# Compte Rendu de Projet AP4A

**Lucas Debouche**

**10 Novembre 2024**

# Table des matières

Compte Rendu de Projet AP4A ..... 0
1. Introduction ..... 2
1.1 Objectif du Projet ..... 2
1.2 Contexte ..... 2
1.3 Technologies Utilisées ..... 2
2. Analyse et Conception ..... 2
2.1 Structure Générale du Projet ..... 2
2.2 Description des Classes ..... 3
3. Implémentation ..... 5
3.1 Méthodes Clés ..... 5
3.2 Gestion des Fichiers CSV ..... 6
4. Tests et Résultats ..... 6
4.1 Exécution des Capteurs ..... 6
4.2 Vérification des Fichiers CSV ..... 6
4.3 Affichage en Console ..... 6
5. Conclusion et Perspectives ..... 7
5.1 Bilan du Projet ..... 7
5.2 Difficultés Rencontrées ..... 7
5.3 Améliorations Futures ..... 7

# 1. Introduction

## 1.1 Objectif du Projet

Ce projet consiste à développer un simulateur d'écosystème IoT permettant de surveiller la qualité de l'air dans un espace de travail. Il repose sur un serveur central qui reçoit et analyse des données provenant de plusieurs types de capteurs : température, humidité, son, et lumière.

## 1.2 Contexte

Le simulateur IoT est conçu pour permettre le suivi de paramètres environnementaux clés. Chaque capteur simule des lectures de données et les envoie au serveur pour être enregistrées et analysées. Le projet utilise les concepts de la programmation orientée objet en C++ et exploite les fonctionnalités d'héritage, de polymorphisme, et d'abstraction pour une architecture flexible et extensible.

## 1.3 Technologies Utilisées

Le projet est réalisé en C++ avec le compilateur g++. La gestion de fichiers se fait via la bibliothèque <filesystem>, et le projet utilise une structure multi-fichier pour une meilleure organisation et maintenabilité.

