

**IZMIR UNIVERSITY OF ECONOMICS**

**SE 311 - SOFTWARE ARCHITECTURE**

**Project Report**

**Project Topic:** Project #3 - Smart City Application

**Lecturer**: Asst. Prof. Ufuk Çelikkan

**Group Members:** Fercan Şen 20150602038

Naz Tekinalp 20160602033

Mert Kesimli 20160602017

**Submission Date:** 22 May 2020

**Table of Contents**

1. **Introduction 2**
2. **Pattern Selection 2**

**2.1 Factory and Singleton Pattern 3**

**2.2 Composite Pattern 4**

**2.3 Observer Pattern 5**

**2.4 Visitor Pattern 6**

1. **Participant Mapping 7**
2. **UML Diagram of the Solution 8**
3. **Classes 9**

**5.1 Abstract Classes 9**

**5.2 Concrete Classes 9**

**5.3 Interfaces 11**

1. **Screen Dumps 11**
2. **Output 21**

**1. Introduction**

With this project, we are designing and developing a smart city application. This application revolves around the usage of sensors. Sensors are mainly installed on apartments and poles. There are four types of sensors: pollution, temperature, congestion, and noise. As a citizen, you subscribe to these sensors. The main purpose of these sensors is that they send a message when a certain event happens. The city has one Data Monitoring Division that periodically checks the sensors for the possibility of malfunction. Engineers that work at Data Monitoring Division can send a request to the sensors to reset themselves. For this project, we used several software design patterns.

This report consists of 7 sections. A detailed explanation of the implementation and design process mentioned in the following sections.

**2. Pattern Selection**

We learned 11 different design patterns during SE 311 course this semester. We used 5 design patterns for the solution of the given scenario.

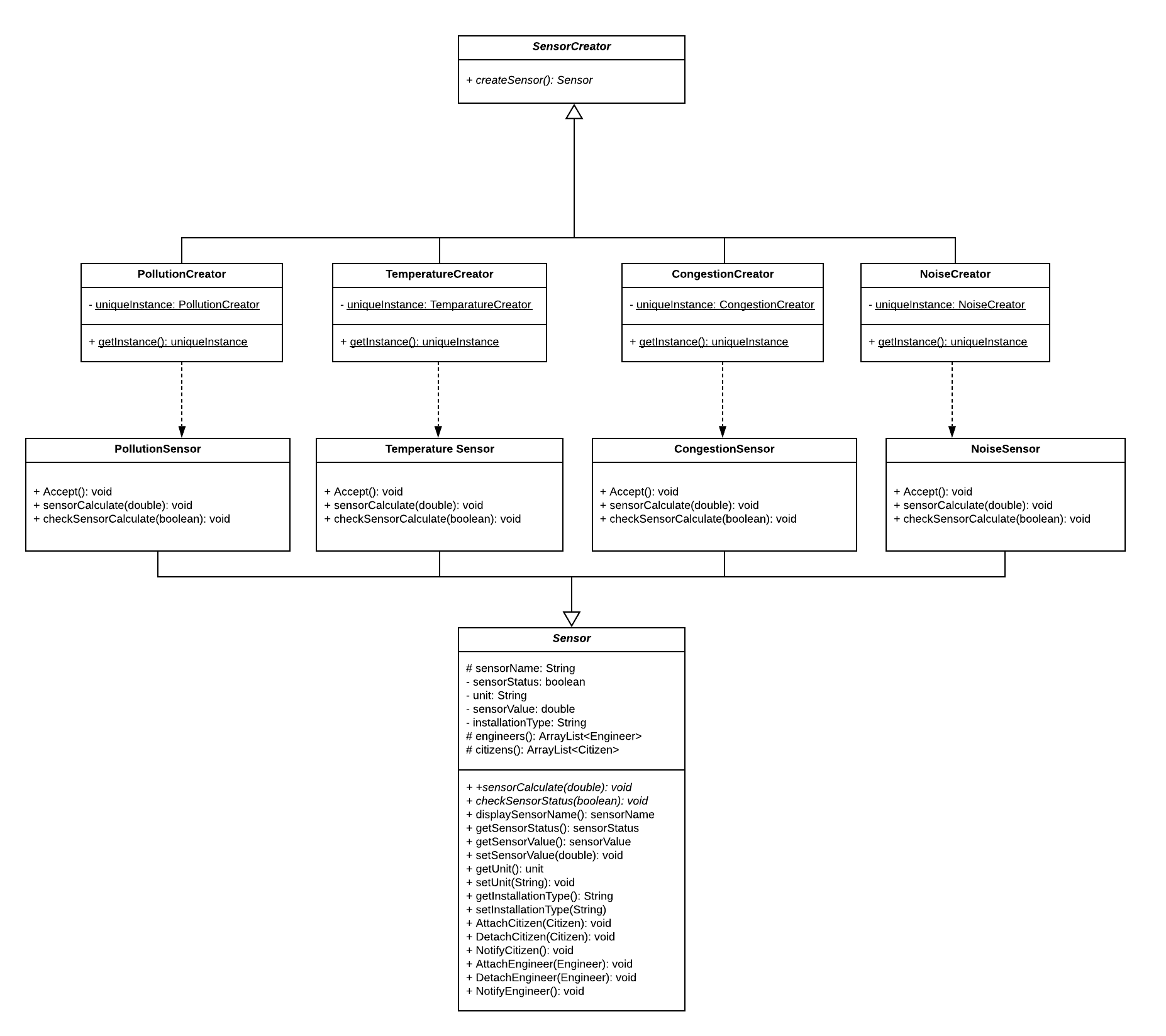
We appreciated the design and implementation procedure. It was compelling but informative. We combined 5 patterns and attempted to implement all required features.

We chose the following 5 patterns in our project:

1. Factory Pattern
2. Singleton Pattern
3. Composite Pattern
4. Observer Pattern
5. Visitor Pattern

The reason for choosing each pattern is explained in detail in the following pages.

**2.1 Factory and Singleton Pattern**

****

**Figure 1: Factory and Singleton Pattern**

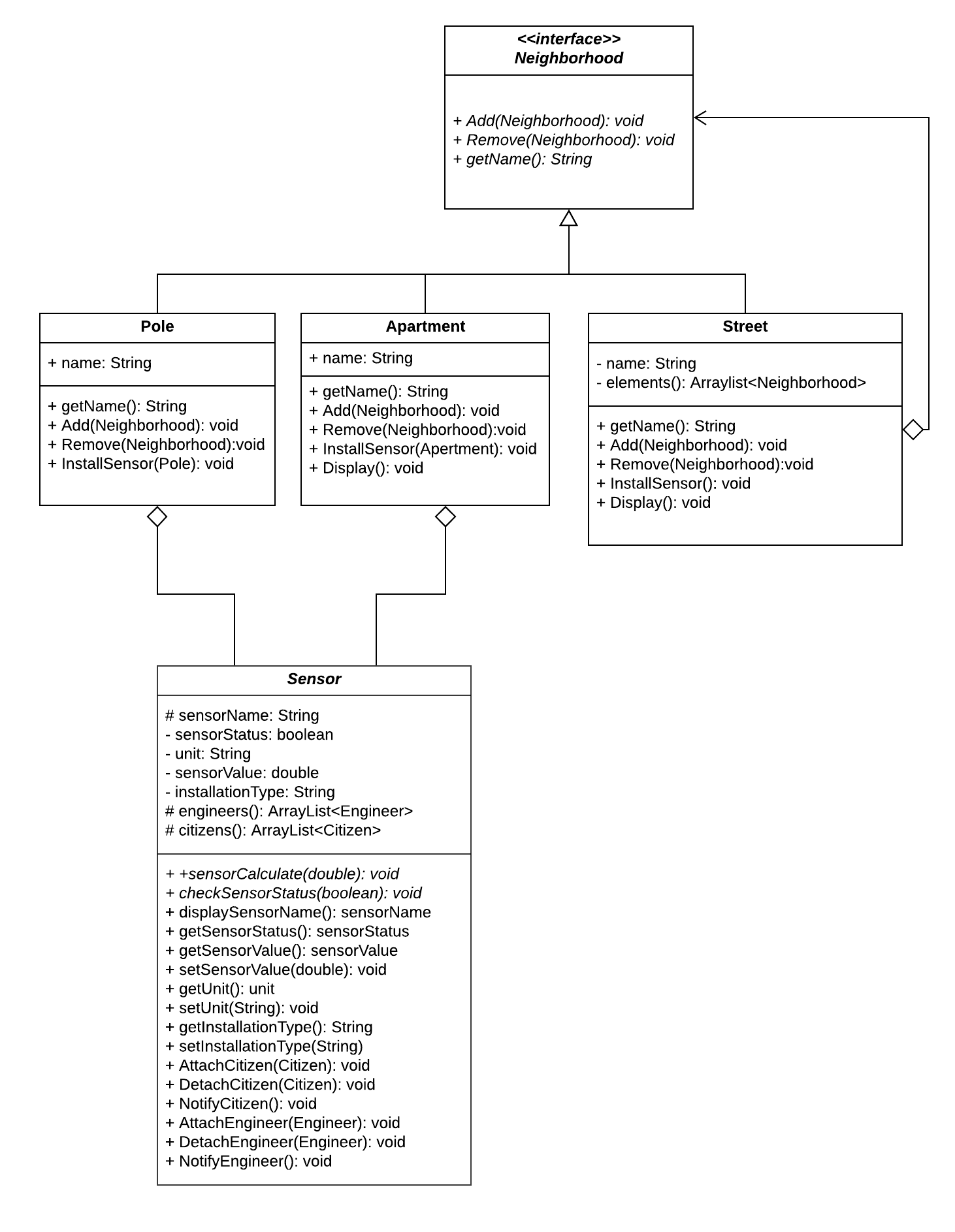
(For higher resolution please [click here](https://bit.ly/2XnjWjY).)

