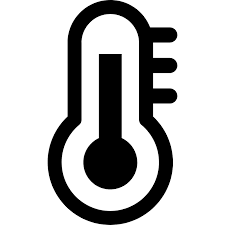
Smart City Application

ışık, gömlek içeren bir resim

Açıklama otomatik olarak oluşturulduoyuncak, tablo içeren bir resim

Açıklama otomatik olarak oluşturuldu

**SE311 - Software Architecture**

**Mehmet Aydın KICIRTI – 20160602105**

**Mustafa Ozan GÜNEY – 20160602093**

**Merve TELLİ - 20160602146**

1.Pattern Choices

**1.1)Factory Pattern**

ekran görüntüsü içeren bir resim

Açıklama otomatik olarak oluşturulduAccording to scenario,in smart city should have 4 type sensor and also these sensors installed everywhere,so Factory Method allows us to create objects from an specific subtype through a Factory class.We have 4 types of sensors: temperature, pollution, congestion and noise.Therefore we think that it should be provided with Factory Pattern and object creation delegation to these sensors.Usage of Factory pattern here will be provided to create different sensors without showing their inner parts.In order to without coupling the client code to concrete classes.

**Participants =>**

**AbstractFactory:** SensorFactory

**ConcreteFactory:**CongestionSensorFactory, NoiseSensorFactory, PollutionSensorFactory, TemperatureSensorFactor

**AbstractProduct:** Sensor

**ConcreteProduct:** NoiseSensor, CongestionSensor, TemperatureSensor, PollutionSensor

**Client:** SensorCreator

**AbstractFactory:** SensorFactory

* The abstract factory declares a set of methods that return different abstract products. Factory classes will implement this abstract class and return their respective sub-class with createSensor().

**ConcreteFactory:**CongestionSensorFactory, NoiseSensorFactory, PollutionSensorFactory, TemperatureSensorFactor

* Every single sensor factory create the base sensor factory with use createSensor(). For example, NoiseSensorFactory create from SensorFactory. These classes implement operations to create concrete products.
* ‘name’and‘sensorValue’parameters create objects belonging to that sensor.

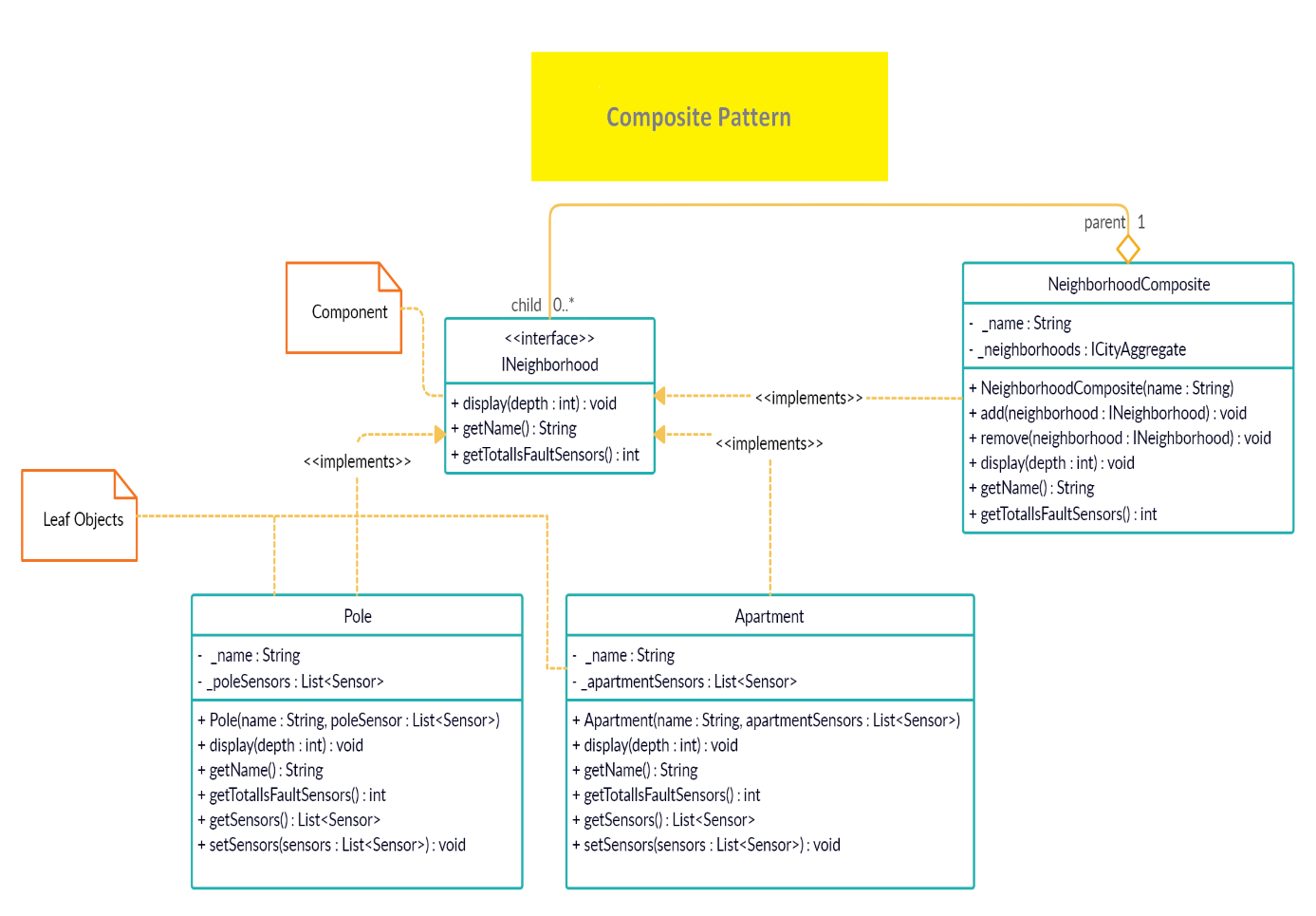
**ConcreteProduct:** NoiseSensor, CongestionSensor, TemperatureSensor, PollutionSensor