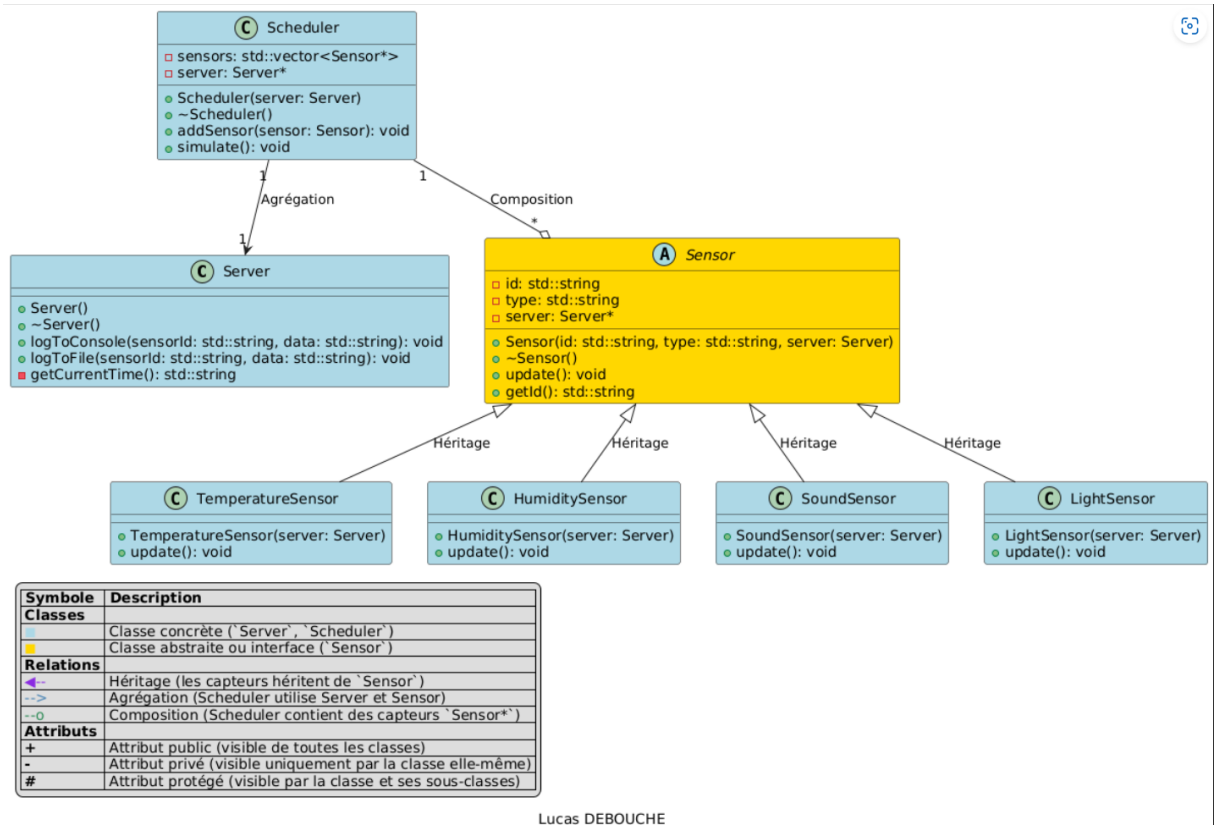
# 2. Analyse et Conception

## 2.1 Structure Générale du Projet

L'architecture du projet est organisée autour de quatre classes principales :

- Server : Représente le serveur central recevant les données des capteurs.
- Scheduler : Responsable de l'activation cyclique des capteurs pour simuler les lectures de données en temps réel.
- Sensor : Classe abstraite qui définit le comportement générique des capteurs.
- Classes spécifiques de capteurs : TemperatureSensor, HumiditySensor, SoundSensor, LightSensor.

**Schéma UML des Classes**




## 2.2 Description des Classes

### Server

La classe Server sert de point central pour la réception et l'analyse des données envoyées par les capteurs. Elle enregistre les données dans des fichiers .CSV dédiés à chaque capteur. Un en-tête (Time | Value) est écrit dans le fichier si celui-ci est vide ou inexistant. Cette classe peut également afficher des données dans la console.

```cpp
#ifndef SERVER_H
#define SERVER_H

#include <iostream>
#include <fstream>
#include <string>
#include <ctime>
#include <unordered_map>

// Classe Server : gère l'enregistrement et l'affichage des données reçues des capteurs
class Server {
public:
    Server();
    ~Server();

    // Affiche les données d'un capteur dans la console
    void logToConsole(const std::string& sensorId, const std::string& data);

    // Enregistre les données d'un capteur dans un fichier CSV spécifique
    void logToFile(const std::string& sensorId, const std::string& data);

private:
    // Récupère l'heure actuelle au format "DD-MM-YYYY HH:MM:SS"
    std::string getCurrentTime();
};

#endif #ifndef SERVER_H
```

### Scheduler

La classe Scheduler gère l'activation des capteurs selon un intervalle de temps fixe. Elle appelle la méthode update de chaque capteur, ce qui permet de simuler des lectures périodiques de données. Scheduler utilise le polymorphisme pour activer tous les capteurs sans connaître leur type spécifique.

```cpp
#ifndef SCHEDULER_H
#define SCHEDULER_H

#include <vector>
#include "Sensor.h"

// Classe Scheduler : active les capteurs périodiquement pour simuler les lectures
class Scheduler {
private:
    std::vector<Sensor*> sensors; // Liste des capteurs à activer
    Server* server;

public:
    Scheduler(Server* server);
    ~Scheduler();

    // Ajoute un capteur à la liste des capteurs
    void addSensor(Sensor* sensor);

    // Lance la simulation en activant chaque capteur périodiquement
    void simulate();
};

#endif // #ifndef SCHEDULER_H
```

