COA – TP

Marc DOUCHEMENT / Romain LE HO

Rapport

ISTIC – Université de Rennes1

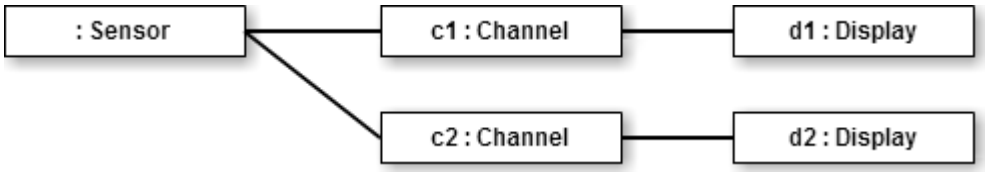
Février 2013

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.

1. 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 :

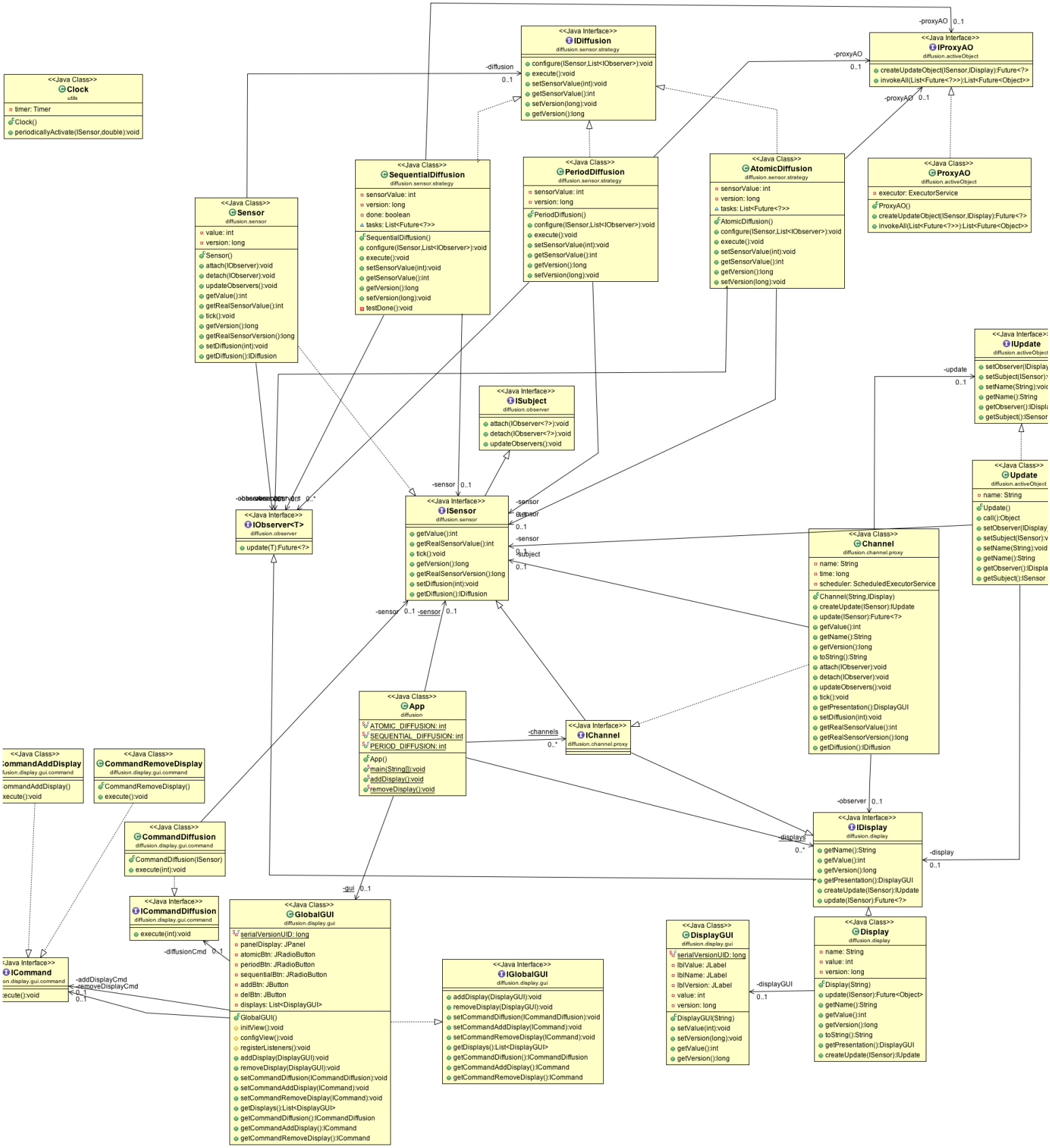


L'architecture comprend donc :

- Une source active (capteur), dont la valeur évolue de façon périodique
- Un ensemble de canaux de transmission avec des délais variables
- Un ensemble d'afficheurs réalisés en utilisant la bibliothèque graphique Swing
