

# Università degli Studi di Napoli “Parthenope”

*Scuola delle Scienze, dell'Ingegneria e della Salute*

*Corso di laurea in Informatica*

*Corso di “Programmazione 3 e Laboratorio di Programmazione 3”*

*Traccia: Monitoraggio ambientale*



*Proff Angelo Ciaramella, Raffaele Montella*

*Candidato: Pierno Francesco Pio*

*Matricola: 0124001360*

*Anno accademico: 2020/2021*

*Tipo di progetto: Progetto individuale*

## Sommario

<b>Descrizione e requisiti del progetto .....</b>	<b>3</b>
<b>UML delle classi.....</b>	<b>4</b>
OBSERVER.....	5
SINGLETON .....	7
STRATEGY .....	8
<b>Elementi importanti del codice .....</b>	<b>9</b>
Principi SOLID .....	9
Dettagli implementativi .....	10
Inserimento di un sensore .....	10
Modifica delle tuple e spostamento del traffico .....	11
Stato dei sensori .....	12
Plot del grafico .....	14
<b>Fonti e note .....</b>	<b>16</b>

# Descrizione e requisiti del progetto

Si vuole sviluppare un sistema per la gestione di una *Smart City*. Una *Smart City* comprende un insieme di strategie di pianificazione urbanistica per migliorare la qualità della vita e soddisfare le esigenze di cittadini, imprese e istituzioni.

Si suppone di gestire un sistema di monitoraggio ambientale. In una città, sono presenti delle centraline contenenti dei sensori di monitoraggio che permettono di registrare i livelli di tre parametri:

- Inquinamento dell'aria
- Temperatura
- Numero di autoveicoli che transitano

L'amministratore del sistema fissa una soglia di guardia per ogni parametro; in questo caso 37° per la temperatura, 60 per l'inquinamento dell'aria e 60 per il numero di autoveicoli.

Il sistema di monitoraggio si può trovare in questi tre stati:

- Codice verde – Se tutti i tre parametri sono sotto soglia
- Codice giallo – Se i primi due parametri (inquinamento dell'aria e temperatura) sono sopra soglia
- Codice rosso – Se tutti i tre parametri sono sopra soglia

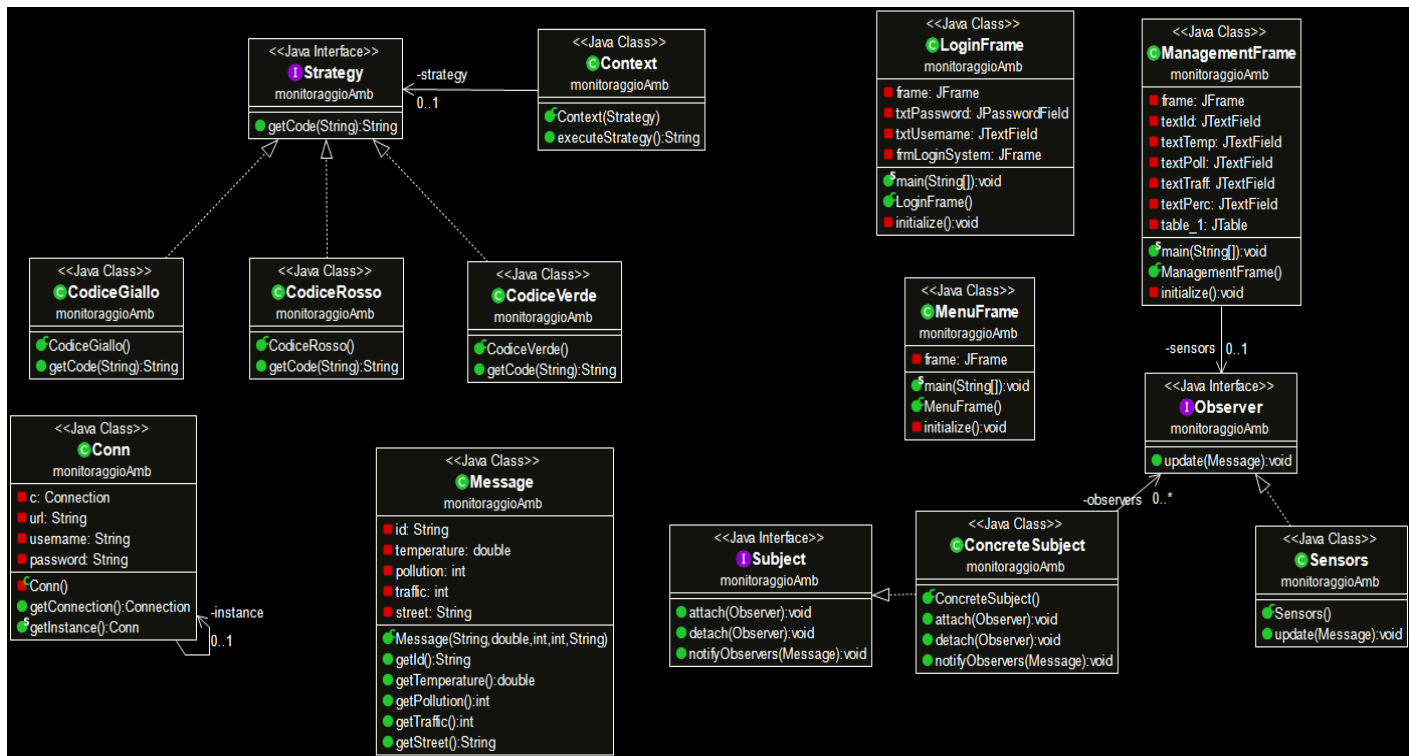
In presenza del codice rosso, l'amministratore può inviare il flusso del traffico su un altro percorso. Inoltre l'admin può aggiungere sensori e visualizzare il grafico statistico fissato per un certo periodo.

