

# Sommario

---

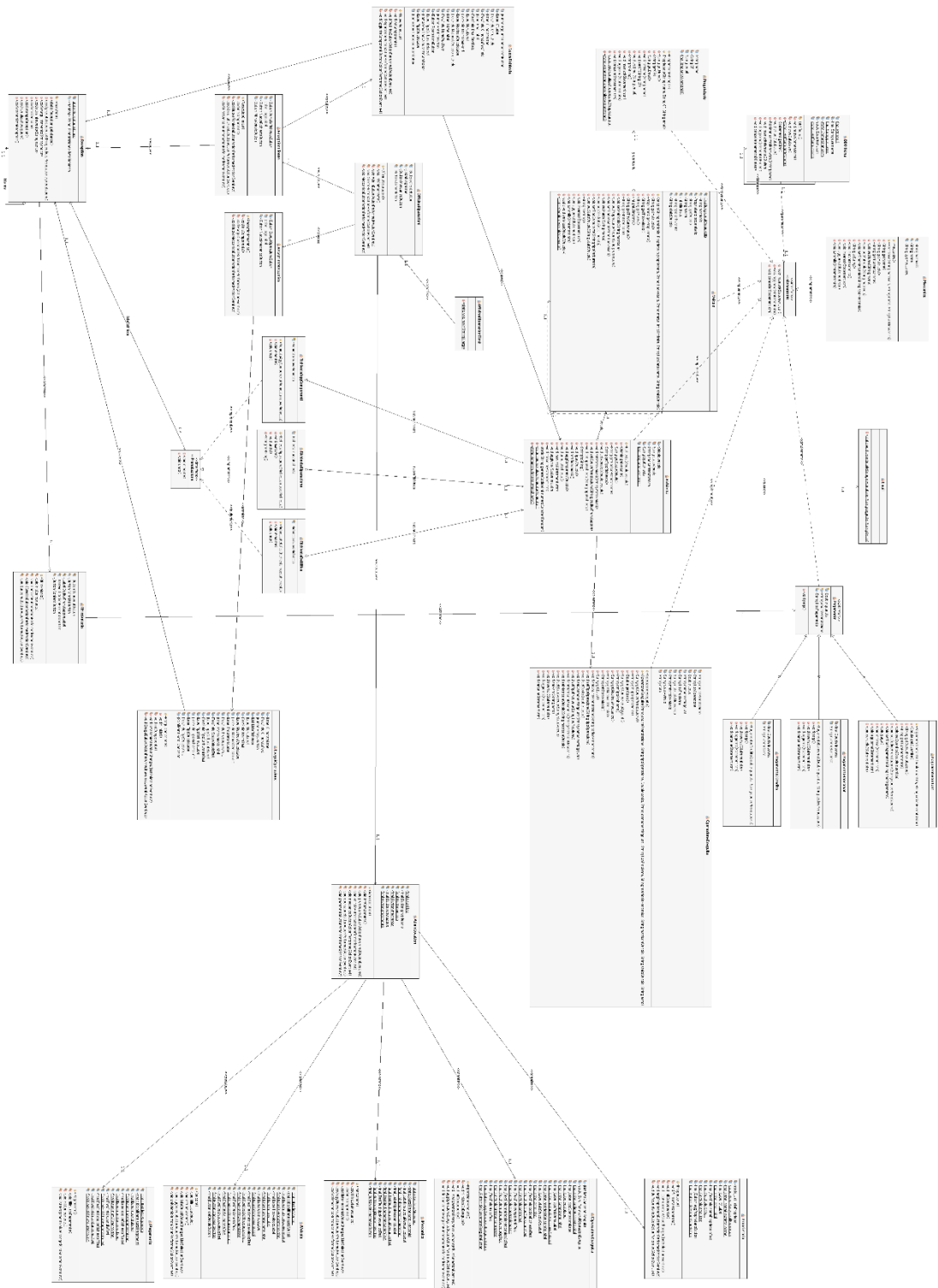
<b>1. Analisi dei Requisiti</b> .....	2
<b>2. Diagramma delle classi UML</b> .....	3
<b>3. Implementazione Java</b> .....	4
<b>3.1 Command Pattern</b> .....	4
<b>3.2 Singleton Pattern</b> .....	6
<b>4. JDBC</b> .....	7
<b>5. Serializzazione</b> .....	10
<b>6. Email</b> .....	12
<b>7. Interfaccia Grafica</b> .....	13
<b>8. Test di Esecuzione</b> .....	15
<b>8.1 Cliente</b> .....	15
<b>8.2 Meccanico</b> .....	18
<b>8.3 Admin</b> .....	19

