Dossier design application

par Milo Fournier, Cassandre FILLOL-ANDRE, Kouceïla ABDELLAOUI et Gaspard SERPINET

Justification du choix d'architecture

Dans un premier temps nous avons choisi d'utiliser une architecture de type Model-View-Controller (MVC). En effet, cette dernière nous permet de changer les données indépendamment de leur représentation et vice-versa. Cette architecture sépare la façon dont les données sont stockées et utilisées de la manière dont on présente les résultats à l'utilisateur. Le lien entre l'utilisateur et les services proposés est fait avec la partie contrôleur qui gère les entrées-sorties.

La partie modèle représente la couche de données de l'application. Elle est responsable de la gestion et de la manipulation des données et offre une interface de programmation pour permettre aux autres parties de l'application de récupérer ou de modifier les données stockées.

La partie vue est responsable de l'interface utilisateur. Elle affiche les données et les résultats de manière graphique pour qu'elle puisse être compréhensible par l'utilisateur. De plus, on associera à chaque acteur de l'application (Particulier, Agence gouvernementale, Fournisseur) un affichage en fonction de ses besoins.

Enfin, la partie contrôleur est responsable de la coordination entre la partie vue et la partie modèle. Elle reçoit les entrées de l'utilisateur et déclenche les actions appropriées pour effectuer les opérations demandées. En outre, elle peut également agir comme une interface avec d'autres services externes à l'application.

Voici un schéma représentatif de l'architecture que nous avons choisie :

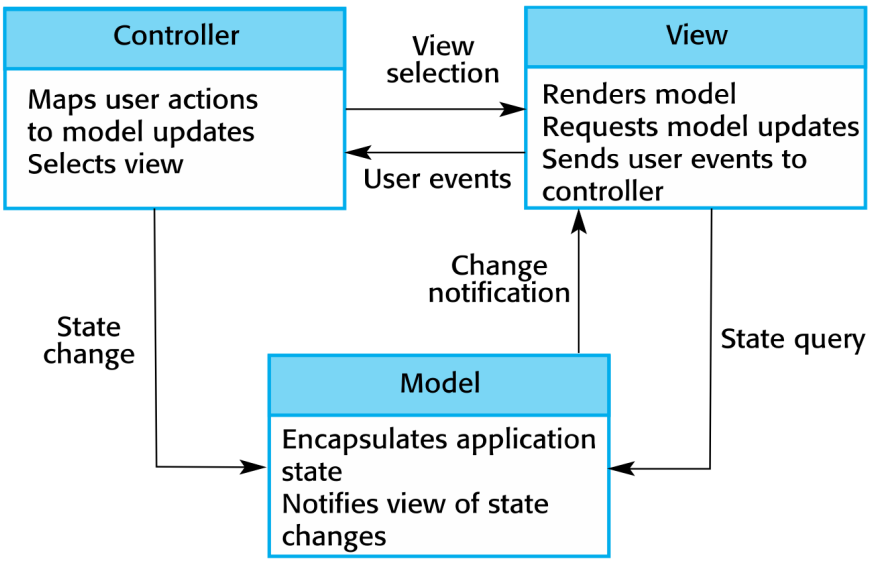
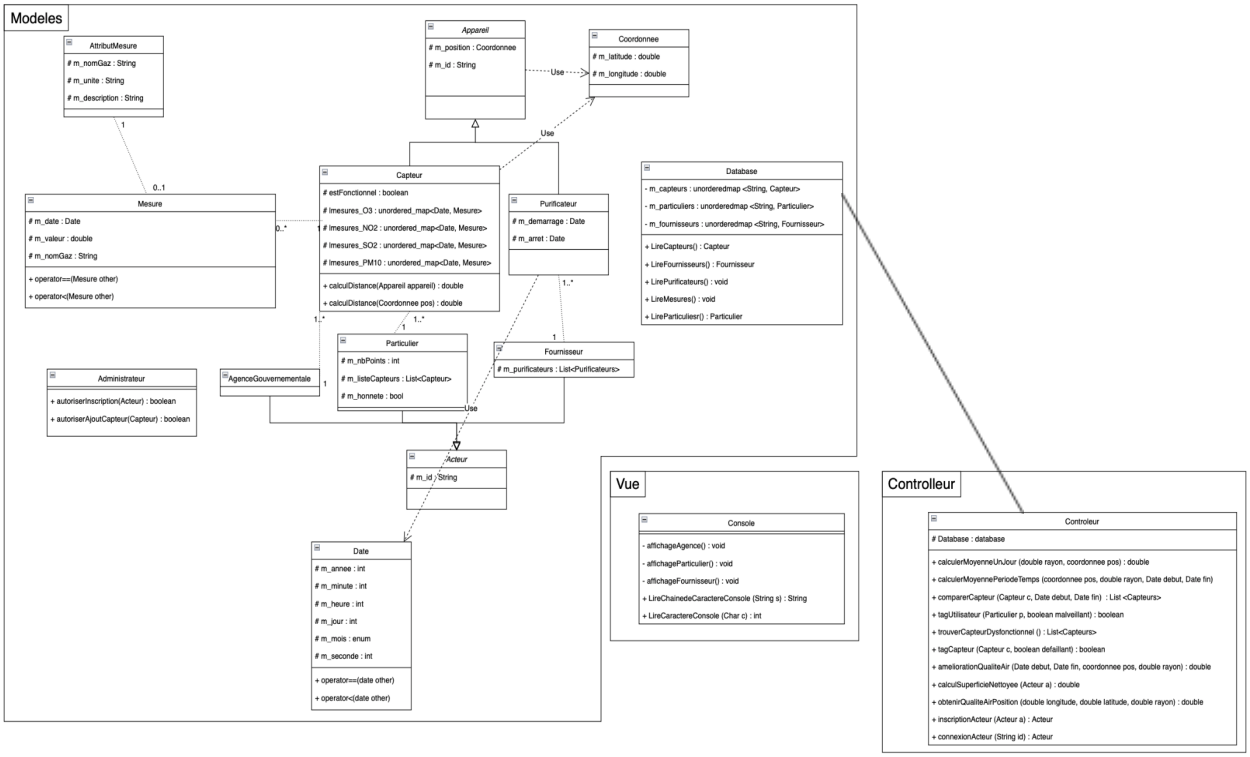


