INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

INSA de Lyon

Computer Sciences Department

Software Engineering & UML

GL-UML Lab
AirWatcher

Autors:
Melisse Cochet
Saad ElGhissassi
Jassir Habba
Sekyu Lee
Simon Perret
(B3431 / B3423)

Professors:
Mrs. Laforest
Mr. Hasan

June 14, 2024

Contents

Introduction and presentation of the subject

In this lab series, we embark on the development of 'AirWatcher', an advanced software application commissioned by a government agency for environmental protection. The application is pivotal in analyzing data from a comprehensive network of air quality sensors, ensuring meticulous surveillance over an extensive territory. AirWatcher's capabilities extend beyond mere data collection; it is designed to identify and troubleshoot malfunctioning sensors, synthesize data into actionable insights, and deliver real-time air quality evaluations.

Central to AirWatcher's operation is the government agency, the system's primary user, charged with the critical task of regional air quality oversight. The agency leverages AirWatcher for continuous monitoring, sensor management, and detailed analytical reporting to inform policy-making and public engagement. Furthermore, the system plays a strategic role in assessing the influence of air purifiers provided by third-party companies, gauging their efficacy in improving environmental conditions.

Providers of these air purifiers, integral users of AirWatcher, utilize the software to track the performance of their products, manage operational data, and make informed decisions based on environmental feedback. These collaborations enrich the agency's understanding of intervention impacts and contribute to a cohesive environmental management strategy.

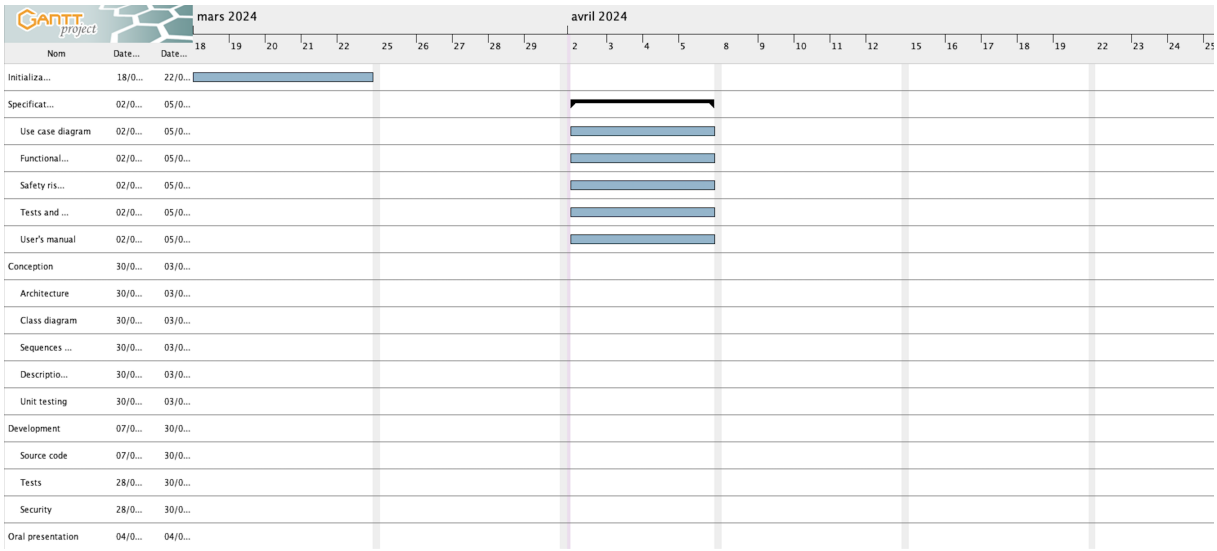
The application also empowers private individuals, volunteer contributors to the sensor network, to share their localized air quality data. Their engagement is facilitated by a points system that rewards contributions, providing a personal stake in environmental health. In return, they receive insights into local air quality and feedback on sensor reliability, fostering a community-driven approach to environmental monitoring.

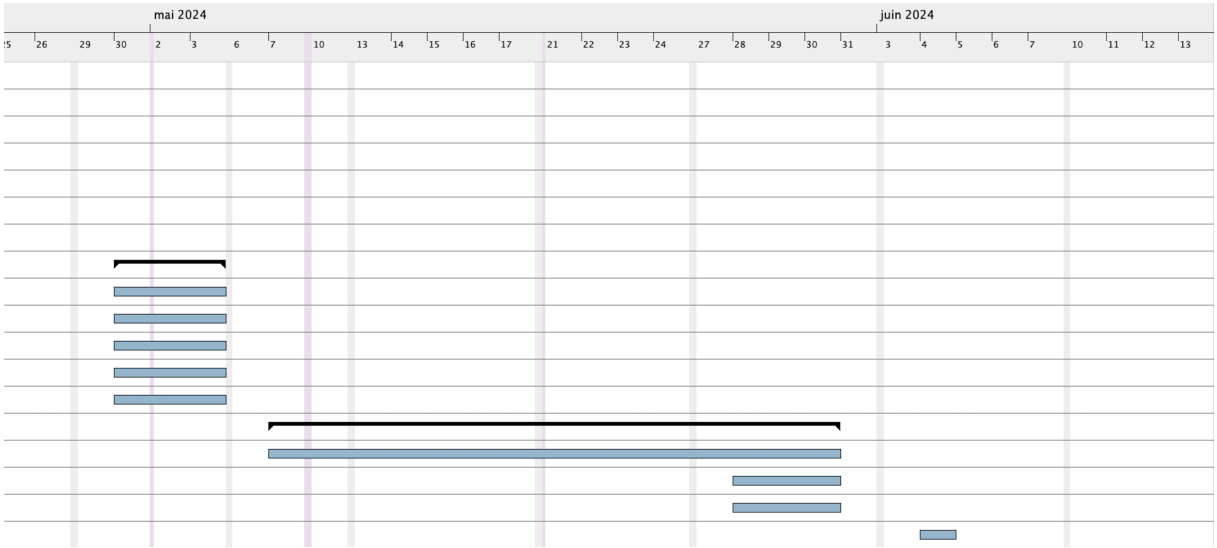
By incorporating stringent security measures, AirWatcher ensures data integrity is paramount, allowing for the isolation and correction of unreliable inputs. The application is underpinned by a commitment to performance, with algorithms optimized for swift and precise execution, thus enhancing the agency's responsiveness to environmental changes.

The ensuing documentation maps out the development journey, detailing our structured approach to fulfilling the agency's ambitious vision for a comprehensive air quality monitoring system.