Il progetto è stato implementato secondo le seguenti linee:

- Per i programmi standalone con supporto grafico si deve far uso di almeno due design pattern. I pattern utilizzati per lo svolgimento della traccia sono **Observer**, che permette di definire una dipendenza uno a molti fra oggetti, in modo tale che se un oggetto cambia il suo stato interno, ciascun degli oggetti viene notificato automaticamente; **Strategy**, che consente di isolare un algoritmo al di fuori di una classe, per far sì che quest'ultima possa variare dinamicamente il suo comportamento; **Singleton**, che ha lo scopo di garantire la creazione di una sola istanza della classe (nel nostro caso con la connessione al database).
- Rispetto di alcuni dei principi **SOLID**
- Sufficienti commenti per il codice e in **javadoc**
- **Interfaccia grafica** che permette una chiara visualizzazione all'utente
- Uso dei database per conservare e gestire le informazioni relativi ad ogni sensore presente nella città

# UML delle classi

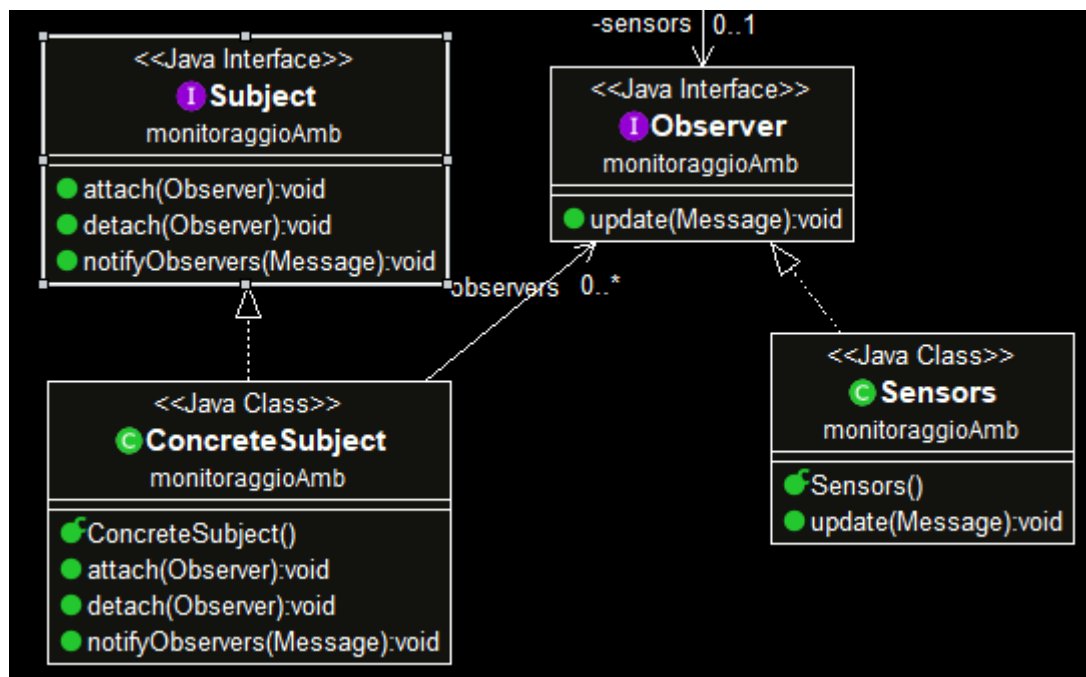
In questo paragrafo viene riportato il diagramma UML creato tramite il plugin di Eclipse *Objectaid UML Diagram*. Le classi **LoginFrame**, **MenuFrame**, e **ManagementFrame** sono state create tramite il plugin di Eclipse *WindowsBuilder*.



