

INSA DE LYON
COMPUTER SCIENCES DEPARTMENT
SOFTWARE ENGINEERING & UML

GL-UML Lab
AirWatcher

Auteurs:

Melisse COCHET
Saad ELGHISSASSI
Jassir HABBA
Sekyu LEE
Simon PERRET

Professors:

Mrs. LAFOREST
Mr. HASAN

Contents

1	Conception	1
1.1	Architecture	1
1.2	Class diagram	2
1.3	Sequences diagram	3
1.4	Description and pseudo-code	4
1.4.1	Analyze data to make sure sensors function correctly	4
1.4.2	Observe the impact of cleaners on air quality	6
1.4.3	Classify Individuals as Unreliable	6
1.5	Unit testing	7
1.5.1	checkSensor	7
1.5.2	distance	7
1.5.3	calculateMeans	7
1.5.4	calculateWeightedMeans	7
1.5.5	checkCleaner	8
1.5.6	checkPrivateSensor	8

1 Conception design in English

1.1 Architecture

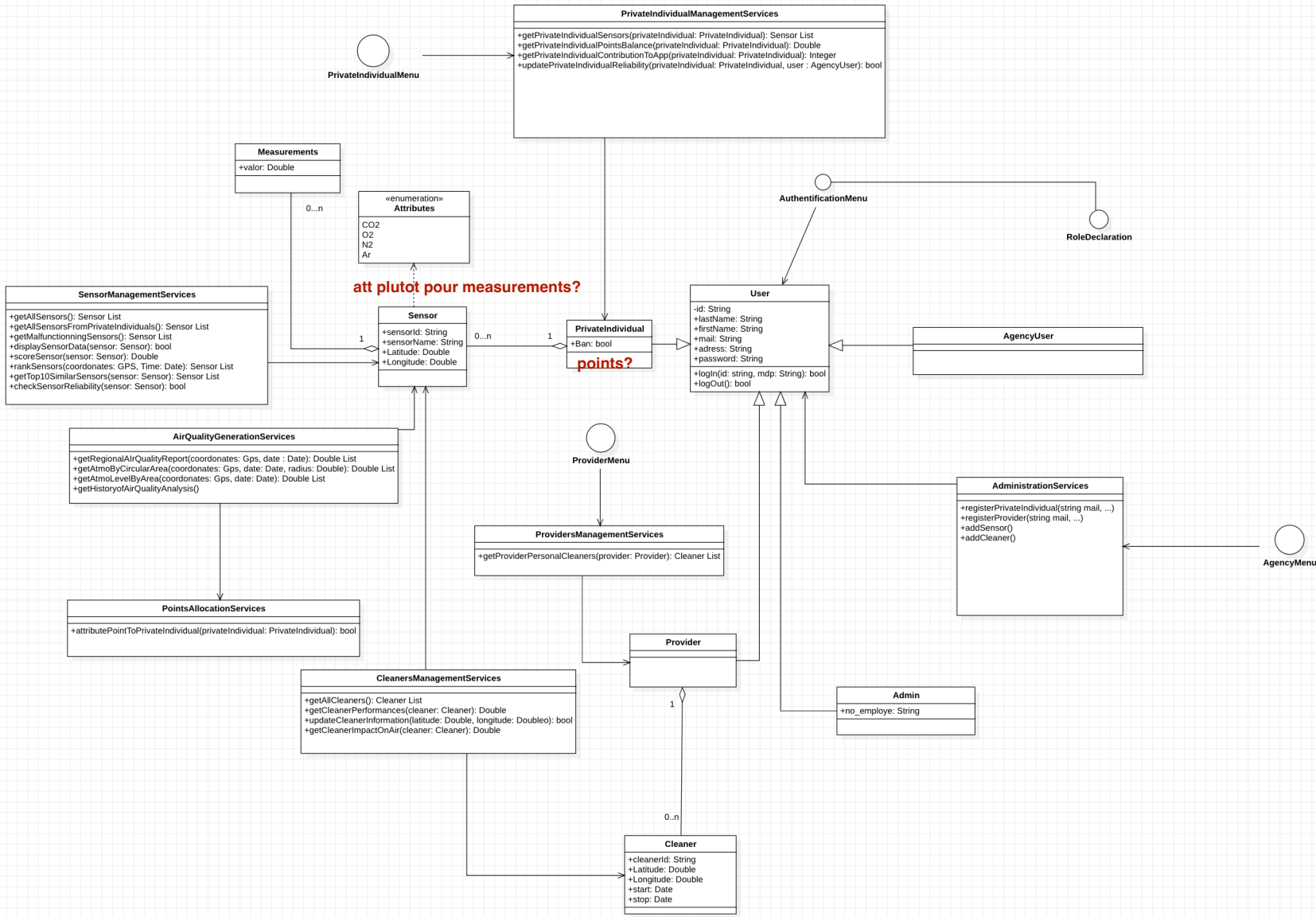
We have chosen to develop our application on a layered based architecture. This enables a concrete Separation of Concerns while developing (Interface Layer - Services Layer - Data Layer) and a better coding organization. The decoupling between layers also allows easier maintenance and updating of the system and a better scalability as the system could handle large amounts of data. Reusability was also a crucial point for choosing this architecture as components developed within each layer, especially within the business logic and data access layers, can be reused across different parts of the application or even in different projects within the same organization. Eventually, this architecture ensures improved security of our application by isolating the layers, making sensitive data operations abstracted away from user interfaces, which was one of our security requirements.