1 Requirements specification

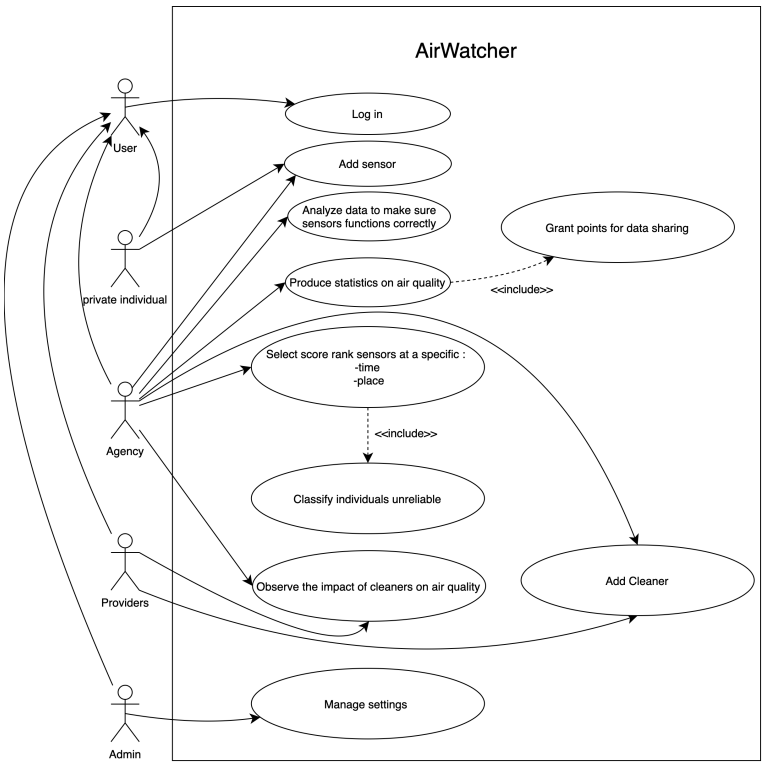
1.1 Gantt diagram





Name	begin	end
Initialization documents	18/03/2024	22/03/2024
Specification of requirements	02/04/2024	05/04/2024
Use case diagram	02/04/2024	05/04/2024
Functional and non-functional requirements	02/04/2024	05/04/2024
Safety risk analysis	02/04/2024	05/04/2024
Tests and validation	02/04/2024	05/04/2024
User's manual	02/04/2024	05/04/2024
Conception	30/04/2024	03/05/2024
Architecture	30/04/2024	03/05/2024
Class diagram	30/04/2024	03/05/2024
Sequences diagram	30/04/2024	03/05/2024
Description and pseudo-code	30/04/2024	03/05/2024
Unit testing	30/04/2024	03/05/2024
Development	07/05/2024	30/05/2024
Source code	07/05/2024	30/05/2024
Tests	28/05/2024	30/05/2024
Security	28/05/2024	30/05/2024
Oral presentation	04/06/2024	04/06/2024

1.2 Use case diagram



1.3 Functional/non-functional requirements and tests

Dataset Context

- Total Sensors: 100 sensors, mapped across France.
- Measurements: Daily readings of O3, SO2, NO2, PM10, taken simultaneously during the year 2019.
- User Types: Two private users, one honest, one dishonest.
- Air Cleaners: Two, with differing effectiveness.
- Task: Identify dishonest users, effective air cleaners, and impact zones through AirWatcher functionalities.

For the functional requirements, we will explain the tests directly linked with the functionality. A CSV file will be provided for testing the functionalities that are developed.

1.3.1 Functional requirements

Sensors Analysis (developed functionality)

Description Make sure sensors function correctly by analyzing data they provide at a specific period of time, and pinpoint reliable and unreliable sensors.

Process The system compares the coherence of the measurements provided by each sensor with the ones provided by the four other closest sensors to pinpoint malfunctioning sensors during a specific period of time. To do this, for each sensor, we compare all the measurements provided in the specified period to the weighted average of the measurements provided by the closest sensors at a same moment. The weighting coefficients when computing the weighted average are the inverse of the distances to the selected sensor. If the error of the measurement to the weighted average is greater than 20%, it is considered inconsistent. If the number of incorrect measurements provided by the interest sensor is greater than 20% of the total of collected measurements at the specified period, the sensor is to be considered malfunctioning, and therefore, unreliable.

Inputs
- Concentrations O3, SO2, NO2, PM10 of each sensor from measurements.csv

- Sensors with their coordinates from sensors.csv
- Start date and end date

Output List of reliable (well-functioning) and unreliable (malfunctioning) sensors at the specified period

Tests
- A sensor providing coherent concentrations is labeled reliable
- A sensor providing one incoherent concentration in its measurements is labeled unreliable
- A sensor providing entirely incorrect concentrations is labeled unreliable
- If there are no measurements for the four closest sensors during the period of time, the sensor is still considered reliable

Dataset The files contained in ValidationTests/SensorAnalysis contain measurements sets for Sensors 57, 66, 67, 68, 77. We set the same 4 measurements for each sensor at 2019-01-01 , 2019-02-01, 2019-03-01.

Let's consider Sensor67. Its nearby sensors are the four others. From 2019-01-01 to 2019-01-07 : all the sensors provide coherent data. They are declared reliable
From 2019-02-01 to 2019-02-07 : we changed Sensor 67 measurement of O3 to make it incoherent to the others sensors measurements of O3, so Sensor 67 presents more than 20% of its measurements deviating. It is considered unreliable.
Identically, if any other, or all measure is incoherent, the sensor will still be considered unreliable, so no need to set another data set for such a scenario.
From 2019-03-01 to 2019-03-07 : Sensor 67 is the only sensor which provides data : we have no measurements to compare it with and it is therefore considered reliable.
NB : As we have only set 5 sensors in the data set, deviation of Sensor67 could influence deviation of other sensors due to the method used to compute weighted means. We therefore neglect all other sensors while running these tests (use of analyseSensor service that only analyzes one sensor at a time).

Air Quality Statistics Production (developed functionality)

Description Generate air quality statistics for a specified time and location.

Process The quality of air is measured according to theATMO index determined after computing the average value of each measurement of concentration provided by all the sensors in the area in the specified period. The area is circular and defined by its center (latitude/longitude coordinates) and a radius (kilometers) + Data Sharing Points Allocation. To determine the ATMO index in the specified area at a specified period of time, the Application searches for all the local sensors comprised in the area. It aggregates the measurements provided by the local sensors by pollutant, selects the aggregated measures comprised in the specified time period, and therefore computes the average value measured for each pollutant in the area. Afterwards, it determines the ATMO index of each pollution, and chooses the highest ATMO index as the global ATMO index of the area.

Inputs
- Start date and end date
- Coordinates (latitude/longitude) of the center of the area
- Sensors data from measurements.csv
- Sensors coordinates from sensors.csv
- Optional radius (default 10 km)

Output ATMO index value, or -1 if no value could be determined

Tests
- The quality level is correct
- The lowest quality level is displayed when measurements are between quality levels
- If no sensors are found in the initial area, notify the user and return null
- If no measurements are available for the specified time, notify the user and return null

Dataset The files contained in ValidationTests/Statistics contain 5 sensors that measured pollutants concentrations from 2019-01-01 to 2019-12-31. Let's focus on the area centered around Sensor11, with a radius of 1000 km. The specified area therefore comprises sensors 11 , 12 , 13 , 21 , 22. The manually determined average concentration of pollutants is :

68.974 for NO2 –> NO2 ATMO Index = 3

65.255 for O3 –> O3 ATMO Index = 3

30.132 for PM10 –> PM10 ATMO Index = 5

88.936 for SO2 –> SO2 ATMO Index = 3

Therefore, for lat = 44 , long = 0 , radius = 1000, start date = 2019-01-01, end date = 2019-12-31, the global ATMO Index of the area must be 5.

If there are no sensors in the specified area : Let's center the measures around a single point where no sensor is placed : for lat = 70, long = 70, radius = 0, start date and end end date don't matter, the index must be null.

If there are no measurements in the specified time period : for lat = 44 , long = 0 , radius = 1000, start date = 2020-01-01, end date = 2020-12-31, the global ATMO Index of the area must also be null.

Sensor Similarity Scoring and Ranking

Description Score and rank sensors based on similarity to a selected sensor during a specified time.

Process The system computes the quality of air of all the sensors, and displays the 10 sensors or least with the same ATMO index to the selected sensor. The highest ranked sensors will be those located closest to the reference sensor + Data Sharing Points Allocation

Inputs
- Sensor selected
- Sensors data from measurements.csv
- Start date and end date

Outputs
- Selected sensor's ATMO index score
- Top 10 similar sensors

Tests
- Correct ranking with less than 10 sensors with the same ATMO index
- Correct ranking with more than 10 sensors with the same ATMO index
- If no sensor has the same ATMO index, the ranking is empty
- If no measurement is available for the specified time, notify the user and return an empty ranking

Air Cleaner Impact Observation

Description Assess the impact of air cleaners on air quality.

