni](#)

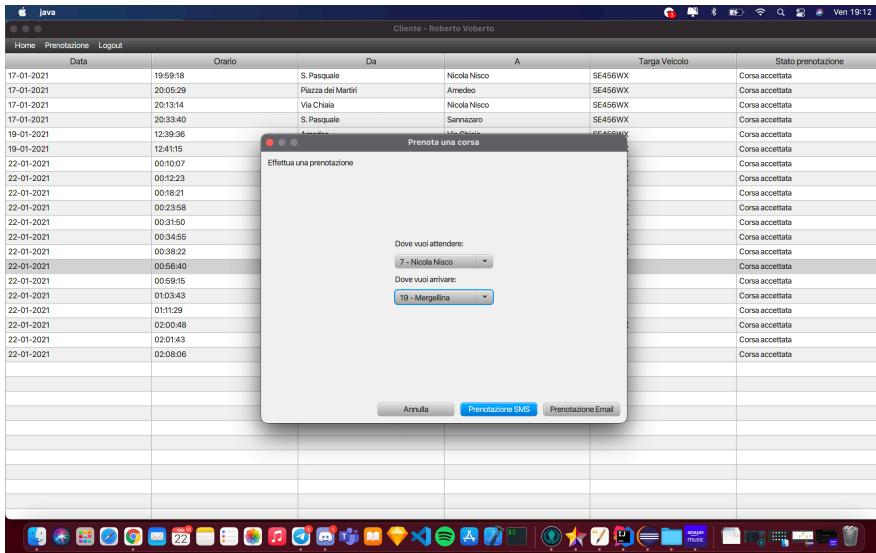
Qui potremo effettuare una registrazione inserendo opportunamente i dati richiesti. Una volta registrati si verrà rimandati in area di login per effettuare l'accesso con le credenziali precedentemente registrate, per testarne l'utilizzo l'utente è libero di creare un proprio account oppure di utilizzare quello di prova con le seguenti credenziali:

- Username: Test
- Password: 1234

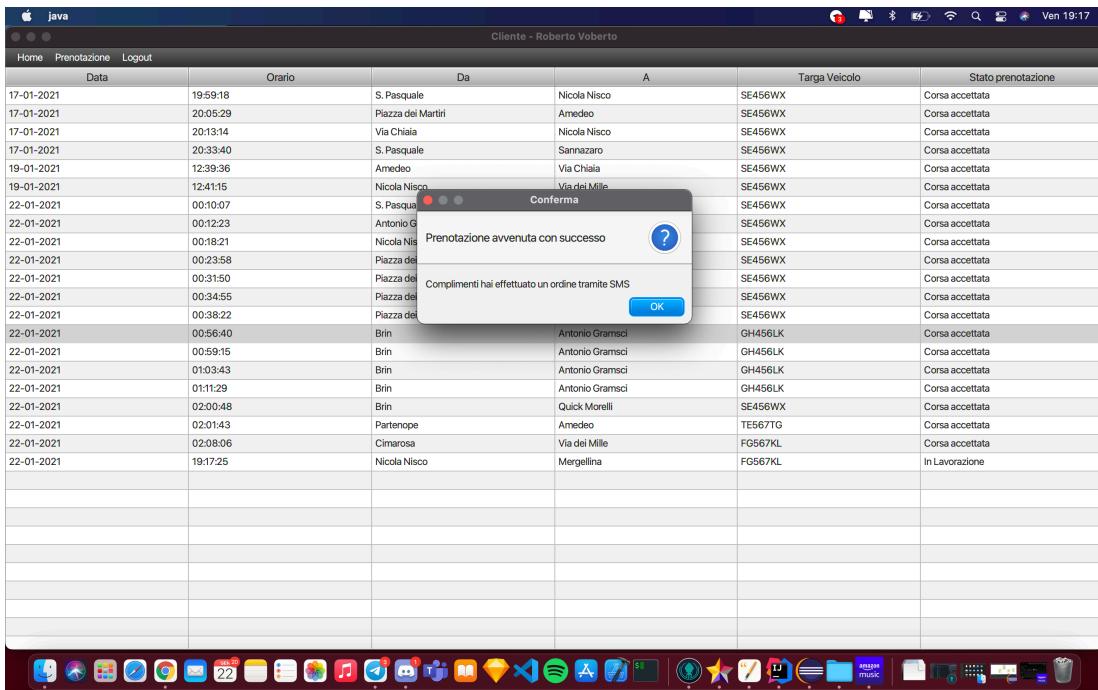
Accedendo potremo visualizzare la lista delle ordinazioni effettuate da parte dell'utente con informazioni annesse:

Data	Orario	Da	A	Targa Veicolo	Stato prenotazione
17-01-2021	19:59:18	S. Pasquale	Nicola Nisco	SE456WX	Corsa accettata
17-01-2021	20:05:29	Piazza dei Martiri	Amedeo	SE456WX	Corsa accettata
17-01-2021	20:13:14	Via Chiaia	Nicola Nisco	SE456WX	Corsa accettata
17-01-2021	20:33:40	S. Pasquale	Sannazzaro	SE456WX	Corsa accettata
19-01-2021	12:39:36	Amedeo	Via Chiaia	SE456WX	Corsa accettata
19-01-2021	12:41:15	Nicola Nisco	Via dei Mille	SE456WX	Corsa accettata
22-01-2021	00:10:07	S. Pasquale	Nicola Nisco	SE456WX	Corsa accettata
22-01-2021	00:12:23	Antonio Gramsci	Riviera di Chiaia	SE456WX	Corsa accettata
22-01-2021	00:18:21	Nicola Nisco	Quick Morelli	SE456WX	Corsa accettata
22-01-2021	00:23:58	Piazza dei Martiri	Via dei Mille	SE456WX	Corsa accettata
22-01-2021	00:31:50	Piazza dei Martiri	Mergellina	SE456WX	Corsa accettata
22-01-2021	00:34:55	Piazza dei Martiri	Mergellina	SE456WX	Corsa accettata
22-01-2021	00:38:22	Piazza dei Martiri	Amedeo	SE456WX	Corsa accettata
22-01-2021	00:56:40	Brin	Antonio Gramsci	GH456KL	Corsa accettata
22-01-2021	00:59:15	Brin	Antonio Gramsci	GH456KL	Corsa accettata
22-01-2021	01:03:43	Brin	Antonio Gramsci	GH456KL	Corsa accettata
22-01-2021	01:11:29	Brin	Antonio Gramsci	GH456KL	Corsa accettata
22-01-2021	02:00:48	Brin	Quick Morelli	SE456WX	Corsa accettata
22-01-2021	02:01:43	Portopecce	Amedeo	TE567TG	Corsa accettata
22-01-2021	02:08:06	Chiaia	Via dei Mille	FG567KL	Corsa accettata

Nel menu vi è una voce prenotazione che permetterà di effettuare una nuova prenotazione per una corsa (ammesso che non ne sia in corso una, in questo caso causerebbe un errore mostrato a video):



La prenotazione può avvenire tramite SMS o Email, ed una volta effettuata il sistema ne verificherà il successo in caso vi sia un tassista:



Come possiamo notare l'algoritmo di Dijkstra ha selezionato automaticamente il tassista più vicino associandolo all'ordine:

```

Parcheggio: Quick Morelli
Lunghezza: 1.5034517455377463
Parcheggio: Cimarosa
Lunghezza: 0.959103579939673
Parcheggio: Mergellina
Lunghezza: 1.6974821809432434
Parcheggio: Brin
Lunghezza: 7.430555772422862

```

La prenotazione appena effettuata rimane in lavorazione fintanto che il tassista non decide di partire selezionando una delle due modalità di raggiungimento, inoltre il suo stato passerà ad occupato come possiamo notare dal seguente screen della sezione “gestore”:

Codice Fiscale	Nome	Cognome	Data di nascita	Genere	Email	Targa Taxi	Nr. Licenza	Stato
VCRR193R06F839A	Roberto	Vecchio	11-01-2002	UOMO	robertovecchio@taxifinder...	SE456WX	LKPNFKN384DFNFSD	Libero
SPSVLR75E44F839R	Valeria	Esposito	19-01-1995	DONNA	valeriaesposito@taxifinder...	TE567TG	45346532	Libero
PSCFNN65M13F839C	Fernando	Pascariello	13-01-1988	UOMO	fernandopascariello@taxi...	LA345PE	98837894657	Libero
RSSMRA80A01F839W	Mario	Rossi	09-01-1998	UOMO	mariorossi@taxifinder.com	FG567KL	243494U2	Occupato

Ora effettuiamo l'accesso con il tassista che deve effettuare la corsa (tutti i tassisti hanno come username, il codice fiscale e come password, 1234, dove l'username è di default uguale al codice fiscale), qui possiamo visualizzare la lista degli ordini già effettuati, mentre per quello in corso vi è la sezione apposita nella toolbar:

The screenshot shows a mobile application interface for a taxi driver named Mario Rossi. At the top, there is a toolbar with icons for Home, Parcheggio, Logout, Visualizza ordini effettuati (which is highlighted in blue), and Nuovo ordine in pendenza. The status bar indicates the device is connected to a network, has a signal, and the time is Ven 19:29. A red dot in the top right corner of the status bar indicates the driver is Occupato (Busy). The main screen displays a table of completed trips with columns for Data, Orario, Da, A, Cliente, and Tipo Raggiungimento. The table contains five rows of data, all from 22-01-2021 at various times between 00:56:40 and 02:08:06, with destinations like Brin, Antonio Gramsci, Cimarosa, and Via dei Mille, and clients like Roberto Voberto and Antonio Gramsci. The last row is partially visible. Below the table is a large empty space, likely for map or other information. At the bottom, there is a dock with various app icons.