# 1. Analisi dei Requisiti

---

I requisiti richiesti per la nostra applicazione riguardano la gestione di una officina per riparazioni di varie tipologie di veicoli, ognuno dei quali è contraddistinto da diverse caratteristiche come le informazioni sul tipo di veicolo, la matricola, la marca, la cilindrata ed il tipo di carburante utilizzato. Si memorizzano inoltre le informazioni circa il proprietario e si dà ad esso la possibilità di prenotare una specifica operazione nella suddetta officina, che può essere una riparazione, una rettifica, o una aggiunta di componenti al veicolo. Al termine della prenotazione il cliente ottiene una ricevuta, mediante la quale ritirare il veicolo al termine delle operazioni. Una volta effettuata la prenotazione lo stato del veicolo stesso all'interno del sistema è settato come *"in\_entrata"*, sarà dunque compito dei meccanici prendere in analisi tale richiesta e procedere nel completare la relativa operazione. Anche gli stessi meccanici sono memorizzati all'interno del nostro sistema e si presuppone che siano presenti nella nostra officina  $m$  meccanici per  $n$  tipologie di problema. Eseguita l'operazione richiesta il meccanico procede a memorizzare le informazioni relative ad essa e a notificare al cliente quanto effettuato, oltre all'esito dell'esecuzione, tramite l'invio di un'email e a cambiare il relativo stato del veicolo, settato come *"in\_uscita"*. Ricevuto il messaggio di notifica il cliente può dunque recarsi in officina e, tramite l'utilizzo della ricevuta da lui posseduta, può ritirare il proprio veicolo, non prima però di aver effettuato il pagamento del costo delle operazioni, possibile in contanti, carta di credito o bancomat.

# 2.Diagramma delle classi UML



## 3.Implementazione Java

---

L'implementazione e la risoluzione della problematica relativa ai requisiti analizzati avviene mediante l'utilizzo del linguaggio Java, in particolar modo cercando di sfruttarne al meglio le potenzialità e di rispettare le proprietà relative al paradigma di programmazione Object Oriented. Si fa uso in ciascuna classe di opportuni costruttori ed inoltre esse vengono dotati di *"accessor method"* ed *"mutator method"* al fine di mantenere per esse l'incapsulamento, oltre all'utilizzo di ereditarietà ed interfacce al fine di rendere estendibile l'intero codice. Infine l'intero software si basa su una soluzione di progettazione valida come il *Command Pattern*, ritenuto valido al fine di rispondere ai requisiti richiesti. Si noti infatti che il nostro sistema deve prevedere quelle che sono varie tipologie di richieste da parte dei client e che esse non vengono immediatamente processate, ma inserite in coda e al momento opportuno eseguite. Si ha dunque la necessità di accodare quelle che sono le varie richieste al fine di permettere ai meccanici di andarle ad analizzare quando risultano essere disponibili. Inoltre si fornisce ai clienti un metodo per annullare le richieste stesse.

### 3.1 Command Pattern

Il software sviluppato si basa dunque sull'applicazione di una soluzione progettuale verificata quale il Command Pattern. In particolar modo tale pattern incapsula una richiesta all'interno di un oggetto, consentendo la parametrizzazione dei client con diverse tipologie di richieste, accodare o rintracciare le richieste o supportare le operazioni di undo.