- Un ensemble de politiques de diffusions de la valeur du capteur aux afficheurs (séquentielle, atomique et époque)

2. Diagramme de classe

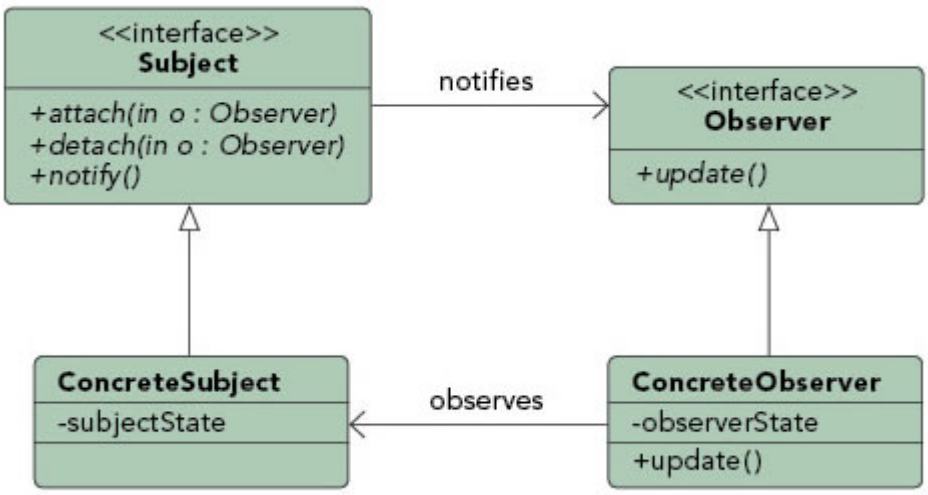
Ce schéma se trouve également dans le dossier « diagram » du projet.



3. Patrons de conception utilisés

3.1.Observer

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

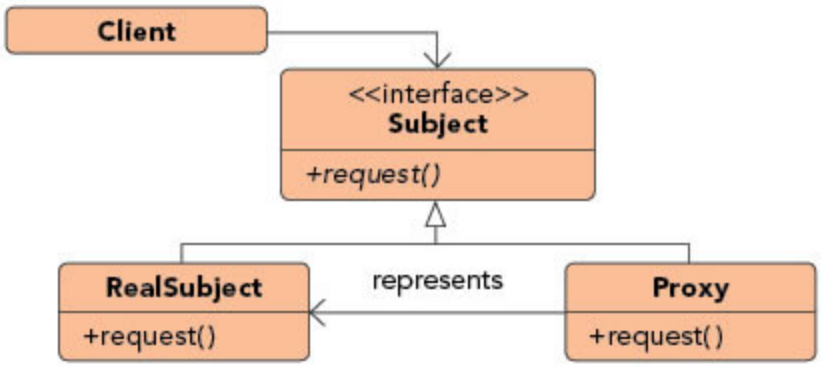


Corrélation entre ce patron et nos classes :

- 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

3.2.Proxy

Afin de gérer le délai de transmission 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.

