INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

Hugo SAYSANA, Thibaut CHANTREL (B3421)
Mathis NGUYEN, Martin NIZON (B3427)

Document de conception

Table des matières

Hugo SAYSANA, Thibaut CHANTREL (B3421)
Mathis NGUYEN, Martin NIZON (B3427)

Architecture et décomposition en modules

Pour implémenter notre application, nous avons choisi une architecture de type « **Layered Architecture** ».

Notre application peut être décomposée en plusieurs « couches » :

- le support système lié à la base de données
- les objets et les services métiers
- l'interface utilisateur

- Support système

Le composant « Système » s'occupera de gérer la base de données. Ce composant stockera des données provenant des fichiers .csv, c'est-à-dire les utilisateurs, les capteurs, les cleaners et les providers dans des listes. Il permettra alors de renvoyer les données nécessaires qui seront utilisées par les services.

- Objets métiers

Le composant « Utilisateur » contiendra les classes permettant d'instancier les différents types d'utilisateurs.

Le composant « Données » contiendra les classes permettant d'instancier tout ce qui concerne la qualité de l'air (capteur, mesure, cleaner, provider).

- Services métiers

Le composant « Services » contiendra les classes de services (service de statistique, service d'authentification, service d'information). Ce composant permettra de répondre aux différentes exigences fonctionnelles listées identifiées. L'accès à ces services suivant le type d'utilisateur sera contrôlé via l'affichage et l'interaction avec la console.

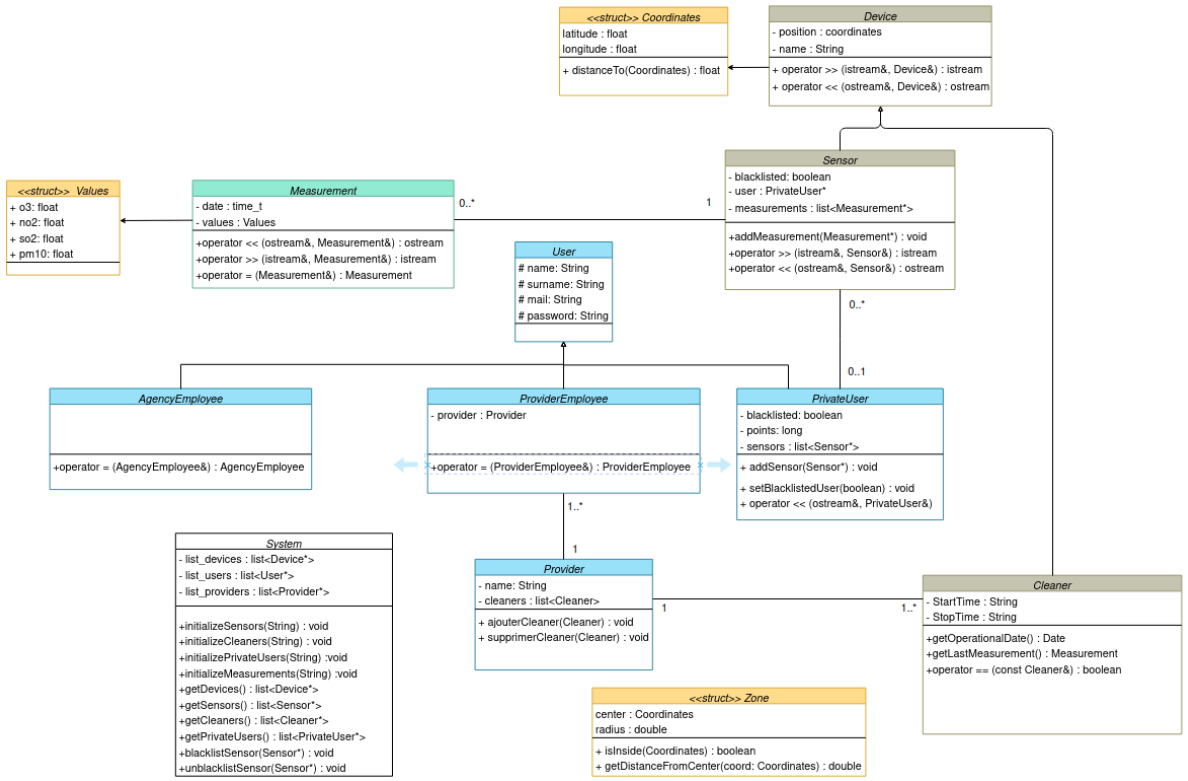
- Interface utilisateur

Le composant « Interface » sera notre classe Main qui réalisera des affichages sur la console.

