**Design Document**

# Diagramme de classe

Une image contenant texte, diagramme, Parallèle, Plan

Description générée automatiquement

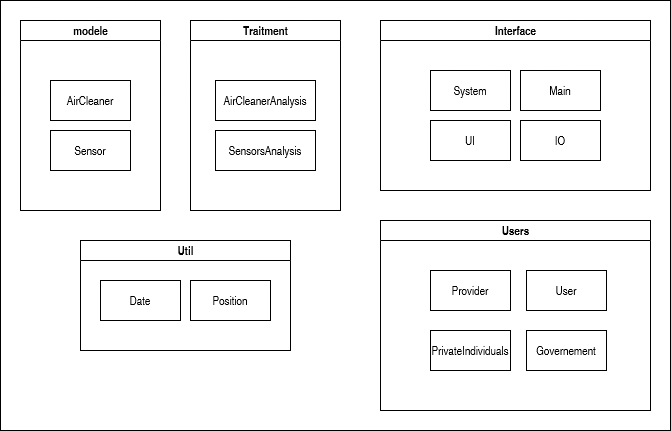
*Figure 1 : diagramme de classe de AirWatcher*

La classe **Main** contient uniquement le menu.

Il commencera par créer un **System**. Le **System** créé la **Fabrique** qui construit tous les éléments à partir des fichiers (**Sensor**, **AirWatcher** et **User**). Ils sont ensuite récupérés par le **System**.

Les **SensorAnalysis** et **CleanerAnalysis** sont créés lors de l’appel d’une méthode et détruits à la fin de la méthode. Cela permettrait de faire du multithreading lors du lancement d’une fonctionnalité.

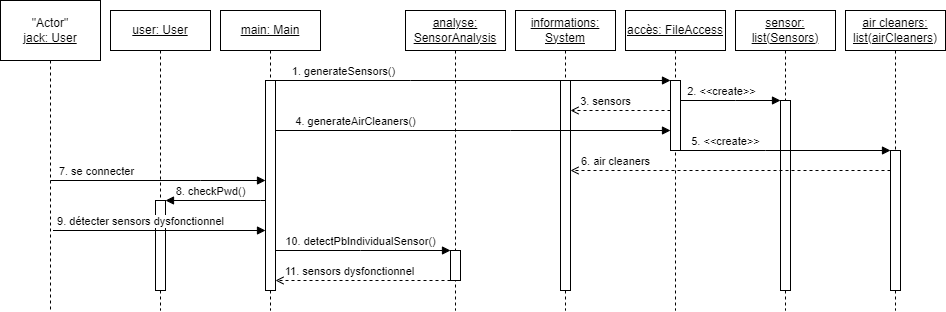
# Décomposition modulaire



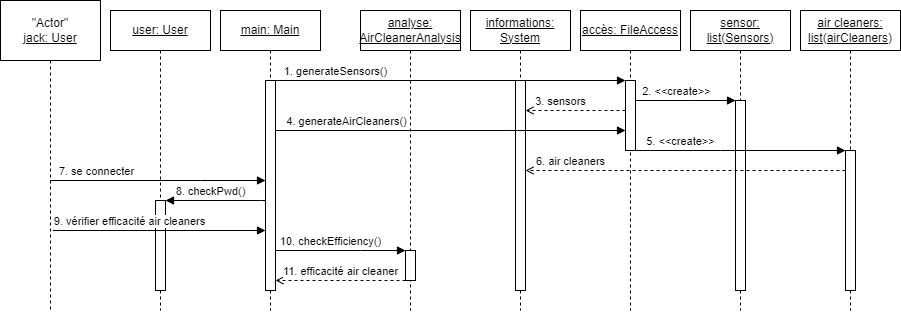
*Figure 2 : Diagramme de décomposition des classes*

# Diagrammes de séquences

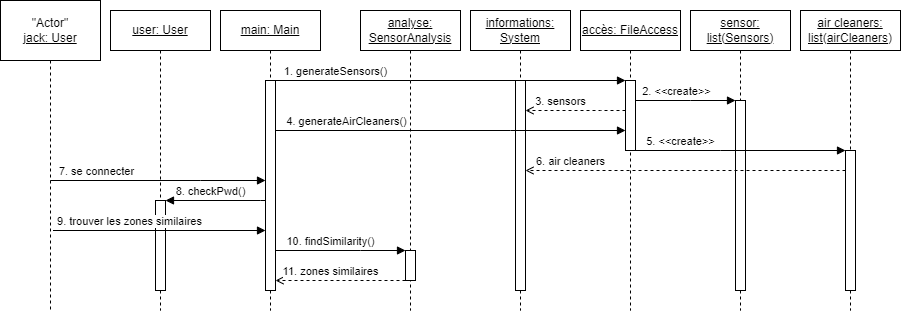
Dans le premier cas, on considère un utilisateur gouvernement souhaitant vérifier quels capteurs sont dysfonctionnels.