Factory pattern is a type of design pattern that comes under the creational pattern as this pattern provides to create an object.

We derived the idea of using Factory Pattern from the following sentence in the project description: “There are 4 sensors: pollution, temperature, congestion, noise.” Our factory “produces” products, which are our sensors. And there are four concrete products, which are our sensor types in this case. Since we may have one for each factory, we used it for each concrete creator.

So, we have one unique instance of each concrete creator, we figured the Singleton Pattern would go very well with the Factory Pattern. So we used them together in harmony.

**2.2 Composite Pattern**

****

**Figure 2: Composite Pattern**

(For higher resolution please [click here](https://bit.ly/3bUvmRD).)

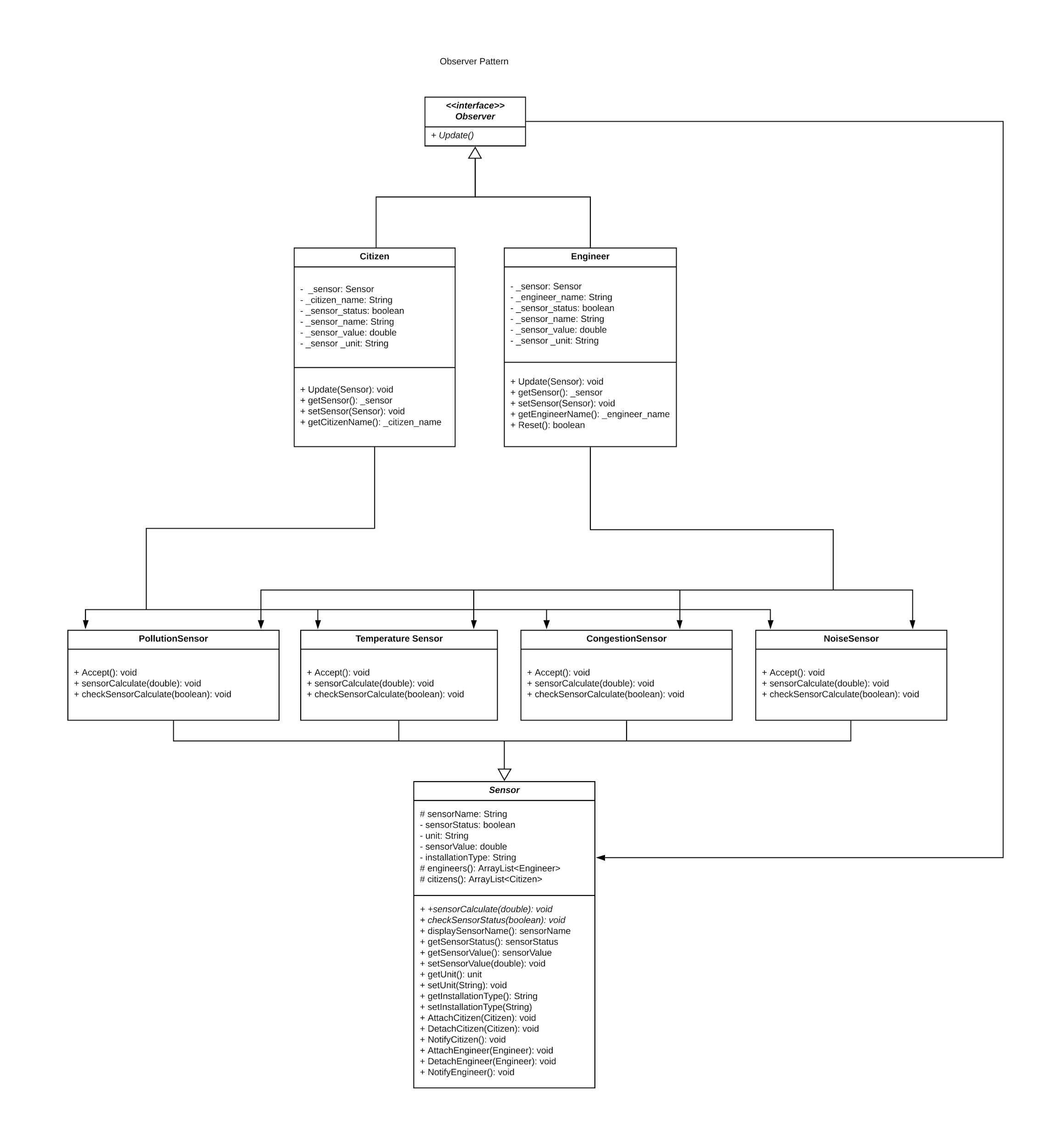
The intent of a composite is to “compose” objects into tree-like structures to represent part-whole hierarchies.

We derived the idea of using Composite Pattern from the following sentence in the project description: “A neighborhood contains streets” and “Streets contain apartments and poles”.

Streets are part of neighborhoods and neighborhoods together form the city.

There’s a hierarchy in the city. We may think it as a tree structure. We used Composite Pattern to apply this hierarchy into our project.

**2.3 Observer Pattern**

****

**Figure 3: Observer Pattern**

(For higher resolution please [click here](https://bit.ly/2A1kFPL).)