Diagramme de classe



Le diagramme de classe suivant représente les différents objets qui permettent le fonctionnement de notre application :

Une classe **Mesure** a été créée avec pour attributs une date, une valeur et le nom du gaz mesuré.

Nous avons également un objet **Acteur** avec comme attributs un ID, dont héritent les classes :

- **Particulier**, avec comme attribut un booléen *"m_honnete"*, qui représente la malveillance du particulier et donc de ses capteurs, une liste de capteurs correspondant aux capteurs lui appartenant, et un nombre de points qui est incrémenté en fonction de l'utilisation des données issues de ses capteurs sur l'application.
- **Fournisseur,** possédant une liste de purificateurs correspondant aux purificateurs qu'il a fournis.
- **Agence gouvernementale**.

En plus de cela, nous avons une classe **Appareil** avec comme attributs un identifiant et une position, dont héritent les classes :

- **Capteur,** avec comme attribut un booléen *"estFonctionnel"* qui nous permet de savoir si le capteur est en état de marche ou non. Nous avons également *4 unordered_map* dans lesquelles sont placées les mesures en fonction du gaz

mesuré, car nous avons constaté que nous accédions souvent aux mesures selon le gaz mesuré. La clé de cette unordered_map est sa date et la valeur est la mesure elle-même. Nous avons choisi la date comme clé car nous avons constaté, lorsqu'il a fallu réaliser les algorithmes, que nous voulions accéder à des mesures selon leur date. Ainsi, nous avons en attributs d'un capteur les mesures que celui-ci a réalisées, classées dans des unordered_map selon le gaz mesuré.

- **Purificateur**, avec comme attributs la date à laquelle son fonctionnement a débuté et la date à laquelle il a cessé de fonctionner.

Nous avons également deux classes qui interviennent dans la création des objets définis ci-dessus. Premièrement, la classe **Database** permet de créer *trois unordered_map* : une pour les capteurs, une pour les particuliers et une autre pour les fournisseurs. Nous avons choisi une unordered_map car nous avons constaté, lors de la conception de nos services et algorithmes, que nous accédions à ces objets en grande partie à partir de leur ID. Nous avons donc mis l'identifiant comme clé de l'unordered_map et l'objet en lui-même comme valeur.