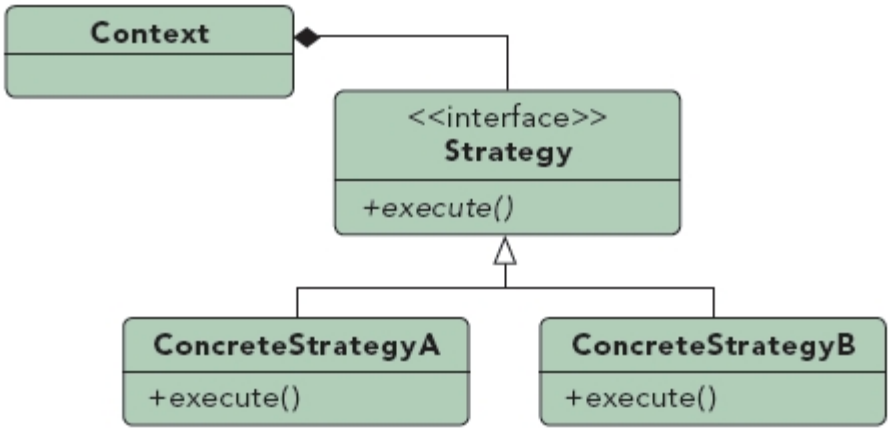


Ce patron est utilisé de fois dont la corrélation entre ce patron et nos classes est :

- 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

3.3.Strategy

Comme l'application intègre différentes politiques de diffusion de la valeur du capteur aux afficheurs (atomique, séquentielle, etc.), le patron de conception Strategy est utilisé.



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

3.3.1. 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).

3.3.2. Diffusion atomique

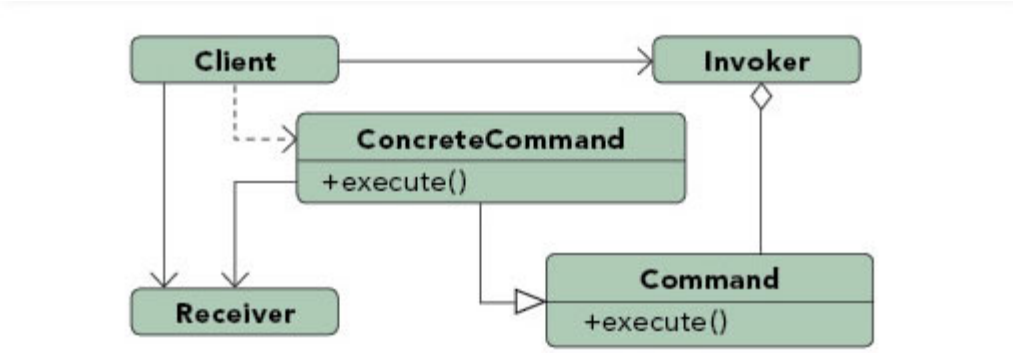
Ici, il faut que tous les affichages se soient mis à jour avant d'obtenir une nouvelle valeur. Cela signifie que le capteur est bloqué : il ne met plus à jour sa valeur tant que tous les affichages ne sont pas à jour.

3.3.3. 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.