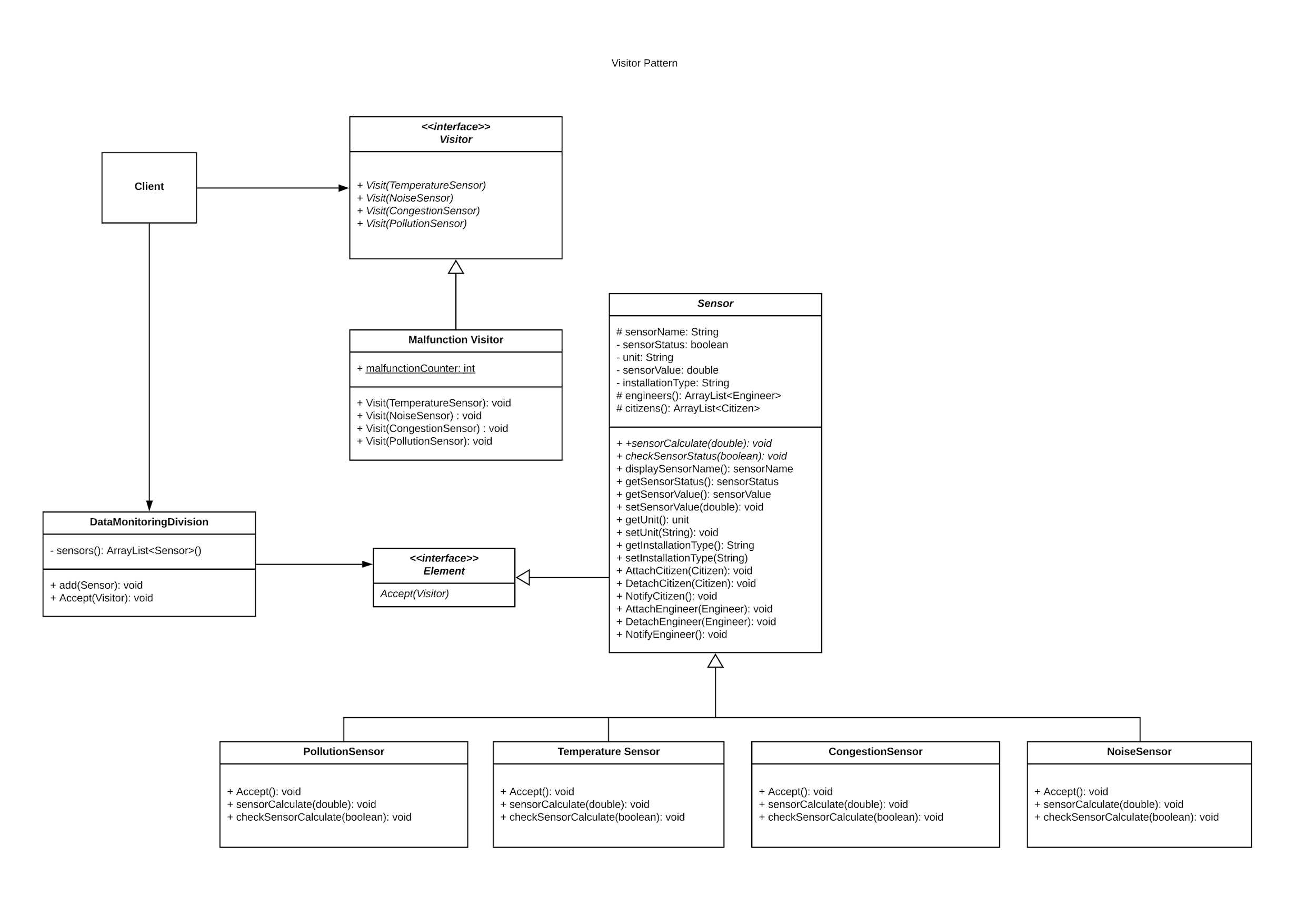
The observer pattern is used when there is a one-to-many relationship between objects such as if one object is modified, its dependent objects are to be notified automatically.

We derived the idea of using Observer Pattern from the following sentence in the project description: “The engineers also send a request to the sensor to reset themselves” and

“You get notified when the temperature falls below 0 degrees, pollution AQI value is above 100, the noise level is above 85 dB and car speed is below 10 km/hr “.

Sensor notification and engineers & citizens subscription would be implemented by Observer pattern. When temperature falls, a notification is sent to the citizens. When there’s a malfunction, a notification is sent to the engineers. For this reason, we used Observer pattern.

**2.4 Visitor Pattern**

****

**Figure 4: Visitor Pattern**

(For higher resolution please [click here](https://bit.ly/2LPZgvO).)

The visitor pattern is used when similar operations have to be performed on objects of different types grouped in a structure.

We derived the idea of using the Visitor Pattern from the following sentence in the project description: “The city has one city-wide Data Monitoring Division that periodically queries the status of all the sensors by sending a status query to them and determines the number of malfunctioning sensors”.

So, in other words, Data Monitoring Division “visits” these sensors and checks for any malfunction. If there is a malfunctioning sensor, the total number of malfunctioning sensors increases.

**3. Participant Mapping**

**3.1 OBSERVER PATTERN**

Observer pattern has 4 participants:

**1. Subject:** Sensor

**2. Concrete Subject:** TemperatureSensor, NoiseSensor, CongestionSensor, PollutionSensor

**3. Observer:** Observer

**4. Concrete Observer**: Citizen, Engineer

**3.2 VISITOR PATTERN**

Visitor pattern has 5 participants:

**1. Visitor:** Visitor

**2. Concrete Visitor:** Malfunction

**3. Element:** Sensor

**4. Concrete Element:** TemperatureSensor, NoiseSensor, CongestionSensor, PollutionSensor

**5. Object Structure:** DataMonitoringDivision

**3.3 FACTORY PATTERN**

Factory pattern has 4 participants:

**1. Product:** Sensor

**2. Concrete Product:** TemperatureSensor, NoiseSensor, CongestionSensor, PollutionSensor

**3. Creator:** SensorCreator

**4**. **Concrete Creator:** TemperatureCreator, NoiseCreator, CongestionCreator, NoiseCreator

**3.4 SINGLETON PATTERN**

Singleton pattern has 1 participant:

**1. Singleton:** SensorCreator

**3.5 COMPOSITE PATTERN**

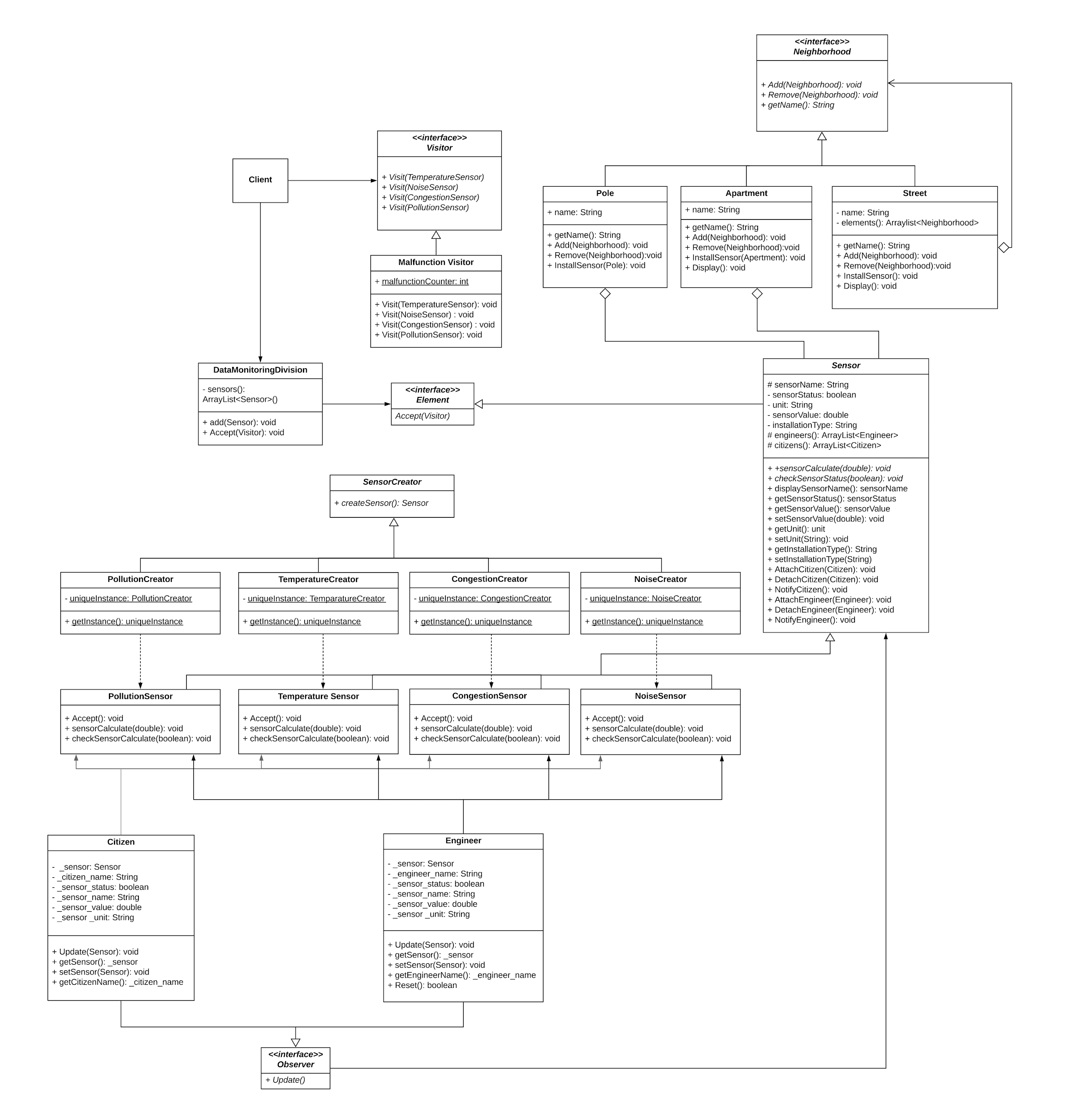
The composite pattern has 4 participants:

**1. Component:** Neighborhood

**2. Composite:** Street

**3. Leaf:** Apartment, Pole

**4. Client:** Client

**4. UML Diagram of the Project**

**Figure 5: UML Class Diagram**

(For higher resolution please [click here](https://bit.ly/3cSPnt9).)