Pour remplir ces structures de données, nous avons réalisé une classe **LireFichier.** Cette classe est composée de 5 méthodes :

- La méthode *lireFournisseurs()* permet de lire le fichier providers.csv et de créer un objet Fournisseur à partir de la ligne lue. On peut ainsi remplir la unordered_map de fournisseurs pour pouvoir l'exploiter dans le contrôleur.
- La méthode *lireCapteurs()* permet de créer un objet Capteur à partir de la ligne lue dans le fichier sensors.csv, ce qui permettra de remplir la unordered_map de capteurs s'ils n'appartiennent pas à des particuliers. Si le capteur appartient à un particulier, on l'ajoute à la liste du particulier correspondant en le retrouvant grâce à son identifiant.
- Une fonction *LirePurificateurs()*, qui permet créer un objet purificateur à partir d'une ligne lue dans de lire le fichier cleaners.csv, ce qui permettra de remplir la liste de purificateur en fonction de l'identifiant du fournisseur.
- Une fonction *lireMesures()*, qui permet de créer un objet Mesures à partir d'une ligne lue dans le fichier measurements.csv correspondant, ce qui permettra de remplir les unordered_map de la classe Capteur, présenté précédemment, en fonction du gaz mesuré.
- Enfin nous avons une fonction *lireParticuliers()*, qui permet de créer un objet Particulier à partir de de la ligne lue dans le fichier users.csv, ce qui permettra de remplir la unordered_map m_particuliers, qui est un attribut de la classe Database.

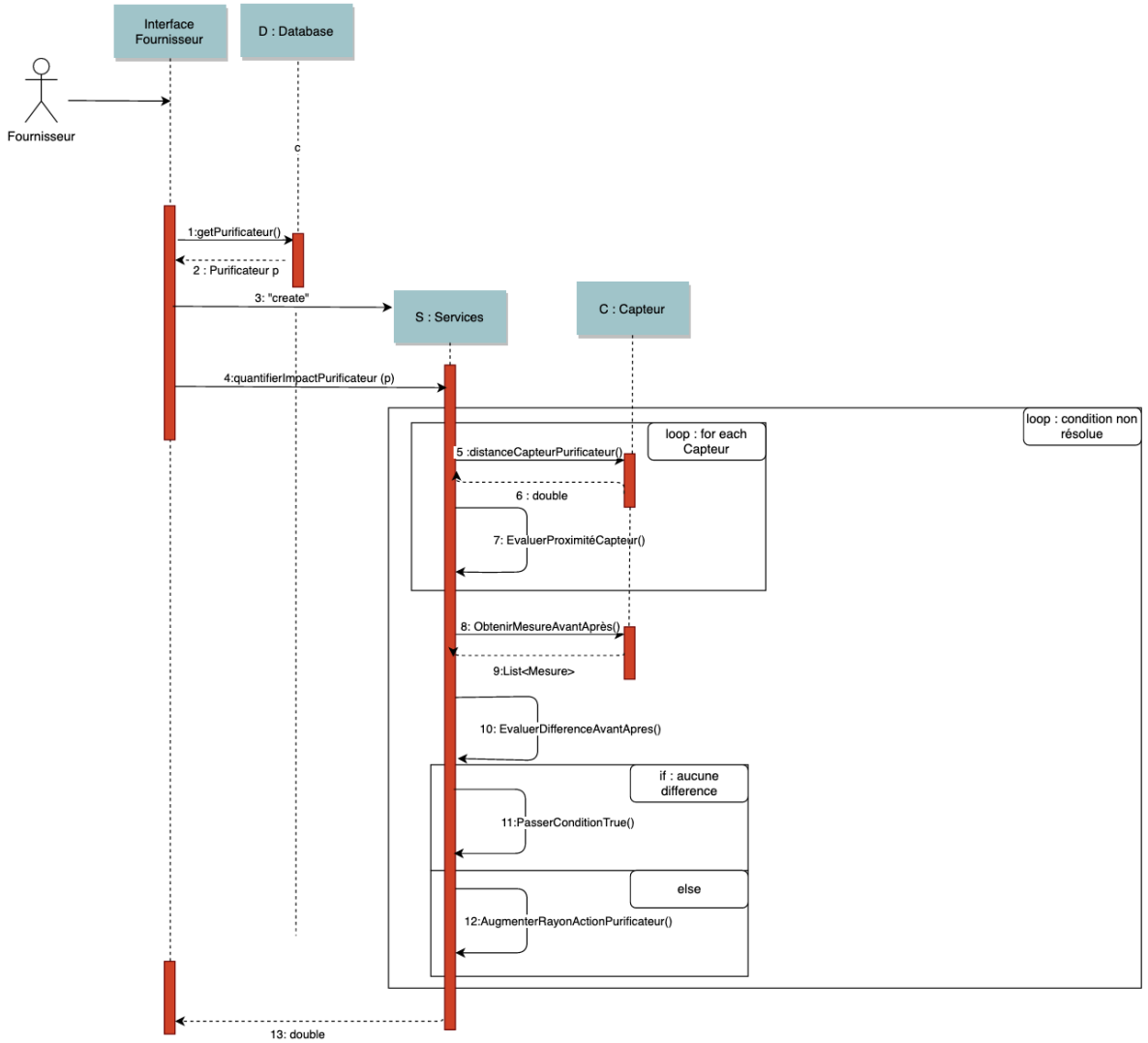
Comme énoncé précédemment, nous avons choisi une architecture MVC, qui suggère la création d'une classe Vue, une classe Contrôleur et une classe Services.