### Sensor (classe abstraite)

Sensor est une classe abstraite qui définit une interface pour les capteurs. Elle contient une méthode virtuelle pure update qui est redéfinie par chaque capteur spécifique.

```cpp
#ifndef SENSOR_H
#define SENSOR_H

#include <string>
#include "Server.h"

// Classe abstraite Sensor : définit le comportement de base d'un capteur
class Sensor {
protected:
    std::string id;      // Identifiant unique du capteur
    std::string type;    // Type de capteur (Température, Humidité, etc.)
    Server* server;      // Pointeur vers le serveur pour envoyer les données

public:
    Sensor(const std::string& id, const std::string& type, Server* server);
    virtual ~Sensor();

    // Méthode virtuelle pure qui sera redéfinie dans chaque type de capteur
    virtual void update() = 0;

    // Retourne l'identifiant du capteur
    virtual std::string getId() const { return id; }
};

#endif // #ifndef SENSOR_H
```

### Capteurs Spécifiques

- **TemperatureSensor** : Génère des valeurs de température et les envoie au serveur.
- **HumiditySensor** : Génère des valeurs d'humidité.
- **SoundSensor** : Simule le niveau sonore en décibels.
- **LightSensor** : Simule la détection de lumière (allumé/éteint).

```h
#ifndef TEMPERATURESENSOR_H
#define TEMPERATURESENSOR_H

#include "Sensor.h"

// Classe TemperatureSensor : dérivée de Sensor, génère des données de température
class TemperatureSensor : public Sensor {
public:
    TemperatureSensor(Server* server);

    // Génère une lecture de température et envoie les données au serveur
    void update() override;
};

#endif // #ifndef TEMPERATURESENSOR_H
```

```h
#ifndef SOUNDSENSOR_H
#define SOUNDSENSOR_H

#include "Sensor.h"

// Classe SoundSensor : dérivée de Sensor, génère des données de niveau sonore
class SoundSensor : public Sensor {
public:
    SoundSensor(Server* server);

    // Génère une lecture de niveau sonore et envoie les données au serveur
    void update() override;
};

#endif // #ifndef SOUNDSENSOR_H
```

```h
#ifndef LIGHTSENSOR_H
#define LIGHTSENSOR_H

#include "Sensor.h"

// Classe LightSensor : dérivée de Sensor, détecte la présence ou absence de lumière
class LightSensor : public Sensor {
public:
    LightSensor(Server* server);

    // Génère un état lumineux et envoie les données au serveur
    void update() override;
};

#endif // #ifndef LIGHTSENSOR_H
```

```h
#ifndef HUMIDITYSENSOR_H
#define HUMIDITYSENSOR_H

#include "Sensor.h"

// Classe HumiditySensor : dérivée de Sensor, génère des données d'humidité
class HumiditySensor : public Sensor {
public:
    HumiditySensor(Server* server);

    // Génère une lecture d'humidité et envoie les données au serveur
    void update() override;
};

#endif // #ifndef HUMIDITYSENSOR_H
```

# 3. Implémentation

## 3.1 Méthodes Clés

### Méthode Server::logToFile

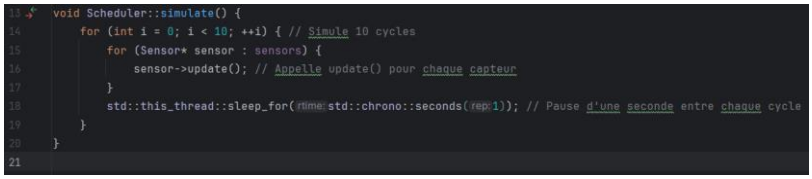
Cette méthode enregistre les données dans un fichier CSV. Si le fichier n'existe pas ou est vide, elle ajoute un en-tête au début. Ensuite, elle ajoute chaque nouvelle donnée sous forme Time,Value.

```cpp
void Server::logToFile(const std::string& sensorId, const std::string& data) {
    std::string filename = sensorId + "_log.csv";
    bool fileExists = std::filesystem::exists(filename); // Vérifie si le fichier existe déjà
    bool isFileEmpty = (fileExists && std::filesystem::file_size(filename) == 0); // Vérifie si le fichier est vide

    // Ouvre le fichier en mode ajout
    std::ofstream logFile(filename, std::ios::app);

    // Si le fichier est vide ou inexistant, ajoute l'en-tête
    if (!fileExists || isFileEmpty) {
        logFile << "Time | Value\n";
    }

    // Récupère l'heure actuelle et écrit les données avec l'horodatage dans le fichier
    std::string currentTime = getCurrentTime();
    logFile << currentTime << " | " << data << "\n";
    logFile.close();
}
```