In addition, you can find the PDF versions of all UML’s in the submitted zip.

**5. Classes**

We have only one java class which is “Sensor.java” in this project. All our patterns(including our main method) are contained within Sensor.java.

We used 15 concrete classes, 2 abstract classes and 4 interfaces in our solution for the given scenario.

**5.1 ABSTRACT CLASSES**

**SensorCreator:** Abstract base class of Concrete Creators (PollutionCreator, TemperatureCreator, NoiseCreator, CongestionCreator). It contains abstract createSensor() method.

**Sensor:** Has main method. Sensor’s properties (name, status, unit, value, installation type) are in this class. These are private variables so we used set & get methods. We combined all patterns in this class. Sensor is the “Product” of Factory pattern, “Subject” of the Observer pattern, “Element” of the Visitor pattern. Composite leafs (Apartment and Pole) has-a Sensor.

**5.2 CONCRETE CLASSES**

**PollutionCreator:** Concrete Creator of Factory Pattern. Extends from SensorCreator. Creates an unique concrete product, which is PollutionSensor. Has Singleton in it.

**TemperatureCreator:**  Concrete Creator of Factory Pattern. Extends from SensorCreator. Creates a concrete product, which is TemperatureSensor. Has Singleton in it.

**CongestionCreator:**  Concrete Creator of Factory Pattern. Extends from SensorCreator. Creates a concrete product, which is CongestionSensor. Has Singleton in it.

**NoiseCreator:**  Concrete Creator of Factory Pattern. Extends from SensorCreator. Creates a concrete product, which is NoiseSensor. Has Singleton in it.

**PollutionSensor:**  Concrete Product of Factory Pattern. Extends from Sensor. Prints according information. Accepts incoming visitors with Accept(). Also checks the status of the sensor with checkSensorStatus(). If there is a malfunction detected, it sends a notification to engineers. Calculates whether the pollution level is above 100 AQI if so sends a notification to citizens with sensorCalculate().

**TemperatureSensor:** Concrete Product of Factory Pattern. Extends from Sensor. Prints according information. Accepts incoming visitors with Accept().

Also checks the status of the sensor with checkSensorStatus(). If there is a malfunction detected, it sends a notification to engineers. Calculates whether the temperature falls below 0 if so sends a notification to citizens with sensorCalculate().

**CongestionSensor:** Concrete Product of Factory Pattern. Extends from Sensor. Prints according information. Accepts incoming visitors with Accept(). Also checks the status of the sensor with checkSensorStatus(). If there is a malfunction detected, it sends a notification to engineers. Calculates whether car speed is below 10 km/hr if so sends a notification to citizens with sensorCalculate().

**NoiseSensor:** Concrete Product of Factory Pattern. Extends from Sensor. Prints according information. Accepts incoming visitors with Accept(). Also checks the status of the sensor with checkSensorStatus().

If there is a malfunction detected, it sends a notification to engineers. Calculates whether noise level is above 85 dB if so sends a notification to citizens with sensorCalculate().

**Pole:**  Leaf of Composite Pattern. Implements Neighborhood. Prints "Cannot add to a PrimitiveElement" and "Cannot remove from a PrimitiveElement." Gets name of pole. Installs sensors on poles. Displays indentation.

**Apartment:** Leaf of Composite Pattern. Implements Neighborhood. Prints "Cannot add to a PrimitiveElement" and "Cannot remove from a PrimitiveElement." Gets the name of apartment. Installs sensors on apartments. Displays indentation.

**Street:** Composite of Composite Pattern. Implements Neighborhood. Gets name of street. Displays indentation. Adds or removes Apartments and Poles to Street with Add() or Remove()

**DataMonitoringDivision:** Object Structure of Visitor Pattern. Adds sensors. Allows the visitor to visit its elements. Accepts incoming visitors. Also, has the list of sensors.

**Malfunction:** Concrete Visitor of Visitor Pattern**.** Implements Visitor. Visits all four sensor types. Looks for any malfunctions. If there is a malfunction, increments malfunctionCounter by one.

**Citizen:** Concrete Observer of the Observer pattern. Implements Observer. Gets notified when certain event happens for the installed sensors. Also gets the citizen name. Stores state that should stay consistent with the subject’s (Sensor).

**Engineer:** Concrete Observer of the Observer pattern. Implements Observer. Checks for malfunctioning sensors, if there is malfunction resets sensor. Stores state that should stay consistent with the subject’s (Sensor).

**5.3 INTERFACES**

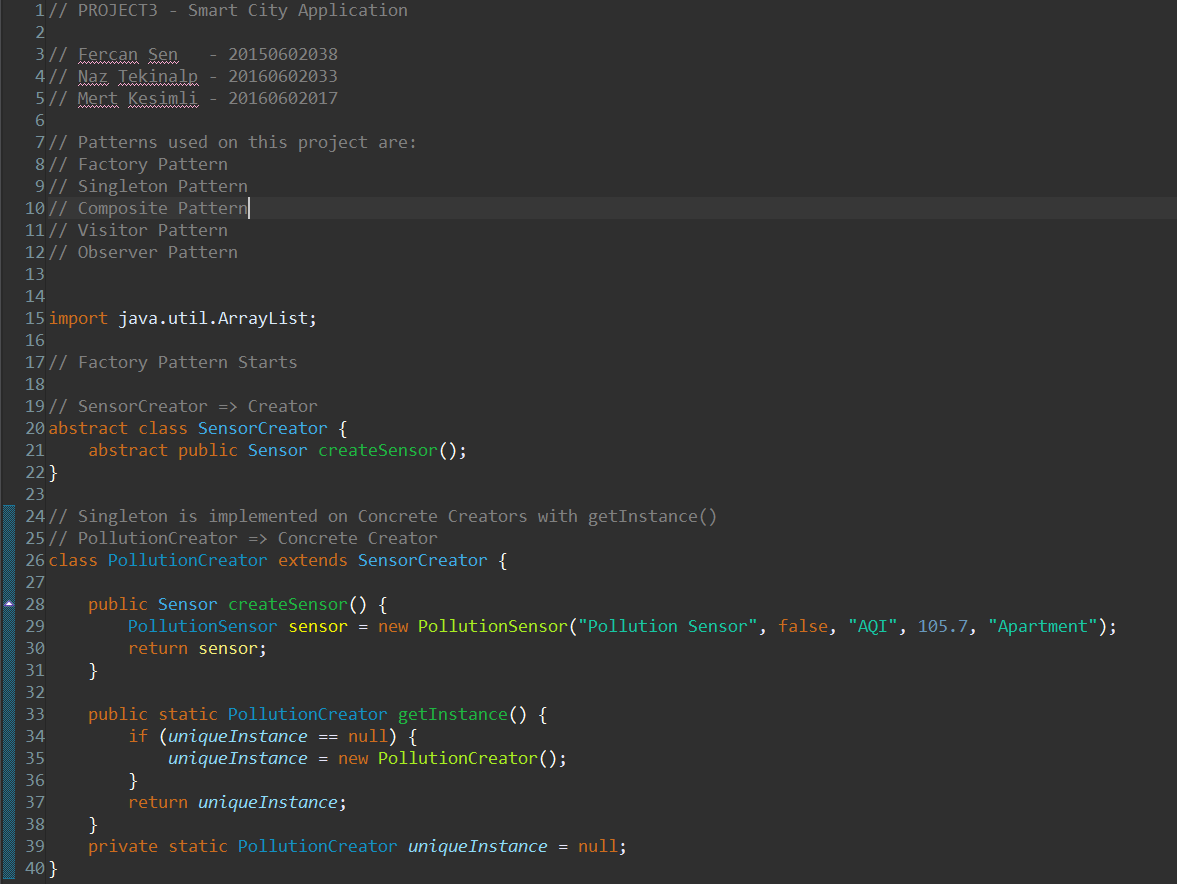
**Neighborhood:** Component of Composite Pattern. Adds and removes neighborhood, displays indentation, gets the name of the neighborhood.

