**COA – TP**

Marc DOUCHEMENT / Romain LE HO

ISTIC – Université de Rennes1

Février 2013

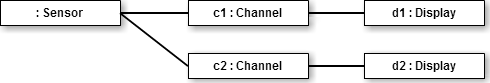
Rapport

# Introduction

Le TP de COA a pour but de gérer l’asynchronisme en Java. Pour cela, nous utilisons les classes Java permettant d’implémenter le patron de conception Active Object. Nous allons gérer la transmission d’une donnée d’un capteur vers des affichages, par le biais de canaux introduisant un délai de transmission.

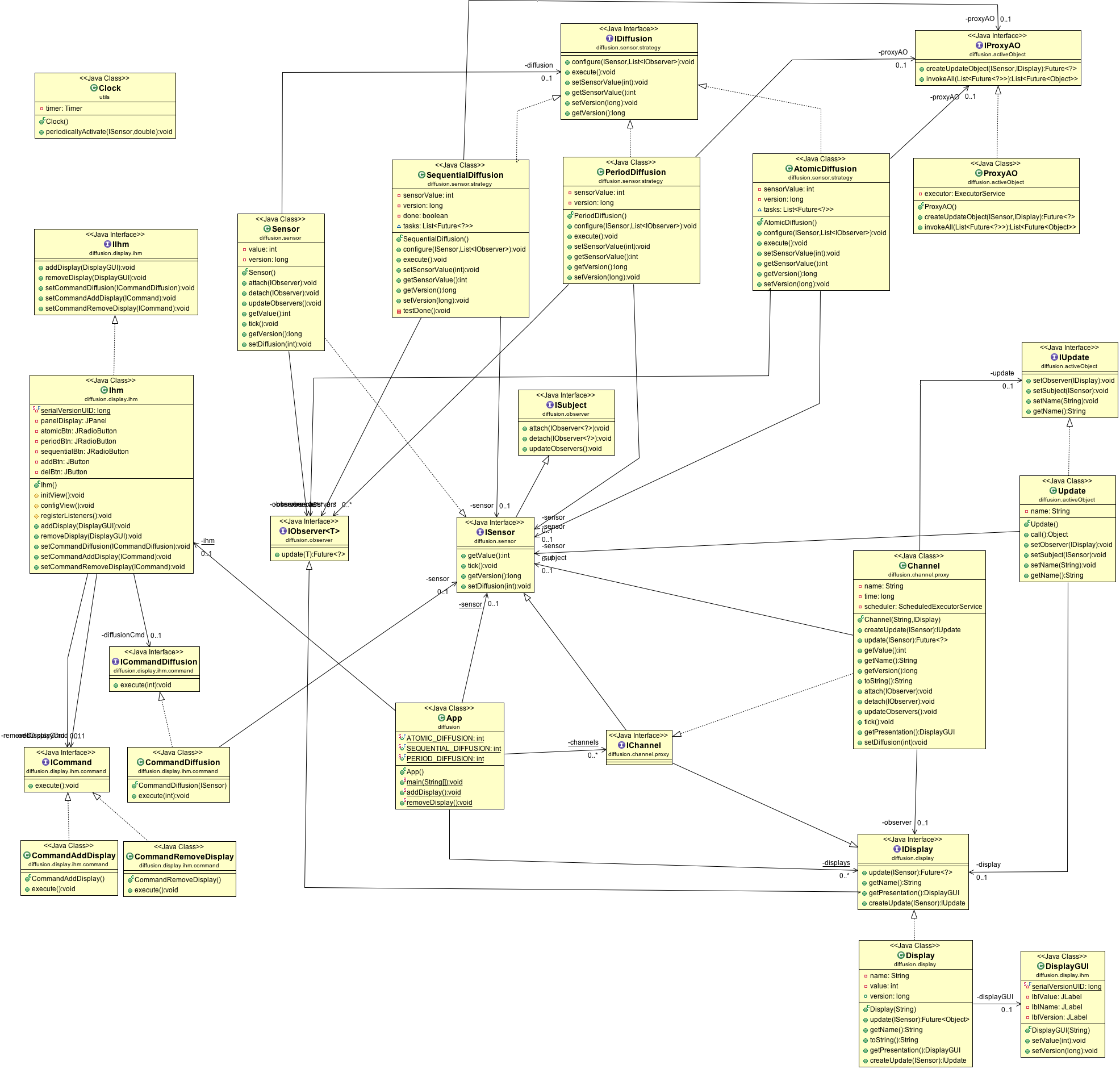
# Fonctionnement

Cette application doit permettre a des afficheurs asynchrones de se mettre à jour lorsque la valeur d’un capteur change (une valeur entière). Afin de mettre en oeuvre un délai de transmission aléatoire, un canal est ajouté entre le capteur et chaque afficheur comme vous le montre la figure suivante :



Encore du blabla

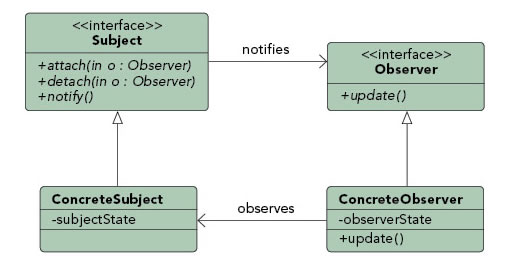
# Diagramme de classe



# Patrons de conception utilisés

## Observer

Pour mettre les afficheurs à jour lorsque le capteur change de valeur, le patron de conception Observer a été implémenté.