Possiamo notare che le classi **LoginFrame**, **MenuFrame**, **ManagementFrame** andranno a gestire tutti gli eventi all'interno del sistema di monitoraggio, tra cui il login dell'amministratore, visualizzazione del grafico statistico all'interno del menù di monitoraggio, e la gestione dei sensori, in cui si potrà eseguire l'inserimento dei sensori, visualizzare lo stato di ogni sensore presente in città per ogni strada, e la modifica/spostamento del traffico su un altro percorso

Il **LoginFrame** prenderà in input i dati inseriti all'interno dei campi *txtPassword* e *txtUsername*. Se inseriamo dati errati per il login, spunterà un messaggio di errore, tramite la classe *JOptionPane*. In caso in cui l'admin effettua l'accesso, potrà entrare nel **MenuFrame**.

## OBSERVER



Il **ManagementFrame** utilizzerà i dati inseriti nei vari `JTextField` e verranno salvati in una variabile di tipo **Sensor**, che implementa l'interfaccia **Observer**. Quest'ultima infatti, andrà ad implementare il design pattern **Observer**, costituita dalle classi **Subject**, che dichiara i metodi

- **Attach(Observer):void** - Metodo che permetterà di aggiungere gli Observer e di collegarli
- **Detach(Observer):void** – Metodo di eliminazione dell'Observer
- **NotifyObservers(Message):void** – Metodo che notificherà il cambiamento di stato a tutti gli Observers collegati

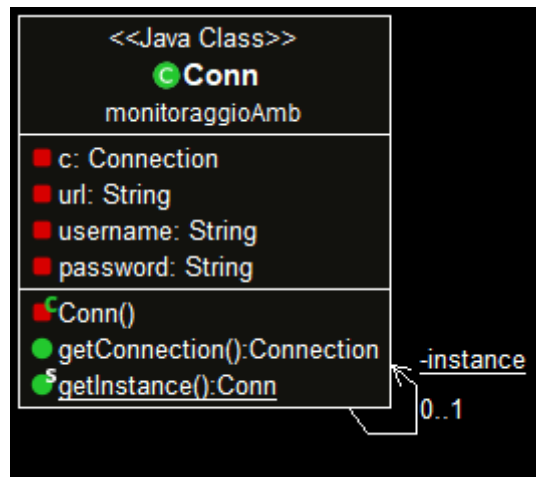


Questi metodi verranno implementati dalla sottoclasse **ConcreteSubject**. Si vuole notare come alcuni metodi prendono in input un oggetto di tipo **Message**. Questa classe ha lo scopo di dichiarare e di restituire tutte le informazioni per i sensori, tramite il proprio costruttore **Message (String, double, int , int, String)** ed i metodi "get"

- **GetId():void** – Metodo che restituisce l'id del sensore
- **GetTemperature():void** – Metodo che restituisce la temperatura misurata
- **GetPollution():int** – Metodo che restituisce la misura relativa all'inquinamento dell'aria
- **GetTraffic():int** – Metodo che mi restituisce il numero di autoveicoli transitati
- **GetStreet():void** – Permette di restituire il nome della strada

La classe **Sensor** implementerà il metodo dichiarato dall'interfaccia **Observer** **update(Message):void** che ha lo scopo di stampare/aggiornare il mio sensore.