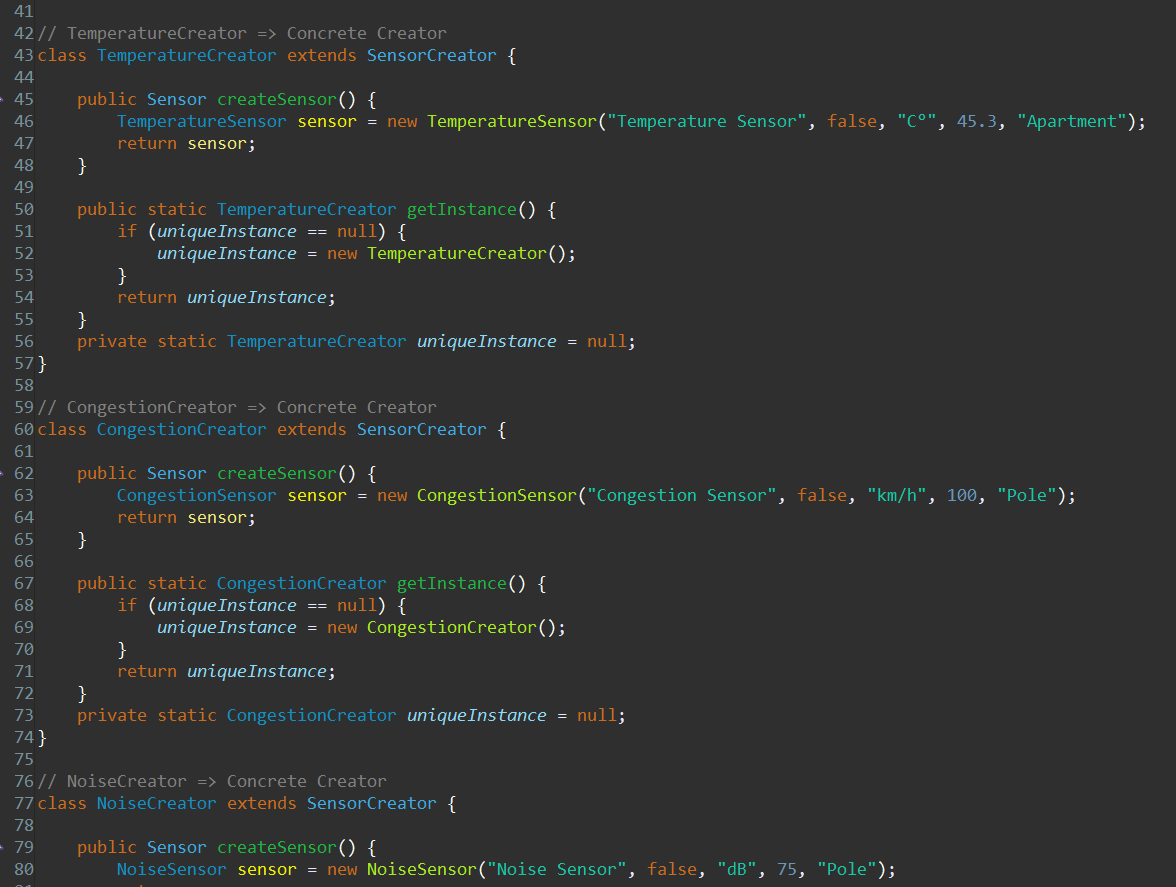
**Observer:** Updates sensor.

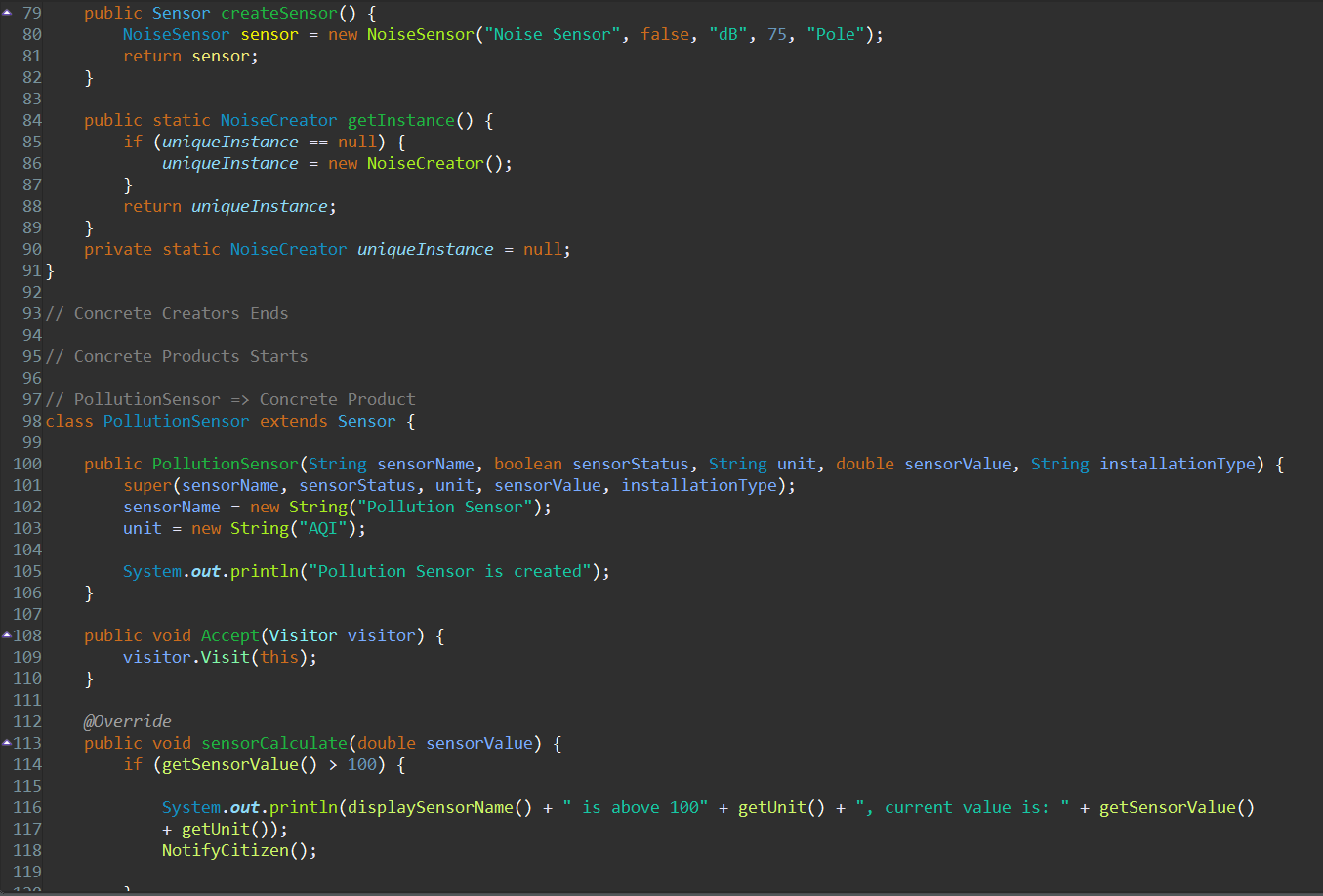
**Element:** Accepts incoming visitors.

**Visitor:** Defines a Visit operation for ConcreteElements (Visits sensors PollutionSensor, NoiseSensor, TemperatureSensor and CongestionSensor).