Process Select the three closest sensors, and calculate their ATMO index during and outside the period of use of the cleaner. If the indexes for all sensors are improved by at least one level, the cleaner is considered to have impacted air quality + Data Sharing Points Allocation

Inputs
- Cleaner selected with its period of use and localisation stored in cleaners.csv
- Sensor locations from sensors.csv
- Sensor measurements from measurements.csv

Outputs The furthest sensor within the impact zone, or null if the cleaner didn't improve the air quality

Tests
- Identify a cleaner with no measurable impact on air quality
- Identify a cleaner with an impact on air quality
- If the three closest sensors did not registered during the time of use of the cleaner, notify the user that no data can be provided on the level of improvement and return null

Individual Reliability Classification

Description Determine the reliability of data from private sensors.

Process The system compares the measures provided by each user's sensors to the average of the measures provided by the three closest sensors at the same date. If more than 10 measures don't meet a 20% tolerance on a certain concentration and if the closest sensor is under 5 km away, the user is BANNED and their measurements will be excluded from all further queries on the application. One unreliable sensor is enough to change the status of a private individual.

Inputs
- Private users and their sensor from users.csv
- Sensors data from measurements.csv
- Sensors with their coordinates from sensors.csv

Outputs
- Update all the values provided by an unreliable user as False in measurements.csv
- Update the status of a user as BANNED in users.csv
- Boolean true if an unreliable individual has been found, false otherwise

Tests
- Identify a reliable user
- Identify an unreliable user (all sensors wrong and only one sensor wrong)
- If there are no sensors within a 5km radius, the sensor won't be taken into account, and a warning is displayed saying that the verification can't be made (return false)
- If there are no correlations with the date, a warning is displayed saying that the verification can't be made (return false)

Data Sharing Points Allocation

Description Reward private individuals for sharing sensor data.

Process Allocates 10 points per instance to the contributing user. All functions that lead to extra points have this in their specifications as a side effect.

Inputs
- Sensor that has been used in a query
- User information from users.csv
- Number of times user sensor's data was used in the system's queries

Outputs Points awarded, total points per user

Tests
- Increment of the right amount of points for the sensor's user

1.3.2 Non-functional requirements

Performance Testing

Requirement : The application must process basic transactions such as analyzing a sensor's data provided during one year within 10 seconds under normal operational conditions.

Testing method: Implementing load testing to simulate a large number of users (around 100) accessing the application simultaneously, numerous measurements (around 150000) or many sensors (around 100). Ensuring that the application maintains its performance benchmarks, such as processing speed and response times. The testing covers various scenarios, including normal, peak, and exceptional load conditions to assess the application's behavior under stress. N.B : we have implemented this test : trying to analyze a sensor's yearly dataset with four measurements per day takes less than ten seconds. To achieve that point, we have decided to break the sensor's data analysis as soon as the 20% deviation rate for the sensor's measurements was reached. We could still improve this performance by deploying an upgrade version of the Application where the sensors measurements are stored in a hash table indexed by their date, so that searching a measures through its date is much faster.

Reliability Testing

Requirement : The application should demonstrate a high degree of reliability, with a defined uptime guarantee. It should prevent failures and minimize facility downtime.

Testing Method: Utilizing uptime tracking software to continuously monitor the availability of the application. Automated testing scripts can be employed to regularly send requests to the system to ensure it's operational. Additionally, reliability can be measured by the Mean Time Between Failures (MTBF) where the system is observed over time to identify the average interval between failures.

Security Testing

Requirement : AirWatcher must adhere to stringent security protocols to protect against unauthorized access and ensure data integrity. Users must authenticate through a secure login process, potentially integrating with existing agency credentials.

Testing Method: Conducting security audits (c.f : Analysis of Security Risks). Regularly update and patch the system in response to newly discovered vulnerabilities.

Usability Testing

Requirement : The system should be user-friendly, allowing users to perform functions with minimal training and effort. The console-based interface must be intuitive, requiring no more than 1 hour of training for new users to perform basic functions.

Testing Method: Conduct user experience (UX) testing sessions where participants perform typical tasks using the system. These sessions will be monitored to identify any usability issues.

Compliance Testing

Requirement : The application must comply with all relevant environmental and data protection legislation.

Testing Method: Conducting compliance audits against the latest standards and regulations. This involves reviewing the system's data handling, storage, and processing against legislative requirements and ensure that all personal data is managed according to privacy laws such as GDPR.

Portability Testing

Requirement : The application should be easily portable to different systems with minimal adjustments. AirWatcher should be adaptable for potential deployment on different server platforms without requiring significant changes.

Testing Method: Testing the application on different operating systems and hardware configurations to assess its adaptability.

Organizational Compliance Testing

Requirement : The development process should align with organizational standards and guidelines.

Testing Method: Implementing code reviews and documentation inspections to ensure compliance with prescribed practices.

1.4 Safety risk analysis

High
- Very costly loss of major tangible assets or resources
- Significant violation of, or harm or impediment to, an organization's mission, reputation, or interest

Medium
- Costly loss of tangible assets or resources
- Violation of, or harm or impediment to, an organization's mission, reputation, or interest

Low
- Loss of some tangible assets or resources
- A noticeable effect on an organization's mission, reputation, or interest

System : AirWatcher					
Asset	**Vulnerability**	**Attack**	**Risk**	**Impact level**	**Countermeasures**
Recovered air condition data	Inadequate password policies	Brute-force attack	Unauthorized account access	High	Enforce strong password policies and regular rotations
Data transfer to databases	Lack of secure transmission	Man-in-the-middle attack	Data interception and manipulation	High	Implement SSL/TLS encryption and digital certificates
Sensor and user data in the database	Insufficient access controls	Unauthorized access	Data leakage or unauthorized modification	High	Apply principle of least privilege, encrypt data at rest, and monitor for unusual access patterns
System operability	Single point of failure (server)	Power outage	System downtime and potential data loss	Medium	Deploy backup power solutions and redundant systems
Sensor data integrity	Malicious data tampering by users	Data corruption	Ingestion of false data	High	Implement anomaly detection systems and user behavior analysis
Unencrypted data	Unsecured data storage and transmission	System intrusion	Uncompensated data exposure	High	Utilize end-to-end encryption for data transmission and storage
Input data validation	Lack of input sanitization	Injection attacks	Storage and processing of erroneous data	High	Enforce strict input validation and sanitization processes

1.5 User manual

1.5.1 Introduction

Welcome to the AirWatcher console interface, where air quality data is analyzed, and sensor information is managed. This manual will guide you through using AirWatcher based on your role - whether you're part of the Agency, a Provider, or a Private Individual.

Getting started

1.5.2 Installation and Compilation

Before launching the AirWatcher application, ensure you have compiled the source code using the provided Makefile. Follow these steps :

- Open the Terminal. Navigate to your terminal or command prompt application on your system.
- Navigate to the Application Directory. Use the cd command to change the directory to the location where the AirWatcher source code is stored. For example:

```
cd path/to/AirWatcher
```

- Compile the Application. Run the make command to compile the application:

```
make
```

  This command will use the Makefile to compile the source code into an executable.

- Run the Application. Once the compilation is successful, start the AirWatcher application by running:

```
./AirWatcher
```

Note : for running the tests framework's interface, please consult the readme file provided with the source code package.

1.5.3 Role Declaration

Upon starting AirWatcher, you will first declare your user status. Please enter your role when prompted:

```
Enter your role (Agency/Provider/Private Individual): [YourRole]
```

1.5.4 Authentication

After declaring your role, you will be prompted to authenticate:

```
Username: [YourUsername]
Password: [YourPassword]
```

1.5.5 Role-Based Main Menu

Once authenticated, you will access a main menu tailored to your role with functionalities specific to your privileges and responsibilities.