3.4.Command

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

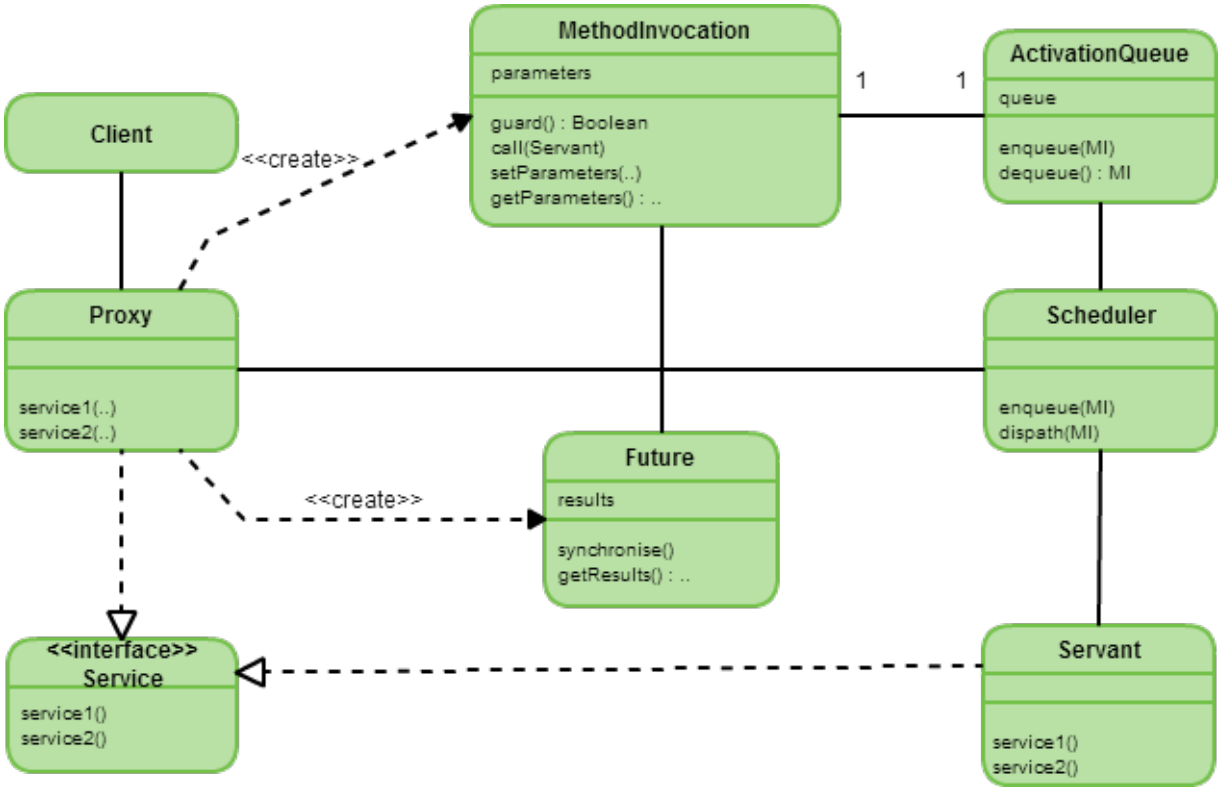


Corrélation entre ce patron et nos classes :

- Changement de stratégie
  - Le Client est la classe App
  - L'Invoker est la classe GlobalGUI
  - La Command est l'interface ICommandDiffusion
  - La ConcreteCommand est la classe CommandDiffsion
  - Le Receiver est la classe Sensor
- Ajout et suppression d'afficheur
  - Le Client est la classe App
  - L'Invoker est la classe GlobalGUI
  - La command est l'interface Command
  - Les ConcreteCommand sont les classes CommandAddDisplay et CommandRemoveDisplay
  - Le Receiver est la classe App