## SINGLETON



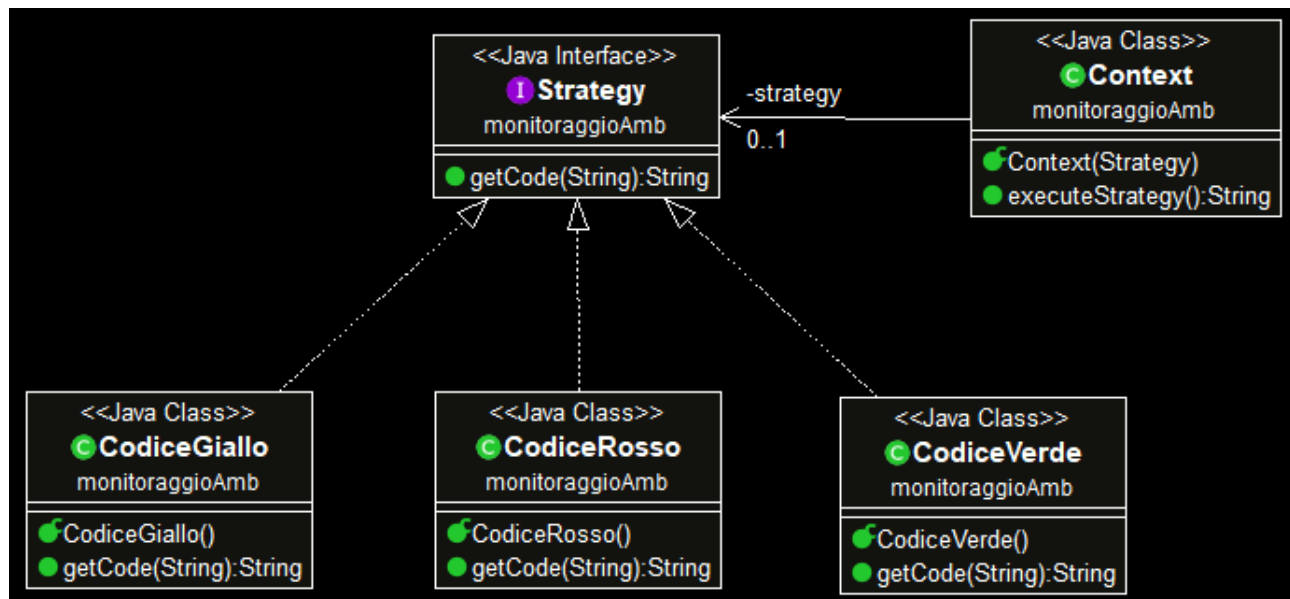
La classe **Conn** consente di andare a collegare il nostro IDE al nostro database MySQL. Per l'implementazione di questa classe è stato utilizzato il design pattern **Singleton**, che come spiegato nel paragrafo precedente, ha lo scopo di creare un'unica istanza di accesso all'interno della classe stessa.

Per collegare il nostro IDE al database, bisogna importare come *"External Jar"* il **JConnector** relativo al nostro database MySQL, in cui si andranno successivamente ad inserire all'interno del costruttore privato **Conn()** accompagnato dalla clausola **throws SQLException** e gestito tramite un blocco try-catch per gestire le eccezioni (si ricorda che il comando **throws** permette di dire al compilatore, che il blocco try-catch, potrebbe lanciare qualche eccezione), i dati che ci servono per effettuare il collegamento tra cui

- L'url al database
- L'username
- Password

Il metodo **getConnection()** andrà a restituire la nostra connessione appena creata ed il metodo **getInstance()** la nostra istanza.

## STRATEGY



Per quanto riguarda la gestione del codice verde, giallo e rosso, è stato utilizzato il design pattern **Strategy**. Si tratta di un pattern comportamentale basato su oggetti e viene utilizzato per definire una famiglia di algoritmi, incapsularli e renderli intercambiabili. Le classi che andranno a costituire tale pattern sono

- **Strategy** – interfaccia che dichiara il metodo `getCode()` e che vorrà invocata dal **Context**
- **Context** – classe che detiene le informazioni di contesto ed ha il compito di invocare l'algoritmo
- **CodiceVerde**, **CodiceGiallo**, **CodiceRosso** – classi che implementano la classe **Strategy** e che effettuano l'overwrite all'interno della classe **Context** all'interno del metodo **`executeStrategy():String`** al fine di ritornare l'implementazione dell'algoritmo.



# Elementi importanti del codice

## Principi SOLID

Il termine **SOLID** viene utilizzato per indicare i cinque principi di progettazione orientata agli oggetti. I principi SOLID sono considerati come linee guida per lo sviluppo di un software estendibile e flessibile. La parola SOLID è un acronimo che serve a ricordare tali principi, e fu coniata da Michael Feathers

- **S** – Single responsibility principle
- **O** – Open-Closed principle
- **L** – Liskov substitution principle
- **I** – Interface segregation principle
- **D** – Dependency Inversion Principle

Il **Single Responsibility principle** afferma che ogni classe dovrebbe avere una ed una sola responsabilità, interamente incapsulata al suo interno. Questo principio viene rispettato in quanto ogni singola classe viene implementata per far sì che abbia una e singola responsabilità. Infatti la classe Message ha lo scopo di andare a definire le informazioni all'interno di ogni sensore; la classe Conn crea la connessione al database e la classe Sensor ha lo scopo di andare a creare i sensori.

Il **Open-Closed principle** afferma che un oggetto o un'entità dovrebbe essere aperta alle estensioni, ma chiusa alle modifiche. Ciò vuole dire che in ogni classe si deve garantire la sua estensione senza cambiamento di codice. Questo principio viene rispettato in quanto si fa esplicito uso delle interfacce e l'uso dello **Strategy pattern**.