Agency menu

```
AGENCY MENU
1. Analyze Sensors Data
2. View Air Quality Statistics
3. Ranking sensors compared to the one selected
4. Observe Cleaner Impact
5. Manage User Points and Classifications
```

```
1. Analyze Sensors Data
a. Analyze one sensor
b. Analyze all sensors
Enter your choice: a
Enter the start date (YYYY-MM-DD): 2019-01-01
Enter the end date (YYYY-MM-DD): 2019-01-31
...
Analyzing Sensor99
Analyzing Sensor97
Analyzing Sensor95
Analyzing Sensor55
...
Reliable Sensors : Sensor1 , Sensor7, Sensor15 ...
Unreliable Sensors : Sensor2 , Sensor9 ...
0:home
```

```
1. Analyze Sensors Data
a. Analyze one sensor
b. Analyze all sensors
Enter your choice: b
Enter the start date (YYYY-MM-DD): 2019-01-01
Enter the end date (YYYY-MM-DD): 2019-01-31
Enter the sensor ID to analyze: 36
Is Reliable: No
Deviation Count: 1178
Total Count: 1460
Deviation Rate: 80%
...
0:home
```

```
2. View Air Quality Statistics
Enter the longitude : 31.6
Enter the latitude : -0.8
Enter the radius (default is 10 by entering a negative value): 12
Enter the start date (YYYY-MM-DD): 2019-01-01
Enter the end date (YYYY-MM-DD): 2019-01-07
...
Air Quality Index: 8
0:home
```

```
3. Ranking sensors compared to the one selected
Enter the start date (YYYY-MM-DD): 2019-01-01
Enter the end date (YYYY-MM-DD): 2019-01-31
Enter the sensor ID to analyze: 9
...
The ATMO index for the selected sensor is 7.
The top 10 similar sensors is Sensor2, Sensor0 ...
0:home
```

```
4. Observe Cleaner Impact
Enter the cleaner ID to analyze: 1
...
The cleaner had a positive impact on the quality of air
0:home
```

```
5. Manage User Points and Classifications
1. View User Contributions
...
1)Max : 450    2)Lou : 390    3)Hugo : 389
2. Identify Unreliable Users
...
```

```
One unreliable user has been found : Karl
Type BAN for banning user : BAN
...
Karl has been banned and his measurements will no longer be used
0:home
```

Provider menu

```
PROVIDER MENU
1. Observe Cleaner Impact
2. Update Cleaner Information
3. View Air Quality Statistics
```

```
1. Observe Cleaner Impact
Enter the cleaner ID to analyze: 1
...
The cleaner had a positive impact on the quality of air
0:home
```

```
2. Update Cleaner Information
a. Register New Cleaner
...
Latitude of the cleaner : 45
Longitude of the cleaner : 6
Data Sharing Code of the cleaner : 479
...
The cleaner has been added, thank you !
b. Desactivate Cleaner
...
Select one cleaner :
a) Cleaner0    b)Cleaner1    c)Cleaner2
...
The cleaner has been removed, thank you !
0. Home
```

```
3. View Air Quality Statistics
Enter the longitude : 31.6
Enter the latitude : -0.8
Enter the radius (default is 10): 12
Enter the start date (YYYY-MM-DD): 2019-01-01
Enter the end date (YYYY-MM-DD): 2019-01-07
...
Air Quality Index : 4
0:home
```

Private individual menu

```
PRIVATE INDIVIDUAL MENU
1. View Personal Sensor Data
2. Check Points and Rewards Status
3. Add a sensor
```

```
1. View Personal Sensor Data
You installed 1 sensor : a)Sensor3
If you want to display the historical of the measurements, select one sensor
...
2020/08/09 Sensor3 O3 56 ; SO2 43 ; NO2 35 ; PM10 20
0:home
```

```
2. Check Points and Rewards Status
You have actually 120 points, you are ranked 7. You don't have any reward yet.
0:home
```

```
3. Add a sensor
Localisation of your sensor (lat,long) : (44, 5)
Data Sharing code : 343
...
Your sensor has been added, thank you !
0:home
```

Select the number corresponding to the action you want to take and press Enter.

1.5.6 Navigating the Interface

Returning to the Previous Menu
To return to the previous menu at any time, type:

```
back
```

Clearing the Screen
To clear the console screen of all previous commands and outputs, type:

```
clear
```

Viewing Help
For a list of available commands or help with a specific command, type:

```
help
```

Using Your Role-Specific Functions
After selecting an option from the main menu, simply follow the on-screen prompts to execute the desired function.

Exiting the Application
To exit AirWatcher safely, select the 'Log Out' option from your main menu.

Support and Assistance
For additional help or to report issues, contact the AirWatcher support team at support@airwatcher.gov.

2 Design

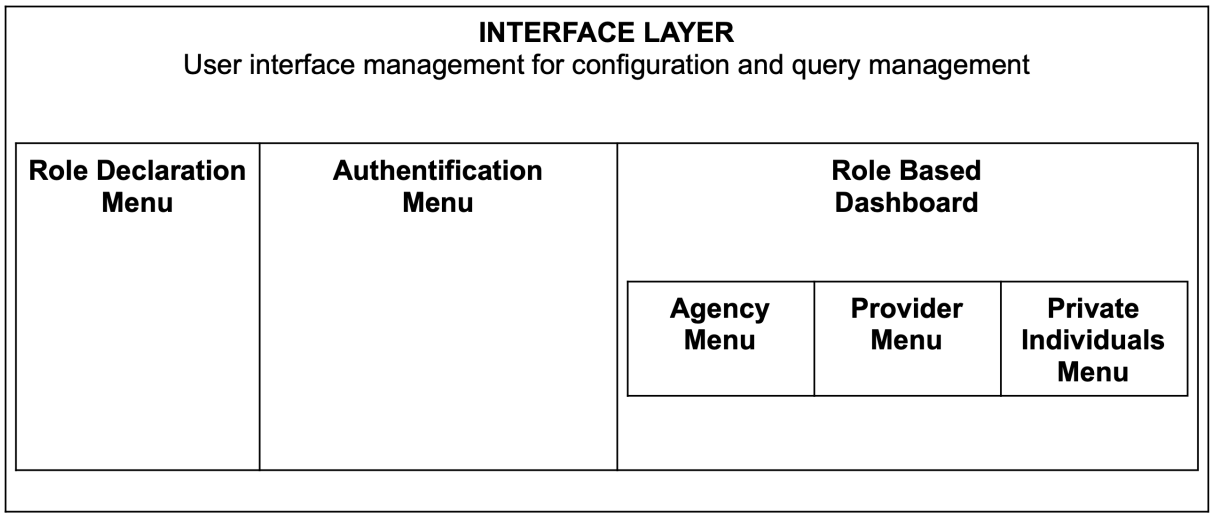
2.1 Architecture

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.

You will find in the table below a representation of the architecture of our application. Please note that it does only show the functions each layer and under-layer fulfills : for detailed architecture of the application, please report to the Class diagram.



SERVICES LAYER

Sensors Management - Get All Sensors - Get All Sensors that belong to private Individuals - Get Malfunctioning Sensors - Display Sensor's Data - Score Sensor - Rank Sensor - Get Top10 Similar Sensors - Check Sensor's Reliability	**Cleaners Management** - Get All Cleaners - Get Cleaner's Performances - Update Cleaner's Information - Get Cleaner's Impact on Air Quality	**Providers Management** - Authenticate Provider - Get Provider's Personal Cleaners
Private Individuals Management - Authenticate Private Individual - Get Private Individual's Sensors - Get Private Individual's Points Balance - Get Private Individual's Contribution to App - Update Private Individual's Reliability	**Administration Services** - Register Private Individual - Register Provider - Add Sensor - Add Cleaner	**Air Quality Generation** - Get Regional Air Quality Report - Get ATMO by circular area - Get AirQuality level by area - Get History of Air Quality Analysis
Points Allocation - Attribute Points to Private Individual		