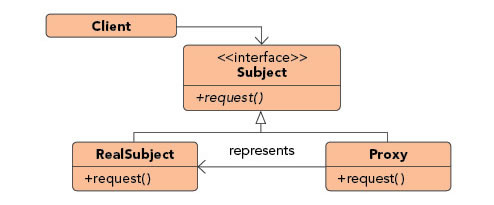


Corrélation entre ce patron et nos classes :

* Le Subject est l’interface ISubject
* Le ConcreteSubject est la classe Sensor
* L’Observer est l’interface IObserver
* Les ConcreteObserver sont les instances de la classe Display

## Proxy

Afin de gérer le délai de transition aléatoire, un canal (Channel) a été mis en oeuvre. Celui-ci devant être entre le capteur et chaque instance d’afficheur, il doit se faire passer dans un premier temps pour l’afficheur (lorsque le capteur notifie l’afficheur de son changement de valeur) et dans un second temps pour le capteur lorsque l’afficheur va récupérer la valeur du capteur (soit le fonctionnement du PC Observer). Afin d’implémenter proprement ce canal, l’utilisation du PC Proxy est fortement indiquée.

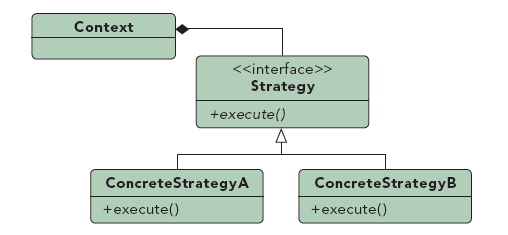


Ce patron est utilisé de fois dont la corrélation entre ce patron et nos classes est :

* Du capteur vers l’afficheur
  + Le client est la classe Sensor
  + Le Subject est l’interface Ichannel (dérivant Isensor)
  + Le RealSubject est la classe Display
  + Le Proxy est la classe Channel
* De l’afficheur vers le capteur
  + Le Client est la classe Display
  + Le Subject est l’interface Ichannel (dérivant aussi IDisplay)
  + Le RealSubject est la classe Sensor
  + Le Proxy est la classe Channel

## Strategy

Comme l’application intègre différentes façons de diffuser la valeur du capteur aux afficheurs (atomique, séquentielle, etc.), le patron de conception Strategy est ici indiqué.



Corrélation entre les rôles de ce patron de conception et nos classes :

* Le Context est la classe Sensor
* La Strategy est l’interface Idifusion
* Les ConcreteStrategy sont les classes AtomicDiffusion, SequentialDiffsusion et PeriodDiffusion

### Diffusion par époque

Cette stratégie de diffusion permet aux affichages de mettre à jour leur valeur uniquement si la valeur récupérée est plus récente que la précédente. Il n’y a pas de vérification quant à la synchronisation entre tous les lecteurs (affichages).

### Diffusion atomique

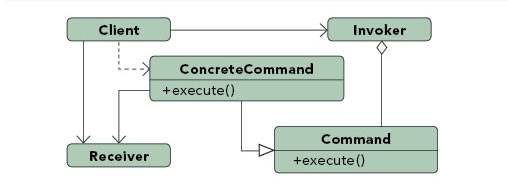
Ici, il faut que tous les affichages se soient mis à jour avant d’obtenir une nouvelle valeur. Cela signifie que le capteur est bloqué : il ne met plus à jour sa valeur tant que tous les affichages ne sont pas à jour.

### Diffusion séquentielle

La diffusion séquentielle suit le même principe que la diffusion atomique, la différence étant que le capteur peut quand même se mettre à jour même si tous les affichages n’ont pas tous récupéré la nouvelle valeur.

## Command

Afin de découpler la présentation (l’IHM) du moteur (App et Sensor) de l’application, toutes les actions (changer de stratégie et ajout/supression d’afficheur) faites par cette présentation utilisent une commande.