### Méthode Scheduler::simulate

La méthode simulate de la classe Scheduler active chaque capteur dans un cycle de 10 itérations, en utilisant une pause d'une seconde entre chaque cycle pour simuler des

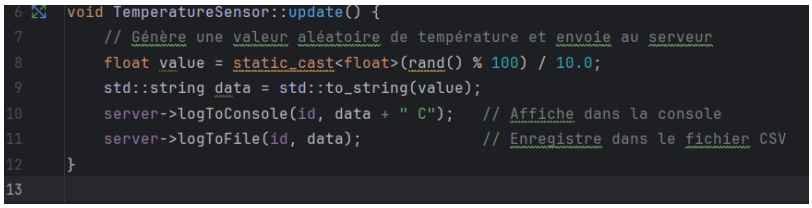
lectures en temps réel. Après chaque update, les données de capteurs sont transmises au serveur.



```
void Scheduler::simulate() {
    for (int i = 0; i < 10; ++i) { // Simule 10 cycles
        for (Sensor* sensor : sensors) {
            sensor->update(); // Appelle update() pour chaque capteur
        }
        std::this_thread::sleep_for(rtime:std::chrono::seconds(rep:1)); // Pause d'une seconde entre chaque cycle
    }
}

```

**Méthode Sensor::update pour les Capteurs Spécifiques**

Chaque capteur spécifique redéfinit la méthode update pour générer et enregistrer ses données de manière polymorphique. Par exemple, TemperatureSensor enregistre des valeurs de température en utilisant un format prédéfini.



```
void TemperatureSensor::update() {
    // Génère une valeur aléatoire de température et envoie au serveur
    float value = static_cast<float>(rand() % 100) / 10.0;
    std::string data = std::to_string(value);
    server->logToConsole(id, data + " C");   // Affiche dans la console
    server->logToFile(id, data);             // Enregistre dans le fichier CSV
}

```

## 3.2 Gestion des Fichiers .CSV

Le serveur génère un fichier .CSV pour chaque capteur, avec un en-tête unique en première ligne (Time | Value). Chaque nouvelle donnée de capteur est horodatée et ajoutée à la fin du fichier.

# 4. Tests et Résultats

## 4.1 Exécution des Capteurs

Chaque capteur est testé pour s'assurer qu'il génère des données et les envoie correctement au serveur. L'affichage en console confirme que les données sont générées de manière réaliste.

## 4.2 Vérification des Fichiers .CSV

Les fichiers CSV générés pour chaque capteur sont vérifiés pour garantir qu'ils contiennent l'en-tête unique en première ligne et des valeurs horodatées pour chaque mesure. Ci-dessous un exemple de fichier généré pour le capteur de température.

## 4.3 Affichage en Console

L'affichage en console pendant la simulation montre que les capteurs envoient régulièrement des données au serveur.

# 5. Conclusion et Perspectives

## 5.1 Bilan du Projet

Ce projet a permis de concevoir un simulateur IoT pour surveiller la qualité de l'air dans un environnement de travail, en utilisant des principes de la programmation orientée objet tels que l'héritage, le polymorphisme et l'abstraction.

## 5.2 Difficultés Rencontrées

Certaines difficultés incluent l'implémentation du polymorphisme, la gestion des fichiers CSV avec un en-tête unique, et l'utilisation des bibliothèques de gestion de fichiers.

## 5.3 Améliorations Futures

Plusieurs améliorations sont possibles :

- Ajout de nouveaux types de capteurs.
- Utilisation de threads pour simuler les lectures de données en temps réel.
- Création d'une interface graphique pour visualiser les données en temps réel.
- Ajout de fonctions avancées d’analyse de données dans la classe Server.