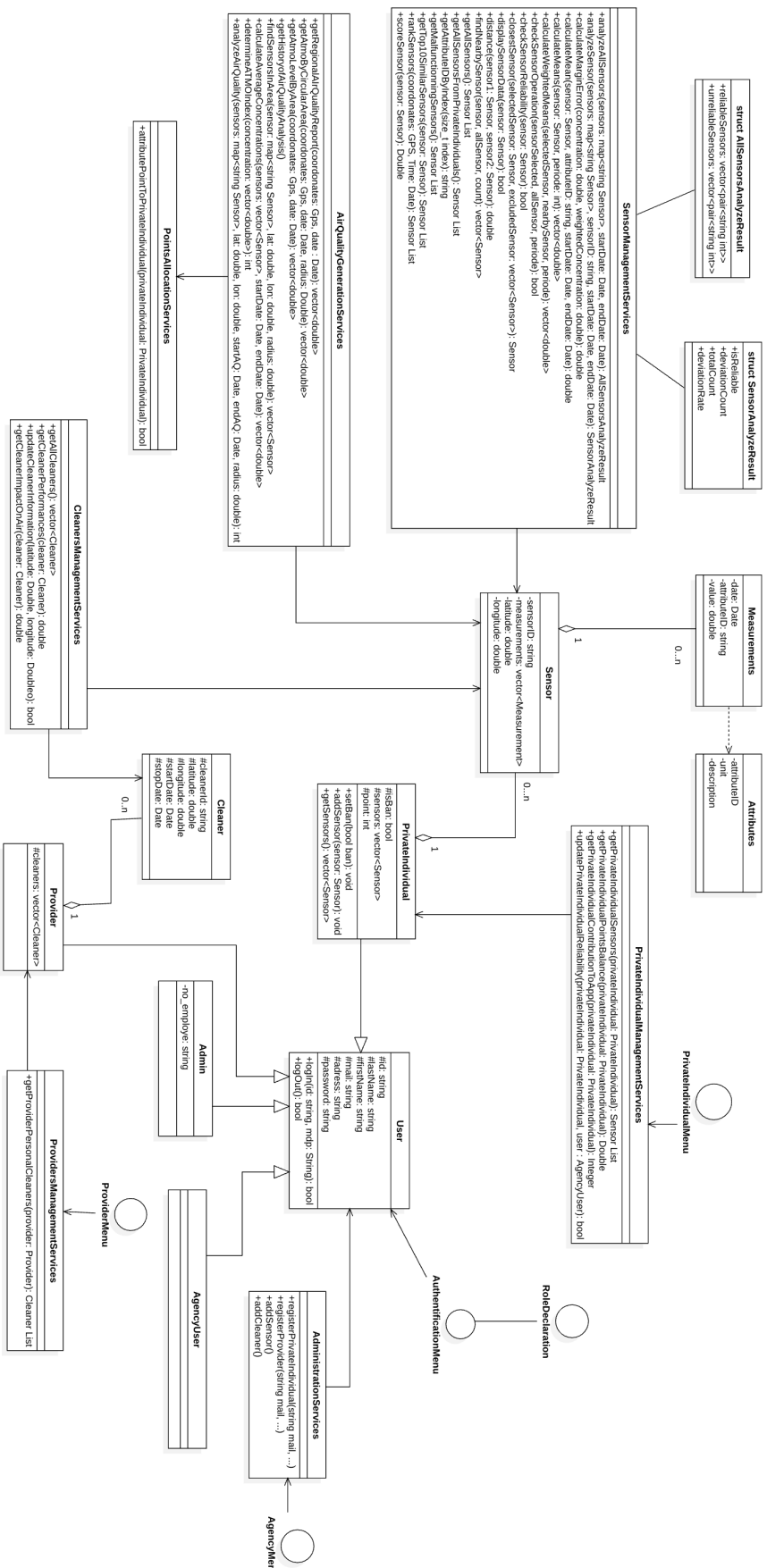
DATA LAYER

Users
Providers
Private Individual

Sensors	**Measurements**	**Attributes**

Cleaners

2.2 Class diagram



2.3 Development of three scenarios

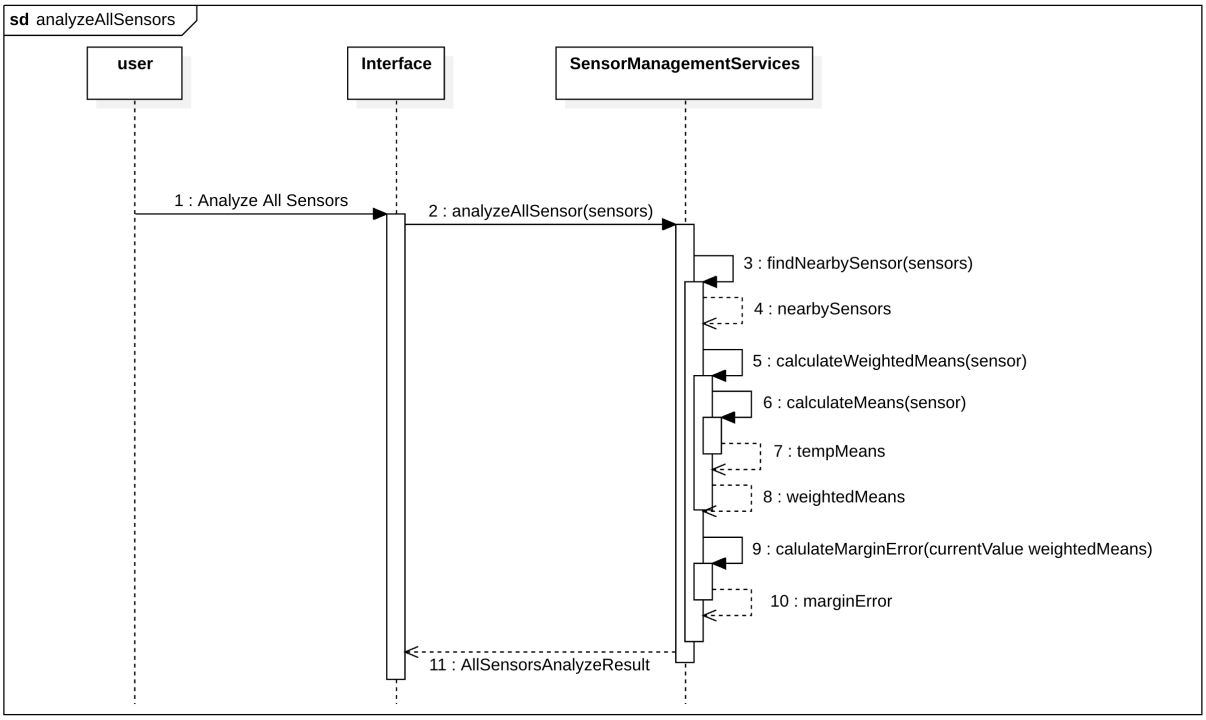
We have chosen to explain 3 main functionalities : one that verifies that the sensors work correctly, one that gives the ATMO index for a specific area and one that classify all private individuals that are considered unreliable. For each one, there are a sequence diagram, the pseudo-code and unit tests.

Please note that we have simplified the sequence diagrams so that they can fit across the width of the page, but each when a method called on a Service object requires reading data, it reaches it by hitting a Data object, and returns it to the service object. Therefore, the methods called on Service objects can imply several back and forth on Data objects.

2.3.1 Analyze data to make sure sensors function correctly

We will select measurements from the sensor provided over a period of time and calculate the means for each concentration (O3, SO2, NO2, PM10). Next, we will identify the four 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 20% for at least one concentration, the sensor will be deemed malfunctioning.

Sequence diagram



NB : findNearbySensors(sensors) actually hits a Data object Sensors and returns nearbySensors to the service. calculateWeightedMeans(sensor) hits a Data object Measurements and returns weightedMeans to the service.