La classe **Service** regroupe tous les services que notre application pourra offrir. La classe **Controleur** quant à elle possède un attribut de type DataBase. Son rôle est de pouvoir prendre les caractères entrés par l'utilisateur dans la console et de réaliser des actions en conséquence, en communiquant avec la classe Services. Enfin, nous avons la classe **Vue**, qui permet un affichage différent selon l'acteur qui utilise l'application.

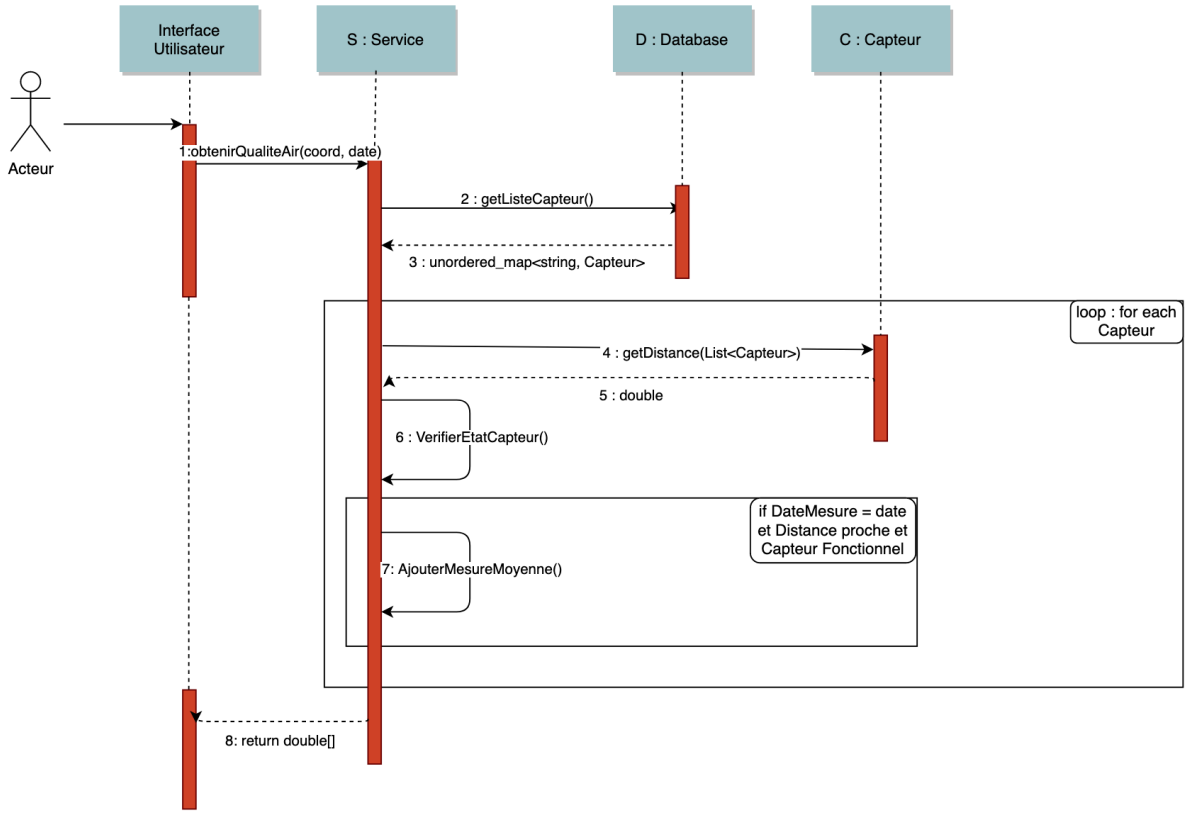
Dans le cadre du code de notre application, le contrôleur et la vue selon néanmoins rassemblés dans le classe Main. De plus, la classe Admin ne sera pas développée dans le cadre de ce projet car non nécessaire aux tests que nous allons mettre en place.