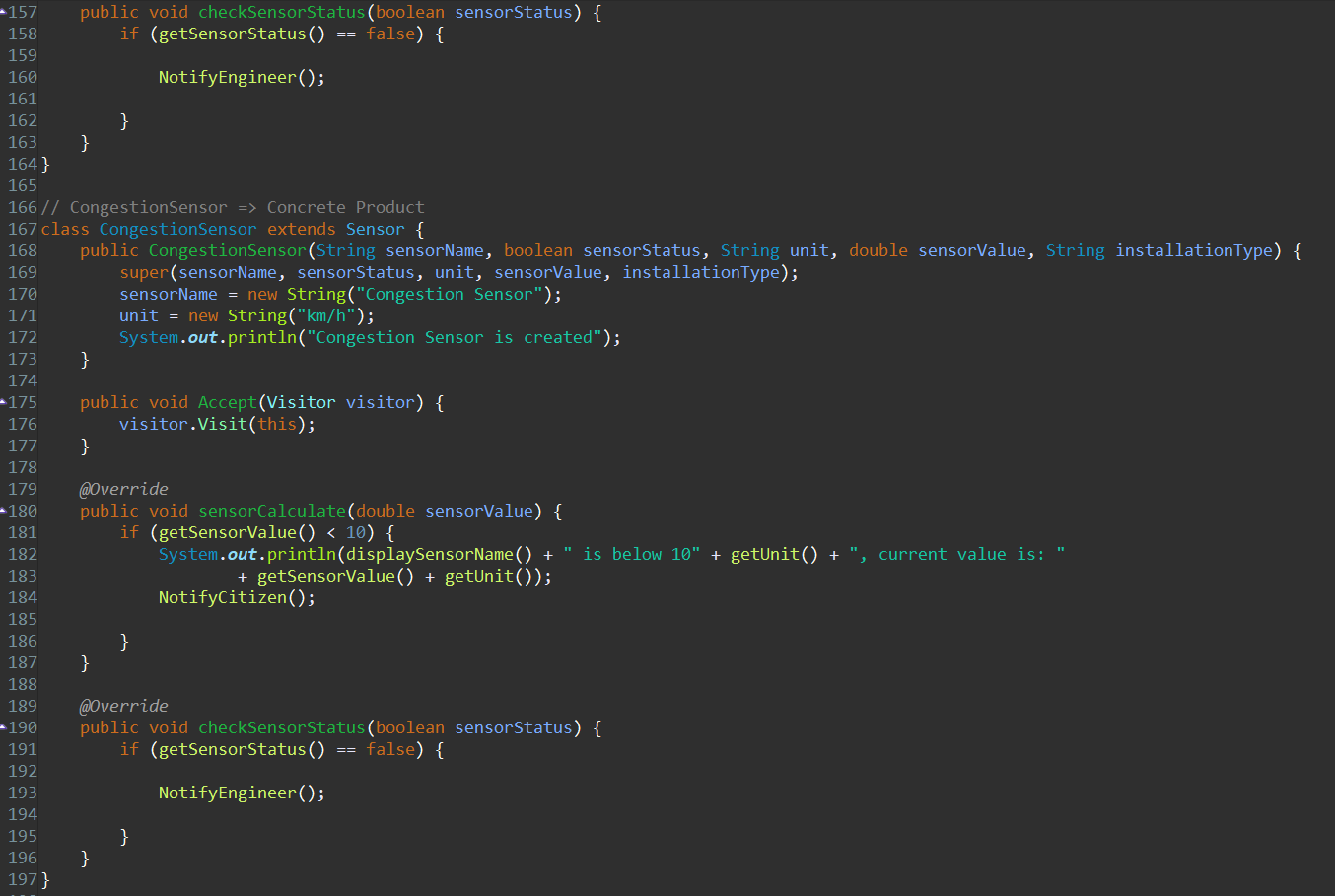
**6. Screen Dumps**

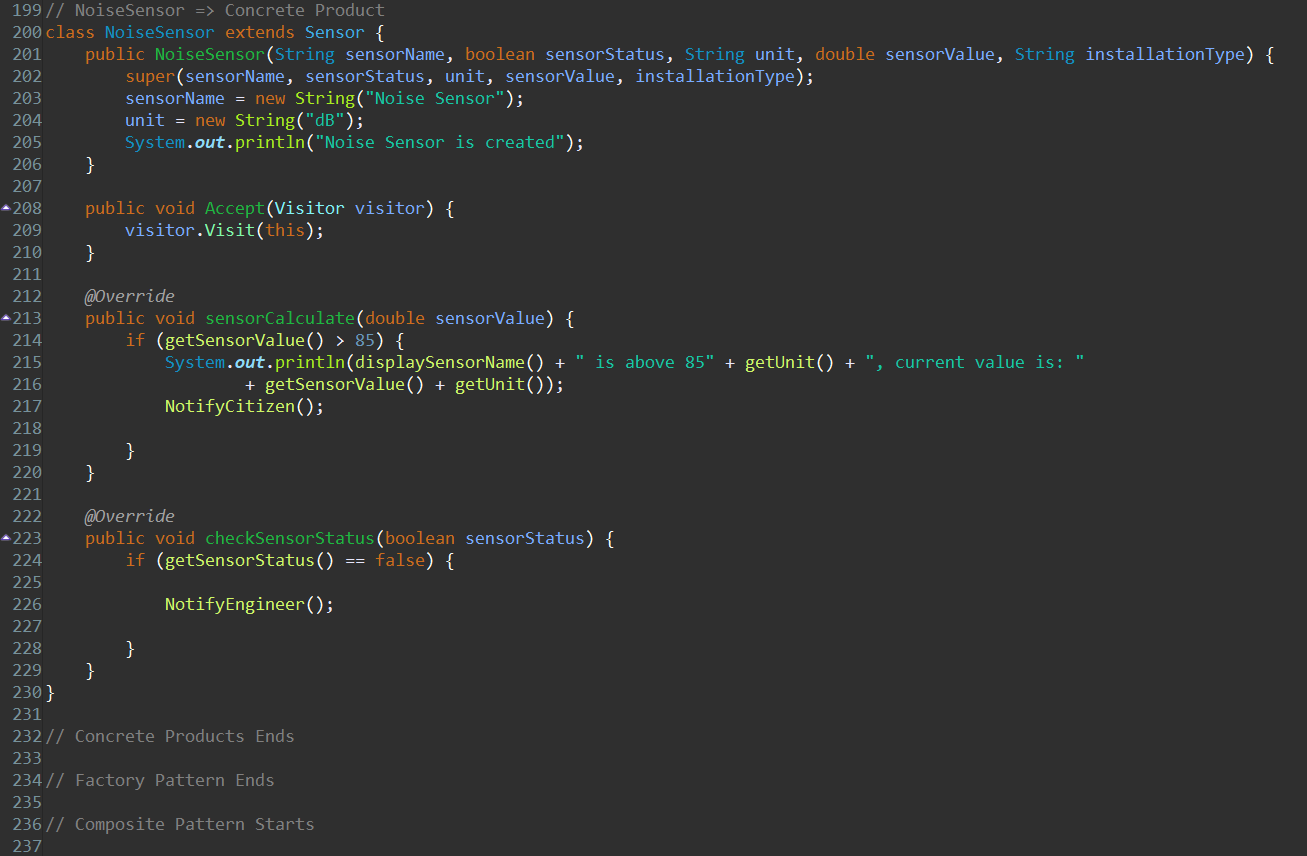
****

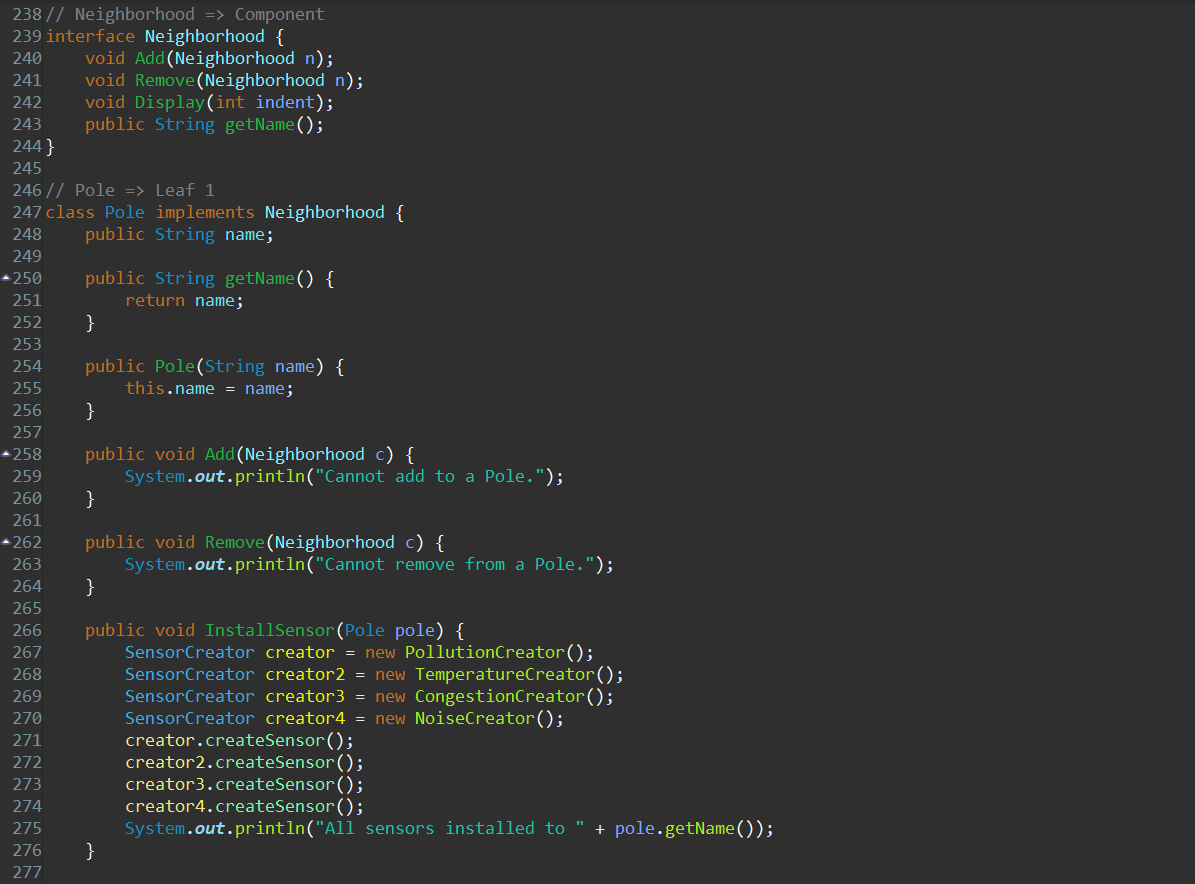