Il pattern "Command" cambia le richieste di operazioni su oggetti non specificati, trasformando le richieste stesse in oggetti. La base di questo pattern è l'interfaccia *"Prenotazione"*, ovvero l'interfaccia per l'esecuzione o annullamento delle operazioni.

```
public interface Prenotazione {  
    public void execute();  
    public void undo();  
}
```

Gli specifici comandi che eseguiti saranno classi che estendono questa interfaccia e specificano la coppia destinatario/azione, immagazzinando il destinatario come variabile di istanza e implementando i metodi *execute* ed *undo* per invocare una specifica richiesta.

In particolar modo le variabili di istanza che risultano essere gli esecutori materiali delle richieste sono oggetti di tipo *Richiesta*, classe che fornisce i metodi necessari al fine di eseguire tutte le possibili operazioni che un client può richiedere.

```
public class Richiesta {

    public void riparaVeicolo() {

        //
    }

    public void rettificaVeicolo() {

        //
    }

    public void aggComponentiVeicolo() {

    }

    public void annullaRichiesta() {

        //
    }

    private Veicolo veicolo;
    private String statoVeicolo;
    private String codicePrenotazione;
    private String tipoRichiesta;

}
```

Allo stesso modo i client non utilizzando direttamente tali tipologie di oggetti, ma viene fornito ad essi una interfaccia pubblica tale da permettergli di effettuare le varie richieste e fare in modo che esse siano mantenute in una coda.

```
public class Reception{

    public void eseguiIntervento(String codice) {

        //
    }

    public void ritiraIntervento(String codice) {

        //
    }

    public void ritiraVeicolo() {

        //
    }

    private HashMap<String, Prenotazione> listaRichieste;

}
```

Fondamentale per il command la coda di “prenotazione” quale “*listaRichieste*” il cui tipo di dato risulta essere un Haspmap, ovvero una collezione di oggetti il cui scopo principale è quello di rendere veloci ed efficienti operazioni quali inserimento e ricerca di elementi, attraverso la memorizzazione di coppie caratterizzate da chiave e valore.

## 3.2 Singleton Pattern

Altra tipologia di pattern utilizzato per lo sviluppo della nostra applicazione risulta essere il Singleton ,soluzione valida nel momento in cui si necessita di una sola istanza di classe e fornire per essa un punto di accesso globale .In particolare si è scelto tale pattern per l'opportuna gestione di alcune risorse come ad esempio l'accesso alla nostra reception oppure al nostro Database, e fornire per essi attraverso un opportuno metodo un punto di accesso valido nei vari punti del programma. L'implementazione fa uso di un metodo *getIstance()* al fine di restituire l'unica istanza della classe creata, dopo aver opportunamente reso private il costruttore della stessa e memorizzato come variabile di istanza proprio un elemento di tale classe che la rappresenti.

```
public class Reception {  
  
    private Reception() {  
        this.listaRichieste = new HashMap<>();  
    }  
  
    public static Reception getIstance() {  
        return istance;  
    }  
  
    private static final Reception istance = new Reception();  
    private HashMap<String, Prenotazione> listaRichieste;  
}
```

## 4.JDBC

---

Altra risorsa utilizzata per lo sviluppo del software officina è il Java Database Connectivity, ovvero lo strato di astrazione software che permette alle applicazioni java di connettersi ad un database che richiede come componenti fondamentali un implementazione del vendor del DBMS ed una applicazione tale da sfruttarlo.