Diagrammes de séquence pour trois algorithmes majeurs de l'application

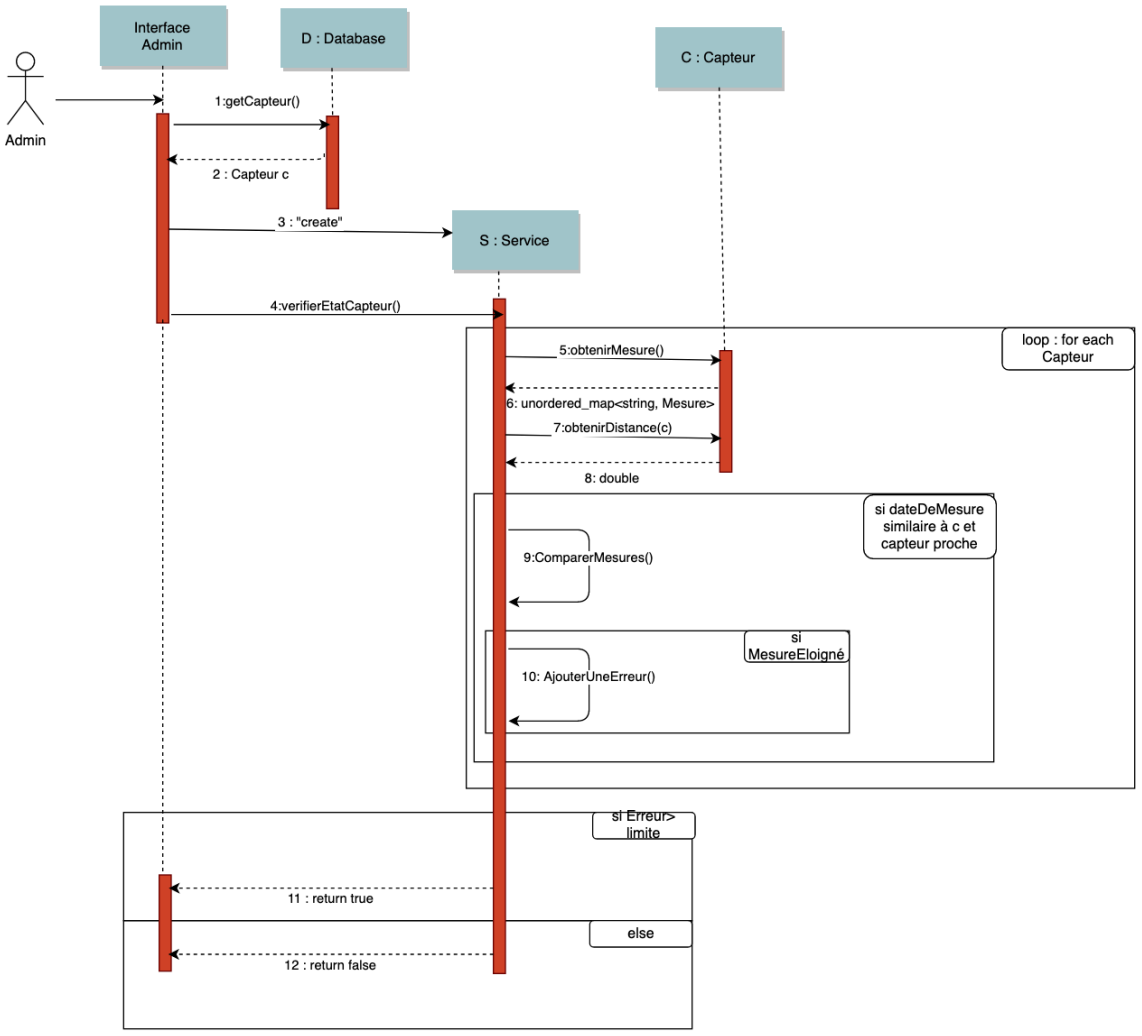
a) Service quantifierImpactPurificateur



b) Service obtenirQualiteAir



c) Service verifierEtatCapteur



Pseudo-codes

Il faut, avant l'exécution de chaque pseudo code, prendre en compte nous nous chargerons d'exécuter les méthodes de la classe LireFichier, qui permettront dans un premier temps d'initialiser les unordered_map dans lesquelles se trouvent les capteurs, particuliers, et fournisseurs, et changer les unordered_map de mesures associé à un capteur et la liste de purificateurs associé à un fournisseur.

