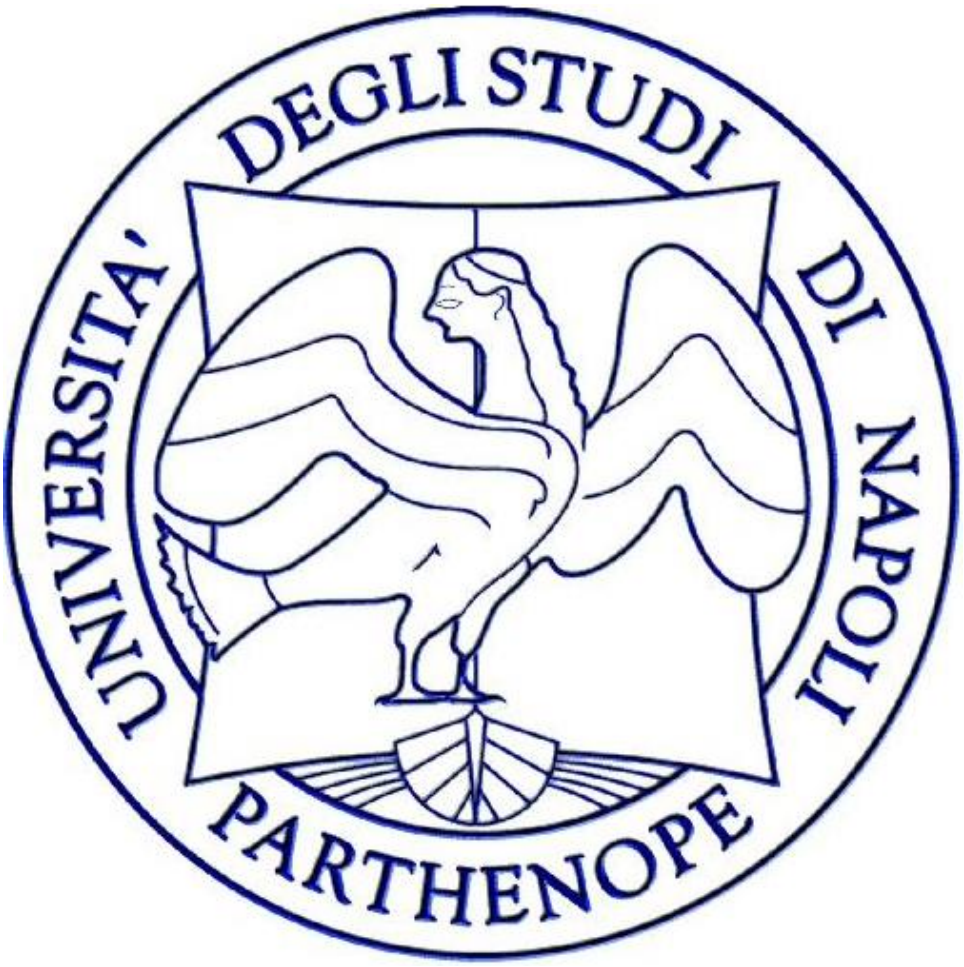
Università degli Studi di Napoli “Parthenope”
Corso di laurea in Informatica