Pseudo-code with unit tests

Algorithm 1 analyzeAllSensors

Require: $sensors, startDate, endDate$
Ensure: $results$

$results \leftarrow$ initialize()
for $sensorPair \in sensors$ **do**
  $sensor \leftarrow sensorPair.second$
  $maxDeviations \leftarrow 0.2 \times 4 \times days$
  $nearbySensors \leftarrow$ findNearbySensors($sensor, sensors, 4$)
  for $date \leftarrow startDate$ **to** $endDate$ **do**
    if $deviationCount > maxDeviations$ **then**
      break
    end if
    for $measurement \in sensor$.getMeasurements() **do**
      if $measurement$.getDate().to_time_t() $\geq$ $date$.to_time_t() **and** $measurement$.getDate().to_time_t() $<$ $date$.addDays(1).to_time_t() **then**
        $currentValue \leftarrow measurement$.getValue()
        $weightedMeans \leftarrow$ calculateWeightedMeans($sensor, nearbySensors, date, date$.addDays(1))
        $weightedMean \leftarrow 0$
        for $i \leftarrow 0$ **to** len($weightedMeans$) $- 1$ **do**
          if $measurement$.getAttributeID() == getAttributeIDByIndex(i) **then**
            $weightedMean \leftarrow weightedMeans[i]$
            break
          end if
        end for
        $marginError \leftarrow$ calculateMarginError($currentValue, weightedMean$)
        if $marginError > 0.2$ **then**
          $deviationCount \leftarrow deviationCount + 1$
        end if
        $totalCount \leftarrow totalCount + 1$
      end if
    end for
  end for
  $deviationRate \leftarrow$ double($deviationCount$)$/totalCount$
  if $deviationRate > 0.2$ **then**
    $results$.unreliableSensors.push_back($\{sensor$.getSensorID(), $deviationCount\}$)
  else
    $results$.reliableSensors.push_back($\{sensor$.getSensorID(), $deviationCount\}$)
  end if
end for
return $results$

Refer to the validation tests for this algorithm in the section 1.3.1.

Algorithm 2 Find Nearby Sensors

Require: $sensor, allSensors, count$
Ensure: $nearbySensors$
1: $nearbySensors \leftarrow []$
2: $distances \leftarrow []$
3: $isSensorIn \leftarrow$ false
4: **for** each $pair$ in $allSensors$ **do**
5: **if** $pair.key \neq sensor.getSensorID()$ **then**
6: $dist \leftarrow$ distance$(sensor, pair.value)$
7: distances.append$(dist, pair.value)$
8: **else**
9: $isSensorIn \leftarrow$ true
10: **end if**
11: **end for**
12: **if** $isSensorIn$ **then**
13: sort$(distances,$ by first component$)$
14: **for** $i \leftarrow 0$ to min$(count,$ length of $distances)$ **do**
15: $nearbySensors.append(distances[i].value)$
16: **end for**
17: **else**
18: Print("The sensor is not in the list of sensors")
19: **end if**
20: **return** $nearbySensors$

Unit tests :
We developeed four tests : a) Correct Test , b) Test with 4 sensor asked but there are only 3, c) Reference sensor doesn't exist , d) No other sensor than the reference in the csv

- The files contained in ValidationTests/FindNearbySensors initially contain 6 sensors : 57 , 66 , 67 , 68 , 77 and 100.
  Let's focus on Sensor 57 :
  For test a : Sensor100 is chosen such as it is too far from Sensor57 : the other four sensors are therefore the nearby ones.
  For test b : we've removed three sensors : only 3 are left in the dataset. These two latter (66 and 67) are the two closest to 57 (can't display four sensors)
  For test c : we want to display the closest sensors to a sensor no set in the dataset. The computation can't be done and nothing is displayed.
  For test d : the only sensor in the dataset is Sensor57, so the computation can't be done and nothing is displayed.

Algorithm 3 calculateMeans

Require: $sensor, startDate, endDate$
Ensure: $means$
1: $measurements \leftarrow sensor$.getMeasurementsForPeriod($startDate, endDate$)
2: $sumCounts \leftarrow$ new unordered_map
3: **for** $measurement \in measurements$ **do**
4: $\quad id \leftarrow measurement$.getAttributeID()
5: $\quad sumCounts[id].first \leftarrow sumCounts[id].first + measurement$.getValue()
6: $\quad sumCounts[id].second \leftarrow sumCounts[id].second + 1$
7: **end for**
8: $means \leftarrow$ new vector
9: $pollutants \leftarrow$ ["$O3$","$SO2$","$NO2$","$PM10$"]
10: **for** $pollutant \in pollutants$ **do**
11: $\quad$ **if** $sumCounts[pollutant].second > 0$ **then**
12: $\quad\quad means$.push_back($sumCounts[pollutant].first/sumCounts[pollutant].second$)
13: $\quad$ **else**
14: $\quad\quad means$.push_back(0)
15: $\quad$ **end if**
16: **end for**
17: **return** $means$

Unit tests :

- Correct mean for Sensor57 : we have set the measurements of Sensor57 to respectively 47 , 50 , 48 , 49 for each parameter; so these measurements must be displayed as the mean values of Sensor57.

  calculateMeans(Sensor57, 2018-12-31 12:00:00, 2019-02-02 12:00:00) : 47 50 48 49

- Mean asked for a sensor not in the csv

  calculateMeans(Sensor0, 2018-12-31 12:00:00, 2019-02-02 12:00:00) : 0 0 0 0

- Mean asked for a sensor with no measurements during the period

  calculateMeans(Sensor66, 2019-03-01 12:00:00, 2019-04-01 12:00:00) : 0 0 0 0

Algorithm 4 distance

Require: $sensor1, sensor2$
Ensure: $distance$
1: $distance \leftarrow \sqrt{(\text{pow}(latSensor1 - latSensor2, 2) + \text{pow}(lonSensor1 - lonSensor2, 2))}$
2: **return** $distance$

Unit tests :

- Correct Euclidean distance between two sensors in sensors1.csv

  distance(Sensor57,sensor66) : 0.80626 which is correct as Sensor57 (coordinates 46.0 ; 3.9) and Sensor66 (coordinates 46.4 ; 3.2)

- Distance with one sensor which is not in sensors1.csv : there is actually no need to run this test, because the Sensor class has a default constructor that provides the instantiated sensor default values for latitude and longitude , so it returns a distance as if it were in the csv.

Algorithm 5 calculateWeightedMeans

Require: $selectedSensor, nearbySensors, startDate, endDate$
Ensure: $weightedMeans$
1: $totalWeightedValues \leftarrow [0.0, 0.0, 0.0, 0.0]$
2: $totalWeights \leftarrow [0.0, 0.0, 0.0, 0.0]$
3: **for** $sensor \in nearbySensors$ **do**
4: $weight \leftarrow 1.0/\text{distance}(selectedSensor, sensor)$
5: $tempMeans \leftarrow \text{calculateMeans}(sensor, startDate, endDate)$
6: **for** $i \leftarrow 0$ **to** $\text{length}(tempMeans) - 1$ **do**
7: $totalWeightedValues[i] \leftarrow totalWeightedValues[i] + tempMeans[i] \times weight$
8: $totalWeights[i] \leftarrow totalWeights[i] + weight$
9: **end for**
10: **end for**
11: $weightedMeans \leftarrow [0.0, 0.0, 0.0, 0.0]$
12: **for** $i \leftarrow 0$ **to** $\text{length}(totalWeightedValues) - 1$ **do**
13: **if** $totalWeights[i] \neq 0$ **then**
14: $weightedMeans[i] \leftarrow totalWeightedValues[i]/totalWeights[i]$
15: **end if**
16: **end for**
17: **return** $weightedMeans$

Unit tests :
The files contained in ValidationTests/CalculateWeightedMeans initially contain 6 sensors : 57, 66, 67, 68, 77 and 100, giving measurements from 2018-12-31 to 2019-02-0
Let's focus on Sensor 57
The closest sensors are Sensor66 , 67 , 68 , 77
The distances to these sensors are 0.806226 , 0.4 , 0.806226 , 0.8 respectively
The means vectors are 47 50 48 40 , 47 50 48 41, 47 50 48 42, 47 50 48 42 respectively

- Correct weighted means with sensors in sensors1.csv and measurements1.csv

  nearbySensors = findNearbySensors(Sensor57, sensors, 4)

  the three first values of the weighted means vector are 47 50 48
  The fourth weighted mean is (40/0.806226 + 41/0.4 + 42/0.806226 + 42/0.8) / (1/0.806226 + 1/0.4 + 1/0.806226 + 1/0.8) = 41.2006

- The selected sensor doesn't exist in sensors1.csv : not runned for the same reasons as distance.