a) Vérifier si un capteur est dysfonctionnel

```
Fonction verifierEtatCapteur(capteurParam: capteur) -> boolean
    distanceDeVerification <- 200km
    limiteDeTemps <- meme jour
    limiteDeMesure <- 5
    compteurErreurs <- 0
    compteur <- 0
    Pour chaque capteur capteurTest
        Si la distance entre capteurParam et capteurTest est inférieur à distanceDeVerification
            Si la date de la mesure de capteurTest date du même jour que celle de capteurParam
                Pour chaque gaz
                    Si la différence entre les valeurs des mesures de capteursTest et capteurParam est plus grande que limiteDeMesure
                        compteurErreur <- compteurErreur + 1
                    Fin Si
                Fin Pour
                compteur <- compteur + 4
            Fin Si
        Fin Si
    Fin Pour
    tauxErreur <- (compteurErreur/compteur) * 100
    dysfonctionnel <- false
    Si tauxErreur > 70%
        dysfonctionnel <- true
    Fin Si
    Retourne etatOk
Fin Fonction
```

La fonction verifierEtatCapteur prend en entrée un objet capteur et renvoie un booléen qui indique si l'état du capteur est considéré comme OK ou non.
La fonction commence par initialiser plusieurs variables locales, telles que la distance de vérification (distanceDeVerification), la limite de temps pour laquelle on considère les mesures comme étant du même jour (limiteDeTemps), et la limite de mesure (limiteDeMesure) qui représente la différence maximale tolérée entre les valeurs de deux capteurs pour un gaz donné.

Ensuite, la fonction boucle sur chaque capteur disponible (capteurTest) et calcule la distance entre celui-ci et le capteur passé en paramètre (capteurParam). Si cette distance est inférieure à la distance de vérification, alors la fonction vérifie si la date de la mesure de capteurTest est du même jour que celle de capteurParam.

Si c'est le cas, la fonction boucle sur chaque gaz et vérifie si la différence entre les valeurs des mesures de capteurTest et capteurParam est supérieure à la limite de mesure. Si tel est le cas, la variable compteurErreur est incrémentée.

Après avoir terminé la boucle, la fonction calcule le taux d'erreur en divisant compteurErreur par compteur (le nombre total de mesures comparées) et en multipliant par 100. Si ce taux est supérieur à 70%, la fonction renvoie true, indiquant que l'état du capteur est dysfonctionnel. Sinon, elle renvoie false.

b) Obtenir la qualité de l'air à une certaine position à un moment donné

```
Fonction obtenirQualiteAirPosition(coordonneesParam: Coordonnees, dateParam: Date) -> double[]
    rayonDeVerification <- 200km
    tableau <- [0, 0, 0, 0]
    Pour chaque capteur
      Si la distance entre coordonneesParam et le capteur trouvé est inférieur à rayonDeVerification
          Si l'état du capteur trouvé est true //appel à verifierEtatCapteur
              Si la date de la mesure est égale à dateParam
                  Pour i allant de 0 à 3
                      tableau[i] <- tableau[i] + la valeur de la mesure
                  Fin Pour
              Fin Si
              compteur <- compteur + 1
          Fin Si
      Fin Si
    Fin Pour
    Pour i allant de 0 à 3
        tableau[i] <- tableau[i]/compteur
    Fin Pour
    Retourne tableau
Fin Fonction
```

Ce code implémente une fonction nommée obtenirQualiteAirPosition qui prend en entrée un objet coordonneesParam de type Coordonnees représentant les coordonnées géographiques d'un lieu, et un objet dateParam de type Date représentant la date pour laquelle on souhaite obtenir la qualité de l'air.

La fonction renvoie un tableau de 4 valeurs, chaque valeur représentant la qualité de l'air pour un type de gaz spécifique. Pour obtenir ces valeurs, la fonction recherche tous les capteurs de qualité de l'air situés à une distance inférieure à 200 km de coordonneesParam. Pour chaque capteur trouvé, la fonction vérifie si son état est "true" (appel à la fonction verifierEtatCapteur), puis elle vérifie si la date de la mesure est égale à dateParam. Si ces conditions sont remplies, la fonction ajoute la valeur de la mesure pour chaque type de gaz dans le tableau.

Enfin, la fonction calcule la moyenne de chaque type de gaz à partir des valeurs ajoutées au tableau précédemment. Ces moyennes sont stockées dans le tableau qui sera renvoyé en tant que résultat de la fonction.

