AOC - Rapport de conception

Description

Projet réalisé dans le cadre des TP d'AOC, cours de Master 2 spécialisation Ingénierie Logicielle en Alternance. Le projet est constitué d'un binôme comportant Erwan IQUEL et Adrien LEBLANC.

Réalisation

Le but de ce projet est de mettre en oeuvre différents patrons de conceptions vus en cours, notamment `Active Object`, sur lequel se repose le projet. L'idée est de parvenir à recréer un effet asynchrone avec le langage Java, en excluant les Threads et autres dérivés.

Nous avons choisi de développer une simple application qui permet à un générateur de valeur, de diffuser cette dernière à des Afficheurs (représentation graphique, sous Java Swing) à travers de Canaux.

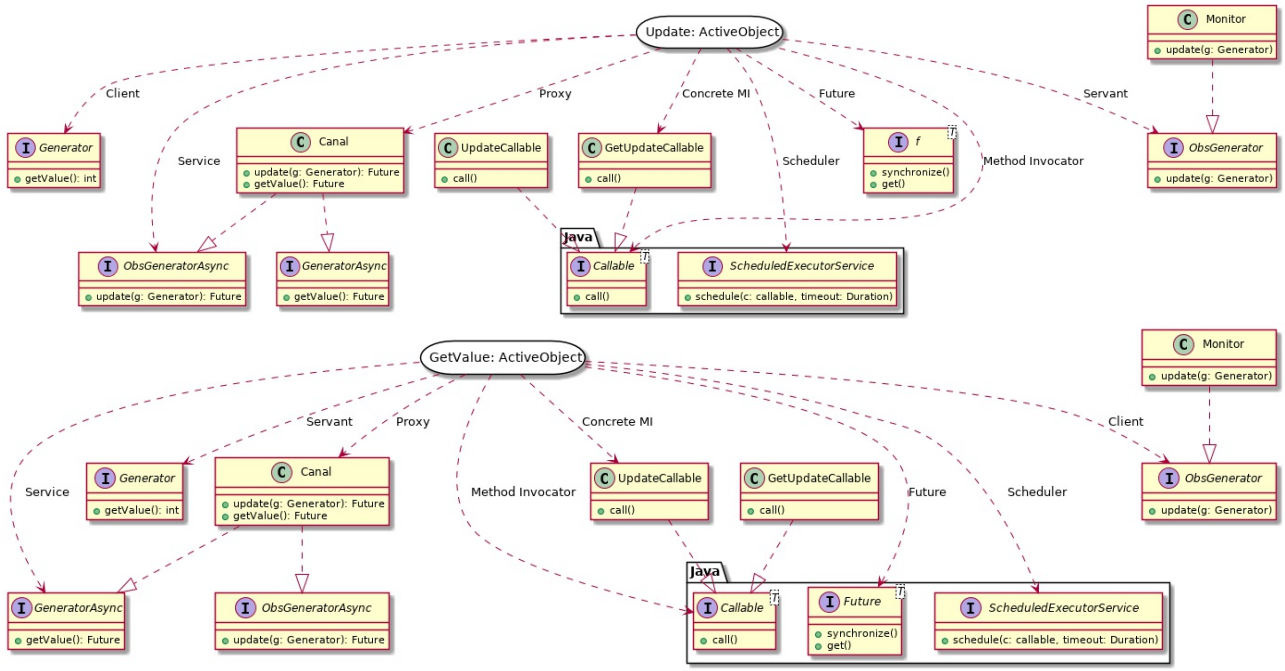
> Ces canaux engrangent un retard de plusieurs centaines de milisecondes, pouvant être interprétés par du "lag".

Développement

Nous nous sommes basés sur un TDD pour développer ce projet. Développer les tests nous aura permis de nous orienter dès le départ vers une réalisation efficace et rapide, tout en utilisant le Pair Programming comme moyen de développement principal.