Progetto Programmazione III e Laboratorio di Programmazione III

Candidato
Franco Riformato
0124001968

Requisiti del progetto

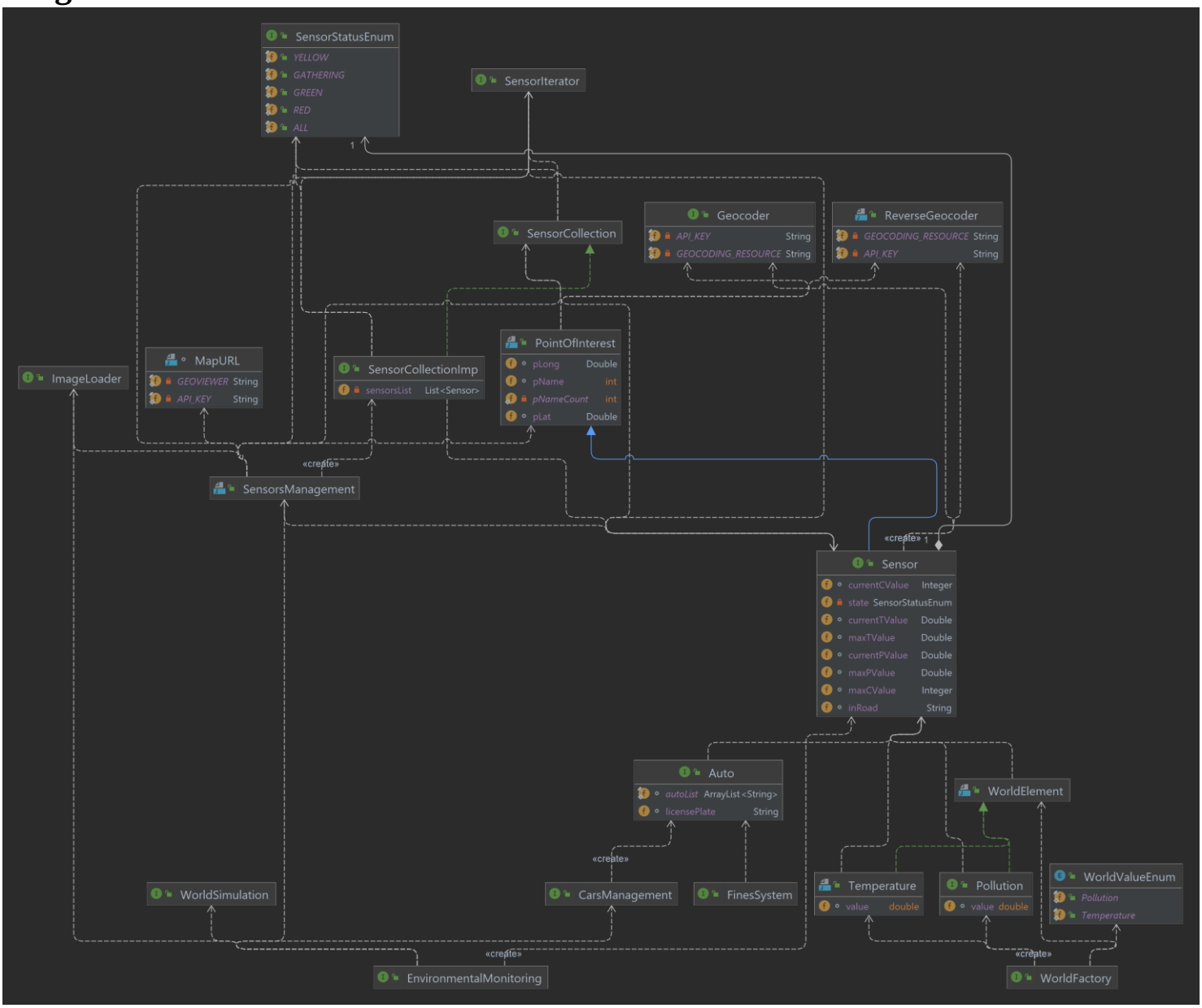
Si vuole sviluppare un sistema per la gestione di una Smart City.
Una Smart City comprende un insieme di strategie di pianificazione urbanistica al fine di migliorare la qualità della vita e soddisfare le esigenze di cittadini, imprese e istituzioni.
Si suppone di gestire un sistema di monitoraggio ambientale centralizzato.
In una città, per ogni strada, sono situate diverse centraline contenenti dei sensori di monitoraggio che permettono di registrare i livelli di tre parametri: inquinamento dell'aria, la temperatura e il numero di autoveicoli che transitano. Per ogni parametro l'amministratore del sistema fissa una soglia di guardia.
Il sistema di monitoraggio si può trovare in questi tre stati:
• codice verde - Se tutti e tre i parametri sono sotto soglia
• codice giallo - Se i primi due parametri sono sopra soglia
• codice rosso - Se tutti i parametri sono sopra soglia
Nel caso si verifichi il codice rosso si può applicare una delle seguenti strategie:
• consentire il traffico solo a targhe alterne. Un dispositivo controlla automaticamente le vetture e procede con l'invio di una segnalazione alla polizia locale in caso di infrazione.
• il flusso del traffico viene inviato su un altro percorso. Il sistema permette, inoltre, di inserire nuovi sensori alla rete e di fare il grafico dei parametri per un periodo fissato.

