Conception Orientée Objet : Bureau d'étude

Gary Romain
gary@etud.insa-toulouse.fr
Combes Pauline
p_combes@etud.insa-toulouse.fr

May 28, 2020

1 Introduction

Ce rapport a pour but de documenter et d'expliquer nos choix de conception pour le bureau d'étude de conception orientée objet. Ce projet est un peu particulier car il a débuté en plein confinement, et nous n'avons donc pas eu accès physiquements aux cartes et capteurs normalement mis à disposition. Lorsqu'on nous a demandé de trouver une solution IoT innovante à un problème que peuvent rencontrer de potentiels utilisateurs, nous avons eu l'idée d'un assistant de jardinage connecté. Nous avons profité du fait que ce bureau d'étude soit complètement simulé pour nous amuser et créer ce petit jardin autonome.

2 Description de l'assistant de jardinage connecté : problématique et fonctionnement

Le système que nous allons considérer se compose de trois plantes au maximum, réparties sur une rangée. A chaque plante est associée une lampe à UV. De plus, un rail sur lequel circule un arrosoir est disposé le long de cette rangée. Ainsi, le but de l'application sera d'effectuer des actions (Allumer les lampes, arroser less plantes, etc.) en fonction de l'état des plantes; retrouné par les capteurs. La figure 1 montre notemment les fonctionnalités à implémenter pour ce système.

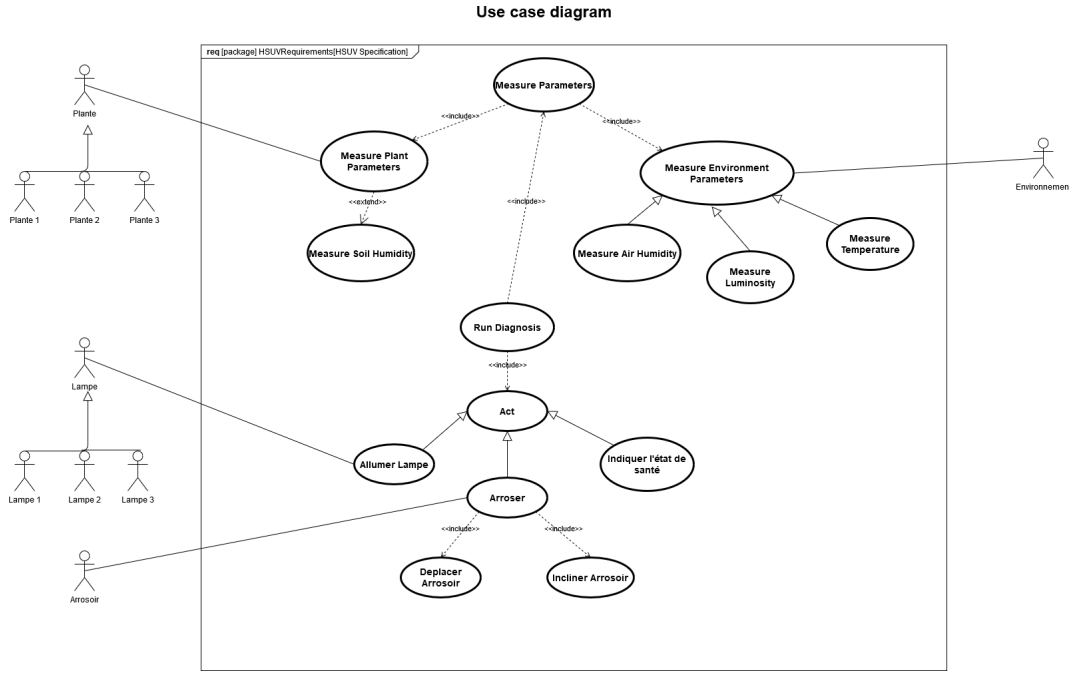


Figure 1: UseCase du sytème d'assistant de jardinage

Au niveau de l'architecture que nous développerons, nous utiliserons une librairie de capteurs, développée en section 3. Les actions à effectuer seront alors décidées par une fonction de décision, expliquée en section 4. Ainsi, nous avons le diagramme de séquence suivant :