* These are our sensors. Every single sensor have own value. These are created from sensor factories.For example, Pollution sensor create from PollutionSensorFactory.

**Client:** SensorCreator

* This class help to our object creation delegation with factory method.Class has createSensor() function and this function is provided by the SensorType parameter.

**1.2)Composite Pattern**

The purpose of this design pattern is to arrange the objects according to the tree structure and establish the upside down relationship in the tree structure. According to this design pattern, the upper and lower objects in the tree structure derive from the same interface class and look like each other. The neighborhoods have streets with apartments and poles. Since this scenario has a hierarchical structure, the use of composite pattern is suitable for this.

**Participants =>**

**Component:** INeighborhood

**Composite** :NeighborhoodComposite

**Leaf:** Apartment, Pole

**Component:** INeighborhood

* The component(INeighborhood) interface declares common operation for objects of a composition.

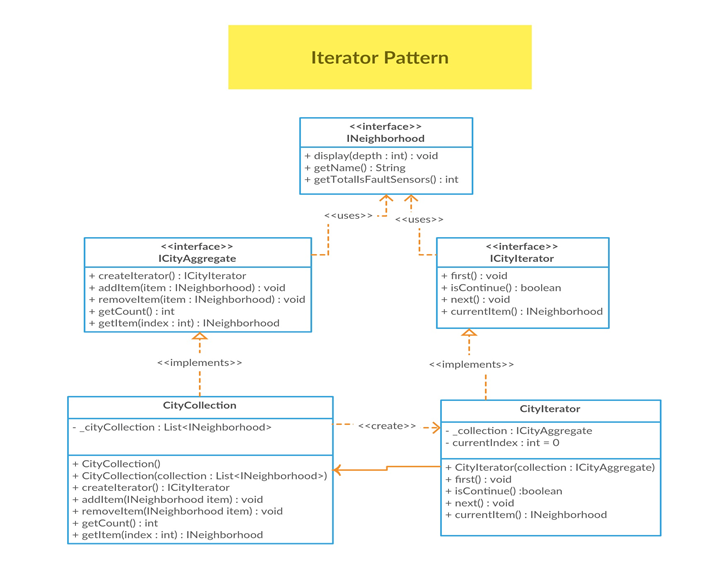
**Composite:** NeighborhoodComposite

* The composite class represent complex componenet thay may have children and composite objects usually delegate the actual work to their children and then “sum up” the result.
* addNeighborhood() and removeNeighborhood() methods enable rotating the component.
* It navigates the leaves with the display() method and also provides those leaves.