The Repository architecture was also considered. It would have probably been easier to manage, but at high costs : Handling concurrent access to a central repository, especially when the data is stored in CSV files, can be complex and inefficient without the sophisticated concurrency control mechanisms typically provided by a DBMS.

Other architectures seemed totally irrelevant, such as the MVC, as AirWatcher's core functionality revolves more around data processing and analysis rather than user interaction, which might not benefit significantly from the MVC's strong UI orientation.

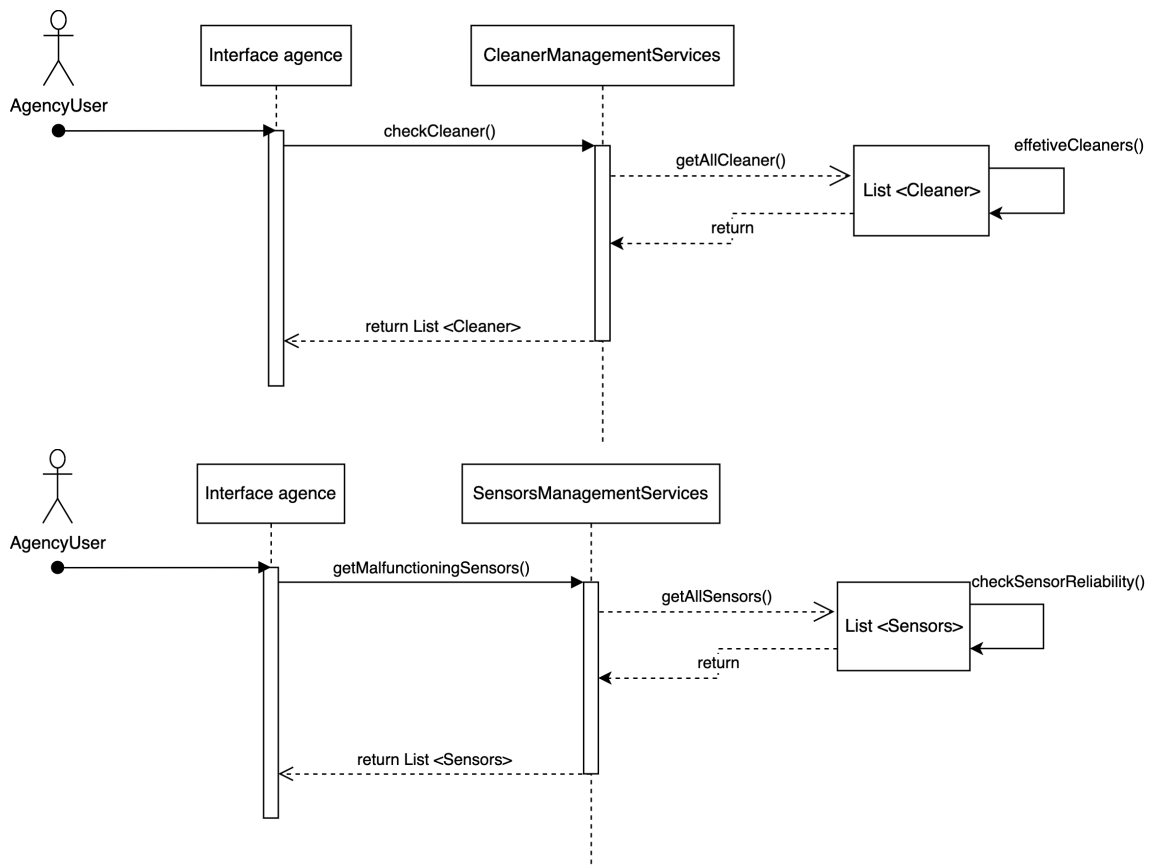
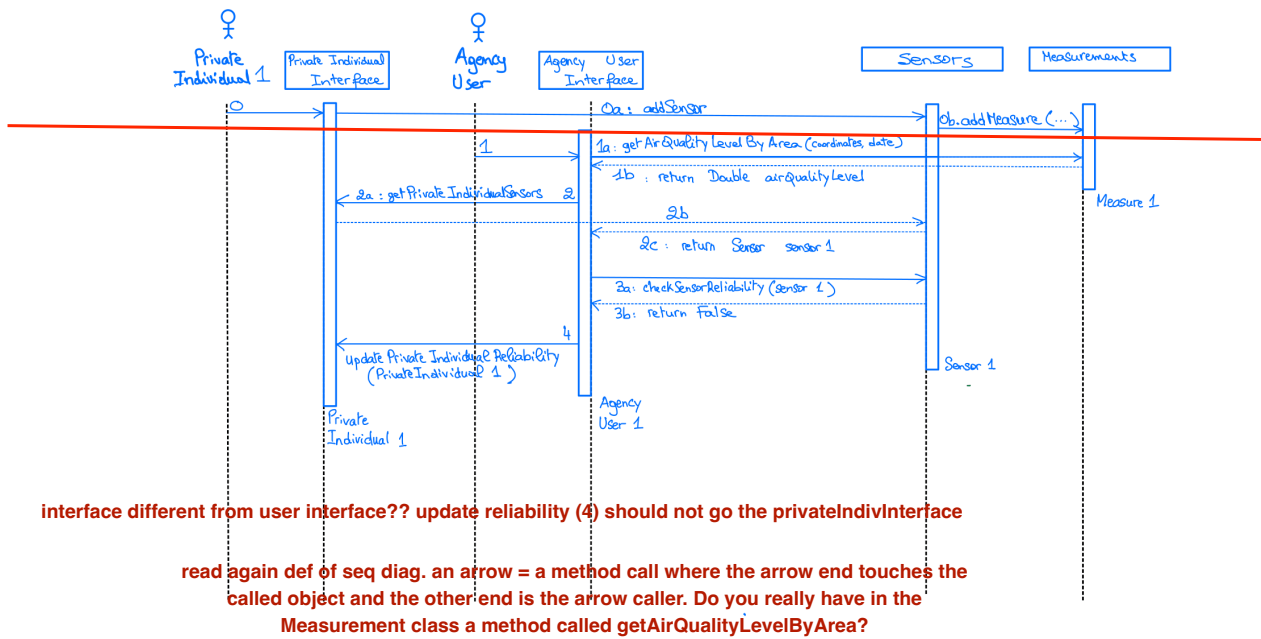
good

1.2 Class diagram



1.3 Sequences diagram

2 diagrams in one! make one diagram per use case



checkSensorReliability is in the List<Sensors> class? I don't see a List<Sensors> class in the class diagram

1.4 Description and pseudo-code

We chose to develop 3 main algorithms such as verifying that the sensors are working correctly, that the air cleaners are effective and finally to verify the veracity of private user sensors.

1.4.1 Analyze data to make sure sensors function correctly

We will select measurements from the sensor provided over the last week and calculate the means for each concentration (O3, SO2, NO2, PM10). Next, we will identify the five closest sensors and assign them weights based on their distance from the selected sensor. Then, we will calculate the means for each concentration during the same period with these weights. Finally, we will compare each concentration with those from the selected sensor, and if the margin of error exceeds 10% for at least one concentration, the sensor will be deemed malfunctioning.