Librerie utilizzate

Al fine di implementare correttamente i requisiti richiesti, sono state utilizzate le seguenti librerie:

- org.jfree.chart per il disegno e la visualizzazione di grafici;
- java.sql.* per l'implementazione del database tramite mySQL;
- java.util.Random per l'utilizzo di valori pseudocasuali;
- java.util.Timer per svolgere delle azioni ogni tot di tempo;
- javax.swing.* per l'interfaccia grafica;
- java.awt.* per la gestione delle finestre;
- java.net per la gestione di URL (al fine di utilizzare HERE).

Diagramma UML



Il diagramma UML permette di mostrare le classi e le loro relazioni.
In particolare, notiamo:

- PointOfInterest, che rappresenta un astratto punto di interesse sulla mappa, la quale permette di definire punti particolari dei quali ci interessano le coordinate geografiche;
- Sensor che rappresenta i sensori per il monitoraggio ambientale, sfruttando PointOfInterest;
- Pollution e Temperature che rappresentano la simulazione dei valori corrispondenti nell'ambiente (utilizzate a fini dimostrativi, in quanto questi valori in un caso reale sarebbero generati direttamente dal mondo esterno e letti dai sensori);
- Auto che permette di rappresentare i veicoli e le loro targhe;
- Geocoder e ReverseGeocoder utilizzate rispettivamente per la geocodifica diretta e inversa.

Design Patterns

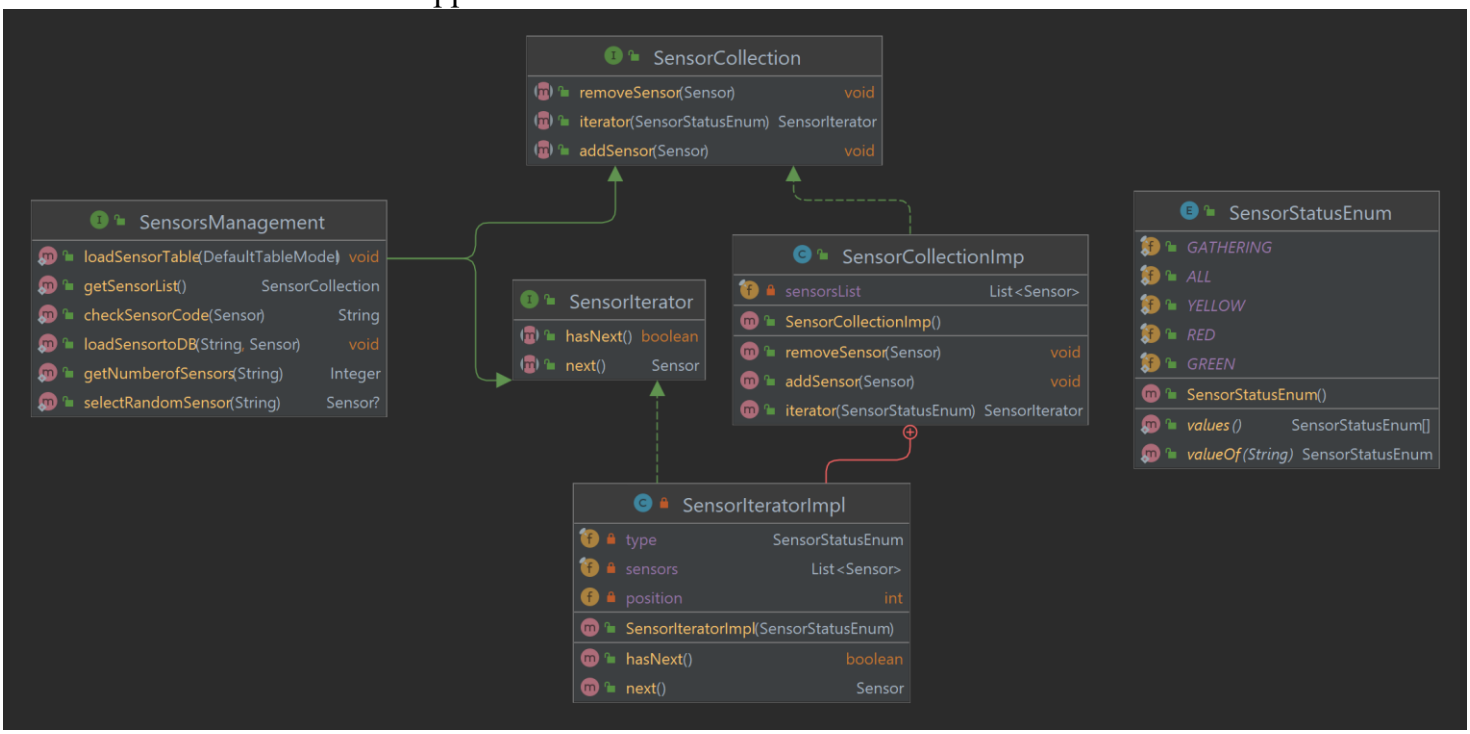
Al fine di implementare al meglio l'applicazione, sono stati utilizzati i design pattern Iterator e Factory Pattern.