Hugo SAYSANA, Thibaut CHANTREL (B3421)
Mathis NGUYEN, Martin NIZON (B3427)

Diagramme de classe

Le diagramme de classe ci-dessous montre les classes qui composent le système, leurs attributs, leurs méthodes et leurs relations, et détaille la conception et l'architecture de notre application. Il est disponible au lien suivant pour zoomer dessus : https://drive.google.com/drive/folders/1BgzI2ygNIlqAkp6qrfQT-RqUFOnYskB8?usp=sharing



StatisticsServices

- getAverageAirQuality(measurements: List<Measurement>)
- getFourClosestSensors(sensors: List<Sensor>, position: Coordinates)
+ getAverageAirQuality(sensors: List<Sensor>, position: Coordinates, date: Date)
+ getAverageAirQuality(sensors: List<Sensor>, zone: Zone, date: Date)

+ getAverageAirQuality(sensors: List<Sensor>, position: Coordinates, startDate: Date, endDate: Date)
+ getAverageAirQuality(sensors: List<Sensor>, zone: Zone, startDate: Date, endDate: Date)
+ getSimilarSensors(otherSensors: List<Sensor>, sensor: Sensor)

+ getCleanerImprovementIndex(cleaner : Cleaner, sensor : List<Sensor>)

AuthenticationServices

+ Authentify(users: List<User>, mail: String, password: String)

InformationServices

+ getFunctionnalSensors(sensors: List<Sensors>)
+ getBlacklistedUsers(all_users: List<Users>)
+ getBlacklistedSensors(sensors: List<Sensors>)
+ getPoints(user: PrivateUser)
+ isBlacklisted(sensor: Sensor)
+ ownBlacklistedSensor(user: PrivateUser)
+ getPoints(users: List<PrivateUser>)

Description des 4 fonctions principales

Les fonctions principales que nous avons choisi sont des fonctions permettant de calculer la qualité de l'air.
Chacunes de ses fonctions appelera la fonction : - getAverageAirQuality(List <Measurements>), pour laquelle nous n'avons pas réalisé de diagramme de séquence mais dont le pseudo-code est donné ci-dessous :

```
Fonction getAverageAirQuality(Measurement measurements[])
      Entrée : la liste des mesures measurements[] dont on calcule la moyenne
      Postcondition : retourne la moyenne de la qualité de l'air de la liste des
mesures
      Déclaration : la moyenne de la qualité de l'air avgX et le nombre nX de
mesures prises en compte dans le calcul de la moyenne, avec X l'élément mesuré.
value la valeur de la moyenne qui sera retournée
      avgO3 ← 0
      avgNO2 ← 0
      avgSO2 ← 0
      avgPM10 ← 0
      nO3 ← 0
      nNO2 ← 0
      nSO2 ← 0
      nPM10 ← 0
      value ← 0

      pour toute measurement m ∈ measurements[] faire
            si m.getO3() != 0 faire
                  avgO3 ← avgO3 + m.getO3()
                  nO3 ← nO3 + 1
            si m.getNO2() != 0 faire
                  avgNO2 ← avgNO2 + m.getNO2()
                  nNO2 ← nNO2 + 1
            si m.getSO2() != 0 faire
                  avgSO2 ← avgSO2 + m.getSO2()
                  nSO2 ← nSO2 + 1
            si m.getPM10() != 0 faire
                  avgPM10 ← avgPM10 + m.getPM10()
                  nPM10 ← nPM10 + 1

      value.O3 ← avg03 / n03
      value.NO2 ← avgNO2 / nNO2
      value.SO2 ← avgSO2 / nSO2
      value.PM10 ← avgPM10 / nPM10
      retourner value
```

Nous utilisons également la méthode `getThreeNearestSensors()` dans le cas où aucune mesure n'a été trouvée, faute de manque de capteurs à la position donnée. Nous aurons