Algorithm 1 Check the operation of a sensor

Require: *sensorSelected*

Ensure: boolean (false if the sensor is faulty)

```
1: checking  $\leftarrow$  true
2: meanO3, meanSO2, meanNO2, meanPM10  $\leftarrow$  calculateMeans(sensorSelected)
3: nearbySensors  $\leftarrow$  empty list
4: for i  $\leftarrow$  1 to 5 do
5:   sensor  $\leftarrow$  closestSensor(sensorSelected, nearbySensors)
6:   nearbySensors.append(sensor)
7: end for
8: weightedMeanO3, weightedMeanSO2, weightedMeanNO2, weightedMeanPM10  $\leftarrow$ 
   calculateWeightedMeans(sensorSelected, nearbySensors, 7)
9: for concentration  $\in$  {meanO3, meanSO2, meanNO2, meanPM10} do
10:  marginError  $\leftarrow$  calculateMarginError(concentration, weightedConcentration)
11:  if marginError > 0.1 then
12:    checking  $\leftarrow$  false
13:  end if
14: end for
15: return checking
```

Required functions

Algorithm 2 distance

Require: *sensor1 sensor2*

Ensure: double

```
1: return distanceEuclidienne
```

Algorithm 3 closestSensor

Require: *sensor1, excludedSensor*

\triangleright List of sensors to exclude

Ensure: Returns the closest sensor

```
1: dist  $\leftarrow$   $+\infty$ 
2: closestSensor  $\leftarrow$  0
3: for sensor not in excludedSensor do
4:   if Distance(sensor1, sensor) < dist then
5:     dist  $\leftarrow$  Distance(sensor1, sensor)
6:     closestSensor  $\leftarrow$  sensor
7:   end if
8: end for
9: return closestSensor
```

Algorithm 4 calculateMeans

Require: *Period* is the timespan over which we want to perform the calculation (*uptonow*), *indays* (integer)

Ensure: return the means for each concentration (O3, SO2, NO2, PM10)

- 1: **for** each *measurement* in measures of *sensor* **do**
 - 2: *period* \leftarrow select the period
 - 3: *avg* \leftarrow average of concentration
 - 4: **end for**
 - 5: **return** *meanO3, meanSO2, meanNO2, meanPM10*
-

Algorithm 5 Calculate Weighted Means of Concentrations

Require: *selected_sensor, nearby_sensors, period* \triangleright Selected sensor, list of nearby sensors, and the period in days

Ensure: Returns the weighted averages for each concentration (O3, SO2, NO2, PM10)

- 1: *weight_tot* \leftarrow 0
 - 2: *meanO3* \leftarrow 0
 - 3: *meanSO2* \leftarrow 0
 - 4: *meanNO2* \leftarrow 0
 - 5: *meanPM10* \leftarrow 0
 - 6: **for** each *sensor* in *nearby_sensors* **do**
 - 7: *weight* \leftarrow Distance(*selected_sensor, sensor*)
 - 8: *weight_tot* \leftarrow *weight_tot* + *weight*
 - 9: *tempMeanO3, tempMeanSO2, tempMeanNO2, tempMeanPM10* \leftarrow Calculate_means(*sensor, period*)
 - 10: *meanO3* \leftarrow *meanO3* + *weight* \times *tempMeanO3*
 - 11: *meanSO2* \leftarrow *meanSO2* + *weight* \times *tempMeanSO2*
 - 12: *meanNO2* \leftarrow *meanNO2* + *weight* \times *tempMeanNO2*
 - 13: *meanPM10* \leftarrow *meanPM10* + *weight* \times *tempMeanPM10*
 - 14: **end for**
 - 15: **if** *weight_tot* > 0 **then**
 - 16: *meanO3* \leftarrow *meanO3* / *weight_tot*
 - 17: *meanSO2* \leftarrow *meanSO2* / *weight_tot*
 - 18: *meanNO2* \leftarrow *meanNO2* / *weight_tot*
 - 19: *meanPM10* \leftarrow *meanPM10* / *weight_tot*
 - 20: **end if**
 - 21: **return** (*meanO3, meanSO2, meanNO2, meanPM10*)
-

1.4.2 Observe the impact of cleaners on air quality

The algorithm aims to evaluate the effectiveness of air cleaners by analyzing air quality data from sensors located around the cleaner's operational area. It focuses on comparing air quality data from before and during the cleaner's operation to ascertain any improvements.

Algorithm 6 Evaluate the effectiveness of an air cleaner

Require: selectedCleaner

Ensure: boolean (true if the cleaner is effective, false otherwise)

```
1: closestSensors[3] ▷ Table to store the closest sensors
2: i ← 0
3: check ← 0
4: result ← false
5: while i < 3 do
6:   sensor ← closestSensor(selectedCleaner, closestSensors)
7:   if sensor ∉ closestSensors then if I understood well closestSensors (in 1.4.1), sensor cannot belong to closestSensors
8:     closestSensors[i] ← sensor
9:     i ← i + 1
10:  end if
11: end while
12: for i ← 0 to 2 do
13:   periodAnte ← definePeriodBefore(selectedCleaner)
14:   periodPost ← definePeriodAfter(selectedCleaner)
15:   meanAnte ← calculateMeans(closestSensors[i], periodAnte)
16:   meanPost ← calculateMeans(closestSensors[i], periodPost)
17:   if meanPost < meanAnte then
18:     check ← check + 1
19:   end if
20: end for
21: if check == 3 then
22:   result ← true
23: end if
24: return result
```