I passi da seguire per l'utilizzo di tale strumento risultano alquanto semplici e prevedono:

1. Caricare i driver
2. Aprire una connessione con il Database
3. Creare un oggetto di tipo Statement per interrogare il Database
4. Interagire con esso
5. Gestire i risultati ottenuti

Le caratteristiche richieste vengono incapsulate in una classe che nel nostro sistema prende il nome di DBOfficina, implementata in Singleton, al fine di gestire al meglio tale risorsa.

```
public class DBOfficina {

    private DBOfficina() {
        creaDatabase();
    }

    private void nuovaConnessione() throws SQLException{

        if( (conn = DriverManager.getConnection(url,username,password)) == null)
            throw new SQLException();

    }

    private void creaDatabase(){
        try{
            nuovaConnessione();
            stm = conn.createStatement();

        }
        catch (SQLException e){
            JOptionPane.showMessageDialog(null,"ERRORE: Creazione Database non riuscita");
        }
        catch (Exception e){
            JOptionPane.showMessageDialog(null,"ERRORE: Accesso al Database non riuscito");
        }
    }

    public static DBOfficina getInstance(){
        if(conn==null || stm==null){
            instance = new DBOfficina();
        }
        return instance;
    }

    public Statement getStm() {
        return stm;
    }
}
```

```

public void chiudiDatabase() {
    try {

        if (stm != null) {
            stm.close();
        }
        if (conn != null) {
            conn.close();
        }
    } catch (Exception e) {
        JOptionPane.showMessageDialog(null, "ERRORE: Chiusura Database non riuscita");
    }
}

public boolean controlPassword(String pswd) {
    if (pswd.equals(password)) {
        return true;
    }

    return false;
}

public void inserisci(elementoDB elem) {
    elem.inserisci(stm);
}

public void aggiorna(elementoDB elem) {
    elem.aggiorna(stm);
}

public void cancella(elementoDB elem) {
    elem.cancella(stm);
}

private static final String url="jdbc:derby://localhost:1527/OfficinaRiparazioni;create=true";
private static final String username="officinaAdmin";
private static final String password="password";
private static Connection conn=null;
private static Statement stm=null;
private static DBOfficina instance = new DBOfficina();
}

```

In tale classe i driver vengono caricati mediante l'istruzione:

```
conn = DriverManager.getConnection(url,username,password)
```

Al seguito della quale è possibile ottenere un oggetto di tipo Statement mediante l'istruzione:

```
stm = conn.createStatement();
```

Infine tali elementi vengono chiusi mediante opportuno metodo di cui si è fornito la classe stessa.

Si noti che vengono supportate le operazioni di CRUD, attraverso l'utilizzo di apposite funzioni che, prevedono come argomento un oggetto di tipo *"elementoDB"* ovvero una interfaccia che astrae le operazioni di inserimento, aggiornamento e cancellazione nel Database, al fine di rendere estendibile il software a cambiamenti futuri che lo stesso potrà avere, o aggiungere ad esso nuove tipologie di elementi con esso compatibili.



```
public interface elementoDB {

    public void inserisci(Statement stm);
    public void aggiorna(Statement stm);
    public void cancella(Statement stm);
}
```

Utilizzando classi che implementano tale interfaccia si fa in modo che essi siano dotate di opportuni metodi che supportano tali operazioni, senza andare a modificare la classe principale che ci permette la gestione del nostro Database. Tutto ciò si basa su una condizione fondamentale, ovvero che tali elementi siano compatibili con il Database stesso, ovvero esistano per essi tabelle fisiche in grado di conservarne le relative informazioni. Un esempio è rappresentato dalla seguente classe Meccanico:

```
public class Meccanico implements elementoDB{

    public Meccanico(String matricola,String nome,String tipoMeccanico) {
        this.nome = nome;
        this.matricola = matricola;
        this.tipoMeccanico = tipoMeccanico;
    }

    @Override
    public void inserisci(Statement stm) {
        String query = String.format("INSERT INTO MECCANICO (MATRICOLA,NOME,TIPO_MECCANICO)"
            + " VALUES ('%s','%s','%s')", this.matricola, this.nome, this.tipoMeccanico);
        try {
            stm.executeUpdate(query);
        } catch (SQLException e) {
            JOptionPane.showMessageDialog(null, "ERRORE: Inserimento Meccanico non riuscito");
        }
    }

    @Override
    public void aggiorna(Statement stm) {
        String query = String.format("UPDATE MECCANICO SET NOME = '%s',"
            + "TIPO_MECCANICO = '%s' WHERE MATRICOLA= '%s'", this.nome,
            this.tipoMeccanico, this.matricola);
        try {
            stm.executeUpdate(query);
        } catch (SQLException e) {
            JOptionPane.showMessageDialog(null, "ERRORE: Aggiornamento Meccanico non riuscito");
        }
    }

    @Override
    public void cancella(Statement stm) {
        String query = String.format("DELETE FROM MECCANICO WHERE MATRICOLA = '%s'",this.matricola);
        try {
            stm.executeUpdate(query);
        } catch (SQLException e) {
            JOptionPane.showMessageDialog(null,"ERRORE: Cancellazione Meccanico non riuscito");
        }
    }

    private String matricola;
    private String nome;
    private String tipoMeccanico;
}
```