- The selected sensor doesn't have nearby sensors in sensors2.csv (only Sensor57) so the weighted means must return the means of Sensor57.

  nearbySensors = findNearbySensors(Sensor57, sensors, 4)

  calculateWeightedMeans(Sensor57, nearbySensors , 2018-12-31 12:00:00, 2019-04-02 12:00:00) : 47 50 48 41.25 (we have set all measurements of the three first parameters to the same values, and made only fourth parameter vary)

- There are no measurements during the period of time in measurements1.csv with sensors1.csv

  nearbySensors = findNearbySensors(Sensor57, sensors, 4)

  calculateWeightedMeans(Sensor57, nearbySensors , 2019-04-02 12:00:00, 2020-01-01 12:00:00) : 0 0 0 0

Algorithm 6 calculateMarginError

Require: $concentration, weightedConcentration$
Ensure: $marginError$
1: **if** $weightedConcentration = 0$ **then**
2: $\quad marginError \leftarrow 0$
3: **else**
4: $\quad marginError \leftarrow \frac{\text{fabs}(concentration - weightedConcentration)}{weightedConcentration}$
5: **end if**
6: **return** $marginError$

Unit tests :

- Correct margin error

  calculateMarginError(45,50) : 0.1

- Margin error with a division by 0

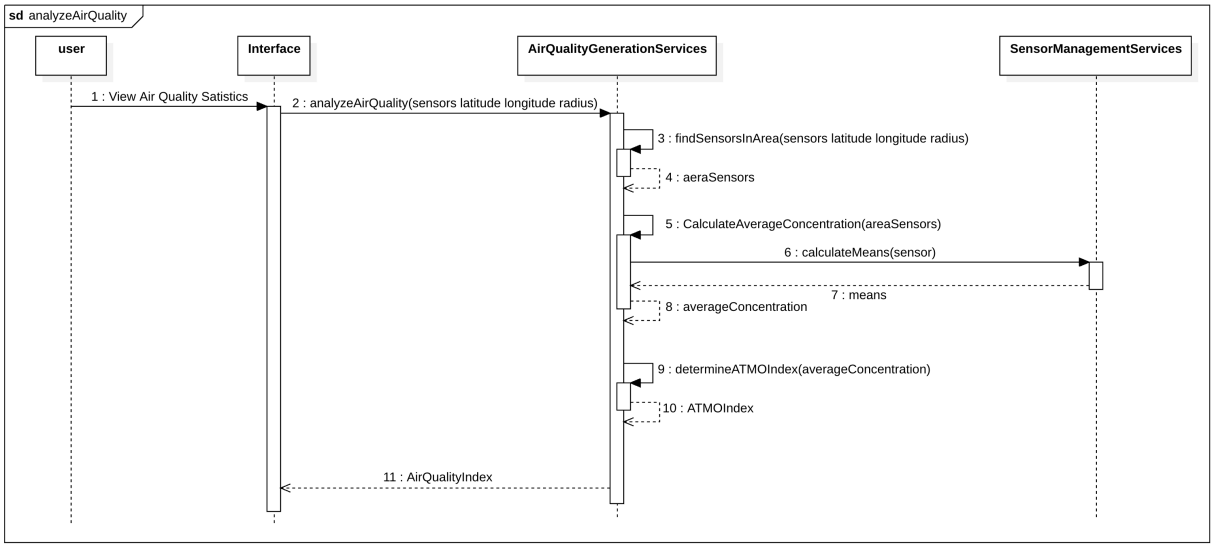
  calculateMarginError(45,0) : 0

2.3.2 Analyze air quality by ATMO index

Our software's air quality analysis functionality leverages a network of environmental sensors to assess the air quality within specific geographical areas. By calculating the ATMO index from average concentrations of key pollutants, this tool provides crucial insights into the environmental health, aiding policy makers and public health officials in their decision-making processes. The associate algorithms are the 5th, 6th and 7th.

NB : findSensorsInArea(sensors, latitude, longituden radius) actually hits a Data object Sensors and returns areaSensors to the service. calculateMeans(sensor) is a SensorManagementServices object called from an AirQualityManagementServices Service object. It hits a measurement Data object and returns means to the service.

Sequence diagram



Algorithm 7 analyzeAirQuality

Require: $sensors, latitude, longitude, startAQ, endAQ, radius$
Ensure: $ATMOIndex$
1: $ATMOIndex \leftarrow -1$
2: $areaSensors \leftarrow$ findSensorsInArea($sensors, latitude, longitude, radius$)
3: **if** not $areaSensors$.empty() **then**
4: $averageConcentrations \leftarrow$ calculateAverageConcentrations($areaSensors, startAQ, endAQ$)
5: $ATMOIndex \leftarrow$ determineATMOIndex($averageConcentrations$)
6: **end if**
7: **return** $ATMOIndex$

Pseudo-code with unit tests Refer to the validation tests for this algorithm in the section 1.3.1.

Algorithm 8 findSensorsInArea

Require: $sensors, lat, lon, radius$
Ensure: $localSensors$
1: $localSensors \leftarrow$ new vector
2: **for** $sensorPair \in sensors$ **do**
3: $sensor \leftarrow sensorPair.second$
4: $distance \leftarrow \sqrt{(sensor.\text{getLatitude}() - lat)^2 + (sensor.\text{getLongitude}() - lon)^2}$
5: **if** $distance \leq radius$ **then**
6: $localSensors$.push_back($sensor$)
7: **end if**
8: **end for**
9: **return** $localSensors$

Unit tests :

- No need for making unit tests as all the functions used in this procedure have already been tested for analyzeAllSensors

Algorithm 9 calculateAverageConcentration

Require: $sensors, startDate, endDate$
Ensure: $averages$
1: $averages, sums, counts \leftarrow [0.0, 0.0, 0.0, 0.0]$
2: **if** $sensors$.empty() **then**
3: **return** $averages$
4: **end if**
5: **for** $sensor \in sensors$ **do**
6: $means \leftarrow$ calculateMeans($sensor, startDate, endDate$)
7: **for** $i \leftarrow 0$ **to** length($means$) $- 1$ **do**
8: **if** $means[i] > 0$ **then**
9: $sums[i] \leftarrow sums[i] + means[i]$ & $counts[i] \leftarrow counts[i] + 1$
10: **end if**
11: **end for**
12: **end for**
13: **for** $i \leftarrow 0$ **to** length($averages$) $- 1$ **do**
14: **if** $counts[i] > 0$ **then**
15: $averages[i] \leftarrow sums[i]/counts[i]$
16: **end if**
17: **end for**
18: **return** $averages$

Unit tests :

- No need for making unit tests as all the functions used in this procedure (or similar functions) have already been tested for analyzeAllSensors

Algorithm 10 determineATMOIndex

Require: $concentrations$
Ensure: $highestIndex$
1: $atmoIndexes \leftarrow [0, 0, 0, 0]$
2: $pollutantRanges \leftarrow$ [array of ranges for each pollutant]
3: **for** $i \leftarrow 0$ **to** 3 **do**
4: $atmoIndexes[i] \leftarrow$ getATMOIndex($concentrations[i], pollutantRanges[i]$)
5: **end for**
6: $highestIndex \leftarrow \max(atmoIndexes)$
7: **return** $highestIndex$

Unit tests :