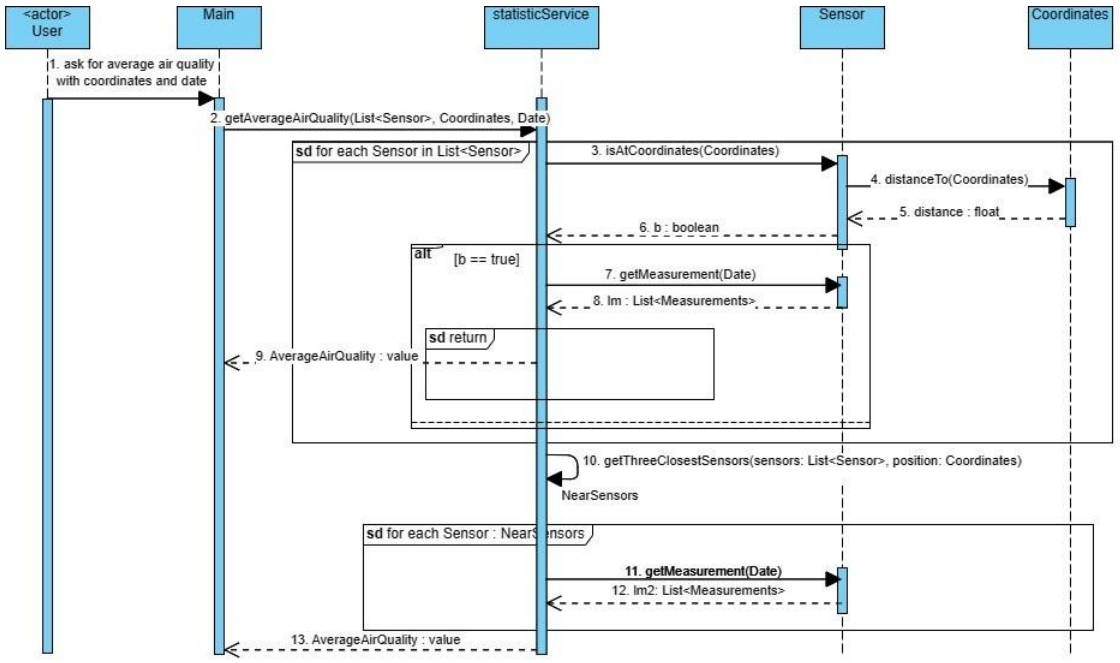
besoin de la fonction de tri trier(liste<>,critere,mode de tri), dont on ne détaillera pas le pseudo code ici. Voici le pseudo-code de la méthode `getThreeNearestSensors()`:

```
Fonction getThreeNearestSensors(Sensor sensors[], Coordinates coordinates)
        Entrée : la liste des capteurs qui ne sont pas dans la liste noire sensors[],
des coordonnées coordinates
        Postcondition : retourne une liste contenant les trois capteurs les plus
proches
        Déclaration : la liste de capteurs closeSensor[] qui sera retournée et la
liste de capteurs sensorTri[] qui contiendra les capteurs triés dans l'ordre
croissant des distances par rapport à coordinates

        sensorSort[] ← []
        closeSensors[] ← []

        sensorSort ← trier(sensors,sensors.getDistanceTo(coordinates),ascending)

        pour i allant de 0 à 2 faire
                closeSensors[i] ← sensorSort[i]
        retourner closeSensors[]
```

getAverageAirQuality (List<Sensor*>, Coordinates, Date)

Cette méthode de la classe statisticService sera utilisée par les utilisateurs pour rendre compte de l'état de l'air à des coordonnées précises à une date précise. Cette méthode retourne un objet de la classe *Values* qui représente la moyenne de l'état de l'air à la position et à la date donnés. Cette valeur est soit obtenue directement à partir des mesures du capteur situé à la position précisée, s'il en existe un, sinon elle est obtenue en effectuant une moyenne pondérée des mesures des 4 capteurs les plus proches.

Diagramme de Séquence



Dans ce diagramme de séquence, getThreeClosestSensors(sensors: List<Sensor>, position: Coordinates) est utilisée dans le cas où aucun capteur n'est présent à l'emplacement exact des coordonnées rentrées en paramètre de la méthode. Cette méthode retourne les 3 capteurs les plus proches de cette zone pour pouvoir par la suite réaliser une moyenne pondérée sur les valeurs des mesures de ces capteurs.