Choix d'architectures

Nous nous sommes au départ inspirés des diagrammes de séquences et de classes, d'un autre binôme, mis à disposistion de tout le monde afin d'aider à la compréhension du sujet. Cette inspiration n'est pas notre résultat final, nous avons dû retoucher à certaines classes, noms de méthodes, tout comme l'ajout, associations, dépendances, etc.



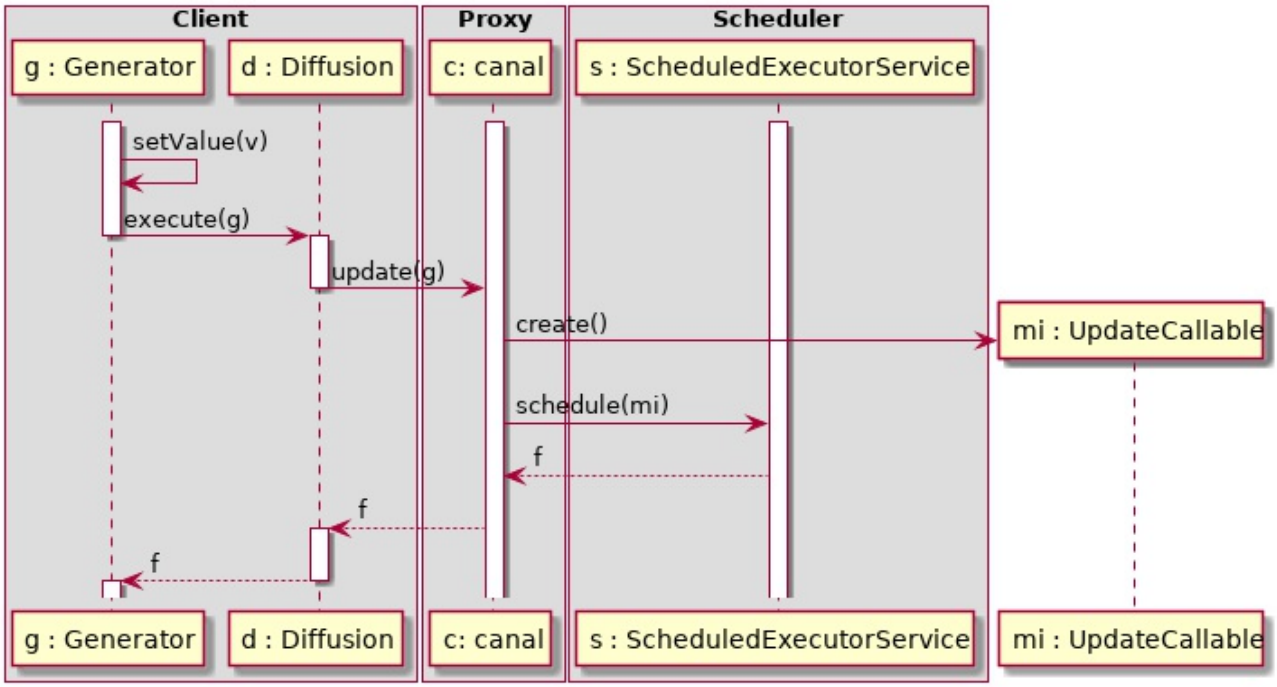
Update d'une valeur

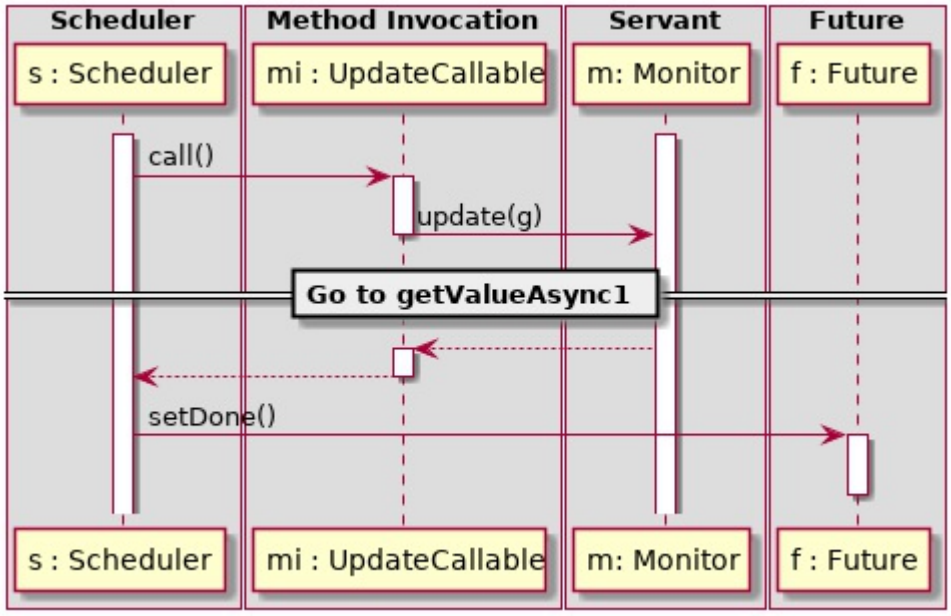