****

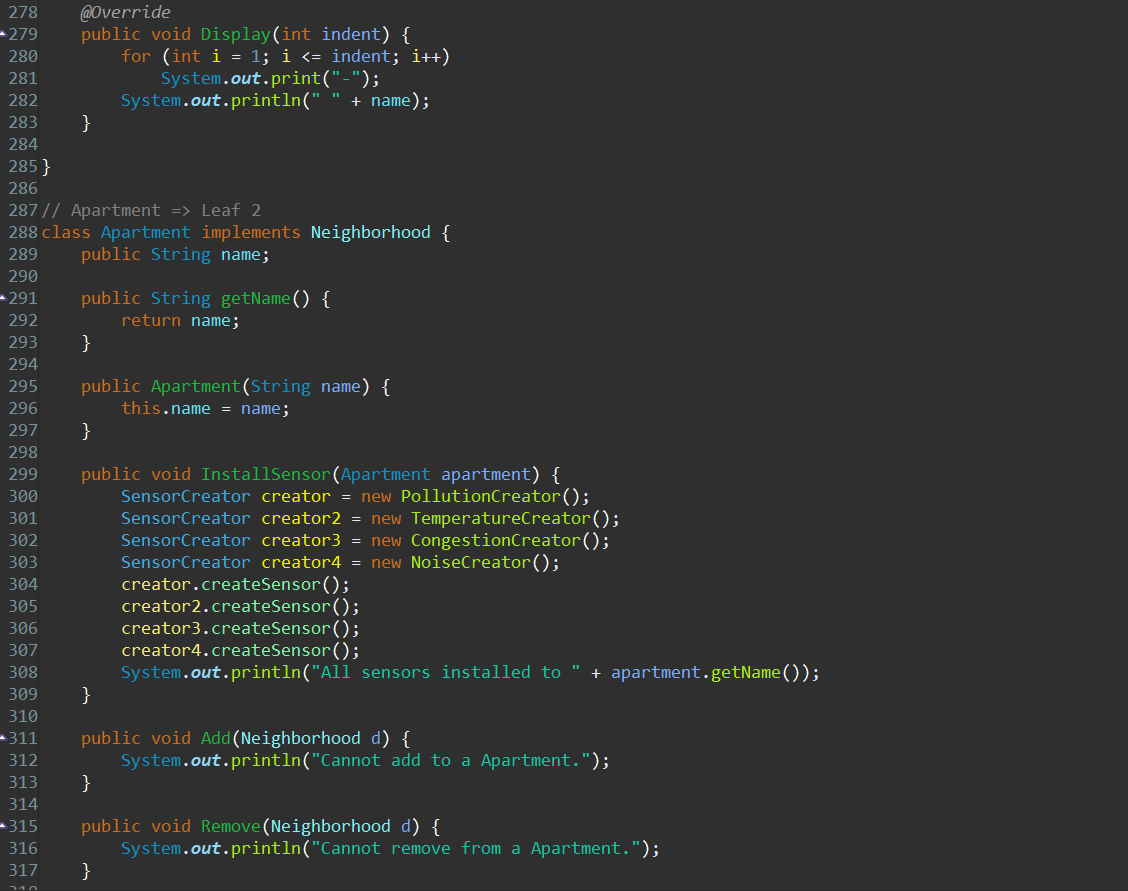
****

****

****

****

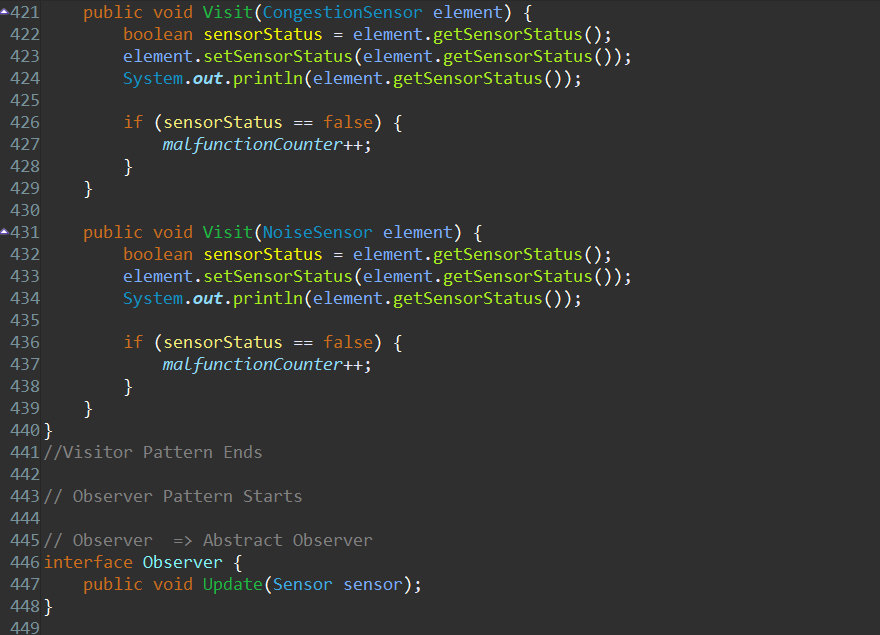
****

****

****

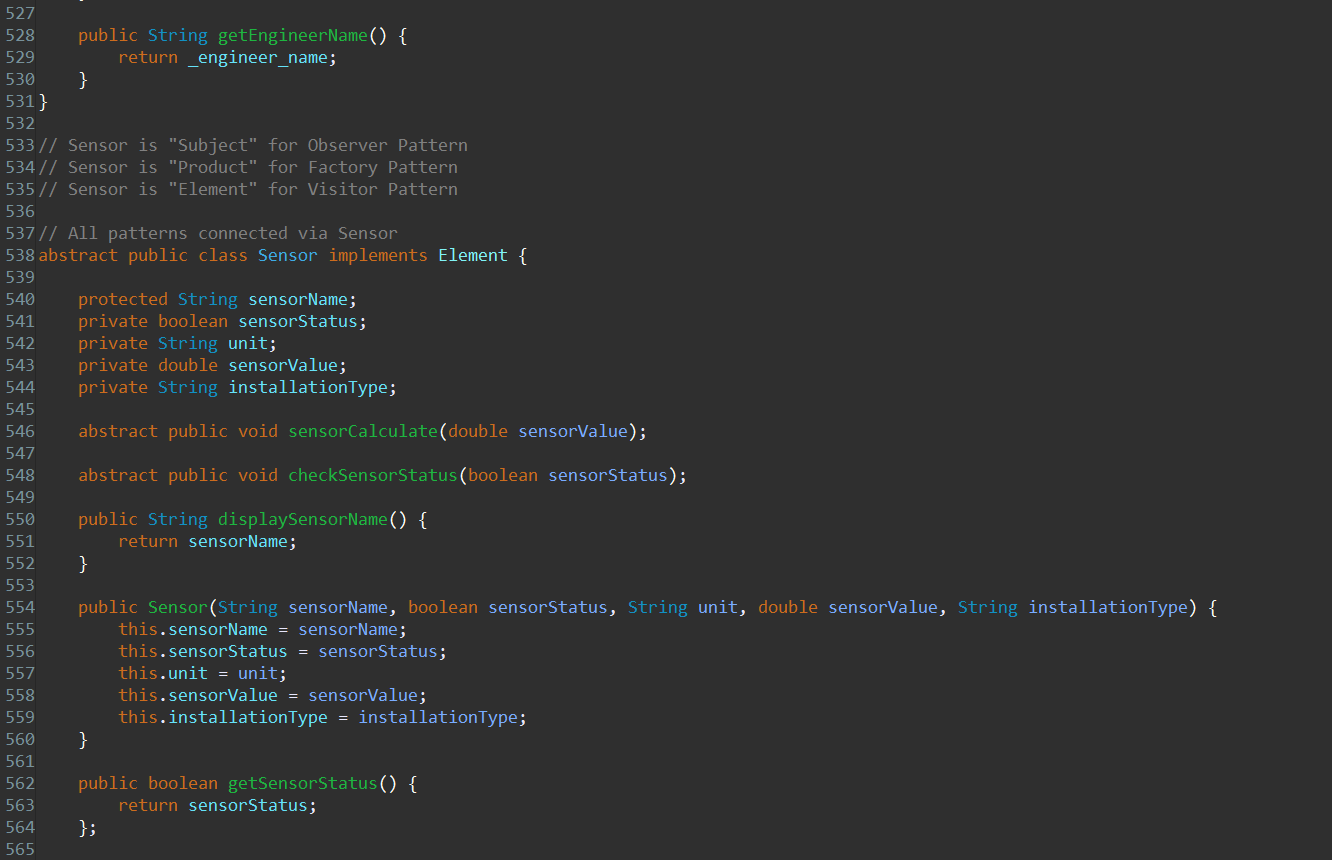