1.4.3 Classify Individuals as Unreliable

The algorithm determines the reliability of data provided by users by comparing their sensor measurements with those of the three geographically closest sensors on the same date. If a user's sensor data consistently deviates from the norm by more than 20% for more than 10 measurements and the nearest sensor is within 5 km, the user is marked as unreliable and banished.

Algorithm 7 Classify Sensor Data Reliability

Require: *UserId, SensorId*

▷ From users.csv

Ensure: Updated reliability status in measurements.csv and users.csv

```
1: user_sensor_data ← GETUSERSENSORDATA(UserId)
2: for each data_point in user_sensor_data do
3:   closest_distance ← FINDMOSTCLOSESTSENSORDISTANCE(data_point.SensorId)
4:   if closest_distance > 5 then
5:     continue
6:   end if
7:   deviations ← 0
8:   closest_sensors ← FINDCLOSESTSENSORS(data_point.SensorId, 3)
9:   for each date_measure in data_point.measures do
10:    average ← GETAVERAGEMEASUREMENTS(closest_sensors, date_measure.date)
11:    if  $|date\_measure.value - average| / average > 0.20$  then
12:      deviations ← deviations + 1
13:    end if
14:  end for
15:  if deviations > 10 then
16:    UPDATEUSERSTATUS(UserId, "BANNED")
17:    UPDATEDATAVALIDITY(data_point.SensorId, False)
18:  end if
19: end for
```

1.5 Unit testing

1.5.1 checkSensor

give the values of the data you will use. Maybe you will need to create specific data for the tests. (directly as objects in the test method, or in dedicated csv files)

- If selectedSensor exists and ineffective, checkSensor(selectedSensor) : True
- If selectedSensor exists and effective, checkSensor(selectedSensor) : False
- If selectedSensor doesn't exist, checkSensor(selectedSensor) : False

1.5.2 distance

- If two sensors exist, Distance(sensor1,sensor2) : dist
- If sensor3 doesn't exist, Distance(sensor1,sensor3) : 0

1.5.3 calculateMeans

- If excludedSensor=[], closestSensor(sensor1,excludedSensor) : closestSensor
- If excludedSensor=[sensor2] with sensor2 the closest with sensor1 , closestSensor(sensor1,excludedSensor) : (second)closestSensor
- If excludedSensor = [allSensors], closestSensor(sensor1,excludedSensor) : 0

1.5.4 calculateWeightedMeans

- If selectedSensor doesn't exist, calculateWeightedMeans(selectedSensor, nearbySensors ,5) : 0
- If nearbySensors is empty, calculateWeightedMeans(selectedSensor, nearbySensors ,5) : 0
- If selectedSensor exist and nearbySensors is not empty, calculateWeightedMeans(selectedSensor, nearbySensors , 5.5) : 0
- If selectedSensor exist and nearbySensors is not empty but there are not datas on the period, calculateWeightedMeans(selectedSensor, nearbySensors , 5) : 0
- If selectedSensor exist and nearbySensors is not empty with datas on the period, calculateWeightedMeans(selectedSensor, nearbySensors , 5) : weightedMeanO3, weightedMeanSO2, weightedMeanNO2, weightedMeanPM10

1.5.5 checkCleaner

- If selectedCleaner exists and not faulty, effectiveCleaner(selectedCleaner): True
- If selectedCleaner exists and faulty, effectiveCleaner(selectedCleaner): False
- If selectedCleaner does not exist, effectiveCleaner(selectedCleaner): False

1.5.6 checkPrivateSensor

- Reliable User: classifyUnreliable(UserId) should return no changes if the user's data is within tolerance.
- Unreliable User: For a user with consistently deviant sensor data, the function should mark their status as BANNED.
- Mixed Reliability: A user with both reliable and unreliable sensors should still be banned if the unreliable data meets the criteria.
- Insufficient Proximity: Users without any nearby sensors (less than 5 km away) are marked reliable, but a warning is displayed.
- Date Mismatch: If no matching date data exists, display a warning about the inability to perform verification.