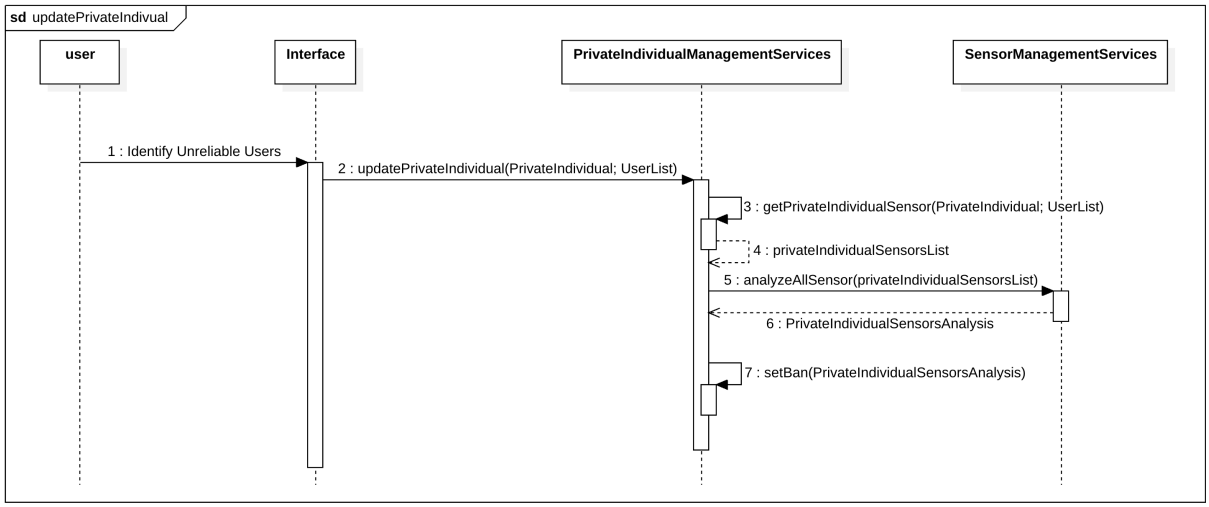
- No need for making unit tests as this function has already been tested in validation tests of analyzeAirQuality

2.3.3 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.

NB : getPrivateIndividualSensor(PrivateIndividual, Userlist) hits a privateIndividuals PrivateIndividual Data object and returns privateIndividualSensorsList to the service. analyseAllSensors(privateIndividualSensorsList) follows the sequence described for analyseAllsensors(...) and returns privateIndividualSensorsAnalysis to the service. setBan(privateIndividualSensorsAnalysis) hits a PrivateIndividual Data object, and returns a boolean to the service.

Sequence diagram



Algorithm 11 Update Private Individual Reliability

Require: $PrivateIndividual, UsersList, dateDebut, dateFin$
Ensure: Updates the reliability status of the private individual

```
PrivateIndividualSensorsList ← GetPrivateIndividualSensors(PrivateIndividual, UsersList)
map < string > PrivateIndividualSensorsList
for user ∈ UsersList do
    if user.first == PrivateIndividual then
        PrivateIndividualSensorsList.add(user.second)
    end if
end for
PrivateIndividualSensorsAnalysis ← analyseAllSensors(PrivateIndividualSensorsList, dateDebut, dateFin)
if PrivateIndividualSensorsAnalysis.unreliableSensors not empty then
    PrivateIndividual.setBan()
end if
```

Pseudo-code with unit tests Unit tests : We have not developed this functionality, but we could validate it with the initially given dataset

- A dataset containing an unreliable Sensor of a private individual must lead to the function banning this private individual (we've established that Sensor36 was unreliable with more than 80% deviation, so User1 of the initial dataset must be banned) but not the other one whose sensor provided reliable data (less than 7% deviation).

3 Software application development

We have chosen to develop two main functionalities of our AirWatcher Application : Analyzing Sensor's data to inquire their reliability, and Assessing Air Quality by enabling the computation of the ATMO index at a specific time and place.

The source code of our application allows to overview the architecture chosen for developing this software (the source code is split up into three packages : Interface, Services and Data, we have constructed a Makefile allowing ergonomic manipulations of the different source files, and we've included a tests framework in the ValidationTests package with its own makefile), as well as the algorithmic choices we have been dealing with.

We will try to enlist the main stakes of our development sessions below.

3.1 Performance evaluation of the implemented and executed algorithms

To evaluate the algorithm performance of our AirWatcher application, we'll focus on several key aspects: the algorithmic complexity, data structures used, and estimated execution time for the given dataset. This analysis will help you understand what's been well done and where improvements can be made.

We will only focus on the Sensors Analysis functionality, as the Determination of Air Quality functionality consumes much less ressources due to its practically constant complexity (always iterates over only four nearby sensors).

3.1.1 Algorithmic complexity

FindingNearbySensors

Function: *std::vector<Sensor> SensorManagementServices::findNearbySensors(const Sensor &sensor, const std::unordered_ map<std::string, Sensor> &allSensors, int count)*
Complexity O(n*log(n)) : The function iterates over all sensors (O(n)) , calculates distances and sorts them (O(n*log(n))) and selects the 4 closest sensors (O(4))

Function: *std: :vector<double> SensorManagementServices: :calculateMeans(const Sensor &sensor, const Date &startDate, const Date &endDate)*
Complexity O(m) : Iterating over all measurements of a sensor: O(m), where m is the number of measurements.

Function: *std::vector<double> SensorManagementServices:: calculateWeightedMeans (const Sensor &selectedSensor, const std:: vector<Sensor> &nearbySensors, const Date &startDate, const Date &endDate)*
Complexity O(k*m): Iterates over nearby sensors (O(k), where k is the number of nearby sensors) , Calculating means for each sensor (O(k*m)), calculates weighted mean (O(k))

Function:*SensorManagementServices: :analyzeSensor**
Complexity O(n*log(n) + d*m*k): Finds nearby sensors (O(n*log(n))), calculates weighted means for each day and each measure O(d x m x k)

3.1.2 Data Structures

Unordered map for sensors : Pros: Fast average-time complexity for insertions, deletions, and lookups (O(1)). **Cons:** Not ordered, so sorting and range queries are not straightforward.

Vector for Measurements Pros: Efficient access and iteration. **Cons:** No direct support for efficient range queries or modifications.

3.1.3 Estimated Execution Time

Given the dataset specifications:
• Number of Sensors: 100
• Number of Measurements per Sensor: $4 \times 365 = 1460$
• Total Measurements: 100 x 1460 = 146,000

Let's estimate the execution time for the main operations:

a. Finding Nearby Sensors
• Complexity: O(n*log(n)) = O(100 log 100) = *(700)
• Time: Negligible for modern processors.

b. Calculating Means
• Complexity: O(m) = O(1460)
• Time: Negligible for modern processors.

c. Analyzing All Sensors
• Complexity: O(n*log(n) + d x m x k) = 0(100 log 100 + 365 x 1460 x 4)
• Approximation: O (700 + 2, 132, 000)
• Time: Could be significant but manageable on modern hardware.

For information purposes : It takes approximately 10 seconds to run the whole analysis of a Sensor that's provided 146,000 measurements in the database on a MacBook Pro M1 , Apple Sillicon ARM64 , 8GB RAM , 8 cores Processor.

3.1.4 Critical Analysis

What's Been Well Done?
• Use of Unordered Map: Efficient sensor lookups.
• Separation of Concerns: Clear separation of different functionalities into services.
• Modular Design: Easy to extend and maintain.

How to Improve?
a. Optimize Finding Nearby Sensors
• Current Approach: O(n*log(n)))
• Improvement: Use spatial data structures like KD-Trees or R-Trees for O(log n) nearest neighbor queries.
b. Parallel Processing
• Current Approach: Sequential processing.
• Improvement: Use multi-threading or parallel processing (e.g., OpenMP) to handle multiple sensors or days concurrently.
Efficient Range Queries
• Current Approach: Linear scans.
• Improvement: Use segment trees or range trees for efficient range queries on measurements.
Memory Usage
• Current Approach: Vectors for measurements.
• Improvement: Consider memory pooling or compression techniques for large datasets. Use hash tables.