**Leaf:** Apartment, Pole

* The leaf classes represent end objects of a composite.Defines the behaviour fort he elements in the composition.We get the names of the leaf classes with the getname().
* Navigates in sensors with display() method and returns faulty sensors with getTotalIsFaultSensor().

**1.3)Iterator Pattern**

According to scenario, you said that “the city is a collection of neighborhoods, So we think that the best solution should be implement of iterator pattern and also we can easily get around without any dependency its underlying representation.

**Participants =>**

**Aggregate:** ICityAggregate

**Iterator:** ICityIterator

**ConcereteAggregate:** CityCollection

**ConcreteIterator:** CityIterator

**Aggregate:** ICityAggregate

* ICityAggregate interface defines the contract for our collection class implemantation.Notice that there are methods to use add() or remove() a Neigborhood.ICityAggregate has a method that returns the iterator for traversal.

**Iterator:**ICityIterator

* This is an interface for accessing and traversing elements.

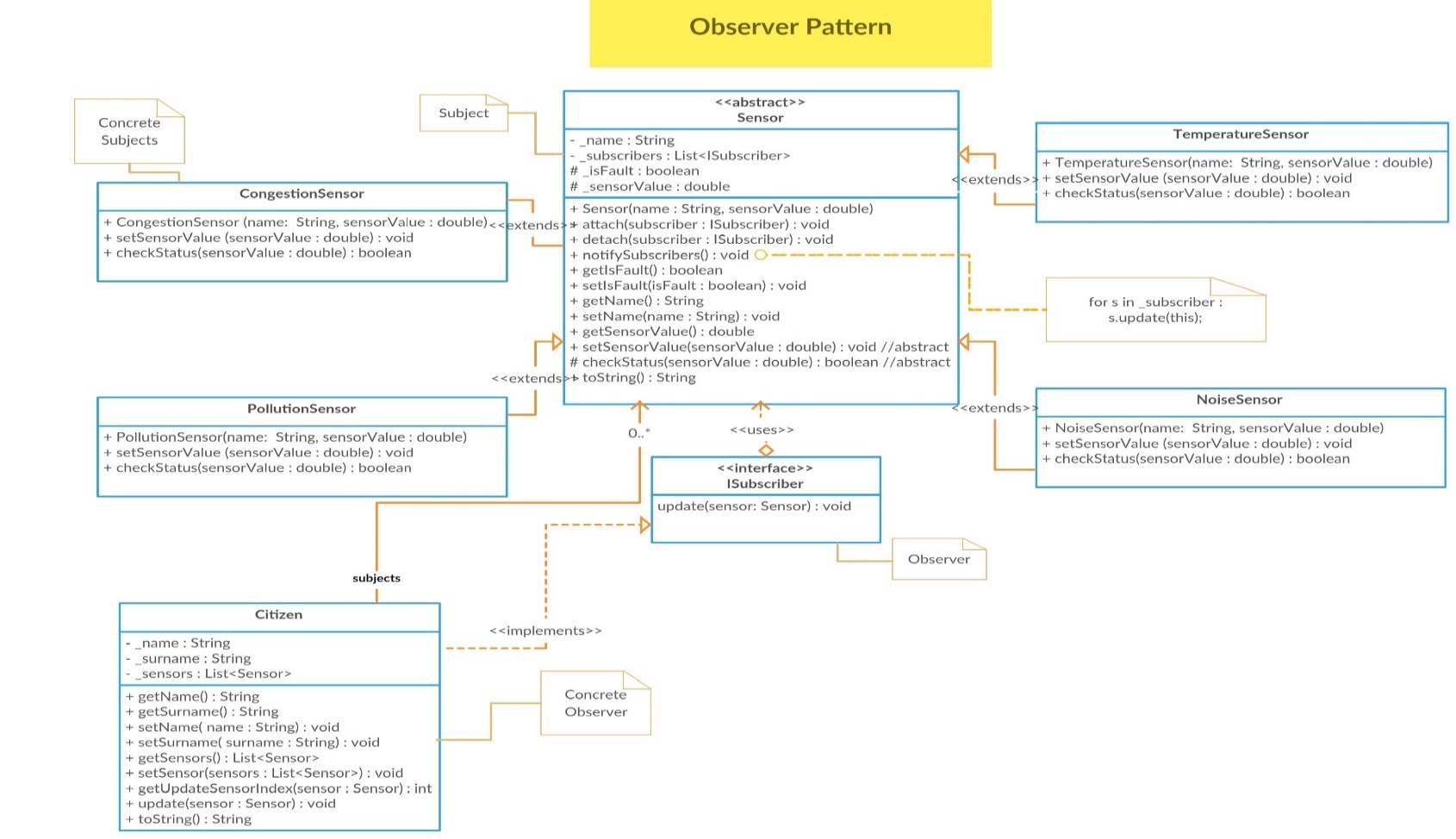
**ConcreteAggregate:** CityCollection

* Each concrete collection is coupled to set of concrete iterator classes it returns. It is a concrete class created using the ICityAggregate. The class contains an array of Neigborhood.

**ConcreteIterator:** CityIterator

* An iterator object traverses the collection independently from other iterators. Therefore it has to store the iteration state.The class implements the iterator interface and defines all the methods of the iterator interface.

**1.4)Observer Pattern**

In the scenario,If it provides the values ​​on the sensors,we should send an information message.In order to solve this situation.Observer pattern can be able to the best choice since we will attach who subscribe to the sensors as an notification.

**Participants =>**

**Subject:** Sensor

**Observer:** ISubscriber

**Comcrete Subject:** CongestionSensor, TemperatureSensor, PollutionSensor, NoiseSensor

**Concerete Observer :** Citizen

**Subject:** Sensor

* This is an abstract class for the subject.Sensor class provides the following;
* Attaching and detaching observers to Concrete Subjects.In order to attached an observer there is any state change on sensor value. Citizen knows.
* Notify function is notifySubscribers() to inform status of sensor value changing.
* We use setSensorValue() function. After changed of sensorvalue the belonging to sensors firstly checking status with checkStatus() function if this provide this condition, will notified subscribers.

**Observer:** ISubscriber

* This is the interface for Observer.As you can see from the name of it.ISubscriber object observes subscribed the sensors.This interface has an important function following in it;
* Update(sensor:Sensor): Function will work after changing status of the sensorValue if condition is provided.It will update.Observer object is taking current state to the incoming state.