## 5.Serializzazione

---

Si fa uso inoltre di un'altra importante caratteristica legata a java, ovvero il concetto di serializzazione. Tale attività permette di salvare lo stato di un'oggetto in uno stream di byte che successivamente può essere memorizzato in un file persistente. Al contrario la serializzazione risulta essere il processo inverso, rispetto cui si ripristina tale stato a partire da uno stream di byte conservato opportunamente. Per eseguire tali attività bisogna necessariamente che le classi di nostro interesse implementino l'interfaccia *Serializable* e definire al loro interno una versione della classe, utile in fase di deserializzazione per verificare che le classi usate da chi ha serializzato l'oggetto e da chi lo deserializza siano tra loro compatibili. Ciò risulta essere quanto implementato ad esempio nella classe *Proprietario*, soggetta proprio a tale procedimento :

```
public class Proprietario implements Serializable{
```

```
    public Proprietario(String nome, String CF, String email) {  
        this.nome = nome;  
        this.CF = CF;  
        this.email = email;  
    }
```

```
    public static void serializzaProprietario(Proprietario p){  
        ObjectOutputStream output=null;  
        try{  
            output=new ObjectOutputStream(new FileOutputStream("dati.dat"));  
        }  
        catch (FileNotFoundException e) {  
            e.printStackTrace();  
        }  
        catch(IOException ioe){  
            ioe.printStackTrace();  
        }  
        try {  
            output.writeObject(p);  
        } catch (IOException e1) {  
            e1.printStackTrace();  
        }  
        try {  
            output.close();  
        }  
        catch (IOException e2) {  
            e2.printStackTrace();  
        }  
        System.out.println("Serializzazione completata.");  
    }
```

```

public static Proprietario deserializzaProprietario(){
    ObjectInputStream ois = null;
    try {
        ois = new ObjectInputStream(new FileInputStream("dati.dat"));
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
    Proprietario p = null;
    try {
        p = (Proprietario) ois.readObject();
    } catch (IOException e1) {
        e1.printStackTrace();
    } catch (ClassNotFoundException e1) {
        e1.printStackTrace();
    }

    return p;
}

private String nome;
private String CF;
private String email;
private static final long serialVersionUID = 1L;
}

```

## 6.Email

---

Il sistema di gestione della nostra officina, secondo i requisiti, richiede inoltre un modo attraverso cui notificare al cliente le avvenute operazioni relative al veicolo da lui posseduto, si fornisce dunque al software un meccanismo tale da permettere l'invio di semplici email al termine delle operazioni da parte di un meccanico. Tutto ciò viene implementato nella classe di utilità *Email* che possiede un unico metodo statico consono a tale caratteristica.

```
public class Email {
    public static void sendEmail(String destinatario,String oggetto,String testo) {

        final String username = "testappdeveloper@yahoo.com";
        final String password = "password94";

        Properties props = new Properties();
        props.put("mail.smtp.auth", "true");
        props.put("mail.smtp.socketFactory.port", "465");
        props.put("mail.smtp.socketFactory.class", "javax.net.ssl.SSLSocketFactory");
        props.put("mail.smtp.host", "smtp.mail.yahoo.com");
        props.put("mail.smtp.port", "465");

        Session session = Session.getInstance(props,
            new javax.mail.Authenticator() {
                protected PasswordAuthentication getPasswordAuthentication() {
                    return new PasswordAuthentication(username, password);
                }
            });

        try {

            Message message = new MimeMessage(session);
            message.setFrom(new InternetAddress("testappdeveloper@yahoo.com"));
            message.setRecipients(Message.RecipientType.TO,
                InternetAddress.parse(destinatario));
            message.setSubject(oggetto);
            message.setText(testo);

            Transport.send(message);
            JOptionPane.showMessageDialog(null, "Email inviata al cliente");

        } catch (MessagingException e) {
            JOptionPane.showMessageDialog(null, "ERRORE : invio email non riuscito");
        }

    }
}
```