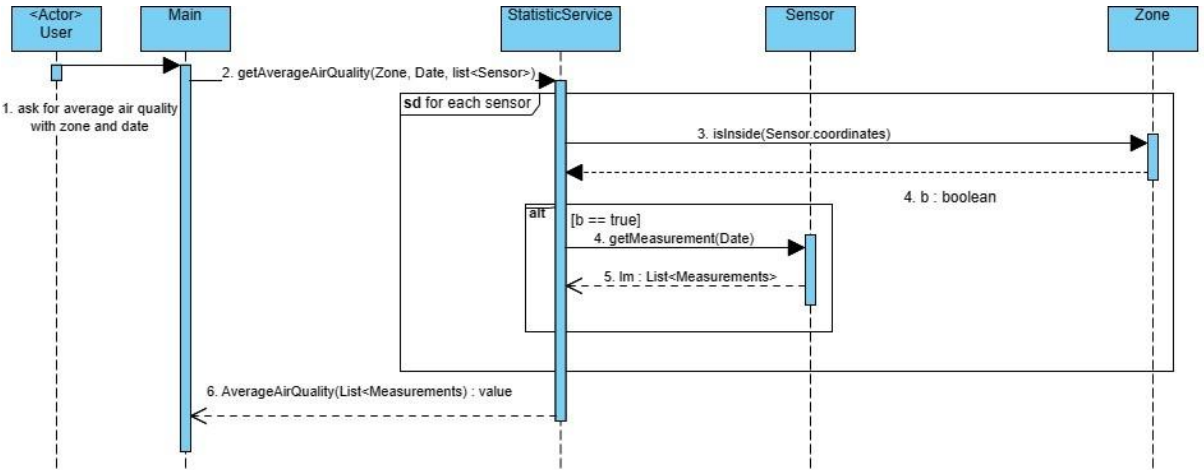
Pseudo-code

```
Fonction getAverageAirQuality(sensor[], coordinates, date)
	Entrée : la liste des capteurs qui ne sont pas dans la liste noire sensor[], des coordonnées coordinates, un instant date
	Postcondition : retourne la qualité de l'air aux coordonnées données et à la date donnée, qu'il y ait un capteur à cet endroit ou non
	Déclaration : le tableau des mesures qu'on passera en paramètre de la fonction pour calculer la moyenne de la qualité de l'air
	measures[] ← null
	pour tout capteur c ∈ sensor[] faire
		si c.getPosition() == coordinates alors
			pour tout measurement m ∈ c.getMeasurements() faire
				si m.getDate() == date
					measures.add(m)
					m.addPoints()

	// Si on n'a pas trouvé de sensor aux coordonnées passées en paramètres
	si measures[] est vide faire
		pour tout capteur c ∈ getThreeNearestSensors(sensor[],coordinates) faire
			pour tout measurement m ∈ c.getMeasurements() faire
				si m.getDate() == date
					measures.add(m)
					m.getUser().addPoints() // s'il existe un user pour m

	retourner getAverageAirQuality(measures[])
```

getAverageAirQuality (List<Sensor*>, Zone, Date)

Cette méthode de la classe statisticService sera utilisée par les utilisateurs pour rendre compte de la qualité de l'air dans une zone précise. La zone est une zone circulaire représentée par des coordonnées (centre) et un rayon saisis par l'utilisateur.
Cette méthode retourne un objet de la classe *Values* qui représente la moyenne de l'état de l'air dans la zone et à la date données.