Dans le deuxième cas, on considère un utilisateur gouvernement souhaitant vérifier l’efficacité des air cleaners.



Dans le troisième cas, on considère un utilisateur quelconque souhaitant trouver les zones similaires à celle d’un capteur.



# Principaux algorithmes

**La méthode scanSensors() permet de détecter les capteurs dysfonctionnels et de les afficher.**

void scanSensors(listeDesCapteurs) {

pour chaque capteur dans listeDesCapteurs:

valeurO3 <- dernière valeur de O3 enregistrée par le capteur

// Idem pour NO2, SO2 et PM10

mapDistanceCapteurs <- multimap

pour chaque autreCapteur dans listeDesCapteurs:

si autreCapteur != capteur:

mapDistanceCapteurs[distance(capteur, autreCapteur)] = autreCapteur

fin si

fin pour

moyenneO3 <- 0

varianceO3 <- 0

// Idem pour NO2, SO2 et PM10

pour les 5 premiers autreCapteur dans mapDistanceCapteurs:

O3 <- dernière valeur de O3 enregistrée par autreCapteur

moyenneO3 <- (moyenneO3 + O3)

moyenneO3 <- moyenneO3 / 5

// Idem pour NO2, SO2 et PM10

fin pour

pour les 5 premiers autreCapteur dans mapDistanceCapteurs:

O3 <- dernière valeur de O3 enregistrée par autreCapteur

varianceO3 <- (O3 - moyenneO3)^2 / 4

// Idem pour NO2, SO2 et PM10

fin pour

si |valeurO3 - moyenneO3| > varianceO3:

afficher “Le capteur “ + capteur + “ semble défectueux. Voici ses informations :”

affichageInformationsCapteur(capteur)

fin si

// Idem pour NO2, SO2 et PM10

}

**La méthode checkEfficiency(airCleaner) permet de calculer l’efficacité d’un airCleaner sur sa période de fonctionnement**

checkEfficiency(airCleaner, listeDesCapteurs) {

mapDistanceCapteurs <- multimap

pour chaque capteur dans listeDesCapteurs:

mapDistanceCapteurs[distance(airCleaner, capteur)] = capteur

fin pour

debut <- date de début de fonctionnement de airCleaner

fin <- date de fin de fonctionnement de airCleaner

efficacite <- tableau[4] // Contiendra l’efficacité pour O3, NO2, SO2, PM10

capteurPlusProche <- premier capteur de mapDistanceCapteurs

mesures <- mesures de capteurPlusProche avant debut et apres fin airCleaner

pour les valeurs de O3, NO3, SO3 et PM10 de mesures (0 <= i < 4) :

efficacite[i] = (mesure debut - mesure fin) / mesure debut

fin pour

si efficacite O3 < 5% ou efficacite NO2 < 5% ou efficacite SO2 < 5% ou efficacite PM10 < 5% :

afficher “L’air cleaner n’a eu aucun effet sur l’air environnant”

sinon

afficher l’efficacite pour O3, NO2, SO2 et PM10

dernierCapteurEfficace <- capteurPlusProche

pour chaque capteur de mapDistanceCapteurs:

mesures <- mesures de capteur avant debut et apres fin airCleaner

efficacite[i] = (mesure debut - mesure fin) / mesure debut

si efficacite O3 < 5% ou efficacite NO2 < 5%  … :

sortir de la boucle

sinon

dernierCapteurEfficace <- capteur

fin si

fin pour

afficher “Rayon d’action : ” + distance(airCleaner, dernierCapteurEfficace)

fin si

}

**La méthode findSimilar() permet de trouver les capteurs similaires à un capteur sur une période donnée**

findSimilar(capteur, debut, fin, listeDesCapteurs) {

moyenneO3 <- moyenne de O3 enregistrées par le capteur entre debut et fin

// Idem pour NO2, SO2 et PM10

afficher “Capteurs similaires à ” + capteur

pour chaque autreCapteur dans listeDesCapteurs:

moyenneAutreO3 <- moyenne de O3 enregistrées par autreCapteur entre debut et fin

varianceAutreO3 <- variance de O3 enregistrées par autreCapteur entre debut et fin

// Idem pour NO2, SO2 et PM10

si |moyenneO3  - moyenneAutreO3 | <= varianceAutreO3 et idem pour NO2, SO2 et PM10 :

afficher “Le capteur “ + autreCapteur + “est similaire. Voici ses informations :”

affichageInformationsCapteur(autreCapteur)

fin si

fin pour

}

# Fichiers de sauvegarde

Nous devons créer des fichiers de sauvegarde afin de se remémorer les login et mot de passe des utilisateurs. Ces derniers ne seront pas cryptés.