## 7. Interfaccia Grafica

---

Si è dotato infine il software di varie interfacce grafiche, al fine di permettere all'utente utilizzatore finale di interagire con il sistema attraverso l'utilizzo delle stesse. In particolare il meccanismo adottato per lo sviluppo delle finestre prevede la creazione di classi che estendo la superclasse *JFrame* e settando per esse i relativi *JPanel* e componenti che le contraddistinguono. Tali componenti inoltre sono in grado di ascoltare determinati input da parte dell'utente e di reagire secondo la logica applicativa per essi prestabilita, attraverso il cosiddetto modello ad eventi che associa ad ognuna di tali componenti che può generare un determinato evento (es. pressione di un bottone) uno o più ascoltatori in grado di gestirlo. In particolare gli eventi di tipo "ActionEvent" vengono generati in corrispondenza di azioni originate dall'interazione dell'utente con le componenti del nostro Frame, come ad esempio il click di un bottone. La gestione di tali eventi avviene attraverso l'implementazione di un *ActionListener*, ovvero una interfaccia caratterizzata da un solo metodo:

```
public interface ActionListener {  
    void actionPerformed(ActionEvent e);  
}
```

Tale gestione prevede:

1. La definizione di una classe che implementi *ActionListener* e ne ridefinisca il metodo *actionPerformed*, specificando in esso la logica di gestione dell'evento.
2. Registrare una istanza di tale classe come listener di un determinato componente dei nostri Frame attraverso il metodo *addActionListener(ActionEvent e)*.
3. Quanto l'utente interagisce con tale componente viene creato un evento *ActionEvent* gestito secondo la logica prestabilita, ovvero invocando il metodo *actionPerformed* dei listener registrati.

Tale schema risulta essere a pieno rispettato durante lo sviluppo del nostro sistema, ed in particolare gli ascoltatori vengono implementati attraverso l'utilizzo delle cosiddette *inner class anonime*, ovvero classi innestate anonime che hanno la prerogativa di accedere agli elementi, anche private, delle classi in cui esse risultano essere definite ed in quanto anonime andranno anche ad istanziate. Esempio di gestione di un evento:

```
public class OfficinaRiparazioni extends JFrame {

    AdminButton.addActionListener(new ActionListener(){
        @Override
        public void actionPerformed(ActionEvent evt){
            AdminButtonActionPerformed(evt);
        }
    });

}

private void AdminButtonActionPerformed(ActionEvent evt){
    JPasswordField pwd = new JPasswordField(10);
    JOptionPane.showConfirmDialog(null, pwd, "Inserire Password", JOptionPane.OK_CANCEL_OPTION);
    if(DBOfficina.getInstance().controlPassword(pwd.getText())){
        AdminLocation ad = new AdminLocation();
    }
    else{
        JOptionPane.showMessageDialog(null, "ERRORE: La password da lei inserita non è valida");
    }
}

private JLabel LogoLabel;
private JButton AdminButton;
}
```

## 8.Test di Esecuzione

---

Vengono in seguito riportati vari esempi di esecuzione del nostro sistema, corredati di screen relativi alle interfacce sviluppate.

### 8.1 Cliente



Intro della nostra applicazione, permette all'utente di loggarsi nel sistema come Cliente, Meccanico o amministratore, quest'ultimo solo attraverso l'inserimento di una password valida.