Diagramme de Séquence



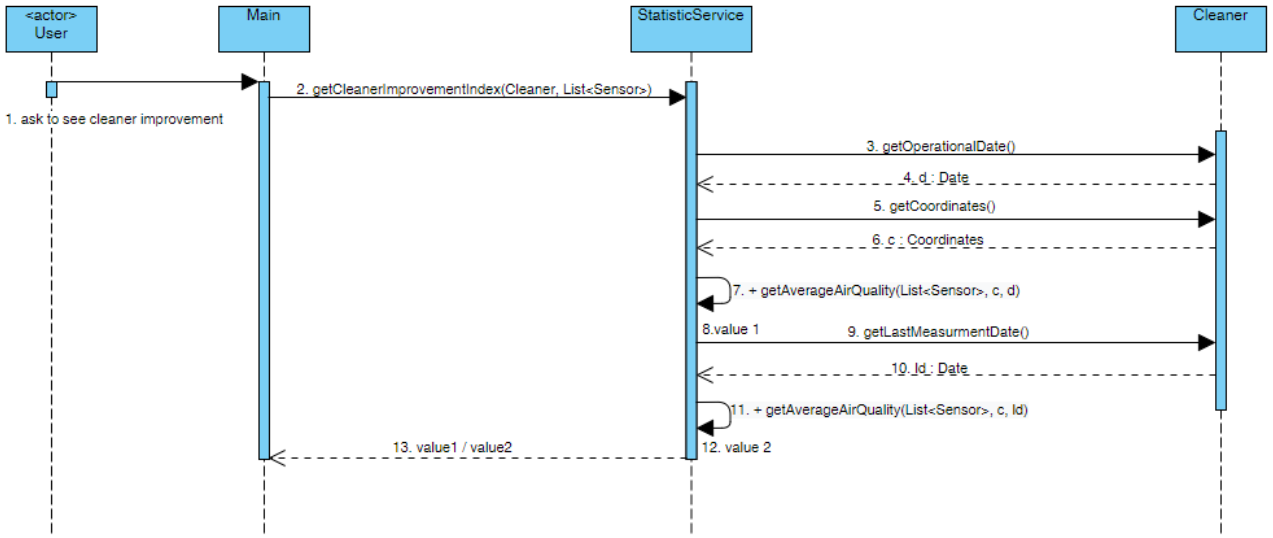
Pseudo-code

```
Fonction getAverageAirQuality(sensor[], zone, date)
      Entrée : la liste des capteurs qui ne sont pas dans la liste noire sensor[],
une zone zone et un instant date
      Postcondition : retourne la moyenne dans la zone donnée, à la date donnée
      Déclaration : le tableau des mesures qu'on passera en paramètre de la
fonction pour calculer la moyenne de la qualité de l'air
      measures[] ← []
      pour tout capteur c ∈ sensor[] faire
            si c.getPosition() ∈ zone
                  pour tout measurement m ∈ c.getMeasurement() faire
                        si m.getDate() = date
                              measures.add(m)
                              m.addPoints()
      si measures[] est vide faire
            afficher "Pas de mesures dans la zone définie"
            retourner []
      retourner getAverageAirQuality(measures[])
```

getCleanerImprovementIndex (List<Sensor*>, Cleaner)

Cette méthode de la classe statisticService sera utilisée par les utilisateurs pour obtenir un indice d'amélioration de l'air suite à la pose d'un cleaner. Cette méthode retourne le rapport entre la moyenne de l'air à la position du cleaner le jour de la date de son démarrage et la moyenne de l'air à la position du cleaner le jour de la date de sa dernière mesure.