Iterator

Iterator è un behavioral pattern usato per fornire un modo standard di attraversare un gruppo di oggetti, senza esporre la rappresentazione interna, all'applicazione.
Il pattern viene implementato attraverso le classi:

- SensorStatusEnum.java, la quale definisce il contratto per la collezione da iterare, definendo gli stati in cui un sensore può trovarsi;
- Sensor.java, dove viene definita la variabile *private SensorStatusEnum state*;
- SensorCollection.java, che dichiara i metodi per l'aggiunta e l'eliminazione di Sensori e un metodo che ritorna l'iteratore.
- SensorIterator.java, la quale dichiara metodi che permettono di sapere se esiste un prossimo sensore e di ottenere il prossimo sensore;
- SensorCollectionImpl.java, che implementa SensorCollection, in modo da definirne i metodi e nascondere l'implementazione interna;

Il suo utilizzo all'interno dell'applicazione si ha nel caricamento della tabella dei Sensori.



Iterator: Applicazione nel codice

SensorStatusEnum.java

```
public enum SensorStatusEnum extends SensorCollectionImp {
    GREEN, YELLOW, RED, GATHERING, ALL
}
```

SensorStatusEnum.java

```
private SensorStatusEnum state;
```

SensorCollection.java

```
public interface SensorCollection {
    SensorIterator iterator(SensorStatusEnum state);

    void addSensor(Sensor s);

    void removeSensor(Sensor s);

}
```

SensorIterator.java

```
public interface SensorIterator {
    boolean hasNext();
    Sensor next();
}
```

SensorCollectionImpl.java

```
public class SensorCollectionImp implements SensorCollection{
    private final List<Sensor> sensorsList;

    public SensorCollectionImp() {
        sensorsList = new ArrayList<>();
    }

    public void addSensor(Sensor s) {
        this.sensorsList.add(s);
    }

    public void removeSensor(Sensor s) {
        this.sensorsList.remove(s);
    }

    @Override
    public SensorIterator iterator(SensorStatusEnum type) {
        return new SensorIteratorImpl(type);
    }
```