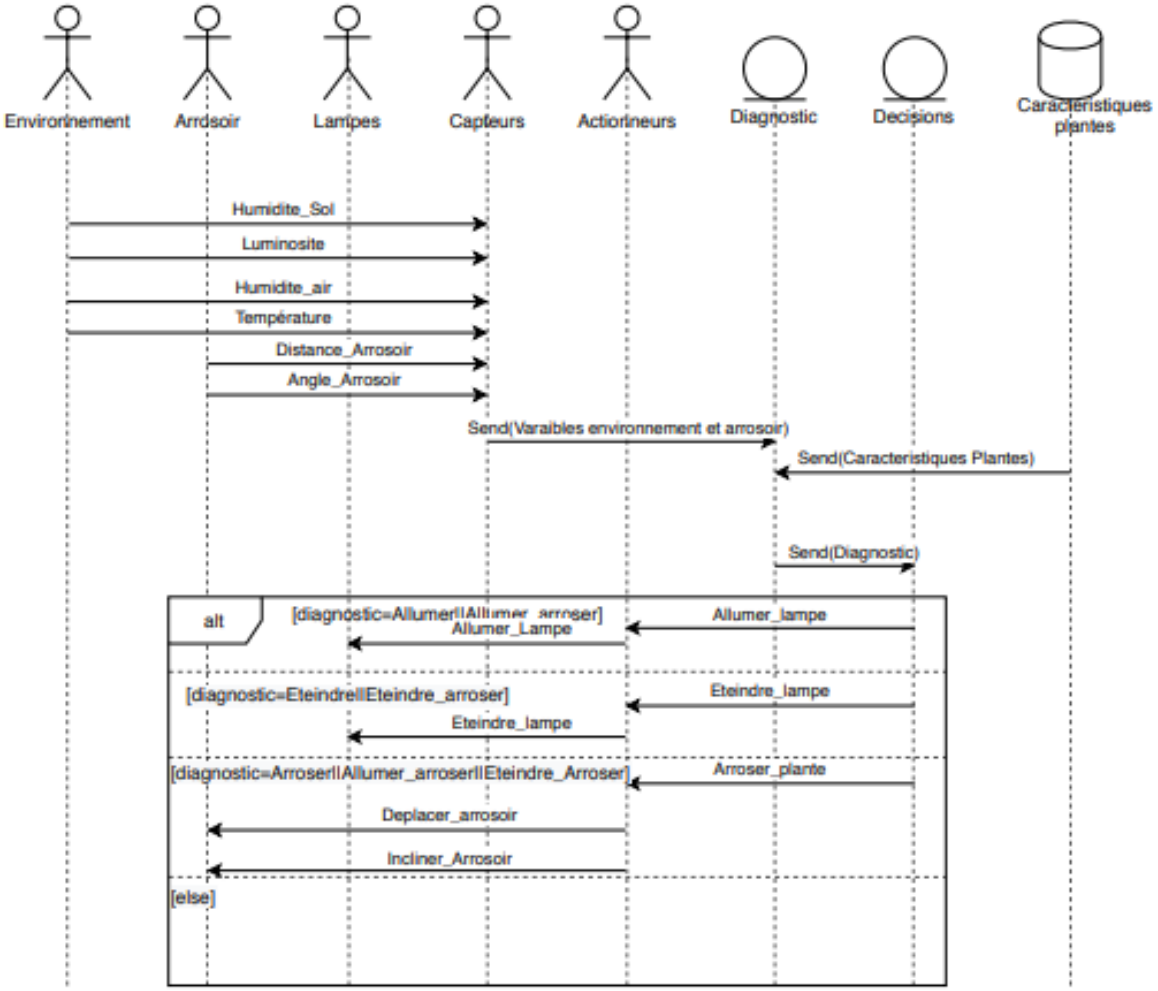


Figure 2: Diagramme de séquence de l'assistant de jardinage

3 Développement de la bibliothèque de capteurs C++

Pour notre projet d'assistant de jardinage, nous avions besoin d'un grand nombre de capteurs et d'un certains nombre d'actionneurs. Les capteurs sont essentiellement des capteurs pour mesurer l'environnement ambiant et sont tous analogiques. Les actionneurs quant à eux sont orientés vers l'arrosage et l'éclairage. Sur la page d'après, vous trouverez le diagramme de classe qui schématise l'arborescence des classes implémentées. Pour une meilleure lisibilité, vous pouvez également le retrouver sur notre Git.

3.1 Les capteurs analogiques

Une classe Sensor hérite directement de la classe Device et rajoute l'attribut temps qui correspond au délai entre chaque prise de mesure du capteur. La classe AnalogSensor hérite elle-même de cette classe et rajoute l'attribut aléa qui correspond aux variations dans l'exactitude des grandeurs mesurés. Ensuite, tous nos capteurs analogiques héritent

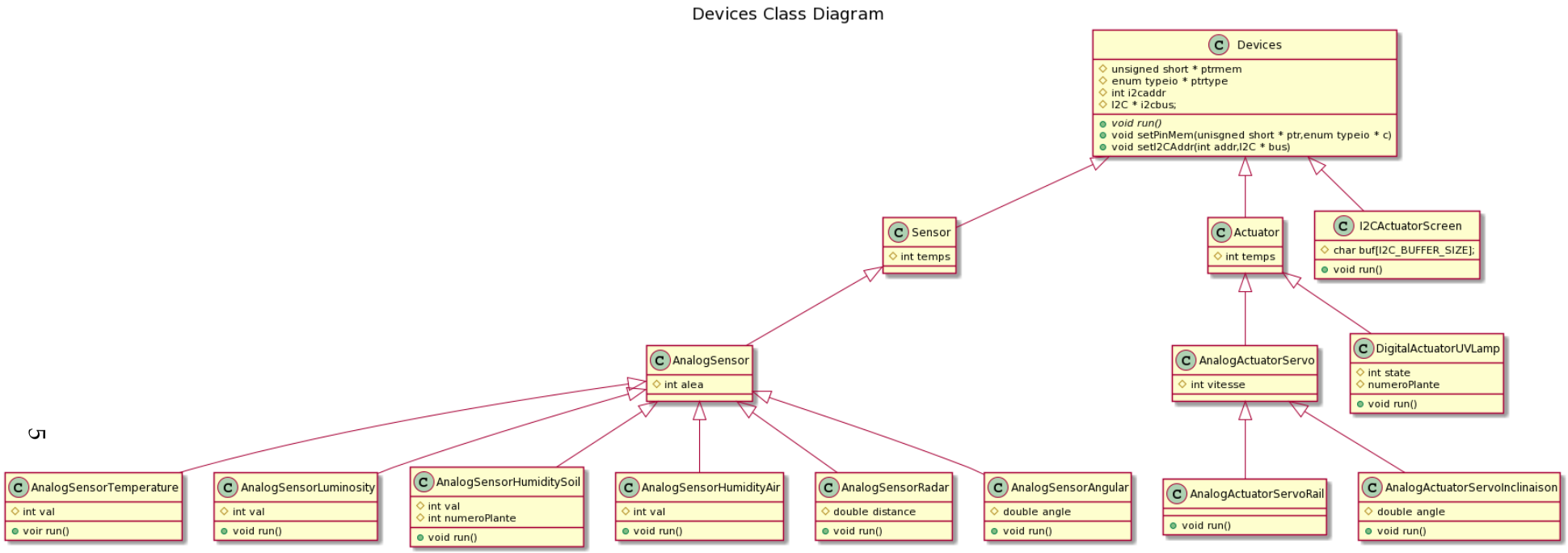
de cette classe avec chacun leurs particularités. Ils redéfinissent tous la fonction void run() d'une manière différente.

3.2 Les actionneurs

De même que pour les capteurs, la classe Actuator hérite de Device en rajoutant l'attribut temps. Nous distinguons ensuite deux actionneurs différents : Les servo-moteurs et la LED intelligente. La classe AnalogActuatorServo est la super classe qui régit les deux actionneurs que nous utilisons pour ce projet : les servo-moteurs gérant déplacement de l'arrosoir sur le rail et son angle d'inclinaison. Enfin la LED intelligente est tout simplement une LED qui augmente la luminosité ambiante lorsqu'elle est allumée.

3.3 Un petit mot sur les unités de mesure utilisées

Concernant les unitées de mesure de chaque capteur nous avons voulu faire au plus simple, ce qui n'est pas forcément le reflet de la réalité lorsqu'on utilise ce type de capteur. Par exemple,le capteur d'humidité dans l'air donne une mesure en % alors que le capteur d'humidité dans le sol donne une mesure en mV. Pour le capteur de luminosité, pour une journée ensoleillé, la réalité voudrait que l'on mesure 100000lux. Pour des raisons pratiques nous avons aussi réduit cette valeur.


Devices Class Diagram
Devices
unsigned short * ptrmem
enum typeio * ptrtype
int i2caddr
I2C * i2cbus;
void run()
void setPinMem(unisgned short * ptr,enum typeio * c)
void setI2CAddr(int addr,I2C * bus)
Sensor
int temps
Actuator
int temps
I2CActuatorScreen
char buf[I2C_BUFFER_SIZE];
void run()
AnalogSensor
int alea
AnalogActuatorServo
int vitesse
DigitalActuatorUVLamp
int state
numeroPlante
void run()
AnalogSensorTemperature
int val
voir run()
AnalogSensorLuminosity
int val
void run()
AnalogSensorHumiditySoil
int val
int numeroPlante
void run()
AnalogSensorHumidityAir
int val
void run()
AnalogSensorRadar
double distance
void run()
AnalogSensorAngular
double angle
void run()
AnalogActuatorServoRail
void run()
AnalogActuatorServoInclinaison
double angle
void run()

4 Développement de l'application : mise en place des fonctions haut-niveau

Dans cette partie, nous expliquons très simplement les algrithmes et fonctionnalités implémenté à un plus haut-niveau pour constituer le corps de notre application. L'application est divisée en deux grandes fonctionnalités : le diagnostiquage des plantes, et les actions à effectuer en conséquence.

4.1 Le diagnostic de l'état de santé des plantes

Comme on peut le voir sur le diagramme de cas d'utilisation, avant de poser le diagnostic, il faut mesurer tous les paramètres environnementaux. La fonction runDiagnostic() fait donc appel à measureHumiditySoil() (en précisant en argument le numéro de la plante qui est diagnostiquée), measureHumidityAir(), measureLuminosity(), et measureTemperature(). Une fonction d'affichage permet ensuite de visualiser ces informations sur le terminal. Ensuite, on applique l'algorithme de décision qui est très simple :

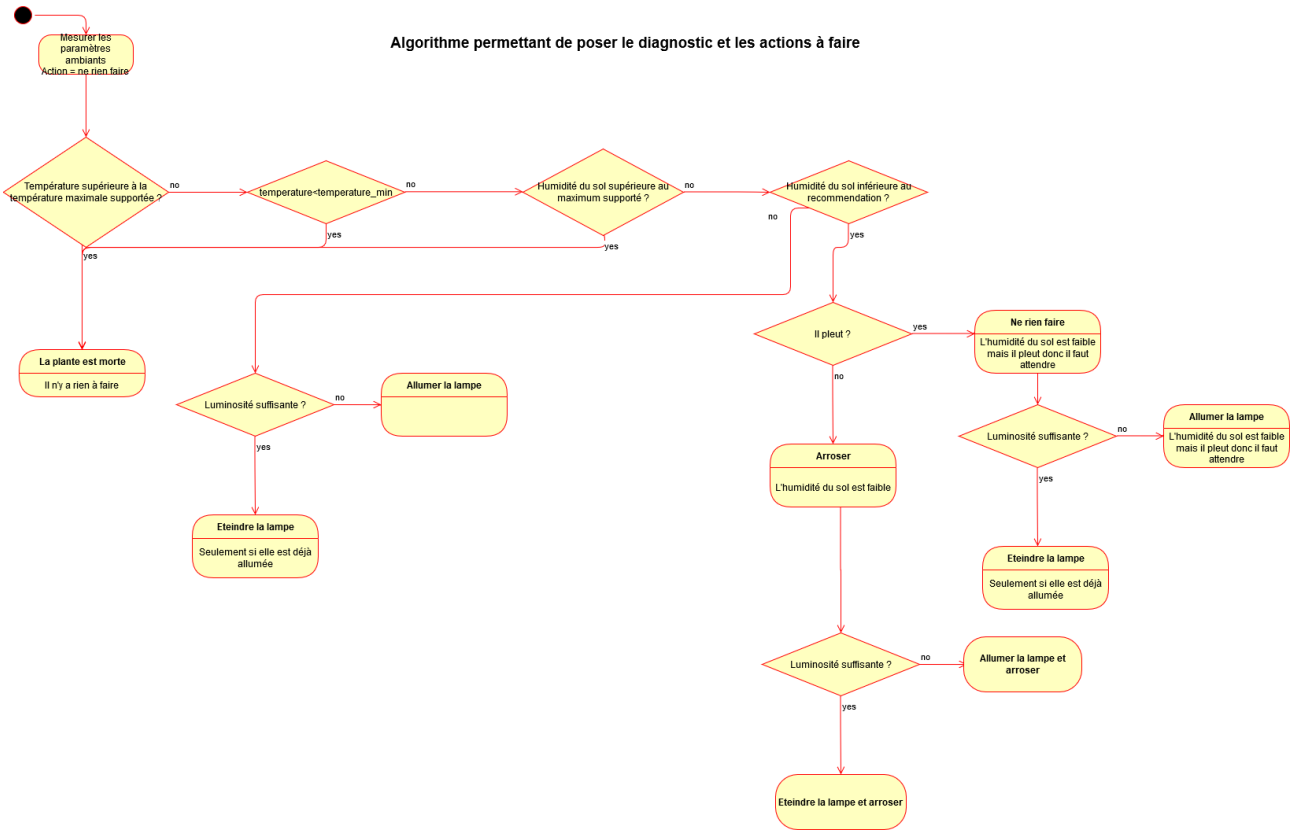


Figure 3: Déroulement de l'algorithme qui diagnostique la plante, à la manière d'un diagramme de machine à états

La fonction runDiagnostic prend en argument un objet de type caracteristiquePlante qui est elle même une classe fille de paramètrePlante et qui regroupe les conditions idéales d'entretien de la plante. C'est à partir de ces données-là que le diagnostic est effectué. La

fonction runDiagnostic() renvoie un int correspondant à une action. Ces actions sont listé dans un enum. Ensuite, il ne reste plus qu'à appliquer les recommendations.

4.2 L'arrosage et l'utilisation des lampes à UV

4.2.1 Gestion de l'arrosage

Au niveau de l'arrosage, grâce au use case en figure 1, nous pouvons identifier deux actions principales lorsque nous voulons arroser une plante : déplacer l'arrosoir et incliner l'arrosoir. Afin de gérer au mieux ces deux actions, nous mettons en place une classe `Arrosoir`, ayant la déclaration suivante :

```
protected:
    EtatArrosoir state;
    bool estArrivé;
    bool finArrosage;
 public:
    deplacerArrosoir(double positionPlante, Board* arduino)
    inclinerArrosoir(double positionPlante, Board* arduino)
    ArroserPlante(int numeroPlante, int humiditeVoulue,Board* arduino)
```

L'attribut state est du type énuméré EtatArrosoir, pouvant être égal à `Arrete`,`EnDeplacement` ou `EnArrosage`. C'est cet attribut qui permet de "coordoner les tâches" lors d'un arrosage. En effet, on ne peut pas arroser la plante lorsqu'on déplace l'arrosoir. Nous mettons donc en place l'algorithme présent en figure 4.

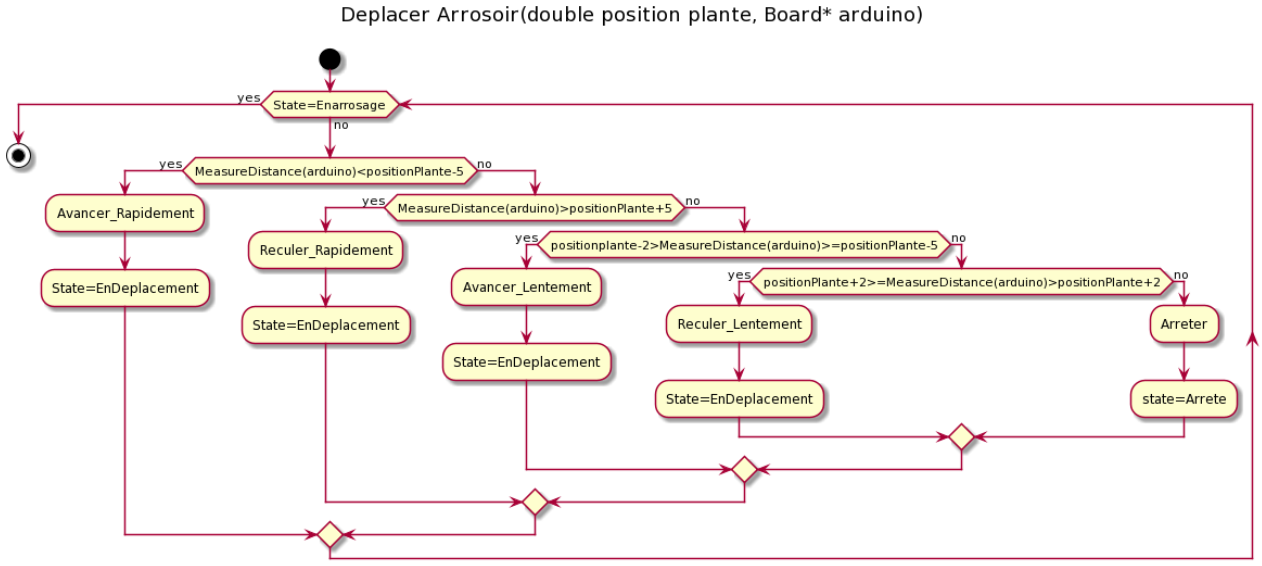


Figure 4: Diagramme d'activité de la fonction DeplacerArrosoir

La fonction `inclinerArrosoir` a la même structure que la fonction `DeplacerArrosage`. Nous procédons ainsi : si on veut arroser la plante, on incline de 45 deg l'arrosoir; si on

arrête d'arroser, on met l'angle à 0 deg.
Enfin, la fonction `ArroserPlante` va utiliser ces deux dernières fonctions pour emmener tout d'abord l'arrosoir a la distance correspondant à la plante. La variable booleenne `estArrive` va alors passe à true, permettant de rentrer dans la boucle inclinant l'arrosoir. A la fin de l'arrosage, `finArrosage` passe à true et on peut alors arrêter le processus d'arrosage.

4.2.2 Gestion des lampes à UV

La gestion des lampes à UV est particulièrement simple. A chaque plante correspond une lampe à UV, s'allumant si la luminosité est trop faible et s'éteignant sinon.

4.3 La coordination des actions à effectuer

Tout d'abord, on peut remarquer que les décisions concernant les lampes à UV n'ont pas besoin d'être coordonnées, puisque chaque action agit sur des capteurs indépendants. Cependant, au niveau de l'arrosage, n'ayant qu'un arrosoir pour les différentes plantes, il est primordial de gérer les décisions prises par la fonction de diagnostic. Pour cela, nous mettons en place un ensemble `set` de plantes à arroser (de classe `CaracteristiquesPlantes`). Afin de les coordonner, nous implémentons un attribut `priorité` dans la classe `CaractéristiquesPlantes` . Cet attribut représente quelle plante est le plus en manque d'eau. Ainsi, nous appliquons les actions d'arrosage une à une, en commençant par les plus prioritaires.

4.4 Simulation de l'environnement extérieur

Notre projet étant entièrement simulé, et pour avoir des scénarios interessants, nous avons simulé l'environnement dans lequel évolue notre jardinière connectée. Nous disposons de plusieurs variables globales qui régissent cet environnement. Grâce à une procédure JournéePrintemps(), nous pouvons modifier ces paramètres suivant si c'est le jour ou la nuie et ajouter une probabilité de pluie. On peut ainsi simuler le dessechement de la motte de terre s'il ne pleut pas, ou au contraire augmenter l'humidité si c'est le cas. Ces modifications restent très rudimentaires et servent uniquement à la demonstration de notre application : les variations de températures, de luminosité, d'humidité sont loin d'être représentatives de la réalité.

5 Conclusion : Discussions et améliorations

Ainsi, malgré les conditions particulières de la mise en place de ce projet, nous avons pu développer une application orientée objet, basée sur simulateur arduino. Ce projet nous a ainsi permis de consolider nos acquis en C++ ainsi qu'en conception orientée objet. Au niveau de notre application, le résultat est plutôt satisfaisant puisque le système réagit comme convenu. Une piste d'amélioration serait par exemple de généraliser ce système à

un grand nombre de plante. De plus, travaillant en simulation, nous sommes conscients que les résultats trouvés dans cette simulation ne fonctionneraient pas sur un matériel réel puisque les données reçues par les capteurs ne correspondent pas à des données qui proviendrait d'un véritable environnement. Ainsi, une seconde piste d'amélioration serait de mener une étude afin que la simulation se rapproche d'un véritable environnement. De plus, tel quel, notre application laisse la lampe allumée toute la nuit... Il serait possible d'ajouter une dimension écologique et économie d'energie avec une utilisation raisonnée des ressources. Enfin, une dernière piste serait de basculer toute la programmation en temps réel afin de pouvoir mieux gérer le partage de donnée, avec des mutex ou des semaphores, pour avoir une vision plus continue sur l'ensemble du système.