**Liskov Substitution principle** dice che se  $q(x)$  è una proprietà che si può dimostrare essere valida per oggetti  $x$  di tipo  $T$ , allora  $q(y)$  deve essere valida per oggetti  $y$  di tipo  $S$  dove  $S$  è un sottotipo di  $T$ . Bisogna essere sicuri che le classi derivate debbano essere capaci di rimpiazzare le classi senza cambiamenti del codice. Tale principio non viene rispettato.

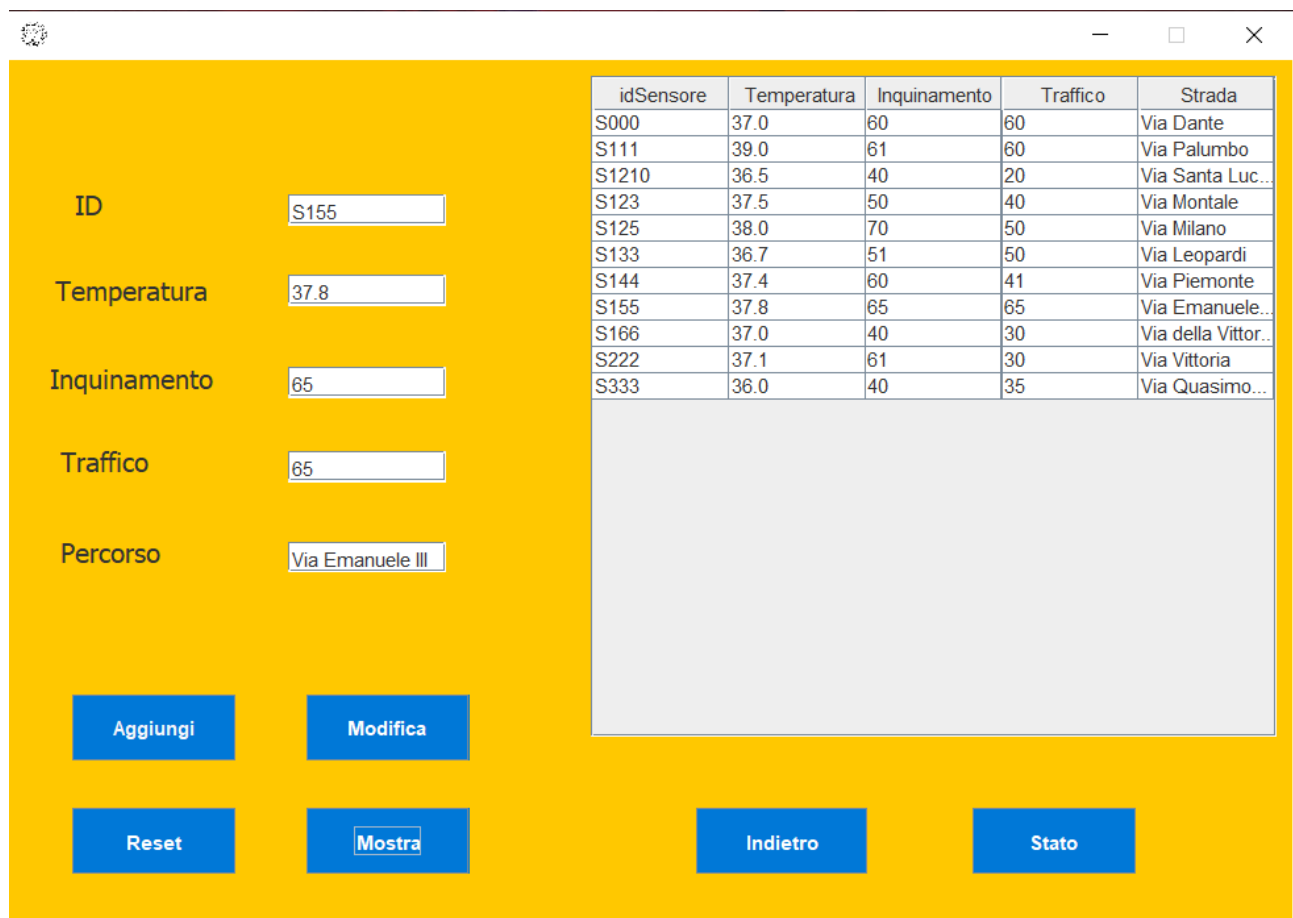
**Interface segregation principle** dice che una classe client non dovrebbe dipendere da metodi che non usa, e che pertanto è preferibile che le interfacce siano molte, specifiche e piccole piuttosto che poche, generali e grandi. Questo principio non viene rispettato in quanto sono state create interfacce singole ed uniche, ad esempio le interfacce Observer, Strategy e Subject.