Diagramme de Séquence



Pseudo-code

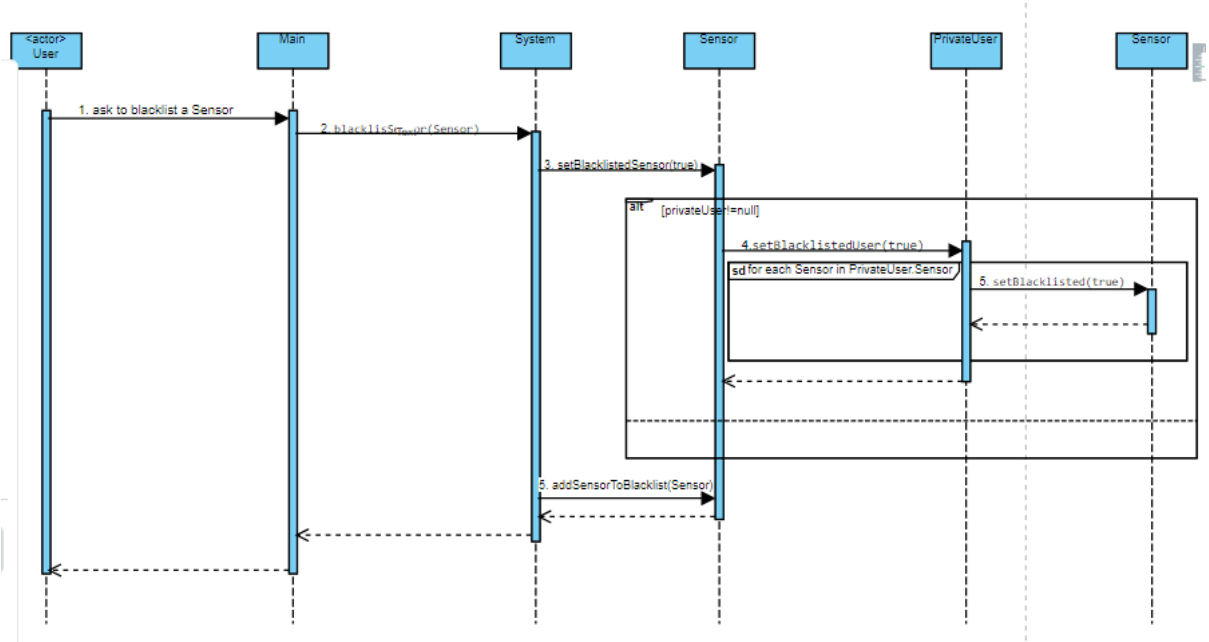
```
Fonction getCleanerImprovementIndex(sensor[], cleaner)
	Entrée : la liste des capteurs qui ne font pas partie de la liste noire et un cleaner
	Postcondition : retourne les bénéfices d'un cleaner à la position où il est installé
	Déclaration : la position pos du cleaner en entrée, la date du démarrage du cleaner dateDebut, la date d'arrêt du cleaner dateFin et le rapport rapp entre dateFin et dateDebut
	pos ← cleaner.getPostion()
	dateDebut ← cleaner.getDateStart()
	dateFin ← cleaner.getDateEnd()

	retourner getAverageAirQuality(sensor[],pos,dateDebut) / getAverageAirQuality(sensor[],pos,dateFin)
```

blacklistSensor(Sensor*)

Cette méthode, de la classe System permet de mettre dans la blacklist les capteurs considérés comme étant défectueux. Un capteur peut être blcklisté par un utilisateur, a priori uniquement un membre de l'agence gouvernementale. Si ce capteur appartient à un particulier, tous les capteurs de ce particulier sont blacklistés.
Remarque : Seul les capteurs ayant été rentré en paramètre se trouveront dans la list<Sensor*> blacklistedSensors énumérant les capteurs blacklisté. Les autres (ceux appartenant à un utilisateur blacklisté) auront simplement true comme valeur de leur attribut

blacklisté. Cette nuance est importante, elle permet de différencier les capteurs réellement défaillants de ceux blacklisté car ils appartiennent à un particulier.

Diagramme de Séquence



Remarque: addSensorToBlacklist(Sensor) n'est pas implémentée dans le code. En effet, dans le code, il s'agit simplement d'un push dans la liste mais le diagramme cette méthode fictive permet d'expliciter ce push.

Pseudo-code

```
Fonction blacklisSensor(sensor*)
	Entrée : pointeur du capteur à blacklister
	Postcondition : Le capteur est blacklisté ainsi que son propriétaire et tous ses autres capteurs si le capteur appartient à un particulier
	sensor->setBlacklistedSensor(true)
	Le capteur est ajouté à la liste des capteur défaillant (blacklisté)

	retourner void

Fonction setBlacklistedSensor(boolean)
	Entrée : boolean qui indique si le capteur doit bel et bien être blacklisté
	Postcondition : Le capteur est blacklisté ainsi que son propriétaire et tous ses autres capteurs si le capteur appartient à un particulier
	this.blacklisted<-boolean
	Si le capteur possède un propriètaire u alors :
		u->setBlacklistedUser(boolean)
	retourner void

Fonction setBlacklistedUser(boolean)
```

Entrée : boolean qui indique si le propriétaire(particulier) doit bel et bien être blacklisté
Postcondition : boolean==true ⇒ propriétaire et tous ses capteurs sont blacklistés mais non ajouté à la liste des capteurs défaillants
this.blacklisted<-boolean
Pour chaque Sensor s du particulier
s.setBlacklisted(boolean)
retourner void

Remarque : Cette fonction fonctionne de paire avec la méthode unblacklistSensor(Sensor*) qui quant à elle, retire le capteur de la liste des capteurs défaillant dans tous les cas. Elle met également son attribut blacklisted à false dans le cas où il ne possède pas de propriétaire ou dans le cas où son propriétaire ne possède plus aucun capteur présent dans la liste des capteurs défaillants. Si son propriétaire ne possède plus aucun capteur défaillant, alors ses capteurs sont de nouveaux utilisables et l'attribut blacklisted de tous ses capteurs ainsi que le sien est mis à false.