Il s’agit du fichiers *data.csv*. Il contient les données dans le format suivant:

LOGIN ; MDP ; TYPE ; [NB de point] ; [NB de capteurs/airCleaners] ; [ID du capteur/airCleaner 1] ; [ID du capteur/airCleaner 2]…

Le type est 1 si c’est un utilisateur disposant d’un capteur ou non, 2 si c’est un fournisseur d’airCleaner ou 3 si c’est un membre du gouvernement.

Les valeurs suivantes sont présentes pour les utilisateurs et les fournisseurs uniquement.

Pour un utilisateur, il s’agit du nombre de point qu’il a accumulé, puis du nombre de capteurs qu’il possède et enfin liste des id de ces capteurs.

Pour les fournisseurs, il s’agit uniquement du nombre de airCleaner déployé ainsi que la liste des id de ces derniers.

# Tests unitaires

User checkpasswd(String login, String password)

Doit renvoyer user si l’utilisateur est dans le fichier *data.csv*

nullptr sinon

données : checkpasswd(user1, mdp) ⇒ user

checkpasswd(user1, mdp2) ⇒ NULLPTR

String getBonReduction()

Si le user.getPoints() est suffisant : Renvoyer vrai

Sinon : Renvoyer faux

données : if user.getPoints() > 150 ⇒ True

user.getPoints() < 150 ⇒ False

bool detectProblem()

Si le capteur possède une anomalie (par exemple un écart de valeur > 10% par heure) on renvoie True

et on appelle aSensor.setIsValid(false)

Sinon on renvoie False

données : pour O2 : 12h 21.1, 13h 22.0, 14h 21.5 ⇒ False

: 12h 21.1, 13h 35.0, 14h 15.5 ⇒ True

String generateStatistics(pos, startTime, endTime)

Renvoie les statistiques(voir Partie 1) sous forme d'une chaîne de caractères

Si pas de données viables, renvoyer "Pas de données"

generateStatistics(0.5:1, 01/01/2023:13h, 01/01/2023:13h)

if unCapteur.pos == 0.5:1 && unCapteur. ⇒ mesure

if tousLesCapteur.pos != 0.5:1 ⇒ pas de mesure

String generateStatistics(pos, r, startTime, endTime)

Renvoie les statistiques(voir Partie 1, pas de capteur à moins de 50km) d'une zone sous forme d'une chaîne de caractères

Si pas de données viables, renvoyer "Pas de données"

vector<Sensor> detectSimilarZones(Sensor aSensor, startTime, endTime)

Renvoie un vecteur de capteurs étant similaire (même moyenne et écart-type) triés par ordre décroissant de similarité.

Renvoie un vecteur vide s'il n'existe pas de capteurs similaires.

(moyenne des données sur le temps passé en paramètre)

Sensor 1 : O3 = 50 NO2 = 45

Sensor 2 : O3 = 49 NO2 = 46 ⇒ les deux capteurs sont similaire

Sensor 1 : O3 = 50 NO2 = 65

Sensor 2 : O3 = 49 NO2 = 70 ⇒ aucun capteurs n’est similaire

double checkEfficiency(AirCleaner)

Renvoie la zone d'éfficacité en mètre du airCleaner

données : Nous avons 5 capteurs disposés chacun à une distance différente du aircleaner. Le cinquième capteur est le plus loin. Pour les quatres premiers capteurs, les mesures sont rédite de 50 %. Or pour le cinquième, elle ne sont pas réduites. Le rayon d’efficacité doit donc se baser sur le capteur 4 avec une fficacité de 50 %

Bool In(position, r)

Renvoie vrai si la position est dans le cercle donné en paramètre

Sinon renvoie faux

données : this.lat : 45 this.long : 0.5

lat = 45 long = 0.7 r = 2000km ⇒ true

lat = 45 long = 0.7 r = 50m ⇒ false