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

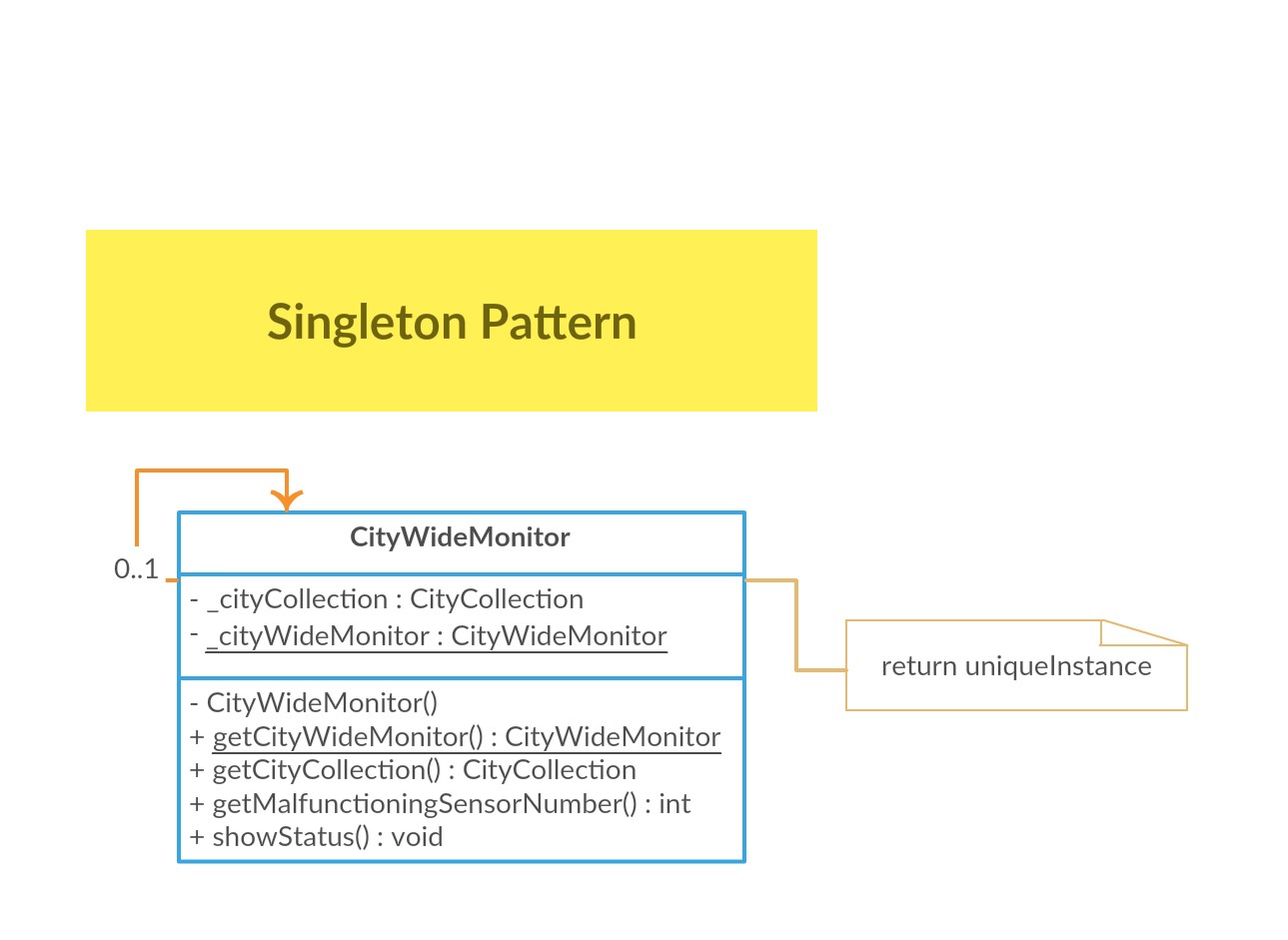
* Derived object from Sensor.These sensors has same operations of with it.Will be used as a condition results provided to notify in our scenario.

**Concrete Observer :** Citizen

* Derived object from ISubscriber.It has also same member of functions and it will work as a Concrete Observer.

**1.5)Singleton Pattern**

Scenario says that should be only one Data Monitoring Division in the city. The usage of the singleton would provide only one instance of monitor for object creation. In our case, it creates CityWideMonitor class.