Pour conclure, cette fonction est utilisée pour obtenir la qualité de l'air à une position géographique donnée et à une date spécifique en utilisant les données de capteurs situés à proximité de cette position.

c) Obtenir le rayon d'action d'un purificateur

```
Fonction quantifierImpactPurificateur(purificateurParam: Purificateur) -> double
    condition <- true
    rayonAction <- 0
    max <- 0
    Tant que condition == true
        min <- +infinity
        capteurProche <- null
        Pour chaque capteur
            distance <- distance entre le capteur et purificateurParam
            Si distance < min et distance > max
                min <- distance
                capteurProche <- capteur
                max <- distance
            Fin Si
        Fin Pour
        tableauAvant <- [0, 0, 0, 0]
        tableauApres <- [0, 0, 0, 0]
        tableauMoyenneAvant <- [0, 0, 0, 0]
        tableauMoyenneApres <- [0, 0, 0, 0]
        Pour i allant de 0 à 3 de capteurProche //i allant de 0 à 3 signifie pour chaque gaz
            compteurAvant <- 0
            Pour chaque date de mesure inférieure à la date de lancement de purificateurParam
                tableauAvant[i] <- tableauAvant[i] + la valeur de la mesure
                compteurAvant <- compteurAvant + 1
            Fin Pour
            tableauMoyenneAvant[i] <- tableauAvant[i]/compteurAvant
        Fin Pour
        Pour i allant de 0 à 3 de capteurProche //i allant de 0 à 3 signifie pour chaque gaz
            compteurApres <- 0
            Pour chaque date de mesure supérieure à la date de lancement de purificateurParam et inférieur à sa date de fin
                tableauApres[i] <- tableauApres[i] + la valeur de la mesure
                compteurApres <- compteurApres + 1
            Fin Pour
            tableauMoyenneApres[i] <- tableauApres[i]/compteurApres
        Fin Pour
        compteurErreurs <- 0
        Pour i allant de 0 à 3
            Si tableauMoyenneApres[i] - tableauMoyenneAvant[i] > 0
                compteurErreurs <- compteurErreurs + 1
            Fin Si
        Fin Pour
        Si compteurErreurs > 2
            condition <- false
        Fin Si
        Si condition == true
            rayonAction <- max
        Fin Si
    Fin Tant que
    Retourne rayonAction
Fin Fonction
```

Ce code implémente une fonction nommée quantifierImpactPurificateur qui prend en entrée un objet purificateurParam de type Purificateur et renvoie la valeur de la variable rayonAction qui représente la portée d'action maximale du purificateur.

Le code utilise une boucle "tant que" pour déterminer la valeur de la variable rayonAction. Dans cette boucle, le code cherche le capteur le plus proche du purificateurParam et calcule les moyennes des mesures de gaz avant et après le lancement du purificateur. Si la

différence entre les moyennes des mesures après et avant est positive pour plus de deux types de gaz, la boucle s'arrête et la valeur de rayonAction est renvoyée.

Pour conclure, cette fonction est utilisée pour quantifier l'impact d'un purificateur d'air sur la qualité de l'air ambiant en mesurant la différence entre les concentrations de gaz avant et après le lancement du purificateur.

Tests unitaires

a) Vérifier état capteur

Description du service :

Nom	VerifierEtatCapteur
Rôle	Analyser les données pour s'assurer du bon fonctionnement des capteurs pour aider l'agence gouvernementale à identifier et réparer les capteurs qui ne fonctionnent pas correctement. Si le service trouve des capteurs dysfonctionnels, alors il fait appel au service tagCapteur pour les définir comme défaillants.
Input	Le capteur qu'on veut vérifier
Output	Renvoie un booleen représentant l'état du capteur : si on renvoie false alors le capteur est dysfonctionnel sinon on renvoie true.

Tests :

1.

Nom	testVerifierCapteurSeul
Dataset	Il n'y a qu'un seul capteur : 2023-05-06 12:00:00 | Sensor0 | O3 | 65.00
Input	Le capteur Sensor0
Output	Peu importe sa valeur, ne pouvant être comparé aux capteurs alentour, il sera supposé juste par l'application. On envoi donc ***FALSE***, car le booléen est nommé Dysfonctionnel

2.

Nom	testEtatCapteurOKdansEtatNonCohérent
Dataset	Il y a 4 capteurs. 3 d'entre eux ont des mesures justes et sont : 2023-05-06 12:00:00 | Sensor0 | O3 | 65.00

	2023-05-06 12:00:00 | Sensor1 | SO2 | 59.00 2023-05-06 12:00:00 | Sensor2 | O3 | 60.00 Le dernier donne des mesures aberrantes qui sont : 2023-05-06 12:00:00 | Sensor3 | O3 | 6500
Input	Le capteur Sensor2
Output	On renvoie ***FALSE***, car le capteur est cohérent avec la majorité des capteurs alentour.