3.5.Active Object

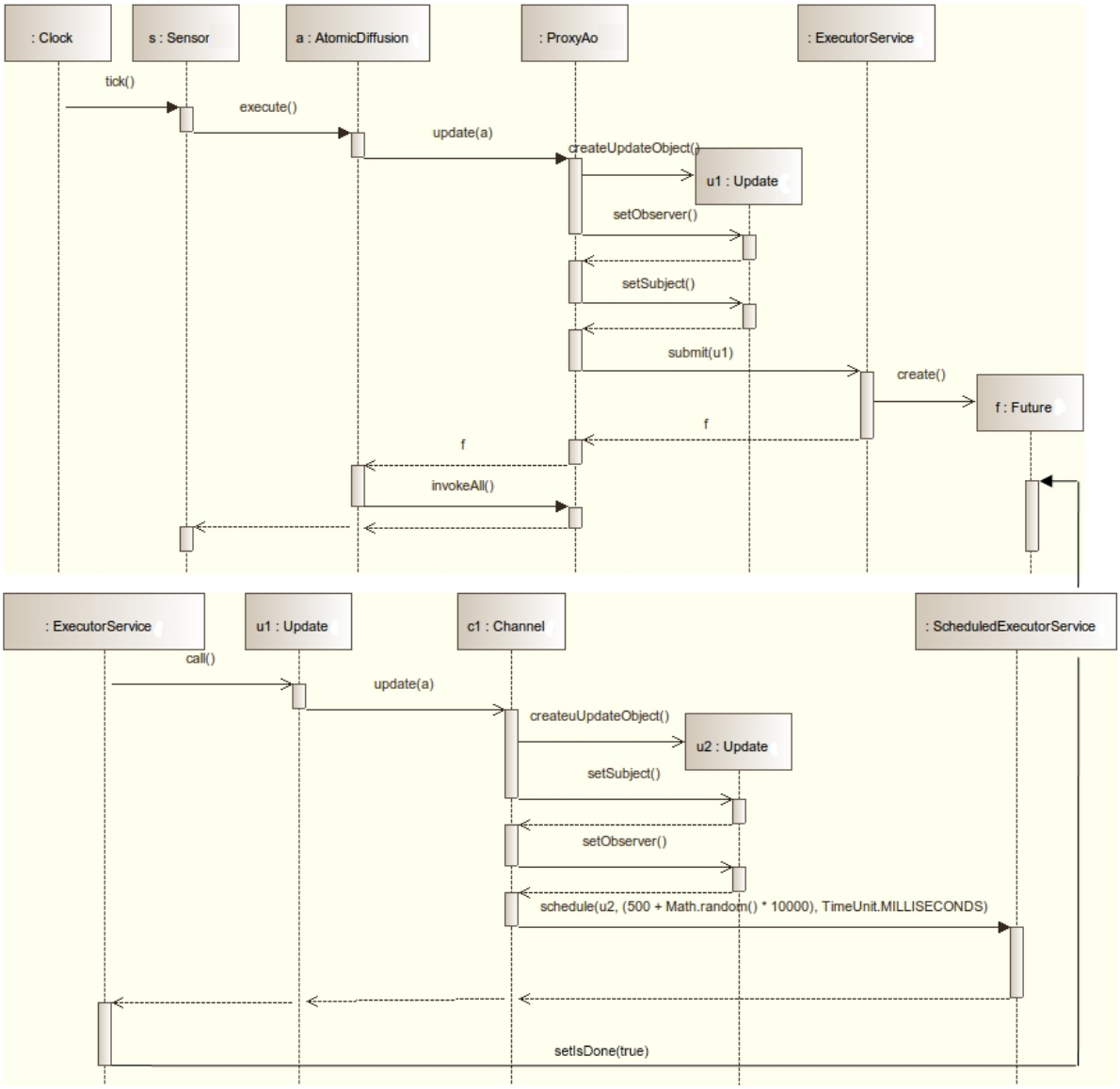
L'application étant fortement axée sur l'asynchronisme de processus (les afficheurs), l'emploi du patron de conception Active Object permet de mettre en oeuvre le concept d'invocation asynchrone d'opération pour structurer la gestion des threads.



Corrélation entre ce patron et nos classes :

- Le Client est l'interface IDiffusion (les stratégies)
- Le Proxy est la classe ProxyAO
- Le service est l'interface IDisplay
- Le MethodInvocation est la classe Update
- Le Future est la classe Future implémentée par Java
- Le Servant est la classe Channel
- Les Scheduler sont l'ExecutorService et le ScheduledExecutorService
- La ActivationQueue est interne à Java

4. Diagramme de séquence



5. Tests

Les différents tests implémentés sont commentés avec de la JavaDoc afin de les détailler. Tous les tests passent. La classe AllTests.java permet d'exécuter tous les tests à la suite.

Classes testées unitairement:

- ProxyAO
- Update
- DisplayGUI
- GlobalGUI
- Display
- Sensor

Classes testées en intégration :

- AtomicDiffusion
- SequentialDiffusion
- Channel (ne fait qu'implémenter 2 autres interfaces)

Toutes les classes ont été testées avec succès.