**Participants =>**

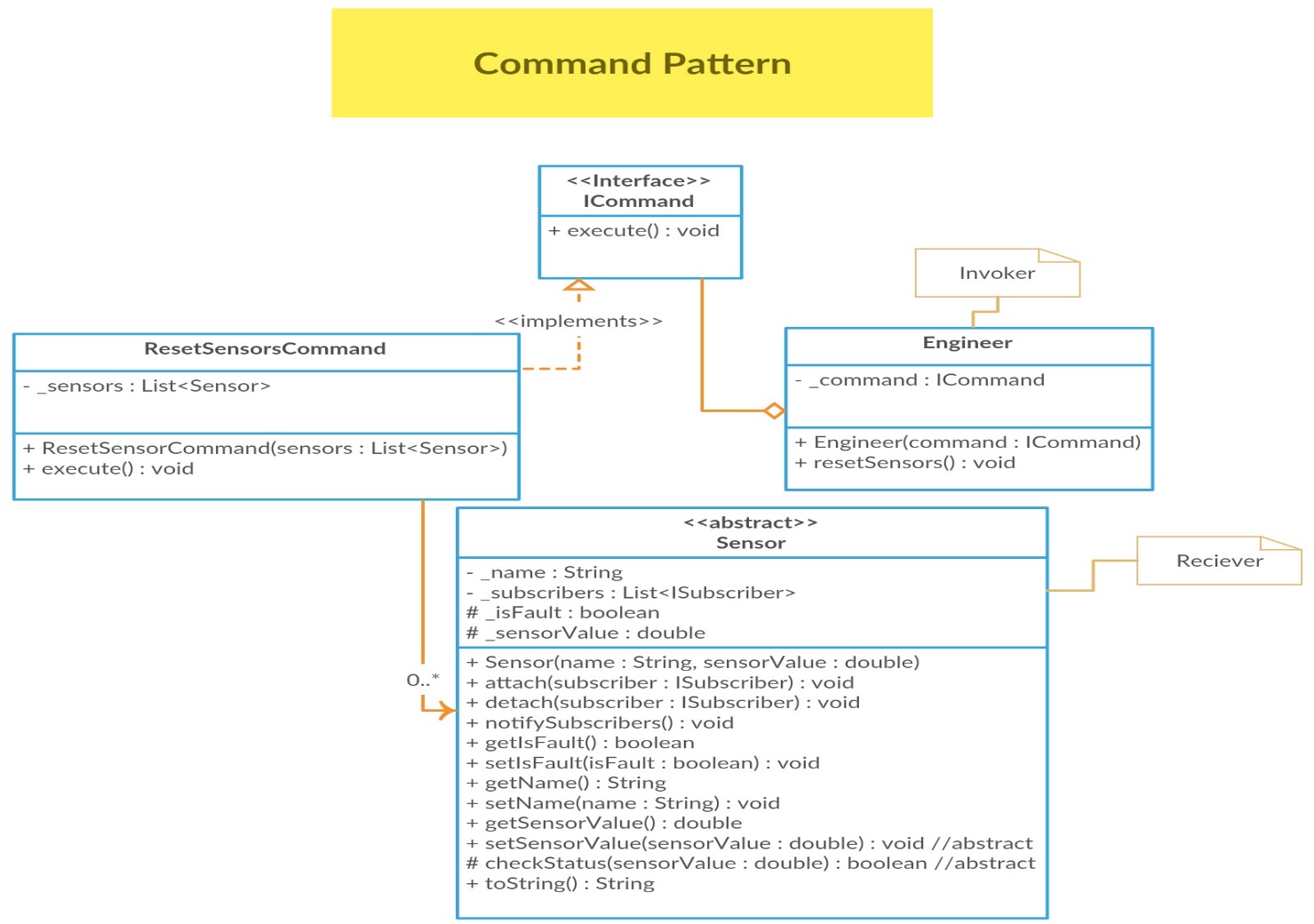
**Singleton :** CityWideMonitor

**Singleton class :** CityWideMonitor

* If we want to construct a singleton class we need to make first constructor private or protected and also class name instance should be static and again private or protected so that when we want to create instance provide by getCityWideMonitor().This function returns either before we did not create firstly class instantiation or directly returns class instatiation.

And also we can use cityCollection when we were creating a neighorhood with this instance provide addNeighbor into collection.Therefore we can get easily status of city with showStatus()function by the way getMalfunctioningSensorNumber() gets us number of malfunctioning sensor into system. We used with showStatus() function.

**1.6)Command Pattern**

According to the scenario, if there is a malfunctioning sensor in city, the engineer sends a request to reset these sensors. Therefore the Command pattern provided to us if there are a request and response relationship in these between subsystems, think of the best solution for us. In order to directly call reset sensors will increase the dependency, instead of realized to implementation of this pattern and also this pattern encapsulate and forward the reset request to the Engineer which performs as an Invoker in there.

**Participants =>**

**Command:** ICommand

**ConcreteCommand:** ResetSensorsCommand

**Invoker:** Engineer

**Reciever:** Sensor

**Command:** ICommand

* This class is the Command of the commands This is an interface class including execute() method for subcheck commands.

**ConcreteCommand:** ResetSensorsCommand

* This class is the Concrete Command of the commands. Implements the ICommand class and implement the execute().

**Invoker:** Engineer

* This class is the Invoker of the commands. We create a engineer(command) for check. If it is necessary use resetSensors() and reset all broken sensors.

