Rapport UML – Projet IOT

Sommaire

Dans ce rapport de projet nous allons vous présenter les diagrammes UML de notre projet de Semestre 3 autour de l'étude de l'IOT à travers le système d'exploitation Contiki et le simulateur Cooja. Comme étude de cas nous avons choisi d'étudier une simple utilisation de l'IOT dans les chaines d'approvisionnement. Afin que vous compreniez mieux notre analyse, nous allons vous expliquer notre projet.

Le but de notre projet est de proposé une solution, en utilisant l'IOT (Internet of Things, tout ce qui tourne autour des objets connectés), pour savoir combien il y a de produits dans un entrepôt. Pour ce faire, nous devons déployer des objets connectés utilisant le système d'exploitation Contiki, qui compterons le nombre d'objet qui rentre dans l'entrepôt et le nombre d'objet qui en sorte. Evidemment, nous ne pouvons pas réaliser les tests avec de réels capteurs, c'est pour cela que nous utilisons le simulateur Cooja, qui va nous permettre de réaliser des simulations de notre entrepôt.

Dans ce rapport, nous allons alors vous présenter notre analyse UML des outils que nous avons utilisés.

Dans un premier temps, nous étudierons différents diagrammes de classes. Le premier est celui d'un mote, qui est un objet connecté de manière générale que nous pouvons utilisés. Le second concerne un SkyMote, qui est un mote particulier qui correspond à ceux que nous utilisons en entrepôt. Le dernier est la représentation de l'entrepôt.

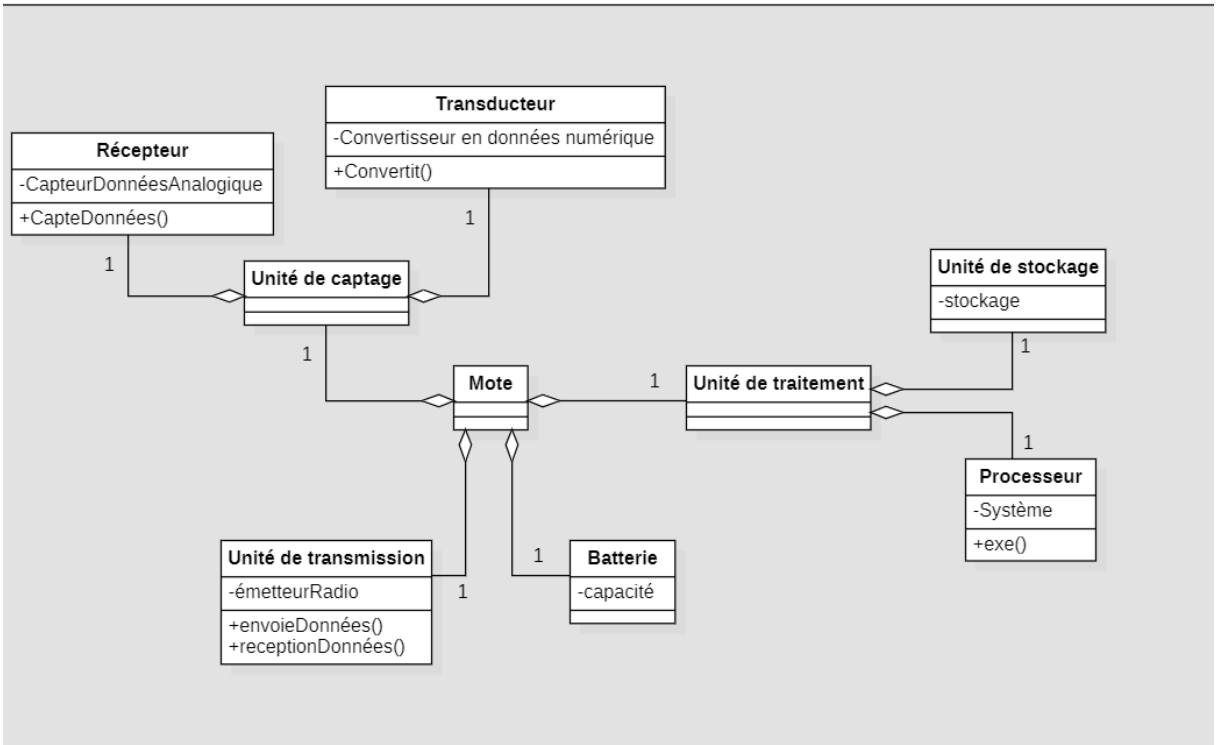
Puis, nous analyserons plusieurs diagrammes de séquences. Le premier est l'envoie d'un message entre le client et le serveur, qui utilisent le protocole RPL. Le second est autour du Simulateur Cooja et le dernier notre étude de cas de l'entrepôt.

Ensuite, nous allons voir les différents états du simulateur et d'un capteur à travers les diagrammes d'états-transitions.

Pour finir, nous étudierons 5 patterns, le singleton et le factory pour modéliser la construction d'un mote. Le décorateur et l'adaptateur pour expliquez plus en détails la structure d'un mote. L'observer pattern pour analyser le comportement d'un SkyMote.

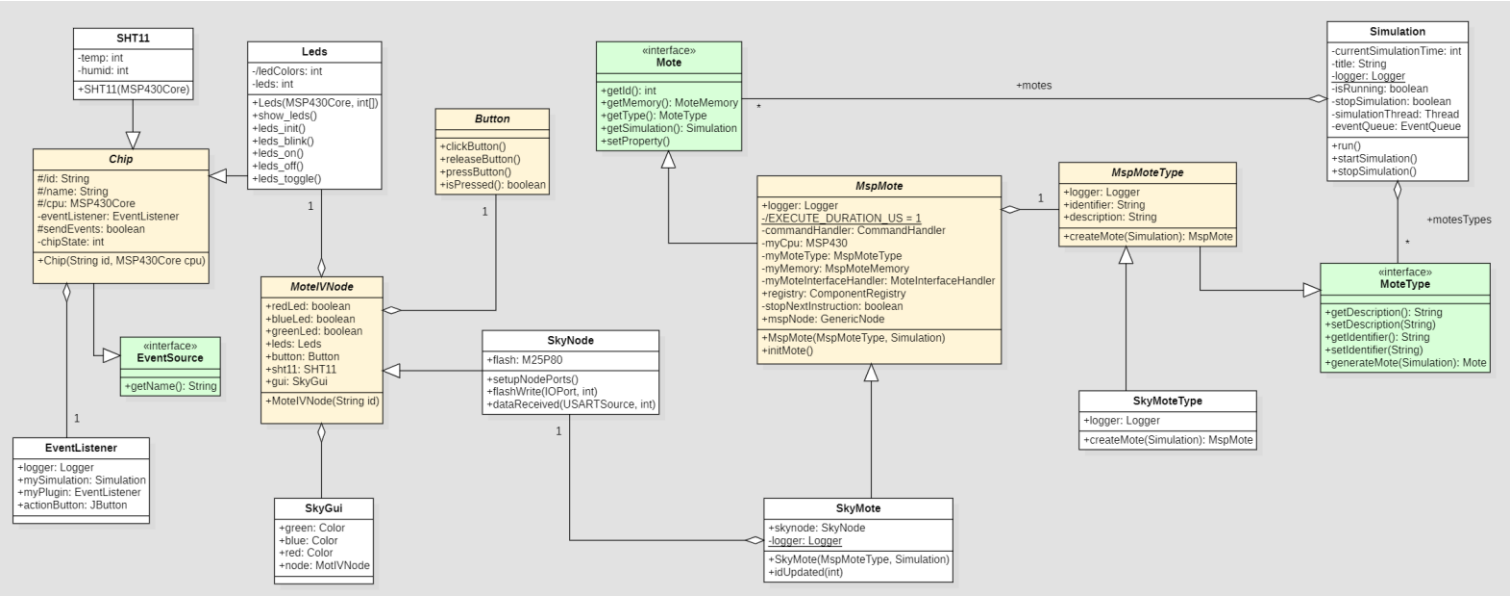
Diagramme de Classes

Une Mote



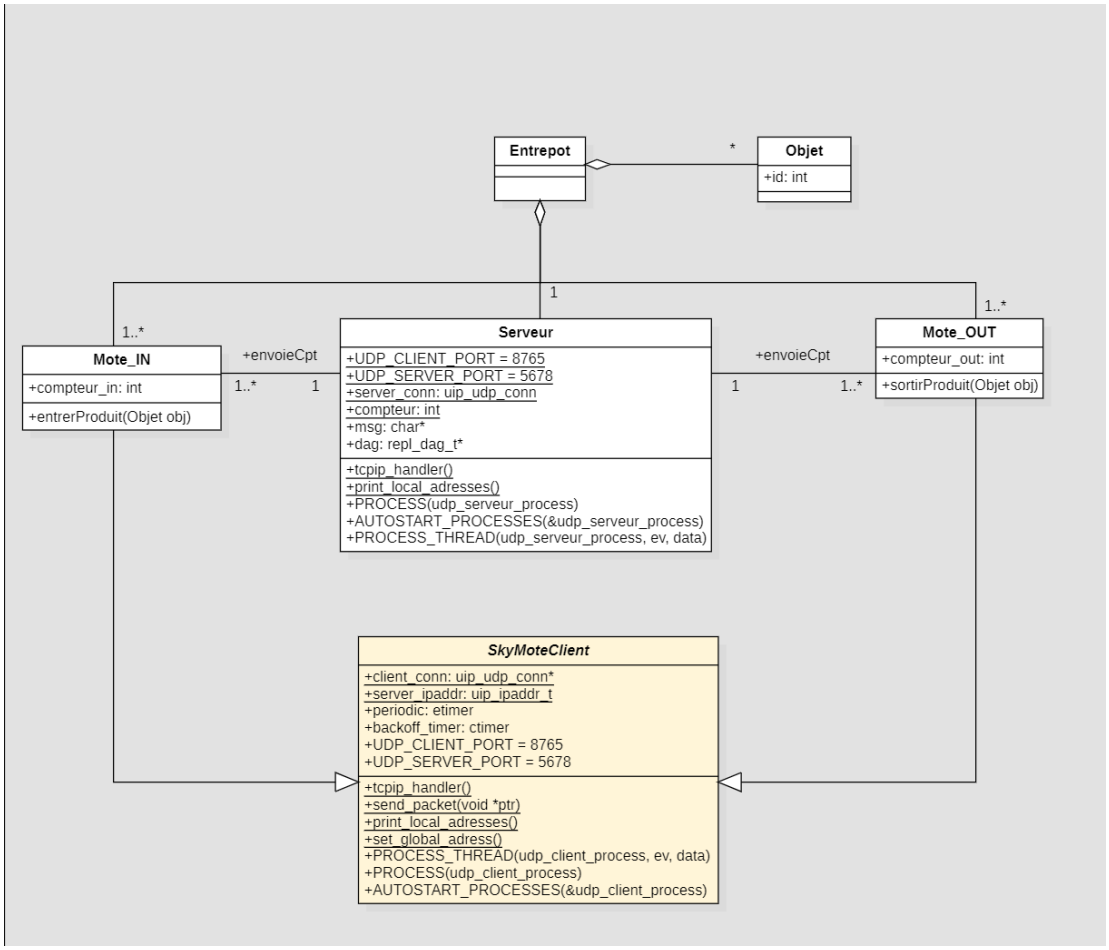
Dans ce diagramme de classe, décrivant un mote, on remarque qu'un mote est composé de 3 unités principales et d'une batterie. L'unité de transmission correspond à la partie de la mote qui réceptionne et envoie les données. Cette unité est équipée d'un émetteur radio. Ensuite, l'unité de captage, comme son nom l'indique, pour capter les données. Cette section est décomposée en 2 parties, le récepteur qui a un capteur de données analogique et le transducteur qui convertit les données analogiques en données numérique. Pour finir, l'unité de traitement est elle aussi divisé en 2. L'unité de stockage est l'endroit où est stocké la mémoire du mote. Le processeur exécute le protocole réseau pour communiquer avec les autres motes, il doit avoir un système Contiki.

Le simulateur et une SkyMote



Afin de réaliser ce diagramme nous nous sommes aidés des classes existantes sur Contiki. De ce fait, nous avons fait un tri dans les méthodes et les attributs pour éviter le superflu. Un SkyMote est un mote que nous utilisons sur Cooja pour modéliser l'étude de cas. Un SkyMote est composé d'un logger, qui est un enregistreur d'événement, et d'une SkyNode, qui correspond à tous les dispositifs liés aux capteurs, d'où les méthodes dataReceived et celles en rapport aux Ports. Cet SkyNode possède un processeur M25P80 et est une spécialisation d'un MoteIVNode. La généralisation gère les événements liés aux leds, buttons, températures et humidité, le gui est l'interface graphique de la node. Les leds possèdent plusieurs méthodes afin de les utiliser. Ces méthodes permettent de les faire clignoter, allumer ou éteindre. On peut également cliquer sur un bouton pour produire un événement. Le MoteIVNode est également composé d'un dispositif nommé SHT11, ce dispositif est celui qui gère le capteur de température et d'humidité. Les leds et les SHT11 sont des spécialisations d'une puce possédant une id, un processeur et un système événement. Pour revenir au SkyMote, il est une spécialisation d'un MspMote. Celui-ci possède les composants d'un mote que nous avons étudié précédemment, il s'agit alors d'un mote d'un point de vue générique. Il possède un type qui est lui aussi une généralisation d'un SkyMoteType. Les dispositifs MSP sont des implémentations de deux interfaces, encore plus générale, qui composent la simulation. Celle-ci possède également un logger et gère les événements des motes. On peut démarrer le simulateur et démarrer la simulation.

L'entrepôt



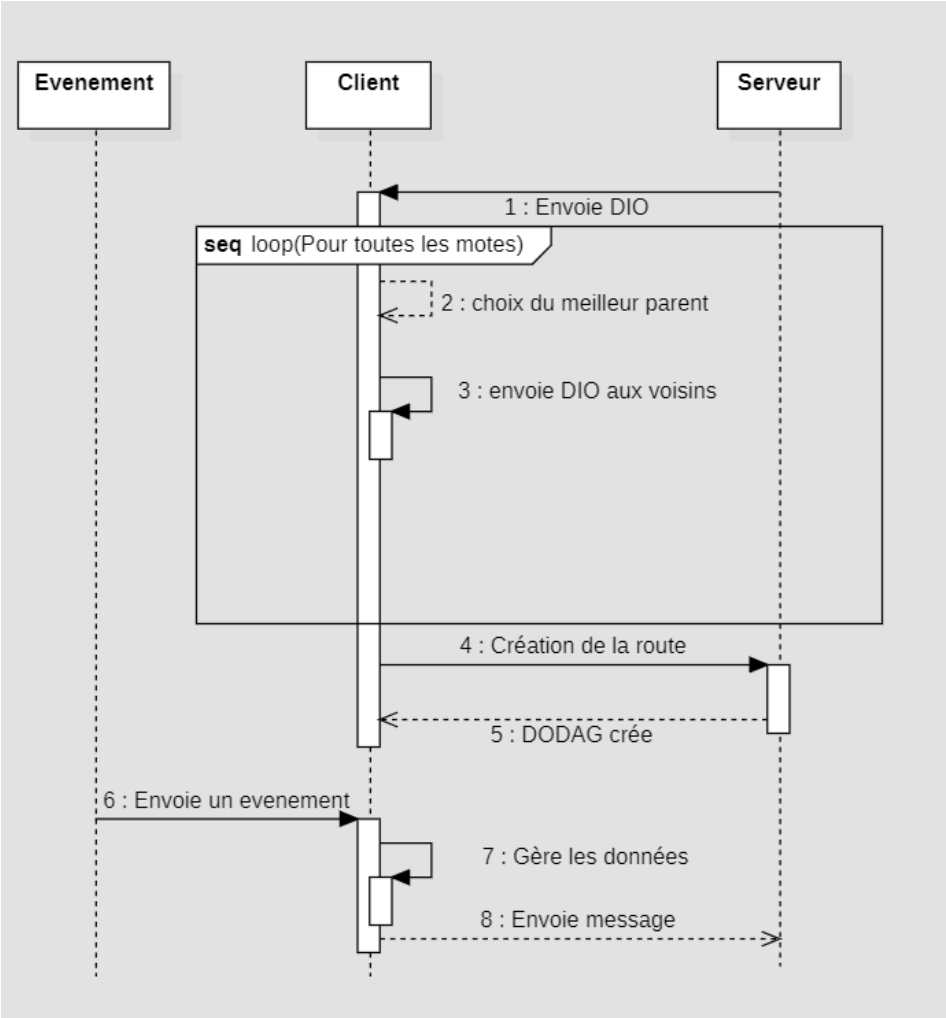
Pour réaliser ce diagramme, nous nous sommes aidés des programmes client et serveur que nous avons réalisé afin de représenter le système de stockage de l'entrepôt. Tout d'abord, nous avons la classe SkyMoteClient. Elle possède un attribut client_conn, la structure udp du client, server_ipaddr, la structure d'adresse du serveur. Nous avons ensuite deux autres attributs qui correspondent à des événements liés aux temps, et les Ports qui sont utilisés pour établir la connexion entre le client et le serveur. Ensuite, il y a différentes méthodes :

- Tcpip_handler : est celle utilisé pour la réception d'un message
- Send_packet : l'envoie d'un message
- Print_local_adresses : afficher son adresse IP
- Set_global_adresses : Etablie l'adresse IP

Les autres méthodes correspondent aux processus qui vont s'exécuter au démarrage de la simulation. SkyMoteClient est une généralisation des classes Mote_IN et Mote_OUT. Les deux classes correspondent aux capteurs qui se situent respectivement à l'entrée et à la sortie de l'entrepôt. A chaque fois qu'un objet rentre ou sors le compteur est incrémenté ou décrémenté. Cette gestion du compteur est faite dans le serveur. Le serveur ressemble beaucoup à la classe client. La méthode tcpip_handler ne va pas être identique, et le processus aussi, dans celui-ci le DODAG va être construit. Il s'agit d'un algorithme lié au protocole RPL. Il permet d'établie une route entre un client et un serveur. Il assure également une topologie dynamique du réseau, lorsque qu'un mote est ajouté, déplacé ou enlevé. Le serveur affiche après le total d'objets dans l'entrepôt.

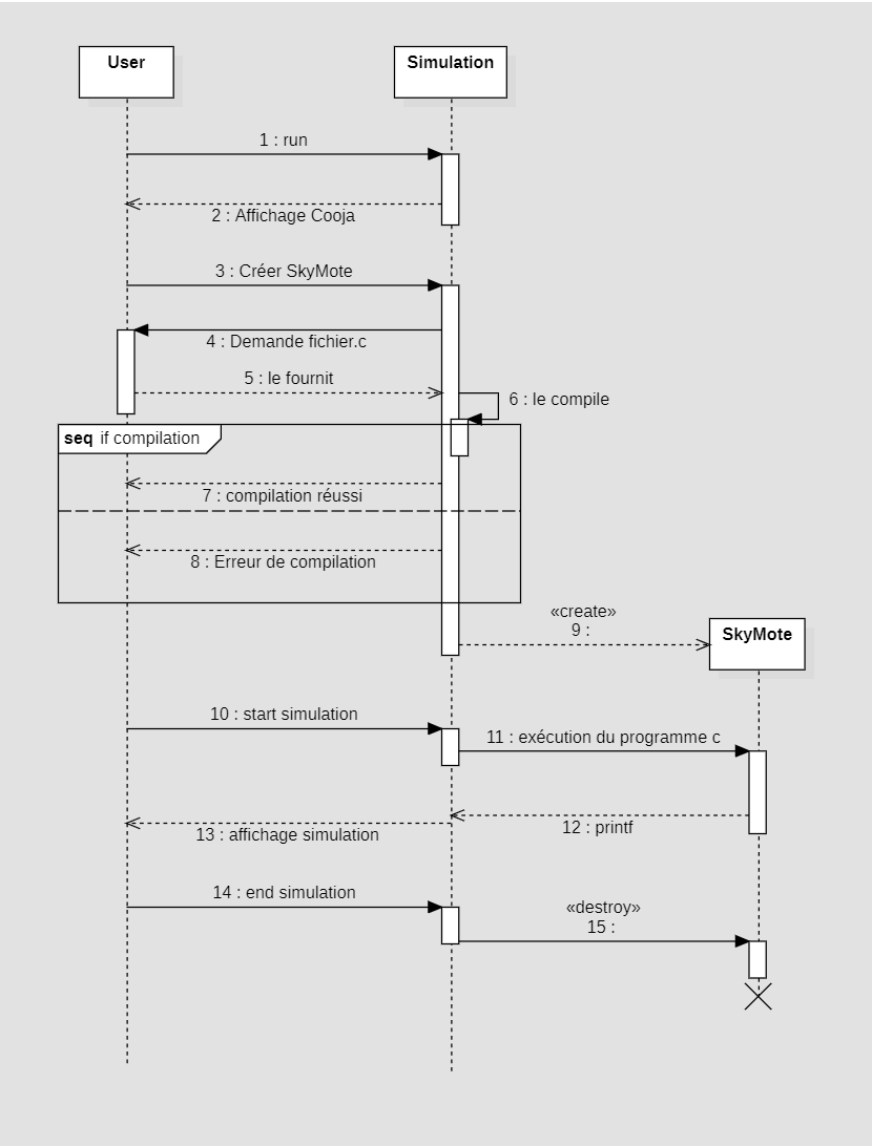
Diagramme de Séquence

Envoie d'un message à travers le protocole RPL



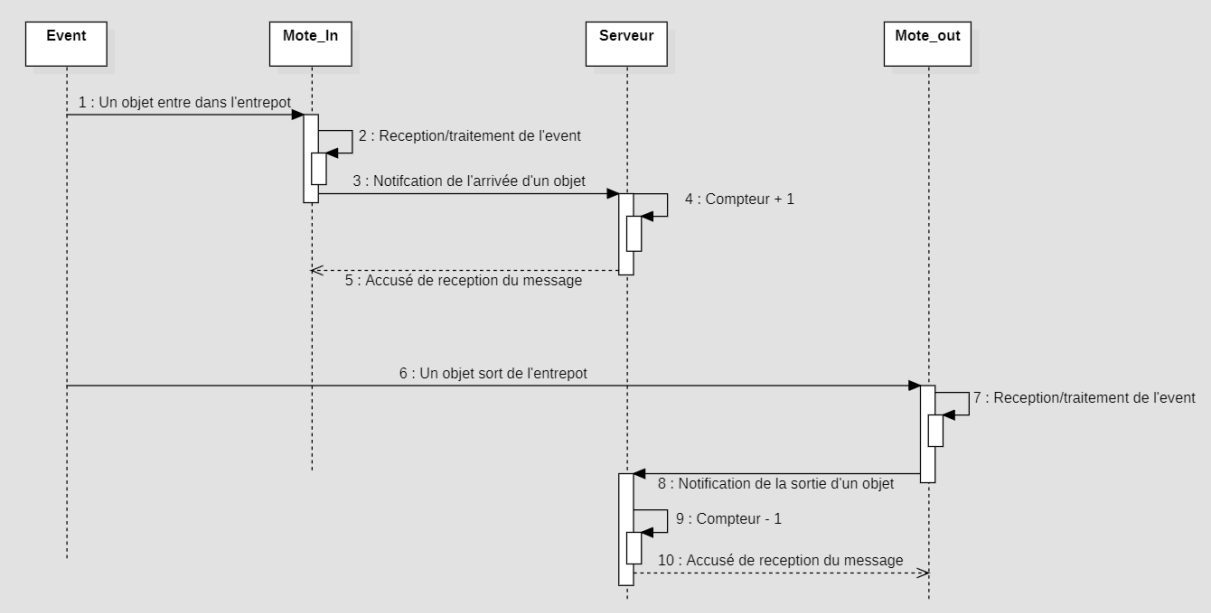
Ce diagramme de séquence décrit l'envoie d'un message entre le client et le serveur à travers le protocole RPL. Lorsque les processus s'exécutent, le DODAG se crée afin d'établie la connexion entre le client et le serveur et pour cela, nous devons passer par différentes étapes. Pour construire le DODAG, le serveur envoie à tous ses voisins un message, dis DIO (DODAG Information Object). Ce message contient l'identifiant du DODAG et le rang du nœud qui diffuse le message. Plus le client est loin du serveur plus son rang est élevé. Pour choisir son meilleur parent, le client choisit un autre client ayant un rang inférieur, ou directement le serveur s'il peut. Le parent est le nœud par lequel les paquets seront envoyés jusqu'au serveur. Le choix s'effectue en fonction du nombre de sauts minimum, la qualité de la communication et la batterie restante. Ensuite, il envoie d'autres DIO à ses voisins pour maintenir le DODAG. Une fois que ses étapes sont effectuées sur toutes les motes, les routes sont créées et le DODAG aussi. Pour finir, lorsqu'un événement est envoyé alors le client gère la réception et envoie un message au serveur.

Le simulateur



Ce diagramme décrit le processus d'exécution du simulateur. Ici nous avons un utilisateur qui décide de démarrer Cooja. Ensuite il demande à Cooja de créer un SkyMote. Pour cela, le simulateur a besoin d'introduire un programme c dans la mote. L'utilisateur le fournit et si le simulateur arrive à le compiler, il crée le SkyMote, sinon non. Une fois le SkyMote crée, l'utilisateur peut démarrer la simulation. Lorsque la simulation est en cours de démarrage, les programmes C s'exécutent. Le simulateur Cooja affiche alors les printf du programme C. Lorsque l'utilisateur met fin à la simulation alors le SkyMote est détruit.

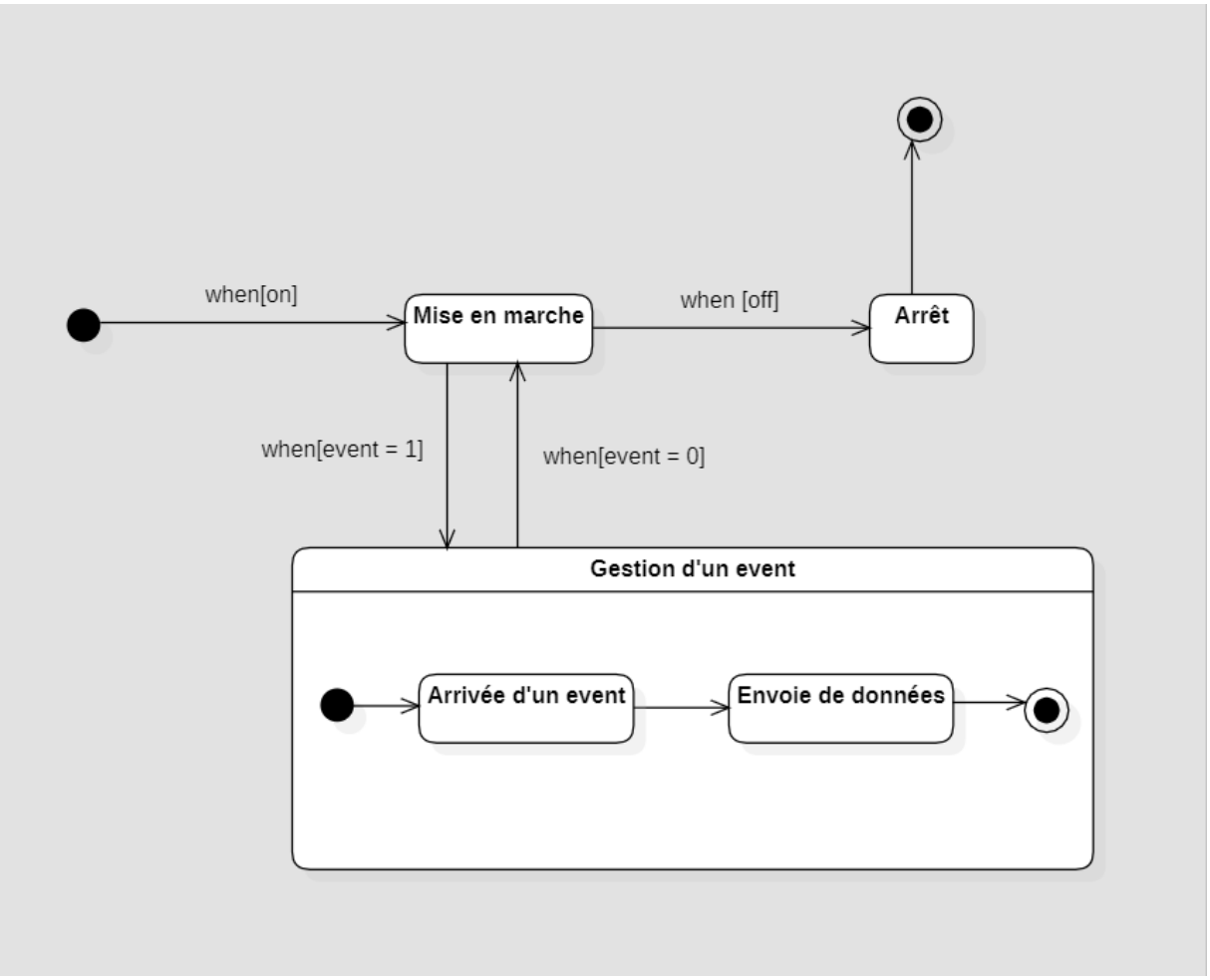
L’entrepôt



Ce diagramme décrit l’arrivée d’un objet et la sortie d’un objet dans l’entrepôt. Pour ce diagramme on suppose qu’un DODAG est déjà fonctionnel. Tout d’abord, nous avons un événement décrivant l’arrivé d’un objet d’un entrepôt. Le capteur situé à l’entrée de l’entrepôt, Mote_IN, réceptionne alors l’événement, le traite à l’aide des différentes méthodes que nous avons vu précédemment et envoie une notification au serveur de l’arrivée d’un objet. Le serveur alors incrémente le compteur du nombre d’objet de 1 et envoie un accusé de réception au mote. Ensuite, il y a un autre événement mettant en scène la sortie d’un objet de l’entrepôt. Le capteur situé à la sortie de l’entrepôt, Mote_OUT, réceptionne alors l’événement, le traite tout comme le Mote_IN et notifie la sortie d’un objet au serveur. Le server alors décrémente de 1 le compteur du nombre d’objet dans l’entrepôt et envoie un accusé de réception au mote.

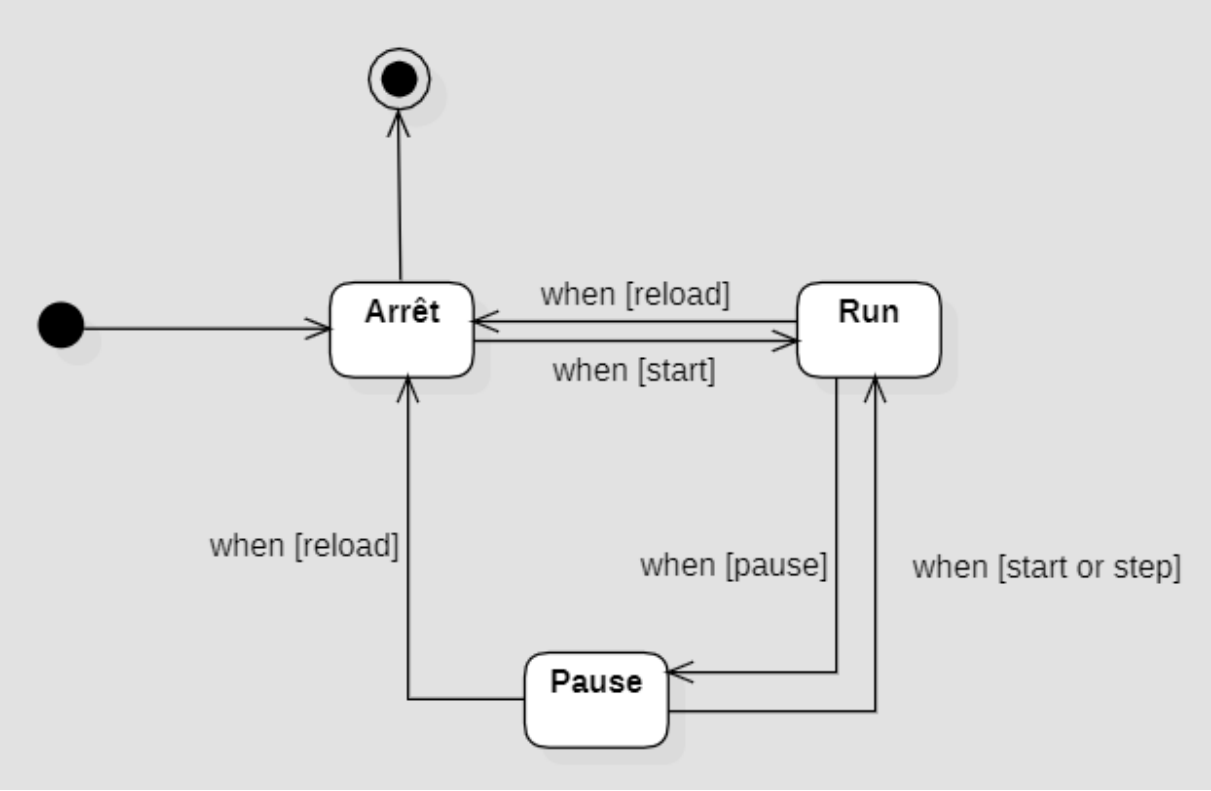
Diagramme d’états-transitions

Capteurs



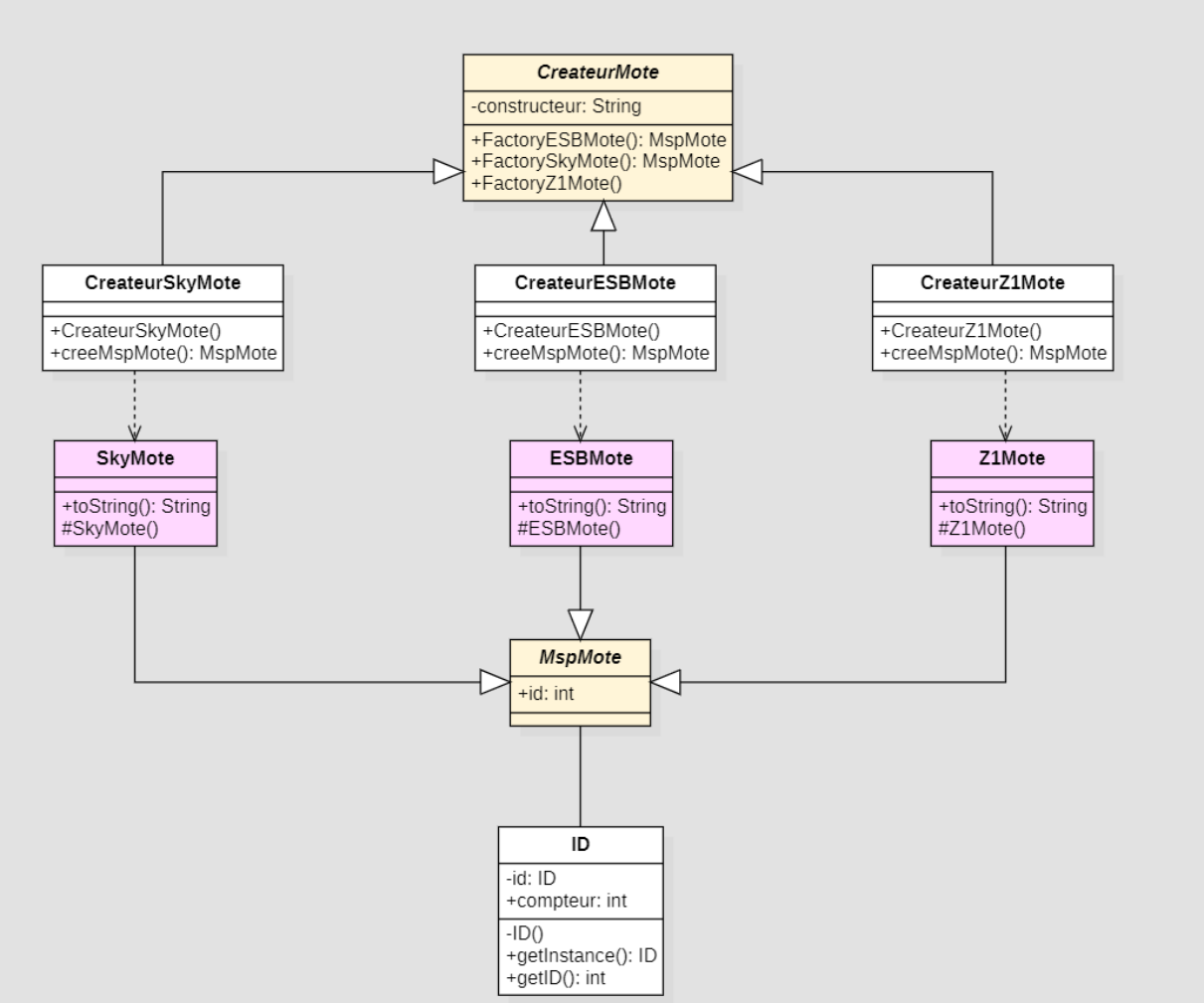
A l’aide du diagramme d’Etats-transitions, nous avons réussi à modéliser les états d’un capteur. Tout d’abord nous avons le bouton on du capteur qui met en marche le capteur. Ensuite lorsqu’un événement est réceptionné alors le capteur entre dans l’état « Gestion d’un event » qui gère l’arrivée d’un événement et l’envoie des données, une fois que l’événement est traité, le capteur revient dans l’état « Mise en marche ». L’utilisateur peut arrêter le capteur s’il appuie sur le bouton off de celui-ci.

Le simulateur



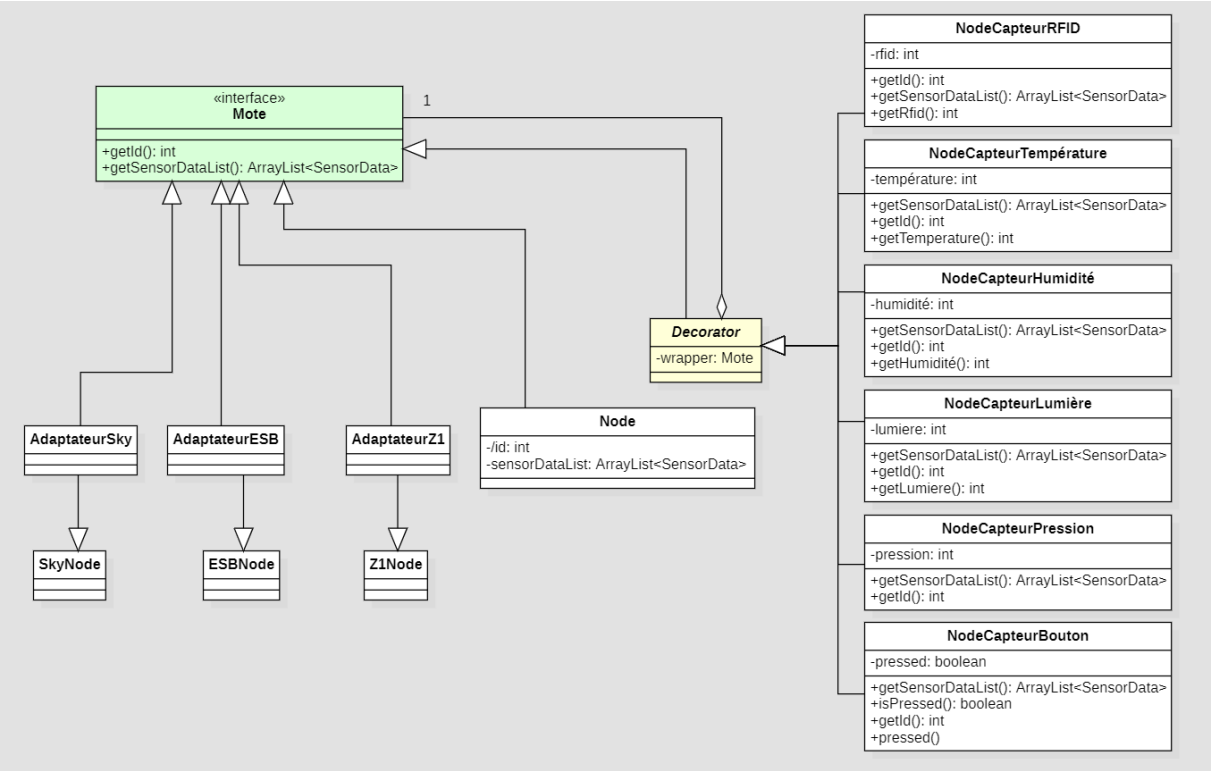
Ce diagramme d'états-transitions, permet d'analyser les différents états liés au simulateur Cooja. Lorsque que nous lançons Cooja et que nous avons créé un mote, alors il est possible de lancer la simulation, mais au départ elle est dans l'état « Arrêt ». Reload permet de remettre la simulation à son état initiale, cette fonction est réalisable à partir de n'importe quel état autre que l'état initial. Start est utilisé afin de passer la simulation de l'état « Arrêt » à « Run ». Il est possible de mettre en pause la simulation avec la fonction pause. Pour finir, nous pouvons faire avancer lentement la simulation avec la fonction step.

Factory / Singleton pattern



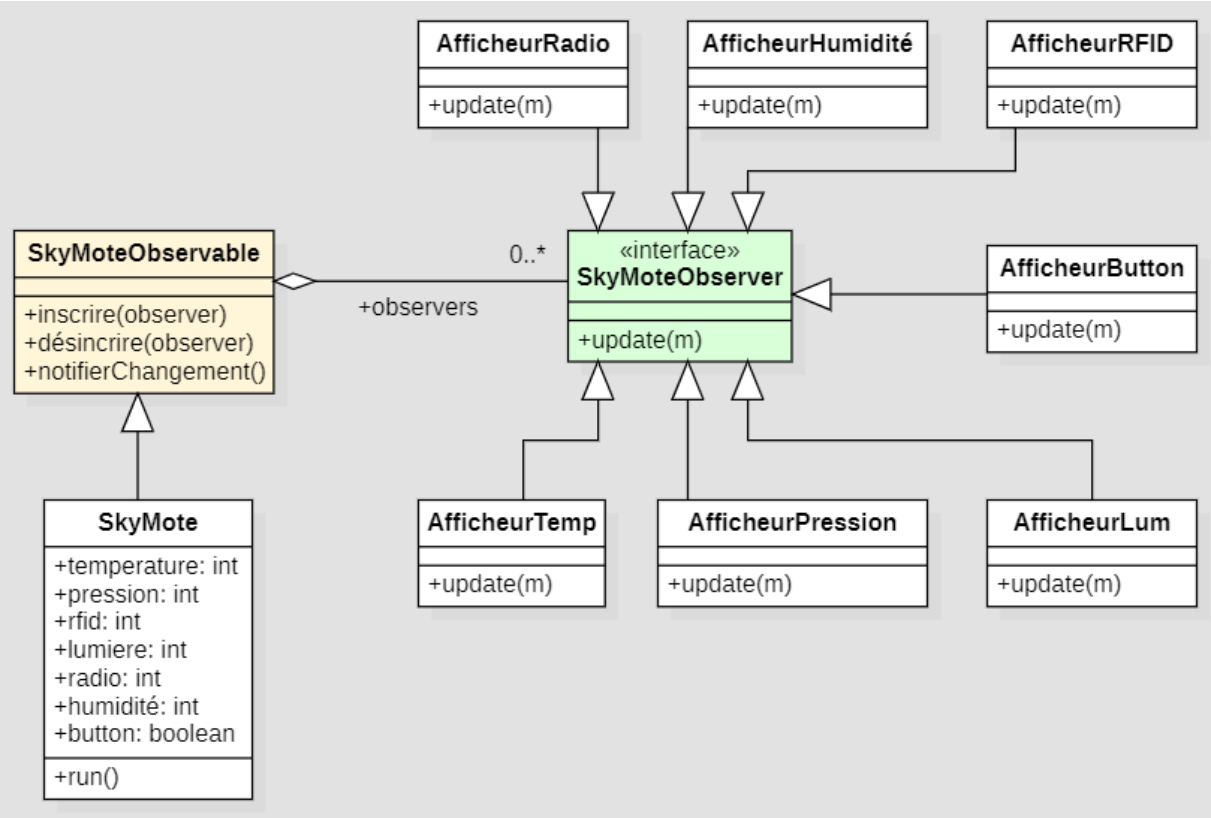
Le singleton et factory pattern sont des patrons de constructions. Ils sont fréquemment utilisés pour déléguer à d'autres classes la construction des objets. Afin de modéliser la construction des SkyMotes, nous avons utilisés le factory pattern et le singleton pattern. Le patron factory permet d'isoler la création des objets, de découpler les classes concrètes de leur utilisateur et de définir un constructeur virtuel. Dans notre exemple, nous l'utilisons afin de créer 3 motes différents, le SkyMote, l'ESBMote et le Z1Mote. Dans notre étude de cas, nous utilisons le SkyMote, mais nous souhaitons montrer qu'il en existe d'autre. Nous avons vu précédemment, dans notre diagramme de classe concernant le SkyMote, que celui-ci et le MspMote possède d'autres attributs et méthodes, mais nous n'avons pas jugé nécessaire de les afficher dans ce diagramme. Le seul attribut que nous avons jugé nécessaire à montrer est l'id car chaque mote durant la simulation possède un identifiant unique. Nous avons donc réalisé un patron singleton gérant les id des motes.

Decorateur / Adaptateur pattern



Le patron décorateur et adaptateur sont des patrons de structurations. Ils tendent à concevoir des agglomérations de classes avec des macro-composants. Nous avons utilisé ces deux patrons afin de modéliser comment une node est structurée. Dans un premier temps nous avons le patron décorateur sur la partie droite du diagramme. Il permet d'ajouter de nouveaux comportements. Dans notre cas, il s'agit de capteurs. Il est possible de « décorer » une node avec 6 sortes de capteurs (en réalité plus mais nous avons décidé d'en modéliser seulement ces 6 -là), le RFID, Température, Humidité, Lumière, Pression, Bouton. A chaque ajout de capteur la liste des capteurs (SensorDataList) est mise à jour. Il est possible d'ajouter ces 6 capteurs sur une node de manière générale, mais également sur une SkyNode, une ESBNode et une Z1Node. Pour le montrer, nous avons réalisé le patron d'adaptation, il permet aux objets avec des interfaces incompatibles de collaborer. Dans notre exemple, il ne nous est pas possible de travailler avec un mote au propre sens du terme, nous somme obliger de passer à travers des motes spécifiques, qui ne fonctionnent pas tous de la même manière. Cependant, pour rendre ce diagramme plus compréhensible par un technicien ou par une personne n'ayant pas de nombreuses compétences dans ce domaine, nous avons décidé de ne pas détailler cet aspect.

Observer pattern



Un patron observateur est un patron de comportement, il tend à répartir les responsabilités entre chaque classe afin de proposer un usage dynamique. Un patron observateur permet de définir un mécanisme d'abonnement pour informer plusieurs objets de tout événement qui se produit sur l'objet qu'il observe. Dans notre cas l'objet que nous observons est un SkyMote. Nous avons vu précédemment, que nous pouvions étudier différents événements à l'aide de capteurs. Nous avons donc pensé qu'il était possible d'établir un patron d'observation sur ces événements. La classe SkyMoteObservale possède une liste « d'observers » en fonction des capteurs qu'il possède, dans notre exemple il possède les 6 présentés précédemment. Lorsque qu'un événement lié à la température arrive, c'est alors AfficheurTemp qui met à jour la température. Au lieu que tout se fasse dans une même classe, et qu'il y ai un couplage fort lors de l'ajout d'un capteur. D'autant plus que Contiki fonctionne à quelques détails près comme cela. Un mote ne possède pas de capteurs prédéfinis, ils sont ajoutés au fur et à mesure en fonction des besoins.