**Dependency Inversion Principle** afferma che i moduli di alto livello non devono dipendere da quelli di basso livello. Entrambi devono dipendere da astrazioni; le astrazioni non devono dipendere dai dettagli, sono i dettagli che dipendono dalle astrazioni. Tale principio viene rispettato in quanto le classi Context e Subject, essendo classi di alto livello, non dipendono dalle classi di basso livello, infatti Context può richiedere il servizio di Strategy ma non usa un servizio di Strategy per raggiungere un terzo oggetto C.

# Dettagli implementativi

## Inserimento di un sensore

Il form permette di inserire una tupla all'interno del mio database MySQL, aggiornando automaticamente quest'ultimo.



idSensore	Temperatura	Inquinamento	Traffico	Strada
S000	37.0	60	60	Via Dante
S111	39.0	61	60	Via Palumbo
S1210	36.5	40	20	Via Santa Luc...
S123	37.5	50	40	Via Montale
S125	38.0	70	50	Via Milano
S133	36.7	51	50	Via Leopardi
S144	37.4	60	41	Via Piemonte
S155	37.8	65	65	Via Emanuele...
S166	37.0	40	30	Via della Vittor...
S222	37.1	61	30	Via Vittoria
S333	36.0	40	35	Via Quasimo...

Form fields:

- ID: S155
- Temperatura: 37.8
- Inquinamento: 65
- Traffico: 65
- Percorso: Via Emanuele III

Buttons: Aggiungi, Modifica, Reset, Mostra, Indietro, Stato

Form di inserimento.

Di seguito, viene riportato una parte del codice che permette questa operazione.

```
public void actionPerformed(ActionEvent e) {

    String idString = textId.getText();
    double temperature = Double.parseDouble(textTemp.getText());
    int pollution = Integer.parseInt(textPoll.getText());
    int traffico = Integer.parseInt(textTraff.getText());
    String stradaString = textPerc.getText();

    try {

        Conn conn = Conn.getInstance();

        PreparedStatement st;
        String query = "INSERT INTO my_db.sensore values(?,?,?,?,?)";
        st = conn.getConnection().prepareStatement(query);
        st.setString(1, idString);
        st.setDouble(2, temperature);
        st.setInt(3, pollution);
        st.setInt(4, traffico);
        st.setString(5, stradaString);
        st.executeUpdate();
        sensors = new Sensors();
        datas.attach(sensors);
        datas.notifyObservers(new Message(idString, temperature, pollution, traffico, stradaString));
        JOptionPane.showMessageDialog(null, "Operazione eseguita");
    }
}
```

## Modifica delle tuple e spostamento del traffico

In una maniera molto simile all'aggiunta dei sensori, viene riportato di seguito il codice che permette l'operazione di modifica e spostamento del traffico.

```
public void actionPerformed(ActionEvent e) {

    String idString = textId.getText();
    double temperature = Double.parseDouble(textTemp.getText());
    int pollution = Integer.parseInt(textPoll.getText());
    int traffico = Integer.parseInt(textTraff.getText());
    String stradaString = textPerc.getText();

    try {

        Conn conn = Conn.getInstance();
        int row = table_1.getSelectedRow();
        String value = (String) (table_1.getModel().getValueAt(row, 0));
        String query = ("UPDATE my_db.sensore SET Temperatura = ?, Inquinamento = ?, Traffico = ?, Strada = ? WHERE idSensore = '" + value + "'");
        PreparedStatement st;
        st = conn.getConnection().prepareStatement(query);
        st.setDouble(1, temperature);
        st.setInt(2, pollution);
        st.setInt(3, traffico);
        st.setString(4, stradaString);
        st.executeUpdate();
        DefaultTableModel model = (DefaultTableModel)table_1.getModel();
        model.setRowCount(0);
        sensors = new Sensors();
        sensors.update(new Message(idString, temperature, pollution, traffico, stradaString));
        JOptionPane.showMessageDialog(null, "Operazione eseguita");


    } catch (Exception e2) {
        e2.printStackTrace();
    }
}

});
```

## Stato dei sensori

Lo stato dei sensori viene controllato tramite l'utilizzo del bottone "Stato". Bisogna selezionare prima una tupla all'interno della JTable e poi utilizzare il Button "Stato". Viene riportato il codice qui.

idSensore	Temperatura	Inquinamento	Traffico	Strada
S000	37.0	60	60	Via Dante
S111	39.0	61	60	Via Palumbo
S1210	36.5	40	20	Via Santa Luc...
S123	37.5	50	40	Via Montale
S125	38.0	70	50	Via Milano
S133	36.7	51	50	Via Leopardi
S144	37.4	60	41	Via Piemonte
S155	37.8	65	65	Via Emanuele...
S166	37.0	40	30	Via della Vittor...
S222	37.1	61	30	Via Vittoria
S333	36.0	40	35	Via Quasimo...