Le générateur change de valeur. S'exécute ensuite la Diffusion courante, qui fait appel à tous les canaux qui observent le Générateur.

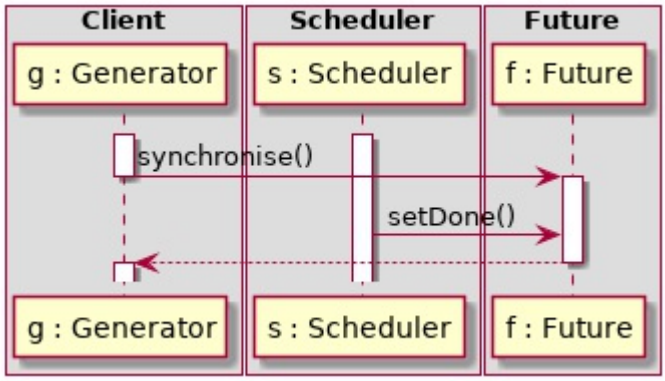
> Notons ici le patron de conception `Observer`, utilisé à plusieurs reprises dans notre architecture.

Ces derniers vont placer la nouvelle valeur du Générateur dans un Update de type Callable, qui sera exécuté par le Scheduler.

Lorsque cette exécution se produira, les Afficheurs se verront attribuer la nouvelle valeur du Générateur et pourront l'afficher.