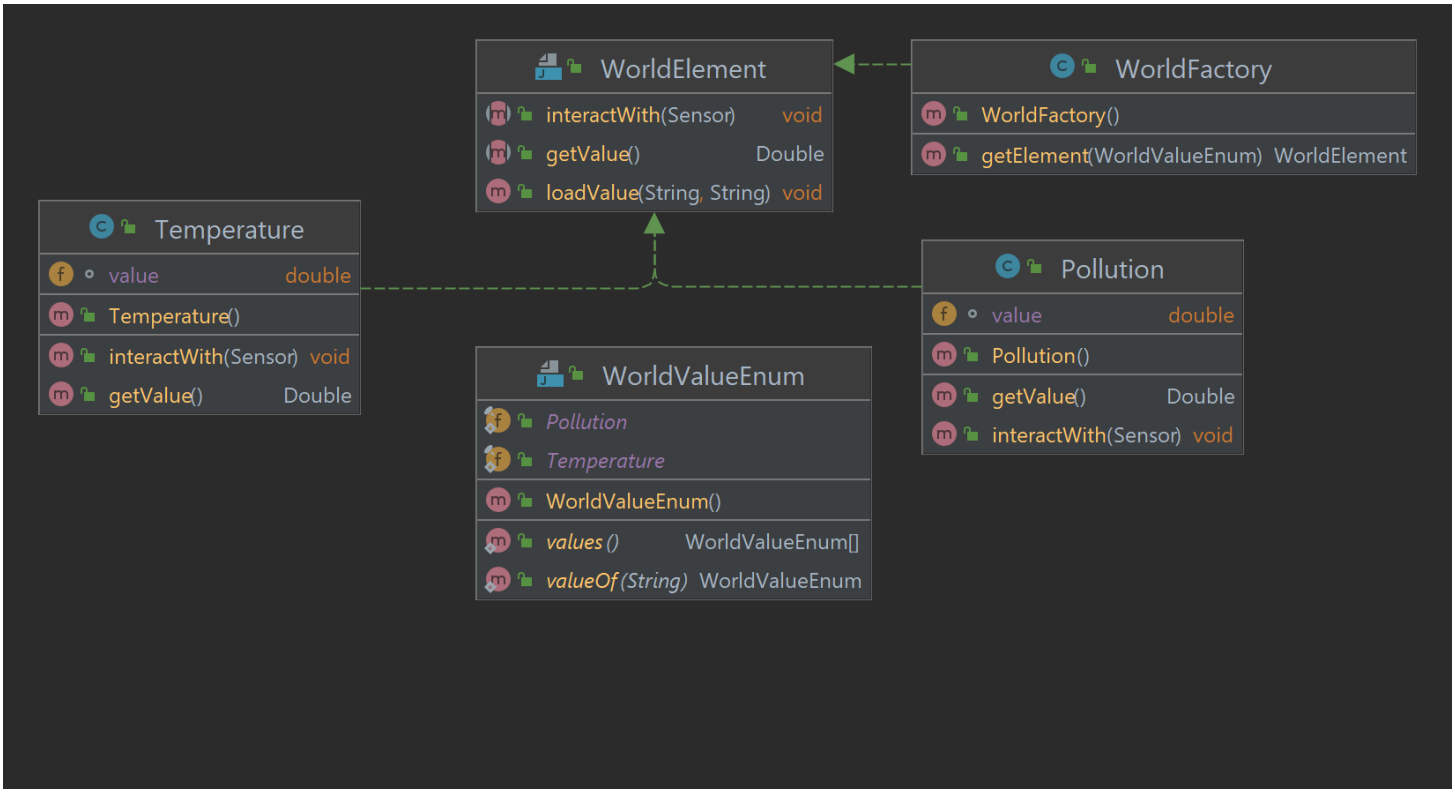
```
    private class SensorIteratorImpl implements SensorIterator {

        private final SensorStatusEnum type;
        private final List<Sensor> sensors;
        private int position;



        public SensorIteratorImpl(SensorStatusEnum state) {
            this.type = state;
            this.sensors = sensorsList;
        }

        @Override
        public boolean hasNext() {
            while (position < sensors.size()) {
                Sensor s = sensors.get(position);
                if (s.getSensorStatus().equals(type) ||
type.equals(SensorStatusEnum.ALL)) {
                    return true;
                } else
                    position++;
            }
            return false;
        }

        @Override
        public Sensor next() {
            Sensor s = sensors.get(position);
            position++;
            return s;
        }

    }
}
```

Factory Method

Questo pattern è utilizzato quando abbiamo una superclasse con multiple sottoclassi e bisogna ritornare una delle sottoclassi in base alla necessità.
Permette di spostare la responsabilità di istanziare la classe dal client alla classe factory.
Il pattern è stato implementato attraverso le classi:

- WorldElement.java, è un'interfaccia che dichiara il metodo per l'interazione con i sensori e il metodo per ottenere il valore dell'elemento, inoltre definisce il metodo di default per caricare un valore nel database;
- Temperature.java, esegue l'override del metodo di interazione con i sensori e del metodo per get per il valore della temperatura;
- Pollution.java, esegue lo stesso di Temperature.java ma per i valori dell'inquinamento;
- WorldValueEnum.java, è un'enumerazione che indica le entità che possono manifestarsi nella simulazione;
- WorldFactory.java, permette l'istanziare di un oggetto Pollution o Temperature a seconda della necessità.

In questo modo, potremo generare nuovi valori di temperatura o di inquinamento in modo casuale, in base alle istruzioni in WorldSimulation.java.
In particolare, viene generato un numero casuale e si ha il 50% di modificare la temperatura o l'inquinamento.