**Codice**  
 **Rosso**  
**OK**

```
public void actionPerformed(ActionEvent e) {  
  
    //Imposto le soglie per i tre parametri.  
    final double tempSoglia = 37;  
    final int traffSoglia = 40;  
    final int pollSoglia = 60;  
  
    //Estraggo i tre valori nella tabella e li inserisco in tre variabili.  
    double temperature = Double.parseDouble(textTemp.getText());  
    int pollution = Integer.parseInt(textPoll.getText());  
    int traffico = Integer.parseInt(textTraff.getText());  
    if(temperature < tempSoglia && pollution < pollSoglia && traffico < traffSoglia)  
    {  
        Context context = new Context(new CodiceVerde());  
        JOptionPane.showMessageDialog(null, context.executeStrategy());  
        System.out.println(context.executeStrategy());  
    }  
    else if(temperature > tempSoglia && pollution > pollSoglia && traffico > traffSoglia)  
    {  
        Context context = new Context(new CodiceRosso());  
        JOptionPane.showMessageDialog(null, context.executeStrategy(), "Codice", JOptionPane.WARNING_MESSAGE);  
        System.out.println(context.executeStrategy());  
    }  
    else if(temperature > tempSoglia && pollution > pollSoglia )  
    {  
        Context context = new Context(new CodiceGiallo());  
        JOptionPane.showMessageDialog(null, context.executeStrategy(), "Codice", JOptionPane.DEFAULT_OPTION);  
        System.out.println(context.executeStrategy());  
    }  
    else  
    {  
        Context context = new Context(new CodiceVerde());  
        JOptionPane.showMessageDialog(null, context.executeStrategy(), "Codice", JOptionPane.DEFAULT_OPTION);  
        System.out.println(context.executeStrategy());  
    }  
}
```

## Plot del grafico

Per la creazione del grafico statistico viene utilizzata la libreria JFreeChart. JFreeChart è una libreria java Open Source per la generazione dinamica di grafici a partire da una fonte di dati. Può essere utilizzata sia in applicazioni Java standalone che Web.