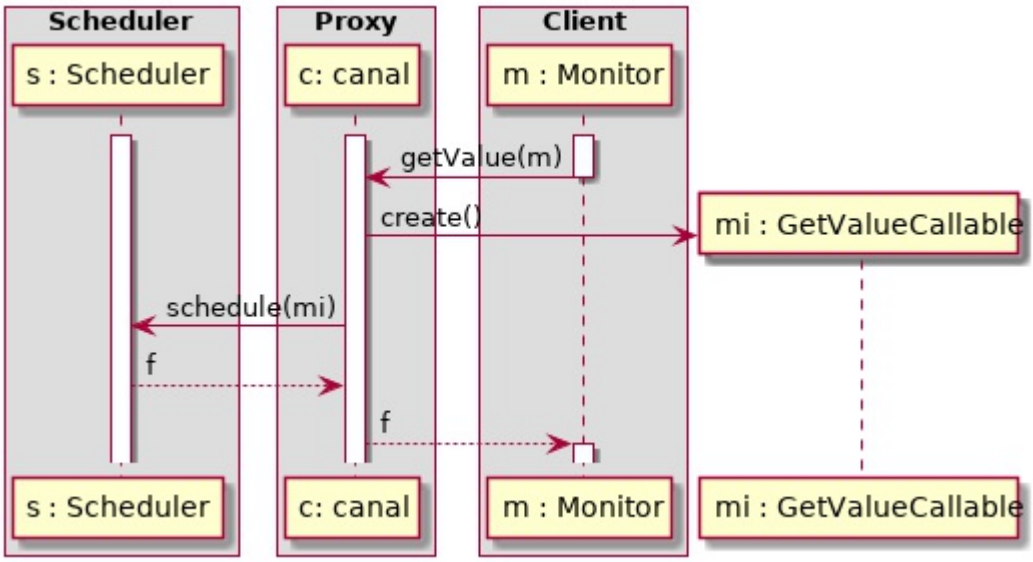


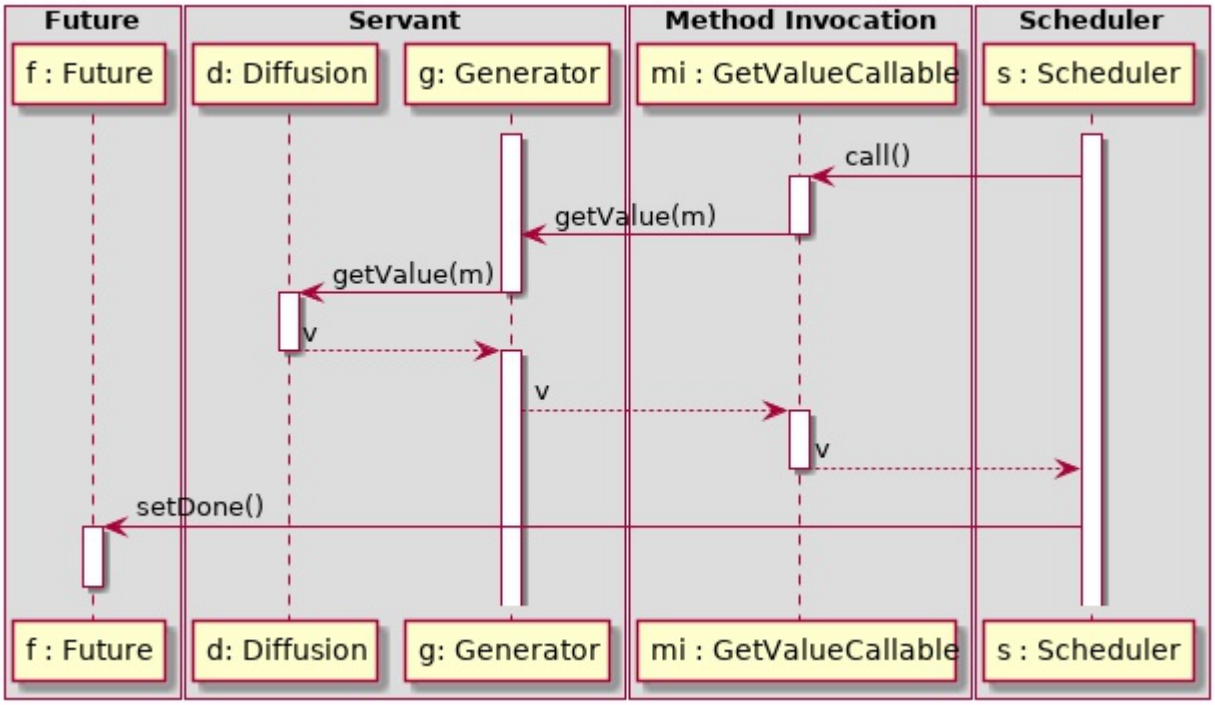


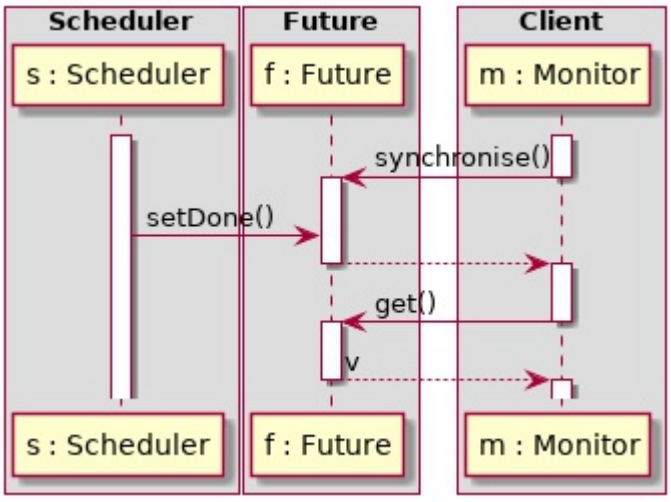


GetValue provenant d'un Afficheur

Ici l'architecture est basiquement la même, mais dans le sens inverse.