Factory Pattern: Applicazione nel codice

WorldElement.java

```
public interface WorldElement {

    void interactWith(Sensor s);

    default void loadValue(String DBString, String dbwhere) {

        //MYSQL DB CONNECTION
        try{
            Class.forName(DBString);
            Connection con= DriverManager.getConnection(
                    "jdbc:mysql://localhost:3306/EMDB","root","root");
            String query = " insert into " + dbwhere
                         + " values (?, ?)";
            PreparedStatement preparedStmt = con.prepareStatement(query);
            java.util.Date date=new java.util.Date();
            java.sql.Date sqlDate=new java.sql.Date(date.getTime());

            preparedStmt.setDate (1, sqlDate);
            preparedStmt.setDouble (2, this.getValue());

            preparedStmt.execute();

            con.close();

        }catch(Exception e){ System.out.println(e);}

    }
    Double getValue();
}
```

Temperature.java

```
public class Temperature implements WorldElement{

    double value = Math.random() * 100;

    @Override
    public void interactWith(Sensor s) {
```

```
        Double currentT = s.getCurrentTValue();

        if (currentT < this.value)
        {
            s.setCurrentTValue(currentT * (1 + 0.1) );
        }

        if (currentT > this.value)
        {
            s.setCurrentTValue(currentT * (1 - 0.1) );
        }
        else
        {
            s.setCurrentTValue(this.value);
        }
    }

    @Override
    public Double getValue() {
        return value;
    }
}
```

Pollution.java

```
public class Pollution implements WorldElement {
    double value = Math.random() * 100;
    public void interactWith(Sensor s) {

        Double currentP = s.getCurrentPValue();
        if (currentP < this.value)
        {
            s.setCurrentPValue(currentP * (1 + 0.1) );
        }

        if (currentP > this.value)
        {
            s.setCurrentPValue(currentP * (1 - 0.1) );
        }
        else
        {
            s.setCurrentPValue(this.value);
        }

    }

    @Override
    public Double getValue() {
```

```
        return value;
    }

}
```

WorldValueEnum.java

```
public enum WorldValueEnum {
    Pollution,
    Temperature
}
```

WorldFactory.java

```
public class WorldFactory {

    public WorldFactory() {
    }

    public WorldElement getElement (WorldValueEnum type) {

        WorldElement retval = null;


        switch (type) {
            case Pollution -> {
                return new Pollution();
            }
            case Temperature -> {
                return new Temperature();
            }
        }

        return retval;
    }
}
```

In WorldSimulation.java

```
WorldFactory factory = new WorldFactory();

Random r = new Random();
float chance = r.nextFloat();

if (chance <= 0.50f)
{
    WorldElement temperature = factory.getElement(WorldValueEnum.Temperature);
    temperature.loadValue(DB, "gentemperature (TempDate, Value)");
    temperature.interactWith(sR);
    System.out.println("Generato: " + temperature);
}
else {
    WorldElement pollution = factory.getElement(WorldValueEnum.Pollution);
    pollution.loadValue(DB, "genpollution (PollDate, Value)");
    pollution.interactWith(sR);
```

```
    System.out.println("Generato: " + pollution);
}
```

Utilizzo del Database MySQL

Per l'utilizzo del database, è stato scelto MySQL implementato tramite la dipendenza di Gradle 'mysql:mysql-connector-java:8.0.25'.
Nel DB, sono presenti le tabelle:

- CARS, cioè l'elenco di auto che possono interagire con i sensori, identificate dalla loro targa. Al fine di simulare le auto, questa tabella viene caricata con un massimo numero di auto (maxCars) indicato in EnvironmentalMonitoring.java;
- FINES, indica le multe inviate alle varie auto che non rispettano le politiche imposte sulla strada. Possiamo trovare al suo interno le targhe e l'ammontare di multa ricevuto.
- GENPOLLUTION, indica l'inquinamento generato a scopo simulativo e la data di generazione;
- GENTEMPERATURE, lo stesso ma per la temperatura;
- HYSTORICALPOLLUTION, mantiene dati per generare il grafico giornaliero relativo all'inquinamento;
- HYSTORICALTEMPERATURE, stesso utilizzo ma per la temperatura;
- ROADS, contiene le varie strade (indirizzo) e la politica associata (ODD, EVEN, NONE o BLOCKED);
- SENSOR, contiene l'ID del sensore, le sue coordinate geografiche (latitudine e longitudine), il massimo valore consentito per l'inquinamento, per la temperatura e per il numero di auto, lo status (GREEN, YELLOW, RED o GATHERING) attuale del sensore e il nome della strada in cui si trova.