Tests unitaires

Tests de getAverageQuality(measurements[])

getAverageAirQuality(measurements[])
Résultat attendu : Values (µg/m3) - O3: 50.375; NO2: 73.25; SO2: 40.375; PM10: 47.625
Test :

```
Measurement m1 = new Measurement("04/05/2023", 50.25, 74.5, 41.5, 44.75);
Measurement m2 = new Measurement("04/05/2023", 50.5, 72, 39.25, 50.5);
List<Measurement> measurements = new List<Measurement>();
measurements.add(m1);
measurements.add(m2);
getAverageAirQuality(measurements).display();
```

getAverageAirQuality(measurements[]) (avec valeur nulle)
Résultat attendu : Values (µg/m3) - O3: 50.375; NO2: 36; SO2: 40.375; PM10: 47.625
Test :

```
Measurement m1 = new Measurement("04/05/2023", 50.25, 0, 41.5, 44.75);
Measurement m2 = new Measurement("04/05/2023", 50.5, 72, 39.25, 50.5);
List<Measurement> measurements = new List<Measurement>();
measurements.add(m1);
measurements.add(m2);
getAverageAirQuality(measurements).display();
```

getAverageAirQuality(measurements[]) (avec valeur négative)
Résultat attendu : Values (µg/m3) - O3: 50.375; NO2: 72.625; SO2: 40.375; PM10: 44,75

Test :

```
Measurement m1 = new Measurement("04/05/2023", 50.25, 73.25, 41.5, 44.75);
Measurement m2 = new Measurement("04/05/2023", 50.5, 72, 39.25, -50.5);
List<Measurement> measurements = new List<Measurement>();
measurements.add(m1);
measurements.add(m2);
getAverageAirQuality(measurements).display();
```

Tests de getAverageQuality(Zone, date)

getAverageAirQuality(sensors[], Zone, date)
Résultat attendu : getAverageAirQuality(Sensor->getMeasurements(date)), Sensor ∈ zone de rayon 1

Test :

```
Zone zone;
zone.coordinates = (45, 1);
zone.radius = 1;
getAverageAirQuality(sensors[], Zone, "03/01/2019").display();
```

getAverageAirQuality(sensors[], Zone, date) (avec zone négative)
Résultat attendu : getAverageAirQuality(Sensor->getMeasurements(date)), Sensor ∈ zone de rayon 1

Test :

```
Zone zone;
zone.coordinates = (45, 1);
zone.radius = -1;
getAverageAirQuality(sensors[], Zone, "04/05/2023").display();
```

Test des méthodes de System

Tests de lecture des fichiers csv et des getters

```
System s("sensors_test.csv", "cleaners_test.csv", "users_test.csv",
"measurements_test.csv");
```

getSensors()
Résultat attendu : Tous les sensors

Test :

```
getSensors().display();
```

getDevices()
Résultat attendu : Tous les devices

Test :

```
getDevices().display();
```

getPrivateUsers()
Résultat attendu : Tous les private users

Test :

```
getPrivateUsers().display();
```

Test de la fonctionnalité de blacklistage

blackListSensor(Sensor) (avec Sensor ∈ PrivateUser)
Résultat attendu : Tous les sensors sauf le Sensor1 et les sensors du PrivateUser1
Test :

```
PrivateUser1.addSensor(Sensor1);
blackListSensor(Sensor1);
getFunctionalSensors().display();
```

unBlackListSensor(Sensor) (avec Sensor ∈ PrivateUser)
Résultat attendu : Tous les sensors
Test :

```
PrivateUser1.addSensor(Sensor1);
blackListSensor(Sensor1);
unBlackListSensor(Sensor1);
getFunctionalSensors().display();
```