Scheduler
Proxy
Client
s : Scheduler
c: canal
m : Monitor
getValue(m)
create()
mi : GetValueCallable
schedule(mi)
f
f
s : Scheduler
c: canal
m : Monitor
mi : GetValueCallable


Future
Servant
Method Invocation
Scheduler
f : Future
d: Diffusion
g: Generator
mi : GetValueCallable
s : Scheduler
call()
getValue(m)
getValue(m)
v
v
v
setDone()
f : Future
d: Diffusion
g: Generator
mi : GetValueCallable
s : Scheduler


Scheduler
Future
Client
s : Scheduler
f : Future
m : Monitor
synchronise()
setDone()
get()
v
s : Scheduler
f : Future
m : Monitor

Diffusions

Nous avons dû mettre en place 3 diffusions différentes, permettant de vérifier que notre système fonctionnait bien. Les 3 se discernent par le fonctionnement de l'affichage des valeurs du générateur sur notre UI.

Atomic Diffusion

```
if(generator == null) {
    throw new BadConfigurationException("generator");
}

T ret = Objects.requireNonNull(this.value);

if(this.generator.numberOfObservers() == this.observers.size()) {
    this.observers.clear();
    this.generator.tick();
}

return ret;
```

Cette diffusion permet d'empêcher qu'un Afficheur soit en avance d'un changement de valeur (ou plus) sur les autres. En effet, afin d'en recevoir une nouvelle, il doit attendre que tous les autres soient à jour. Si ce n'est pas le cas, la génération de la nouvelle valeur est "perdue", aucun Afficheur n'en aura eu connaissance.

La méthode `tick()` du générateur n'est appelée que lorsque les Afficheurs ont tous reçu la valeur. Cette méthode permet la régénération d'une valeur toutes les n secondes.

Sequential Diffusion

```
if(generator == null) {
    throw new BadConfigurationException("generator");
}

T ret = Objects.requireNonNull(this.value);

if(this.generator.numberOfObservers() == this.observers.size()) {
    this.observers.clear();
    this.settable = true;
}

return ret;
```

Cette diffusion n'a pas de contrainte au niveau de la génération, mais doit afficher la même valeur à un instant T sur tous les Afficheurs. Ainsi la contrainte est cette fois-ci sur ces derniers.

Le booléen `settable` est muté à `true` lorsque les Afficheurs ont tous reçus la valeur du générateur. Ce booléen est requis pour le setter.

D'autres valeurs peuvent parvenir jusqu'aux Afficheurs, mais elles seront en attente d'affichage si la précédente n'a pas été affichée par les tous les autres.

Causal Diffusion

```
if(value == null) {
    throw new BadConfigurationException("value");
}

return this.value;
```

Cette diffusion est la plus libre des trois, elle permet de mettre à jour n'importe quel Afficheur sans avoir à se préoccuper des autres. A ce niveau-ci, c'est contrôlé par la machine qui exécute le programme, il est impossible de prédéfinir l'ordre d'arrivée des valeurs, en dépit du "lag".

Répresentation Graphique

L'utilisation de la bibliothèque Swing de Java nous a permis de représenter graphiquement notre travail. Nous avons mis en place un Générateur et 4 Afficheurs, tous liés à ce dernier par un Canal différent.
Voici un screenshot de notre rendu visuel

Outils utilisés

- Git
- Java
- JUnit
- Bibliothèque Swing