3.

Nom	testEtatCapteurNonCohérent
Dataset	Il y a 3 capteurs. 2 d'entre eux ont des mesures justes et sont : 2023-05-06 12:00:00 | Sensor0 | O3 | 65.00 2023-05-06 12:00:00 | Sensor1 | SO2 | 59.00 Le dernier donne des mesures aberrantes qui sont : 2023-05-06 12:00:00 | Sensor2 | O3 | 6500
Input	Le capteur Sensor2
Output	On renvoie ***TRUE***, car le capteur n'est pas considéré comme OK

4.

Nom	testEtatCapteurCohérent
Dataset	Il y a 3 capteurs dans un rayon proche. Les 3 ont des valeurs proches. 2023-05-06 12:00:00 | Sensor0 | O3 | 65.00 2023-05-06 12:00:00 | Sensor1 | O3 | 65.02

	2023-05-06 12:00:00 | Sensor2 | O3 | 65.01
	Le capteur Sensor0 (on peut ensuite faire ce même test avec tous les sensors)
Output	On renvoie ***FALSE***.

b) Obtenir la qualité de l'air à une certaine position

Description du service :

Nom	obtenirQualiteAirPosition
Rôle	L'utilisateur peut grâce à l'application demander à connaître la qualité de l'air autour de coordonnées et à une date choisies par l'utilisateur. Ce service permet de lui renvoyer une moyenne de ces données
Input	Coordonnées et une date
Output	On renvoie un tableau des moyennes des différents gaz

Tests :

1.

Nom	testObtenirQualiteAirSeul
Dataset	Il n'y a qu'un seul capteur près des coordonnées demandée qui a des mesure à la date demandée : 2023-05-06 12:00:00 | Sensor0 | O3 | 65.00
Input	Les coordonnées de ce capteurs et le 5/06/2023
Output	Si le capteur est dysfonctionnel, alors un tableau [0, 0, 0, 0] sera renvoyé, et si le capteur est fonctionnel, les mesures du capteur en question seront renvoyé.

2.

Nom	testObtenirQualiteAirAucunCapteur
Dataset	Il n'y a aucun capteur ayant pris des mesures à la date choisie et près des coordonnées demandée
Input	Par exemple : 20/02/2022 et coordonnées sans capteurs à proximité

Output	[0, 0, 0, 0]

3.

Nom	testObtenirCapteurPlusieurs
Dataset	Il y a 3 capteurs dans un rayon proche ayant pris des mesures à la bonne date. Les 3 ont des valeurs proches. 2023-05-06 12:00:00 | Sensor0 | O3 | 65.00 2023-05-06 12:00:00 | Sensor1 | O3 | 65.02 2023-05-06 12:00:00 | Sensor2 | O3 | 65.01
Input	Une coordonnées au milieu de ces 3 capteurs et 5/6/2023
Output	Un tableau de double renvoyant une moyenne des trois mesures, si les trois capteurs sont fonctionnels.

4.

Nom	testObtenirCapteurIncohérent
Dataset	Il y a 3 capteurs. 2 d'entre eux ont des mesures justes et sont : 2023-05-06 12:00:00 | Sensor0 | O3 | 65.00 2023-05-06 12:00:00 | Sensor1 | SO2 | 59.00 Le dernier donne des mesures aberrantes qui sont : 2023-05-06 12:00:00 | Sensor2 | O3 | 6500
Input	Une coordonnées au milieu de ces 3 capteurs et 5/6/2023
Output	Un tableau de double renvoyant une moyenne des deux mesures des capteurs cohérent qui ignore bien les mesures du capteur incohérent

c) Quantifier Impact Purificateur

Description du service :

Nom	quantifierImpactPurificateur
Rôle	Nous permet de savoir quel est le rayon d'action d'un purificateur d'air.
Input	Le purificateur dont on veut étudier l'impact.
Output	Renvoie un double signifiant le rayon d'action du purificateur. Renvoie 0 sinon.

Tests :

1.

Nom	testQuantifierImpactPurificateurSansCapteur
Dataset	Il n'y a aucun capteur.
Input	Le purificateur.
Output	On renvoie 0.

2.

Nom	testQuantifierAirPurificateurCapteur
Dataset	Il n'y a qu'un seul capteur près des coordonnées demandées qui a des mesures aux dates demandées et qui montre une amélioration.
Input	Le purificateur.
Output	On renvoie la distance entre le purificateur d'air et le capteur.