Corrélation entre ce patron et nos classes :

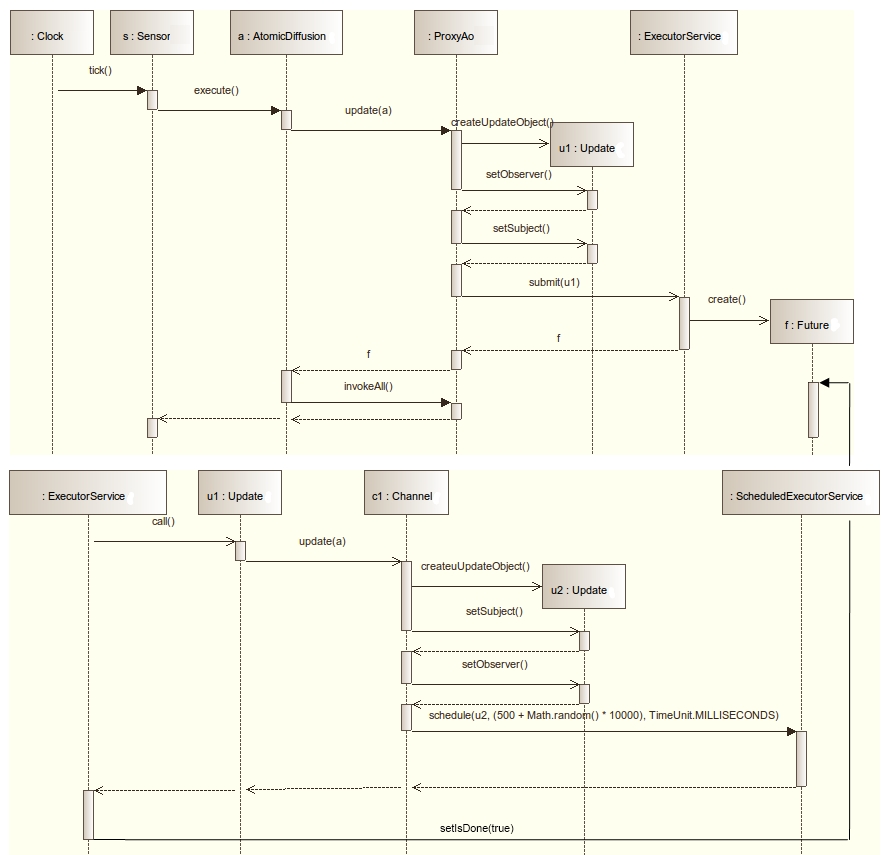
* Changement de stratégie
  + Le Client est la classe App
  + L’Invoker est la classe Ihm
  + La Command est l’interface ICommandDiffusion
  + La ConcreteCommand est la classe CommandDiffsion
  + Le Receiver est la classe Sensor
* Ajout et supression d’afficheur
  + Le Client est la classe App
  + L’Invoker est la classe Ihm
  + La command est l’interface Command
  + Les ConcreteCommand sont les classes CommandAddDisplay et CommandRemoveDisplay
  + Le Receiver est la classe App

## Active Object

[ [Le blablabalbalba]]

Le schéma

# Diagramme de séquence



# Tests

Les différents tests implémentés sont commentés avec de la JavaDoc afin de détailler les tests si besoins.

## Intégration

Ici, les tests d’intégration permettent avant tout de s’assurer du bon fonctionnement des différentes stratégies de diffusion implémentées dans l’application.

### Diffusion atomique

|  |  |  |
| --- | --- | --- |
| N° | Nom et description du test | OK/KO |
| 1 | testValuesConsistencyBetweenSensorAndDisplay | OK |
| 2 | testValuesConsistencyBetweenTwoDisplays | OK |
| 3 | testValuesConsistencySensorDontChangedItsValueDuringTheDisplaysGetValue | OK |

### Diffusion séquentielle

|  |  |  |
| --- | --- | --- |
| N° | Nom et description du test | OK/KO |
| 1 | testValuesConsistencyBetweenSensorAndDisplay | OK |
| 2 | testValuesConsistencyBetweenTwoDisplays | OK |
| 3 | testValuesConsistencySensorDontChangedItsValueDuringTheDisplaysGetValue | OK |

## Unitaires

Les tests unitaires suivants permettent de s’assurer que les classes implémentées retournent les bonnes valeurs et vérifier la non régression du code en cas de modification.

### Sensor

|  |  |  |
| --- | --- | --- |
| N° | Nom et description du test | OK/KO |
| 1 | *testTick* vérifie que la valeur du capteur change à chaque tick | OK |
| 2 | *testGetRealSensorVersion* vérifie que l’horodatage du capteur change à chaque tick | OK |
| 3 | *testSetDiffusionWithAtomic* test le changement de stratégie (atomique ici) | OK |
| 4 | *testSetDiffusionWithSequential* test le changement de stratégie (séquentielle ici) | OK |
| 5 | *testSetDiffusionWithPeriod* test le changement de stratégie (époque ici) | OK |
| 6 | *testGetVersionWithAtomicDiffusion* test l’horodatage du capteur (stratégie atomique) | OK |
| 7 | *testGetVersionWithSequentialDiffusion* test l’horodatage du capteur (stratégie séquentielle) | OK |
| 8 | *testGetVersionWithPeriodDiffusion* test l’horodatage du capteur (stratégie époque) | OK |
| 9 | *testGetValueWithAtomicDiffusion* test la valeur du capteur lu par un afficheur (stratégie atomique) | OK |
| 10 | *testGetValueWithSequentialDiffusion* test la valeur du capteur lu par un afficheur (stratégie séquentielle) | OK |
| 11 | *testGetValueWithPeriodDiffusion* test la valeur du capteur lu par un afficheur (stratégie époque) | OK |