Nel caso in cui sia un cliente ad interagire con il sistema, si permette ad esso di selezionare se desidera effettuare una nuova richiesta, annullarla o ritirare il veicolo, se le operazioni sono terminate.

Schermata che permette all'utente di inserire tutte le informazioni necessarie per registrare nel sistema una nuova operazione da eseguire da parte dei meccanici, e di selezionare il tipo della richiesta stessa.



Se invece l'utente seleziona l'annullamento di una richiesta di essa ne deve essere verificato il codice, inserito dall'utente.

Selezionando invece il ritiro veicolo compare la seguente schermata, in cui si chiede all'utente di inserire il codice che esso ha ottenuto al momento della prenotazione relativo alla sua ricevuta e di selezionare eventualmente il metodo di pagamento.

In particolare il pagamento è gestito praticamente in Java attraverso la definizione di una classe astratta che, nel momento in cui viene estesa implementa nelle sottoclassi le operazioni che contraddistinguono ogni singola tipologia di pagamento.

```
public abstract class Pagamento implements elementoDB{

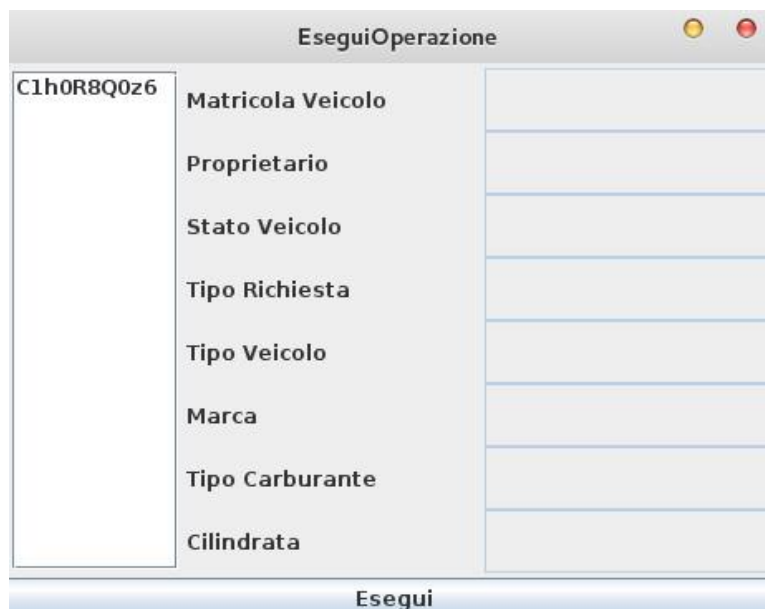
    public abstract void paga();

    Double quantita;
    String codicePrenotazione;
    String tipoPagamento;
}
```

## 8.2 Meccanico



Per quanto riguarda i meccanici invece si dà ad essi la possibilità di eseguire una operazione in coda nel momento in cui essi risultano essere disponibili, oppure di aggiungere nel sistema un nuovo collega.



EseguiOperazione	
C1h0R8Q0z6	Matricola Veicolo
	Proprietario
	Stato Veicolo
	Tipo Richiesta
	Tipo Veicolo
	Marca
	Tipo Carburante
	Cilindrata
Esegui	

La scelta di una determinata operazione da eseguire da parte dei meccanici viene effettuata mostrando ad essi i codici delle varie prenotazioni, e cliccandoli compare a lato l'intera informazione riguardante la richiesta, e analizzandola si può decidere se eseguirla o meno, registrandola nel sistema ed inviando una email al proprietario.

### 8.3 Admin



Se infine un utente esegue il login nel sistema come amministratore inserendo una valida password si permette ad esso di interagire con la gran parte degli elementi della nostra applicazione, consentendo ad esso operazioni di lettura, aggiornamento e cancellazione.



Ne è un esempio la gestione dei veicoli proprio da parte degli amministratori di sistema.