Data	Orario	Da	A	Cliente	Tipo Raggiungimento
22-01-2021	00:56:40	Brin	Antonio Gramsci	Roberto Voberto	LENGTH
22-01-2021	00:59:15	Brin	Antonio Gramsci	Roberto Voberto	LENGTH
22-01-2021	01:03:43	Brin	Antonio Gramsci	Roberto Voberto	LENGTH
22-01-2021	01:11:29	Brin	Antonio Gramsci	Roberto Voberto	LENGTH
22-01-2021	02:08:06	Cimarosa	Via dei Mille	Roberto Voberto	LENGTH

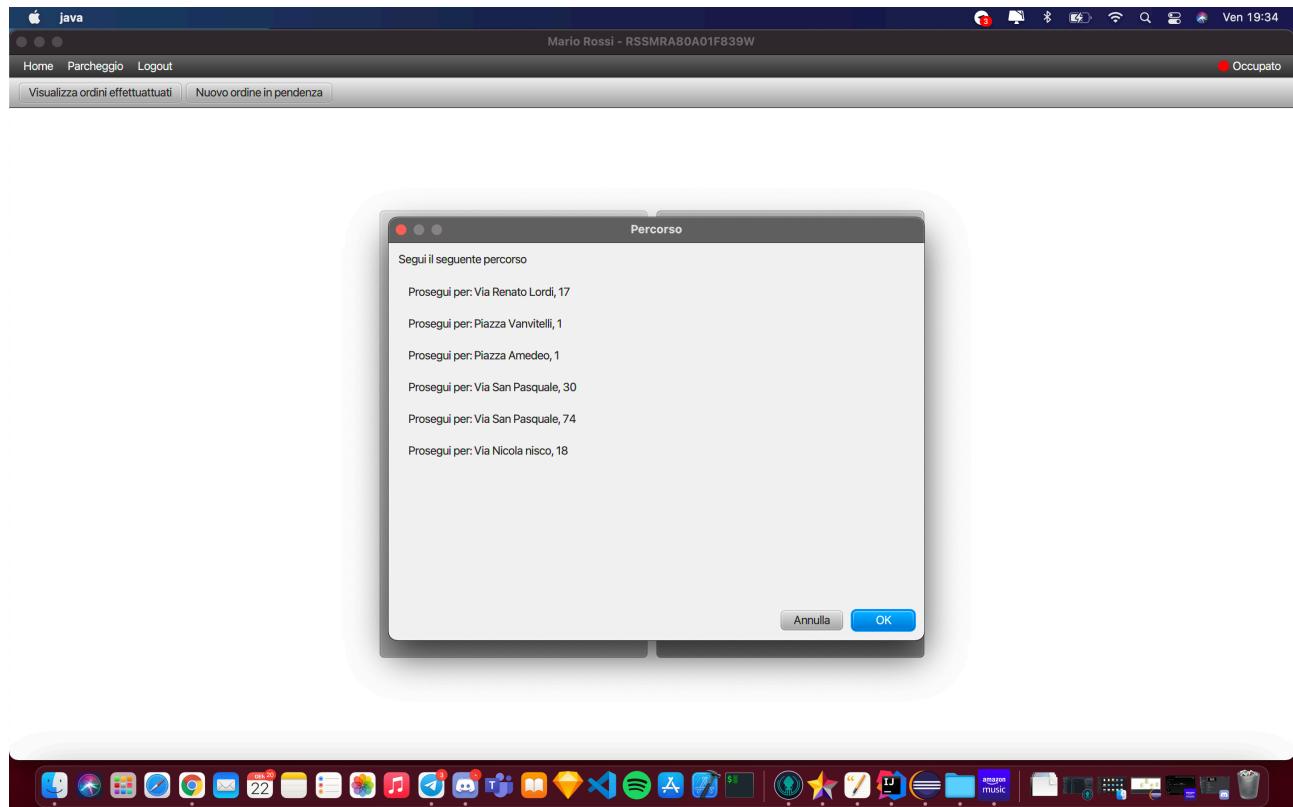
Come possiamo notare vi è un notificatore di stato nell'angolo in alto a destra che ci mostra se il tassista è occupato o meno. Nel caso in cui sia occupato questo potrà accettare l'ordine premendo sul tasto “nuovo ordine in pendenza”:

The screenshot shows the same mobile application interface for a taxi driver. The toolbar at the top is identical to the previous one. The main screen now displays two large rectangular buttons side-by-side. The left button is light gray and labeled "Percorso più breve". The right button is darker gray and labeled "Percorso con meno traffico". This indicates the driver can choose between the shortest route or a route with less traffic to accept the current pending order.

Potremo quindi scegliere:

- Percorso più breve
- Percorso con meno traffico

Cliccando ad esempio su percorso con meno traffico, il sistema visualizzerà a schermo il percorso da seguire:



Cliccando su Ok o annulla il sistema considererà preso in carico il cliente e quindi lo stato del cliente tornerà a libero modificando la view, così come possiamo apprezzare di seguito:

Di default il sistema, una volta completato un ordine re-inserisce il tassista ed il corrispettivo taxi nella una coda del parcheggio in cui era presente precedentemente, ma è comunque possibile notificare il sistema che si è in un parcheggio differente (in quanto il tassista potrebbe voler cambiare parcheggio), in questo caso basterà accedere alla sezione del menu parcheggio per notificare TaxiFinder del cambiamento:

