I.U.T Clermont-Ferrand

Projet JavaFx

Documentation

Sensor Manager

Élèves :
Yannis Mahiou
Benoit Louveau

Enseignant :
Laurent Provot

Décembre 2017 / Janvier 2018

1 Contexte De L'application

Sensor Manager est une application de gestion de capteurs, permettant à l'utilisateur de créer ses propres capteurs et de les manipuler.

L'utilisateur, une fois un capteur créé, peut choisir de lui faire générer des températures factices selon des algorithmes pré-conçus : l'utilisateur peut générer une température de manière aléatoire, par interval de valeurs et finallement par fenêtre glissante.

L'utilisateur peut ensuite afficher un des capteurs de la liste de capteurs, selon différents types d'affichages pré-conçus. Il est possible d'afficher la température d'un capteur de manière digitale, de manière iconique via des icones météorologique (soleil, nuage pluvieux, flocon de neige) ou bien de manière thermométrique via un thermomètre.

Il doit au préalable lancer la génération de temperature en appuyant sur le bouton on des capteurs de la liste. Ainsi, il lance la génération de température, et peut l'arrêter à tout moment.

Par ailleurs, il est également possible de créer des super-capteurs. Un super-capteur est un regroupement de capteurs basiques. La température de ce super-capteur est calculée comme étant la moyenne pondérée de chaque sous-capteurs qu'il possède. Ainsi, l'utilisateur peut choisir par drag and drop un capteur de la liste pour l'ajouter à la liste de sous-capteur du super-capteur.

Au final, l'utilisateur dispose d'une gestion organisée des capteurs, avec différentes possibilités de génération de température, d'affichage, et de création.

2 Description du Diagramme de Classe

2.1 Les packages et leur rôle

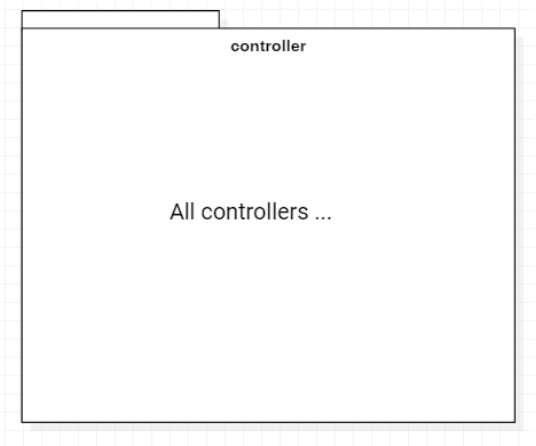
- **ihm** : Contient toutes les interfaces graphiques FXML
- **controller** : contient tous les contrôleurs qui gèrent les interfaces graphiques
- **business_logic** : contient toutes les classes métier
- **cellFactory** : contient des factories pour créer des cellules de composants JavaFX(ListCell)

2.2 Description textuelle

Partie IHM

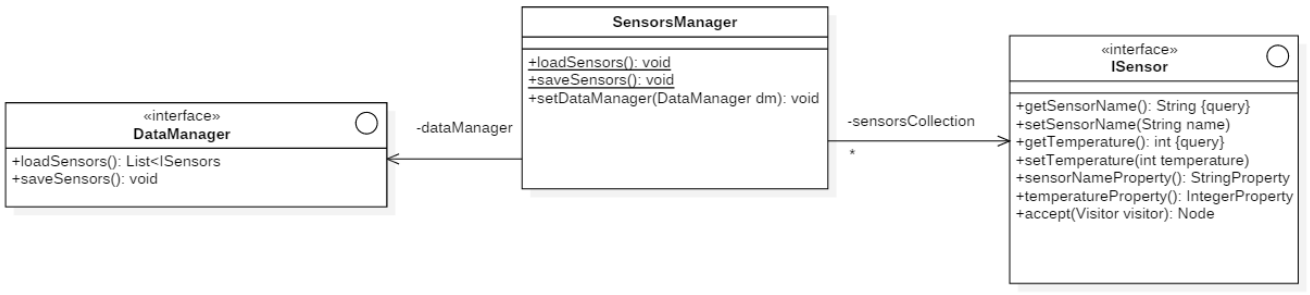
La partie ihm est composée des fichiers FXML qui déterminent l'aspect de l'application. Pour plus d'ergonomie, elles ont été reliées à un fichier CSS qui rend l'application plus agréable en y appliquant un style.

Partie Controller



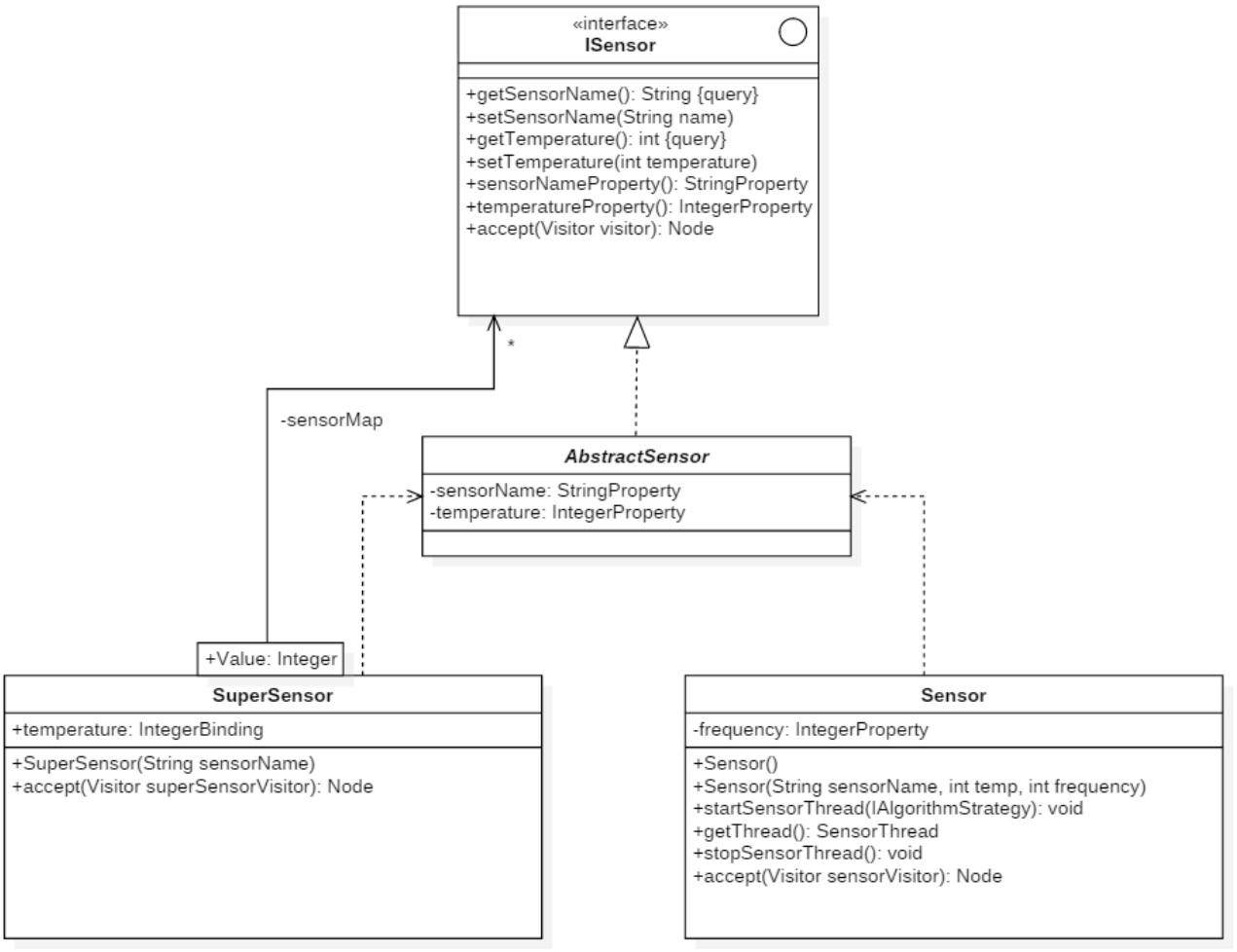
Ces fichiers FXML utilisent les données du modèle grâce à des contrôleurs. Ces derniers permettent d'assigner des méthodes à des composants graphiques afin de manipuler les données du modèles (affichage, suppression, modification, création). C'est grâce à ces contrôleurs que l'on va pouvoir binder des données du modèle et effectuer des actions dessus.

Partie Metier

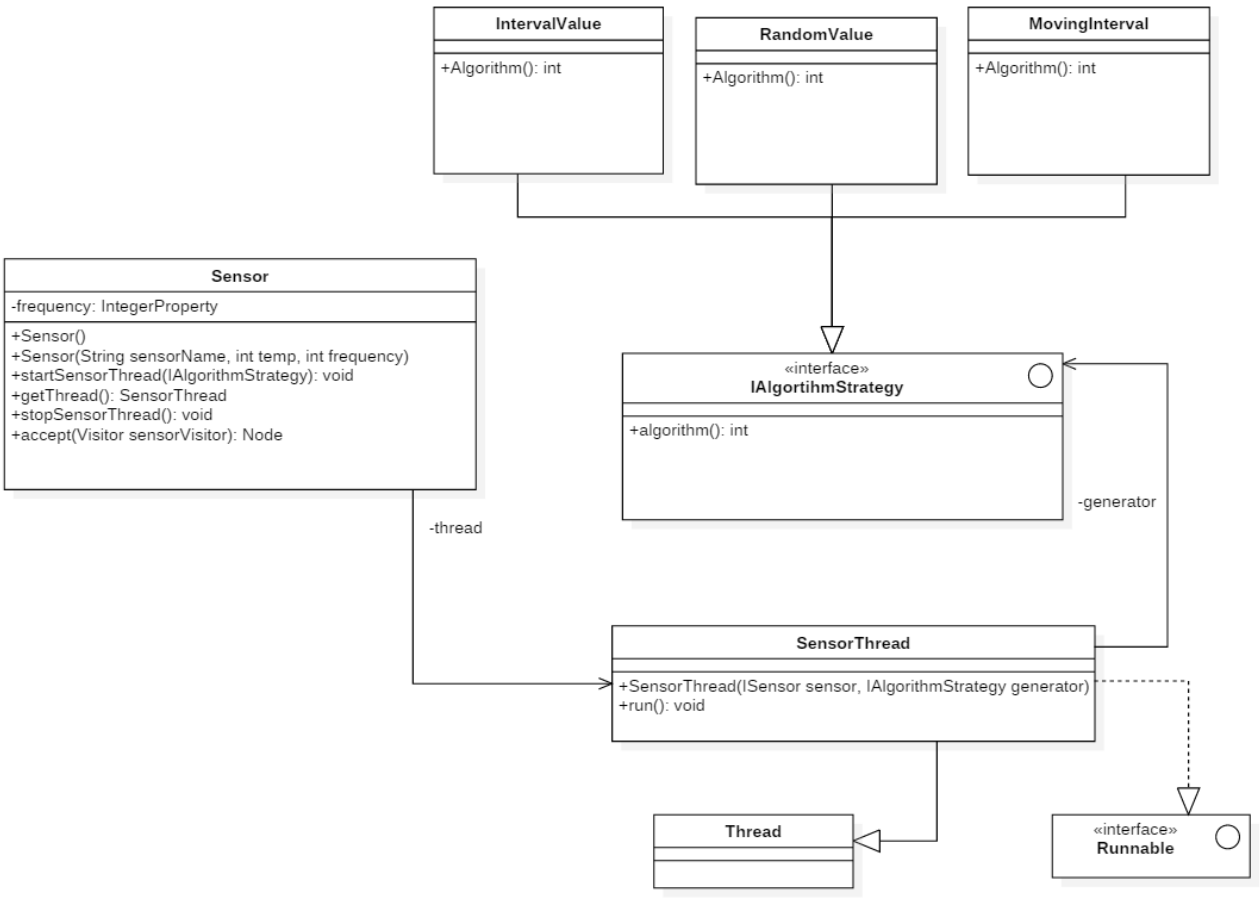


La gestion des Sensors ne se fait que par le SensorsManager, cela permet de manipuler l'ensemble du modèle par une seule classe dont c'est la responsabilité. Ce SensorsManager ne manipule que des ISensor. Ainsi, l'ensemble

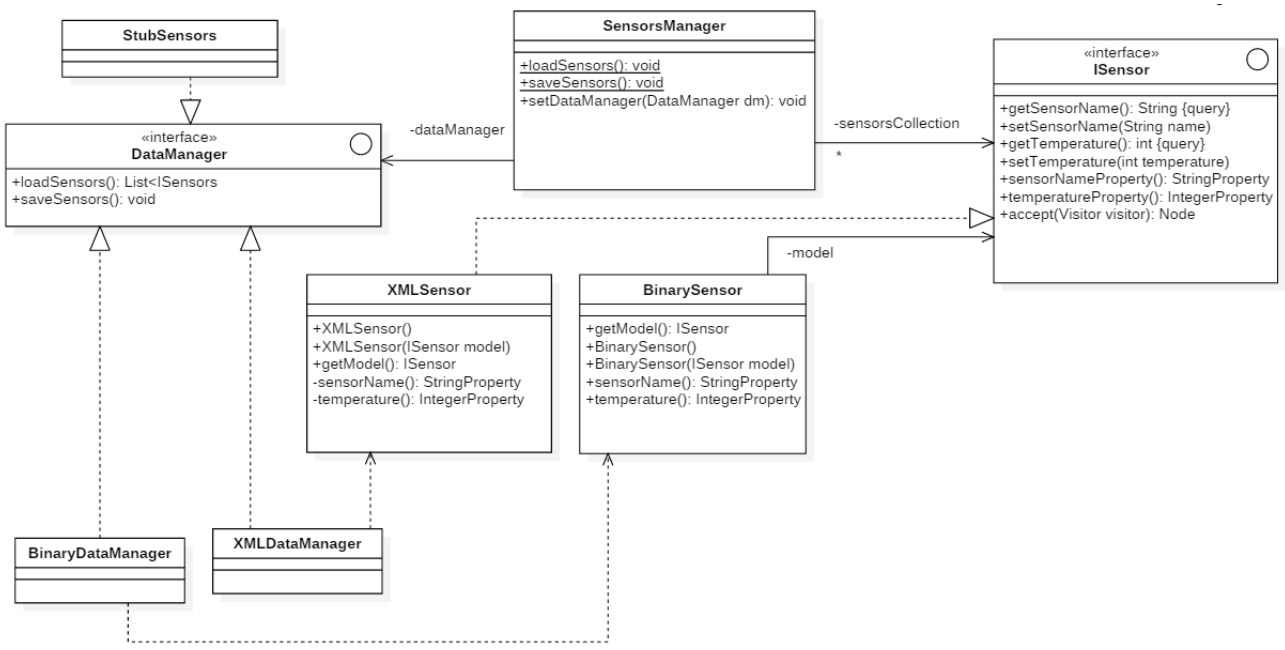
du métier est protégé car celui qui manipule les objets ne manipulera que la couche d'abstraction et n'ira pas modifier directement dans les couches inférieures. On garde ainsi cet esprit de boîtes noires utilisables et non modifiables.



Les capteurs sont modélisés de la manière suivante. On a une interface ISensor, qui sera manipulée par l'ensemble du projet. Cette interface est manipulée par l'ensemble du projet afin de protéger les classes Sensor et SuperSensor. Nous avons ajouté une classe abstraite AbstractSensor entre l'interface et les modules de bas niveau, dans le but de factoriser le code en commun entre Sensor et SuperSensor. Nous obtennons ainsi une structure qui est très facilement utilisable car on ne manipule que des ISensor, et d'un autre côté facilement extensible du fait de AbstractSensor qui implémente ISensor.



Chaque Sensor dispose d'un SensorThread. Le Thread va s'occuper de contrôler la temperature de ce capteur, en l'actualisant selon la frequency d'un Sensor. Le constructeur de SensorThread prend une IAlgorithmStrategy en paramètre. Cette interface est le point d'accès au différents algorithmes de génération de température que nous avons implémenté. Par extension, une nouvelle stratégie de génération de temperature peut être ajoutée dans le futur.



La persistance est gérée par un DataManager. Ainsi, peu importe le type de sérialisation choisi, il suffit de l'indiquer au DataManager. Pour mettre en

place un nouveau type de sérialisation, il n'y a plus qu'à faire une classe qui implémente DataManager et qui met en place son type de sérialisation. Ainsi, après avoir créé un stub, nous avons décidé de mettre en place une persistance XML en créant un XMLDataManager qui manipule un XMLSensor qui est Sensor sérialisable. D'un autre côté, nous avons mis en place une persistance Binaire en créant un BinaryManager qui manipule un BinarySensor. Nous disposons actuellement de 2 types de persistances. On peut bien évidemment rajouter une autre persistance dans le futur(JSON ...).

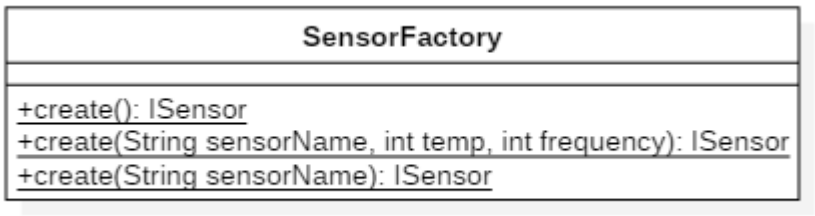
2.3 Les Principes S.O.L.I.D

Nous avons, tout au long du projet, cherché à respecter au mieux les principes S.O.L.I.D

S : Single Responsability Principle
Chaque classe doit avoir une responsabilité unique.

Appliquer ce concept permet de découpler les responsabilités pour que le changement d'une classe n'ait pas d'impact sur plusieurs responsabilités.

Nous avons appliqué ce principe dans notre projet et chaque classe a une responsabilité unique :



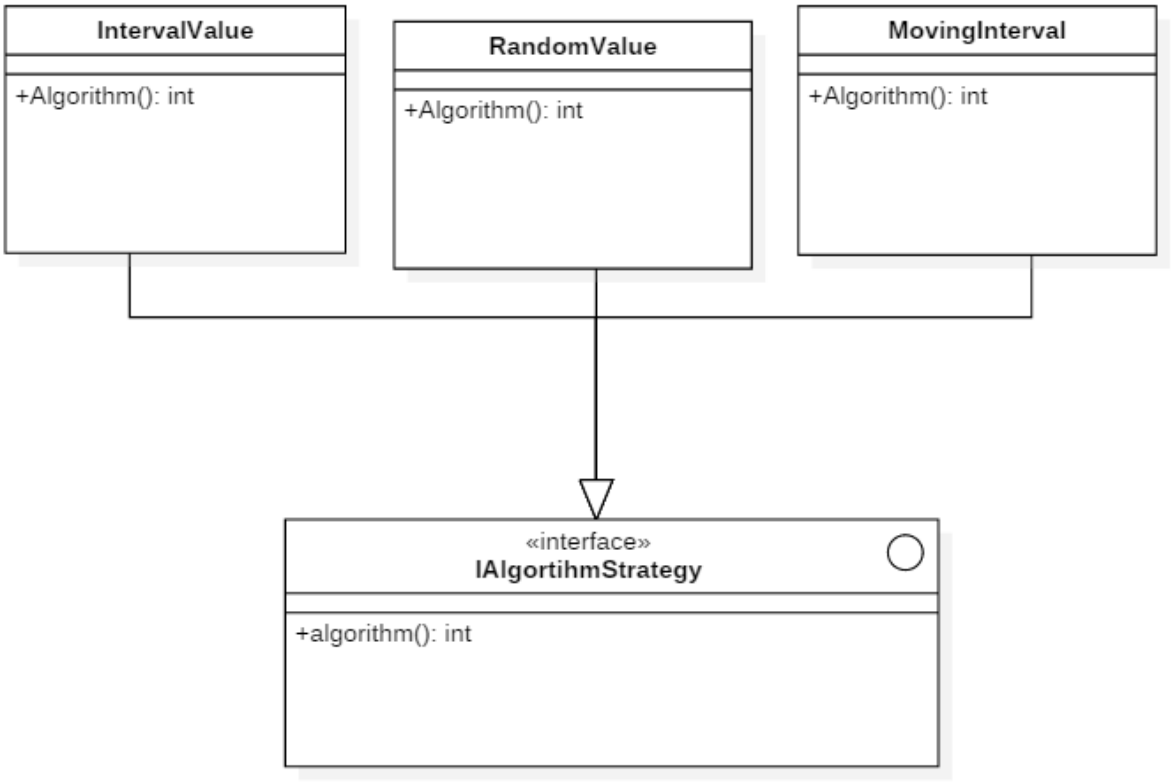
Par exemple, nous avons créé une factory qui n'a pour seule responsabilité que de créer des objets IUser. Cette responsabilité est unique et propre à cette classe. Ainsi, en cas de modification de cette factory aucune autre

responsabilité n'en sera affectée.

O : Open/Close Principle
On doit créer une application dont le modèle est ouvert à l'extension et fermé à la modification.

Appliquer ce concept permet de découpler les responsabilités pour que le changement d'une classe n'ait pas d'impact sur plusieurs responsabilités.

Nous avons appliqué ce principe dans notre projet et chaque classe a une responsabilité unique :

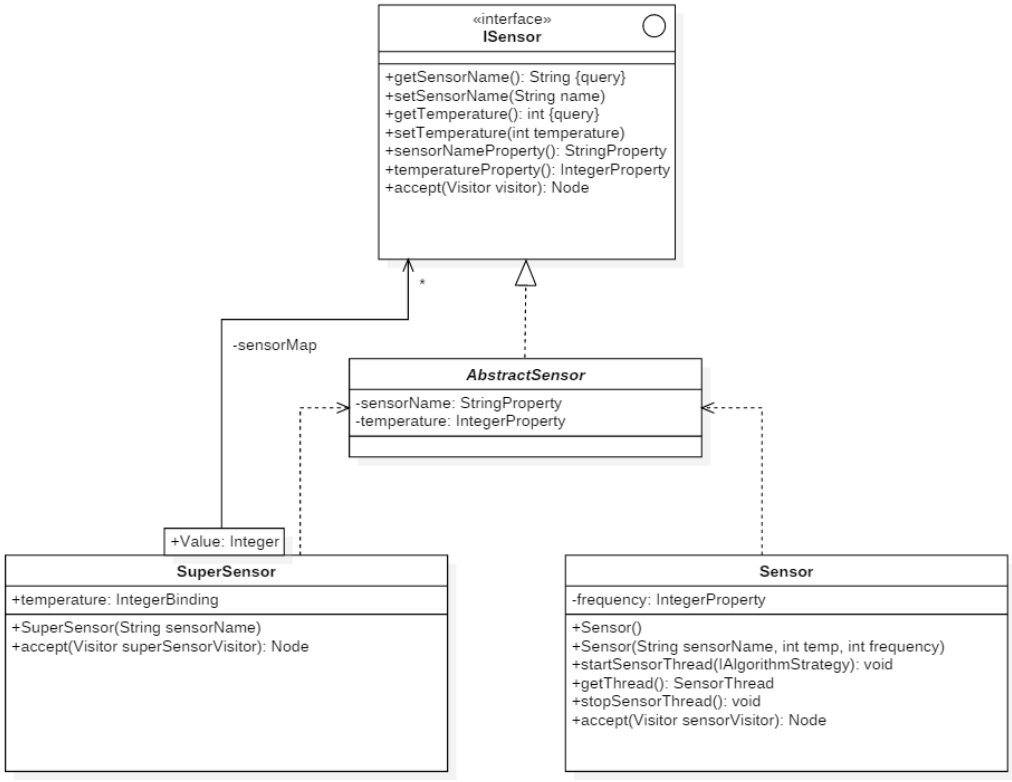


Pour réaliser la sérialisation, nous avons utilisé une interface DataManager. C'est cette dernière qui est manipulée dans le projet. Ainsi, peu importe le type de sérialisation utilisé, il suffit simplement de choisir si on veut utiliser le DataManager XML ou le stub. De plus, si on veut changer de type de sérialisation et implémenter une base de données ou une sérialisation binaire par exemple, aucune classe n'est modifiée mais le DataManager est étendu par héritage. L'application est alors beaucoup plus facilement maintenable.

L : Principe de la substitution de Liskov

Une fonction qui utilise un objet d'une classe mère doit pouvoir utiliser toute instance d'une classe dérivée sans avoir à la connaître.

C'est le principe qui permet de profiter énormément du polymorphisme et qui joue sur la substituabilité. Ainsi, on va chercher à placer les méthodes le plus haut possible pour que toutes les classes filles puissent y avoir accès.

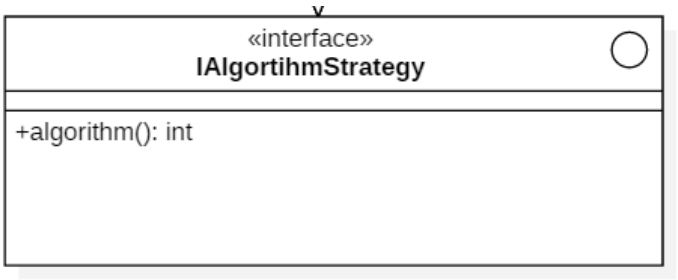


Ainsi, le maximum de méthodes sont remontées et s'appliquent sur tous les enfants. On peut alors manipuler un ISensor sans problème et sans savoir si ce dernier est un Sensor ou un SuperSensor.

I : Principe de ségrégation d'interface

Ce principe consiste à découper une interface en plusieurs petites interfaces si celle-là s'avère trop importante.

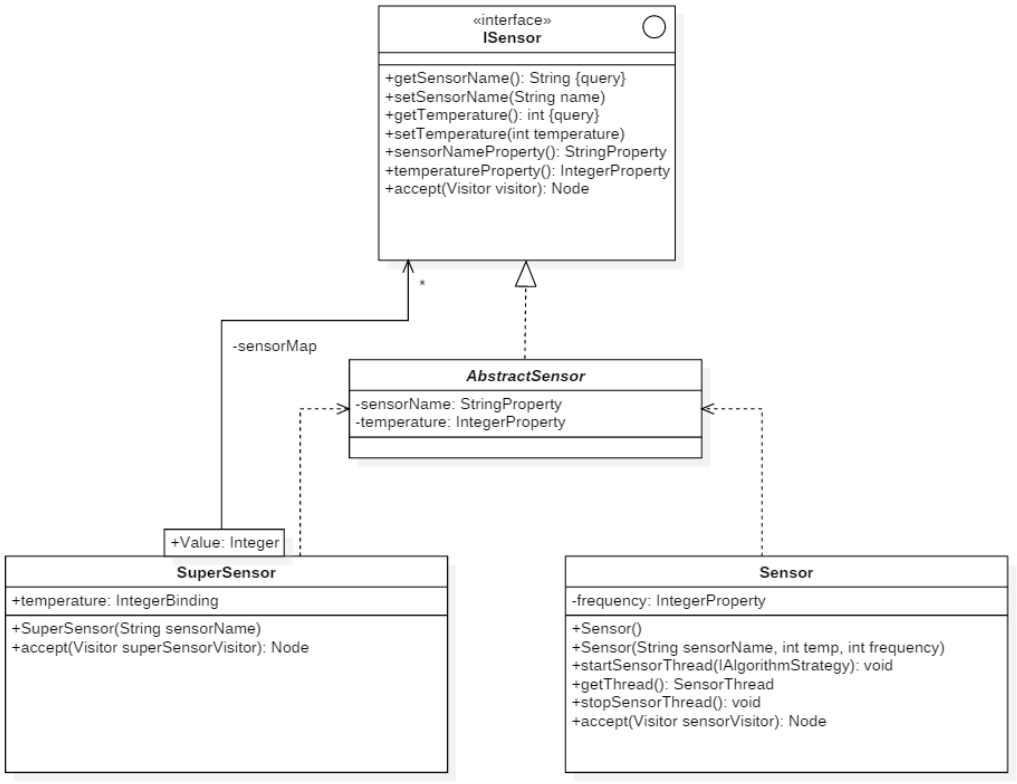
Nous ne l'avons respecté que pour notre interface IAlgortihmStrategy afin de respecter le principe d'Interface Fonctionnelle (une seule méthode par interface).



D : Principe d'inversion de dépendance

Les modules de haut niveau ne doivent pas dépendre des modules de bas niveau et les modules de bas niveau doivent dépendre d'abstraction.

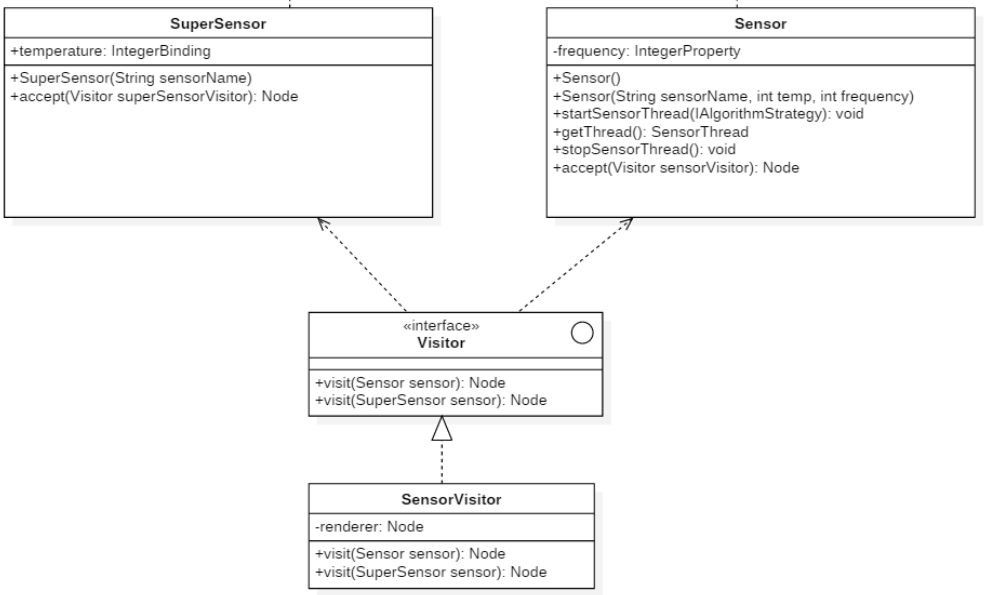
Nous avons essayé de respecter au mieux ce principe, en voici un exemple :



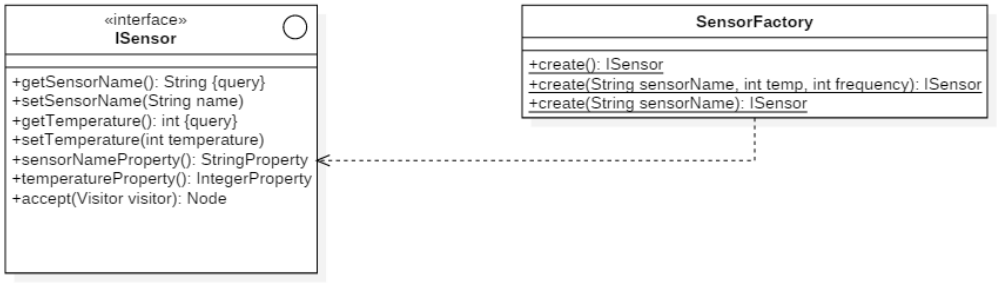
Il y a un contrat passé entre l'interface ISensor, Sensor et SuperSensor, par l'intermédiaire de la classe abstraite AbstractSensor. Ainsi, de l'extérieur du modèle on ne manipule que des ISensor pour protéger les classes de bas niveau qui dépendent d'abstraction. Les module du bas dépendent donc de 2 couches d'abstraction. On crée ainsi une boîte noire et l'on peut utiliser des ISensor sans savoir ce qu'il se passe derrière.

3 Justification des Patrons de Conception

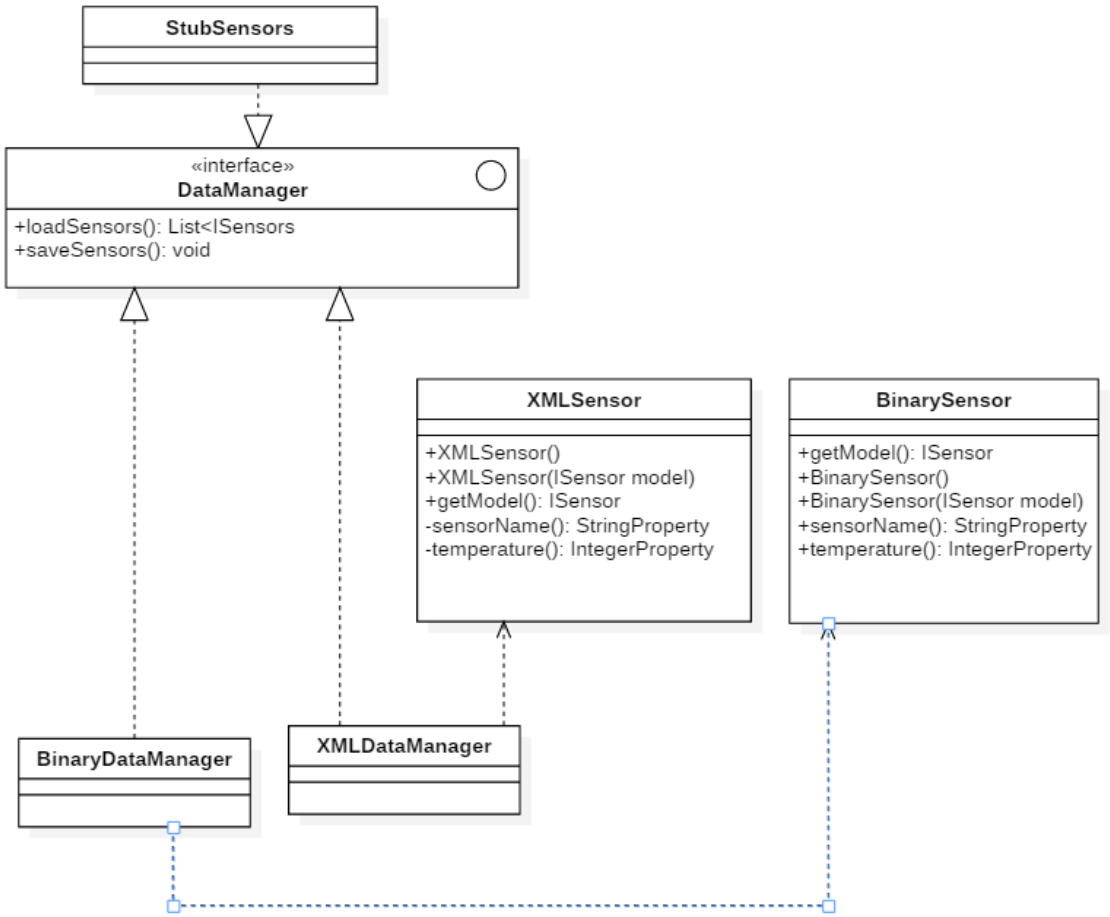
Nom	But initial	Pourquoi nous l'utilisons
Visiteur	Fait la représentation d'une opération applicable aux éléments d'une structure d'objet. Il définit une nouvelle opération, sans qu'il soit nécessaire de modifier la classe des éléments sur lesquels elle agit	On veut pouvoir générere 2 types d'affichages différents selons le type d'instanciation d'ISensor. Ainsi, on obtient des SensorCell différentes et on respecte le O de SOLID.

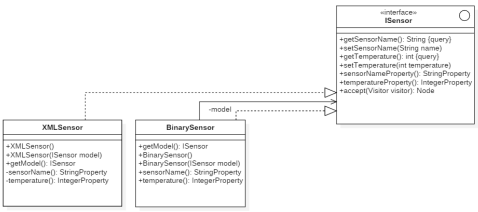


Nom	But initial	Pourquoi nous l'utilisons
Fabrique	Définit une interface pour la création d'un objet, mais en laissant à des sous-classes le choix des classes à instancier. La Fabrique simple permet à une classe de déléguer l'instanciation à des sous-classes	Notre fabrique SensorFactory permet de créer une instance d'objet ISensor. Cette responsabilité lui est déléguée et c'est la seule qu'elle assume. Toutes les instanciations de ISensor sont donc assurées par cette Factory et on s'assure ainsi que toutes nos classes n'ont qu'une seule responsabilité.



Nom	But initial	Pourquoi nous l'utilisons
Strategie	Définit une famille d'algorithmes, encapsule chacun d'entre eux et les rend interchangeables. Le Stratégie permet aux algorithmes d'évoluer indépendamment des clients qui les utilisent	La stratégie nous permet de pouvoir faire plusieurs types de sérialisation et de les gérer via un DataManager. Ainsi, on peut ajouter autant de types de sérialisation que l'on veut seulement par extension et sans modification de code. Le modèle de conception est alors plus facilement maintenable et évolutif.



Nom	**But initial**	**Pourquoi nous l'utilisons**
Procurateur	Fournit à un tiers un mandataire ou un remplaçant, pour contrôler l'accès à cet objet	Un ISensor n'est pas sérialisable. Pour effectuer une sérialisation XML, nous avons donc créé un Sensor sérialisable (XMLSensor ou BinarySensor) qui se fait passer pour un ISensor. On peut donc créer un ISensor à partir d'un XMLSensor ou BinarySensor et inversement. Ainsi, on peut sérialiser et désérialiser à l'aide du procurateur
		

Nom	**But initial**	**Pourquoi nous l'utilisons**
Composite	Compose des objets en des structures arborescentes pour représenter des hiérarchies composant/composé. Permet au client de traiter d'une unique façon les objets et les combinaisons d'objets	Nous voulions qu'un SuperSensor puisse disposer d'une arborescence de ISensor. Ainsi, le composite est très adapté à une structure arborescente d'objets. C'est pourquoi nous l'avons choisi