SEP - Compte rendu

PERRET Marc
GUIHAIRE Clément

Introduction

L’objectif de ce projet est d’implémenter le patron de conception Observer asynchrone, comportant un sujet et plusieurs observer (4 en l'occurrence).
Pour cela nous allons également implémenter le patron de conception Active Object dans les deux sens de communication.

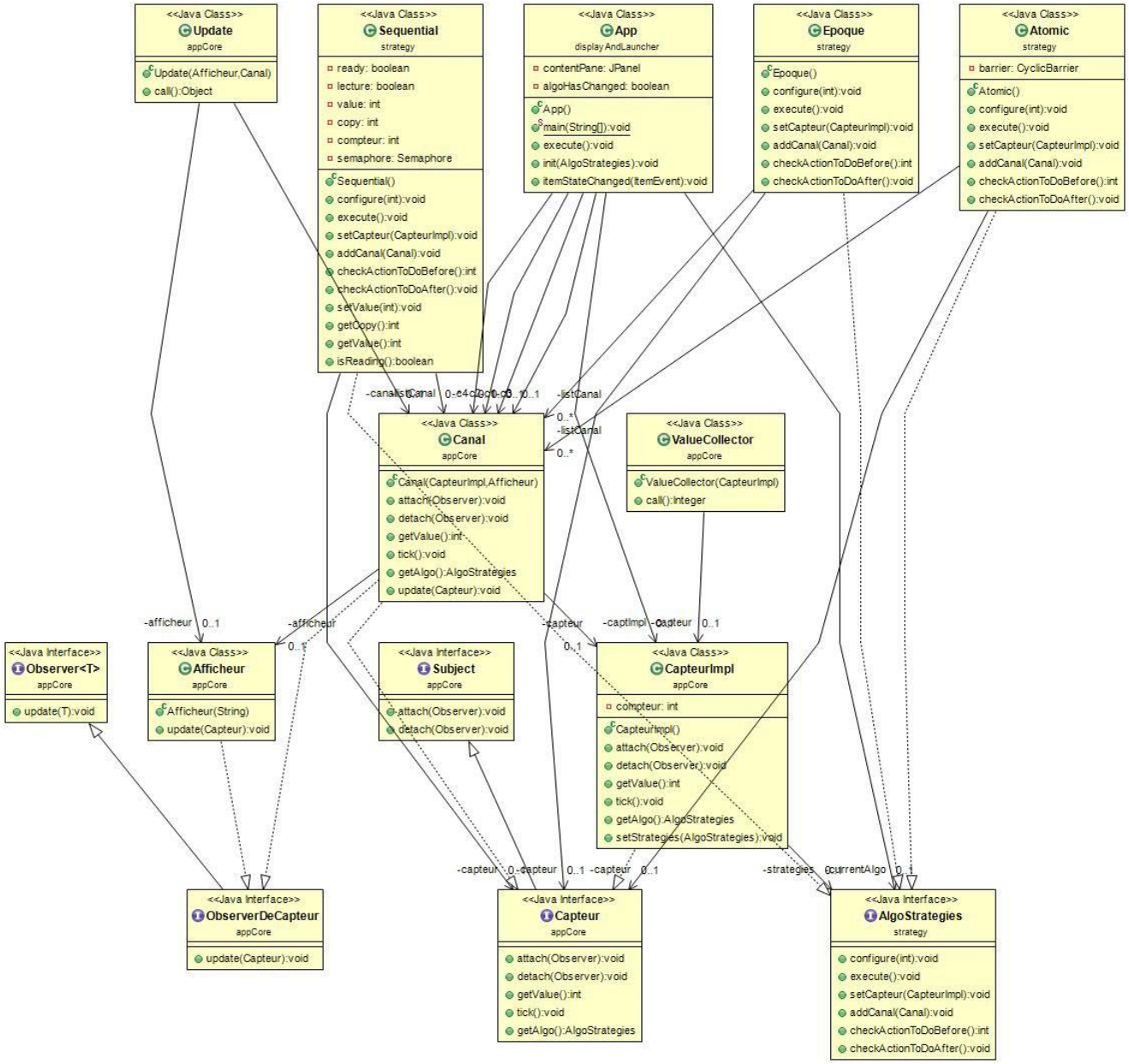
Mise en place

Les outils

Afin de développer cette application, nous avons utilisé le langage de programmation JAVA, ainsi que l’IDE Eclipse.
Pour l’affichage graphique, nous avons utilisé Swing.
Nous avons également utilisé un gestionnaire de version : Github.

Architecture

diagramme de classe:



Les 3 stratégies de gestion de la cohérence

Pour ces algorithmes, l'objectif est que le sujet possède un compteur, l'incrémente régulièrement et transmettre la valeur de ce compteur à 4 observateurs. Pour réaliser cette application, nous utilisons 3 stratégies différentes de gestion de la cohérence : Atomic, Sequential et Epoque.

Atomic

Cet Algorithme a pour objectif de gérer la cohérence en obligeant le sujet à attendre que tous les observateurs aient affiché la nouvelle valeur avant de pouvoir incrémenter le compteur.
Pour cela, nous avons mis en place une barrière qui oblige le sujet à attendre que l'ensemble des observateurs aient reçu l'information au travers d'un canal et qu'ils l'aient affiché.

```
@Override
public void execute() {
    for(Canal canal : listCanal){
        canal.update(capteur);
    }

    this.checkActionToDoBefore();
}
```

Une fois que tous les observateurs ont reçu la nouvelle donnée, le sujet peut alors incrémenter à nouveau son compteur et attendre une nouvelle fois...

Sequential

Dans cet algorithme, l'objectif est de laisser le sujet incrémenter un compteur sans jamais le gêner. Les observateurs vont donc récupérer une valeur de copie.
L'algorithme permet de vérifier quand la valeur du compteur a changé et ainsi de mettre à jour la valeur de la copie.

Ensuite chaque observateur va aller lire la nouvelle copie.
Si la valeur a changé, on envoie l'information aux observateurs

```
@Override
public void execute() {
    if(ready){
        ready = false;
        for (Canal l : listCanal) {
            l.update(capteur);
        }
    }
}
```

Le premier observateur va bloquer les autres à l'aide du sémaphore. Puis il va copier la valeur du capteur dans copy. Les autres observateurs vont donc directement lire la copie.

```
@Override
public int checkActionToDoBefore() {
    // TODO Auto-generated method stub
    try {
        semaphore.acquire();
    } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
    if(isReading()){
        this.copy = capteur.getValue();
        this.setValue(copy);
    }

    else{
        this.copy = this.getValue();
    }
    semaphore.release();

    return copy;
}
```

Epoque

Contrairement aux autres algorithmes, il n'y a pas de contrôle de cohérence. Le sujet envoie les informations aux observateurs au travers de canaux, et cela sans aucun contrôle. Peu importe si les observateurs affichent des valeurs différentes, peu importe s'ils loupent des valeurs.

```
@Override
public void execute() {
    for(Canal canal : listCanal){
        canal.update(this.capteur);
    }
}
```

Quand chaque observateur reçoit une nouvelle valeur, il l'affiche.

Conclusion

https://github.com/Fmeuu/SEPCe projet nous a donc permis de mettre en oeuvre et d'implémenter les patrons de conception étudiés en cours. Cela nous a permis de créer une architecture claire, simple et adaptable en implémentant un grand nombre d'interfaces.

Les différents algorithmes sont donc bien architecturés et il est aisé de passer de l'un à l'autre.