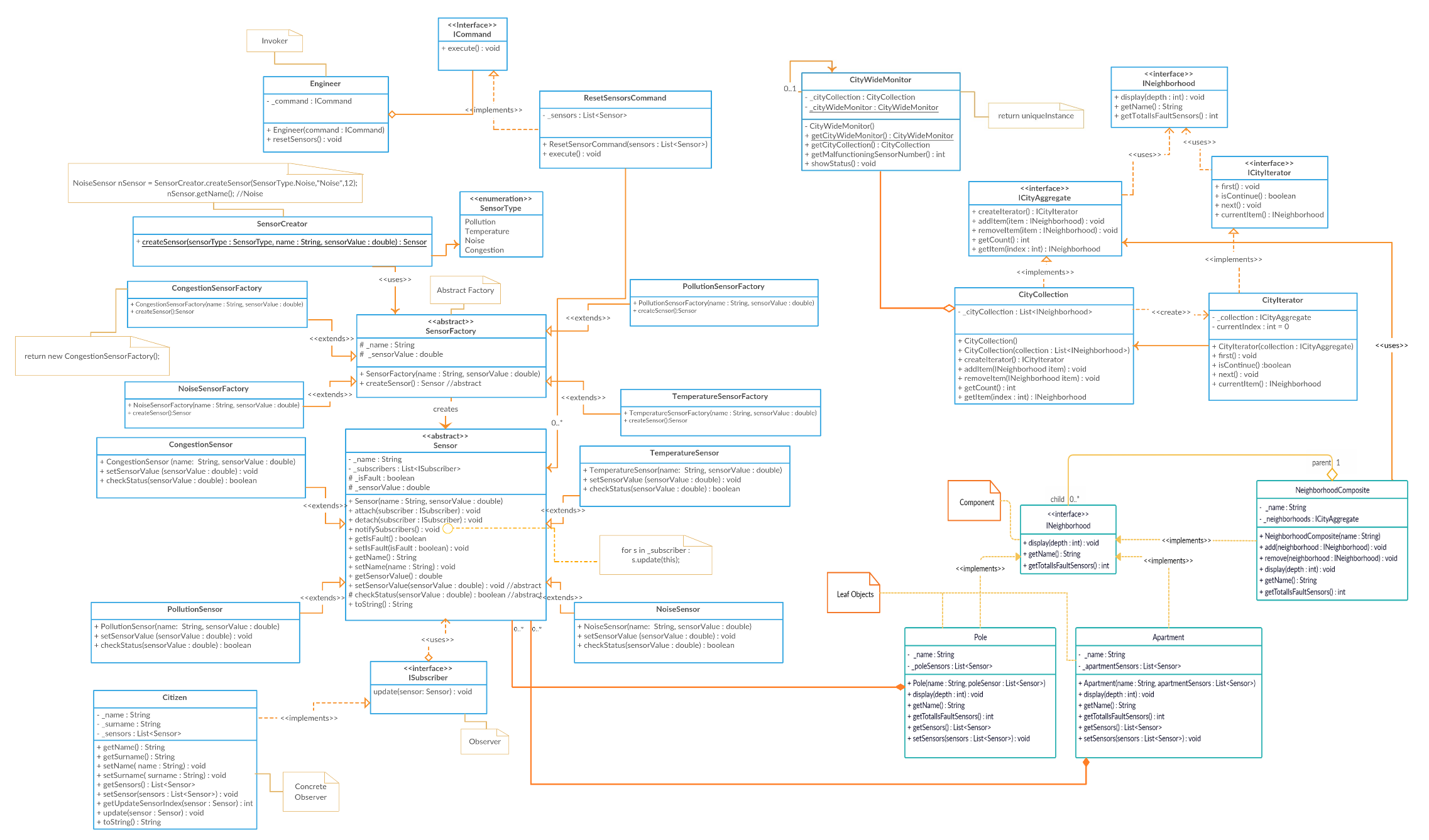
**Reciever:** Sensor

* This class is the Reciever of the commands. Every sensors have different portion for maximum value. When engineer wants to check all sensors,use checkStatus().If any sensor is broken use resetSensor() and reset all broken sensors.

**2.)Conclusion**

In the Smart City Application, we tried to implement 6 patterns. Before drawing the UML, we tried to think about the best choice patterns from the scenario. After that, we started to draw the UML, but this part was so hardest part for us when we were trying to connect classes in UML. We used creately to draw the class diagram, then we were adapting to own structures on application. As a result, we aimed to understand the best solution to the use of design patterns.

UML Diagram



**3.)Running Code**