****

****

****

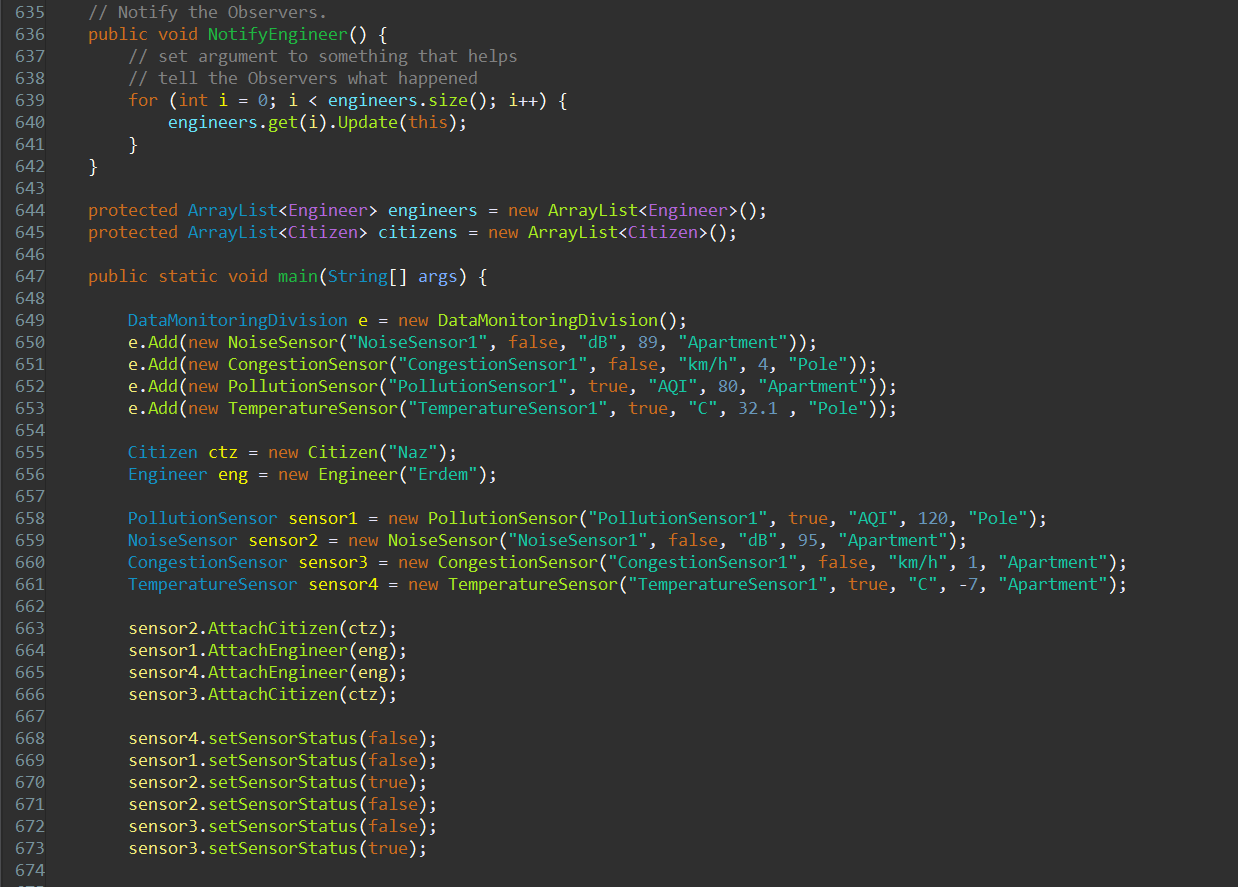
****

****

****

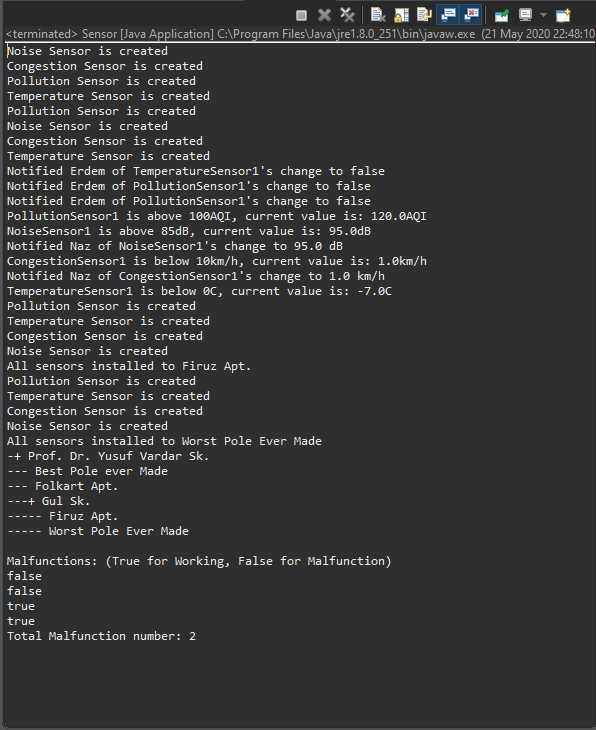
****

****

****

****

**7. Output**

****