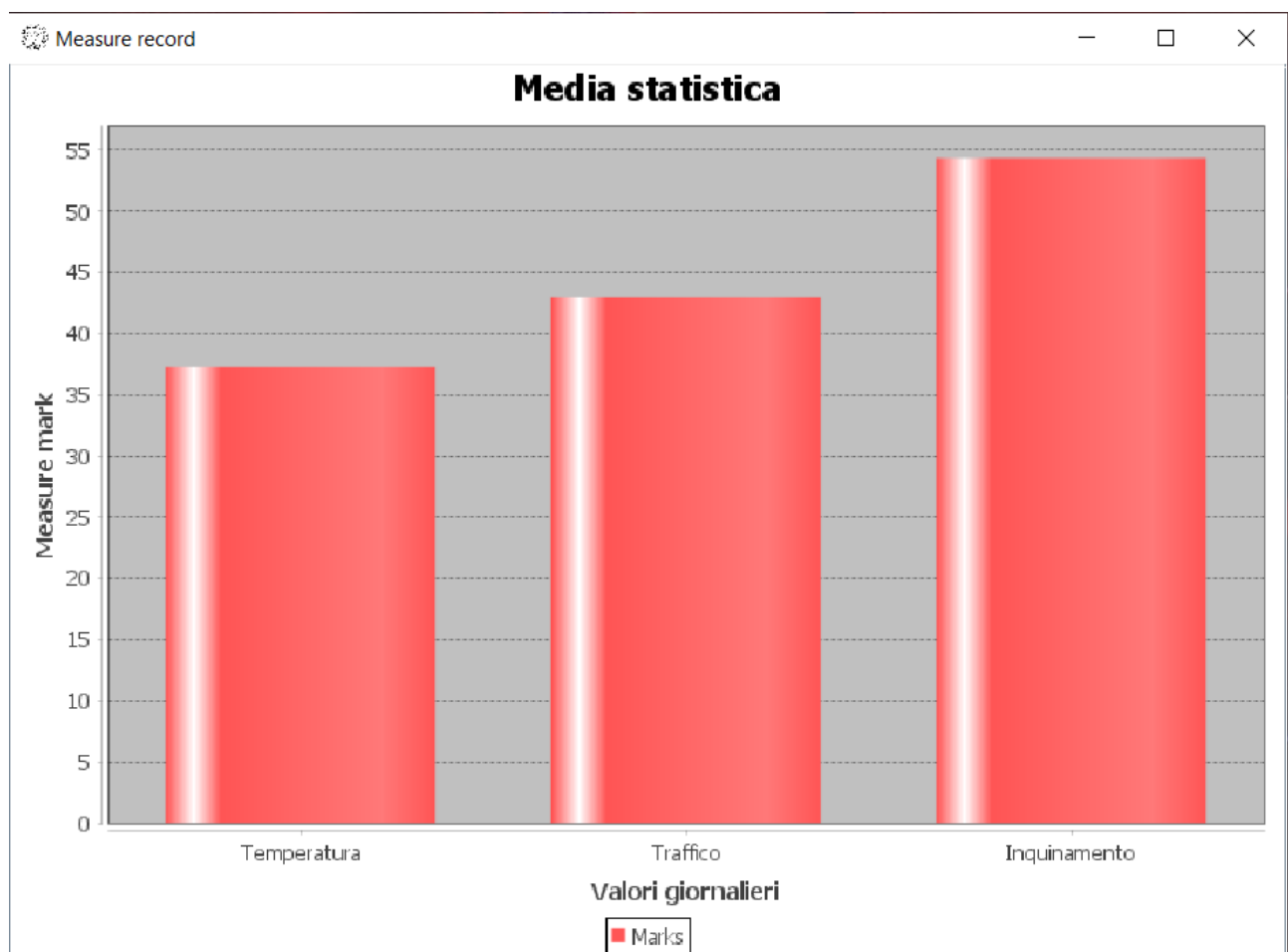


Grafico media statistica.

Di seguito viene riportato la porzione di codice che implementa tale funzione.

```
public void actionPerformed(ActionEvent e) {  
    try {  
        DefaultCategoryDataset data = new DefaultCategoryDataset(); //Tramite la libreria .net posso estrarre i dati dal mio database MySQL per poi inserirli nel grafico.  
        Conn conn = Conn.getInstance();  
        //Utilizzo una query con una funzione aggregata AVG che mi restituisce la media dei tre parametri dando anche un'alias alla colonna risultato.  
        String query = "SELECT avg(Temperatura) as Temperatura, avg(Traffico) as Traffico, avg(inquinamento) as Inquinamento FROM my_db.sensore;";  
        Statement statement = conn.getConnection().createStatement(); //Crea lo statement, che crea l'interfaccia dello statement di MySQL  
        ResultSet rs = statement.executeQuery(query); //Eseguo la query precedentemente dichiarata.  
  
        while(rs.next())  
        {  
            //Inserisco i risultati della query nella variabile data che successivamente li plotta sul grafico  
            data.setValue(rs.getDouble("Temperatura"), "Marks", "Temperatura");  
            data.setValue(rs.getInt("Traffico"), "Marks", "Traffico");  
            data.setValue(rs.getDouble("Inquinamento"), "Marks", "Inquinamento");  
        }  
  
        //Dichiarazione di variabili per personalizzare il grafico.  
        JFreeChart jChart = ChartFactory.createBarChart("Media statistica", "Valori giornalieri", "Measure mark", data, PlotOrientation.VERTICAL, true, true, false);  
        CategoryPlot plot = jChart.getCategoryPlot();  
  
        plot.setRangeGridlinePaint(Color.black);  
  
        ChartFrame chartFrame = new ChartFrame("Measure record", jChart, true);  
        chartFrame.setIconImage(Toolkit.getDefaultToolkit().getImage("C:\\Users\\Asus\\Desktop\\logo.png"));  
  
        chartFrame.setVisible(true);  
  
        ChartPanel chartPanel = new ChartPanel(jChart);  
        chartPanel.setPreferredSize(new java.awt.Dimension(560,400));  
    }  
}
```

## Fonti e note

Per la stesura del codice sono state utilizzate

- Slide relative alle lezioni del prof. Angelo Ciaramella e del prof. Raffaele Montella
- Manuale di Java 9 di Claudio de Sio Cesari – Il Linguaggio Objected Oriented Java

Software utilizzati

- Objectaid UML Explorer, tool di Eclipse, per generare diagrammi UML
- WindowsBuilder, altro tool di Eclipse, che permette di generare e personalizzare con facilità i frame.
- Eclipse IDE 2020-12
- Database MySQL 8.0