Université de Rennes 1

Patron de Conception Active Object

AOC Projet TP

Chakib BENKEBIR
Christophe PLANCHAIS
Youssouf ROUDANI
11/01/2018

INTRODUCTION

Lors de ce TP, notre tâche a principalement été de mettre en œuvre le pc Active Object, et ceci, dans le cadre d'un programme JavaFX : "Mise en œuvre d'un patron de conception Observer asynchrone, comportant un sujet (Générateur) et plusieurs observateurs.", avec un "canal [qui] a pour but d'introduire des délais aléatoires de communication entre sujet et observateurs", le tout avec 3 stratégies de gestion de la cohérence : atomique, séquentielle et par époques.

Un objet passif est un ensemble d'attributs et de méthodes qui est accédé par un ensemble de threads extérieurs. L'accès n'est pas thread-safe. Si l'objet était en charge de l'exécution de ses propres méthodes, le problème d'action non voulues serait résolu. Pour un objet actif, l'invocation de méthodes se fait avec "l'autorisation" de l'objet, sous son thread de contrôle.

L'importance du pc est cruciale puisque, par exemple, le thread de l'interface utilisateur ne doit pas prendre de retard : le pc AO sépare l'invocation de méthode et son exécution.

PARTIE 1 : BIBLIOTHEQUES

1.JAVA FX

JAVA FX est une solution Oracle pour le développement cross plateformes d'interfaces graphiques (GUI's) qui offre des avantages par rapport aux solutions AWT et SWING dans le domaine médias et animations. Dans ce TP nous utilisons la description de l'interface graphique sous forme XML.

Le thread de l'application JAVA FX est le seul pouvant accéder et modifier l'interface graphique.

Notre objet "scene" contient le nœud graphique "root", qui est chargé avec l'URL de fichier FXML getClass().getResource("sample.fxml"). La balise racine AnchorPane de notre fichier FXML contient la spécification d'un contrôleur : fx:controller="fr.istic.FXMLDocumentController">. une classe avec constructeur publique.

Une fois le fichier FXML en mémoire, nous pouvons le modifier programmatiquement à volonté. Cette étape, est matérialisée par la méthode "initialize(URL url, ResourceBundle rb)" de notre contrôleur.

Enfin nous rendons accessible à FXML les méthodes "start" et "stop" grâce à l'annotation « @FXML : <Button fx:id="stopButton" layoutX="197.0" layoutY="127.0" mnemonicParsing="false" onAction="#stop" ». La méthode "stop" est triviale et ne requiert pas d'explication, tandis que la méthode "start" démarre le générateur (cf section suivante).

```
Platform.runLater(() -> labelGen.setText(String.valueOf(integer)))
```

2.ORACLE : Programmation asynchrone

Pour éviter de bloquer l'application avec une instruction prenant trop de temps, le JDK nous fournit l'interface « Runnable » pour confier les portions bloquantes de code à des threads fils. Il suffit de réécrire la méthode « run » puis de démarrer le thread avec « new Thread(myRunnable) ».

Java 5 introduit le framework Executor pour résoudre les problèmes classiques des threads qui sont la complexité et la performance. ExecutorService est une interface qui fournit une solution pour traiter des pools de threads. Executor est un nouveau framework d'exécution des tâches paru en 2004. Il propose la méthode newSingleThreadExecutor() pour créer un nouveau thread de type classique. Executors est la classe qui propose toutes les méthodes statiques nécessaires. La construction d'animations devient facile.

Ainsi, au lieu d'avoir un unique servant, un ensemble de threads est créé pour chaque Active Object. Cela augmente la performance. Chaque servant peut demander au Scheduler de lui assigner une nouvelle tâche, lorsqu'elle sera disponible. La raison de l'existence d'un pool de thread est que la création illimitée de threads peut ralentir le système.

Nous utilisons la classe SCHEDULEDTHREADPOOLEXECUTOR pour exécuter à intervalle fixe la méthode "createValue" de notre générateur. L'usage de cette classe est une alternative à l'objet "Timer" qui ne peut exécuter qu'une tâche à la fois. scheduledExecutorService.scheduleAtFixedRate(generator::createvalue,0,100, TimeUnit.MILLISECONDS). Cette méthode utilise la bibliothèque "Math" pour la génération d'un entier aléatoire.

3.JAVA 8 : traits de programmation fonctionnelle

Lambda Expressions: en 2014, Java 8 propose les expressions lambda, c'est l'implémentation d'une closure, elle peut avoir des paramètres et un résultat : (arguments) -> corps

Une expression lambda est typée de manière statique. Ce type doit être une interface fonctionnelle. Une interface fonctionnelle doit disposer d'une et une seule méthode abstraite (ne doit pas avoir d'implémentation par défaut) et 0, 1 ou plusieurs méthodes par défaut.

Reference de Méthode :Java 8 a introduit les références de méthode pour répondre au besoin de faire référence dans une lambda expression à une méthode existante via son nom. (cf generator::createvalue de la classe FXMLDocumentController.java)

PARTIE 2 : RÔLES THÉORIQUES DE AO

Les appels de méthode sur un AO sont toujours non bloquants. La fonction de l'AO (ou du Scheduler) est de prendre les messages dans la queue et de les exécuter un par un sur son thread. Comme les champs privés ne sont accessibles que par le thread privé, il n'y a pas de soucis de concurrence.

1.Threads et exécution de méthode (côté serveur)

Les méthodes du servant accessibles par le client sont déclarées via l'interface Proxy. D'autres méthodes peuvent être utiles au Scheduler. A la complétion du calcul, le servant obtient le mutex write-lock sur la Future qu'il met à jour.

La méthode "execute" de l'ExecutorService prend en paramètre un runnable, le code est donc compatible avec nos habitudes de création de threads. Cependant, la méthode "run" d'un runnable ne peut retourner de valeur. A l'opposé, la méthode "call" d'un callable retourne un "V". Autre point important, la méthode "submit" de l'ExecutorService qui prend en paramètre un callable ou un runnable renvoie un objet Future <V> (un Future<void> pour un runnable).

Thread pool : au lieu d'avoir un unique servant, un ensemble de threads est créé pour chaque Active Object. Cela augmente la performance. Chaque servant demande au Scheduler de lui assigner une nouvelle tâche, lorsqu'elle sera disponible. Le thread "dispatcher" (l'Active Object ou le Scheduler) est idle jusqu'à ce qu'un "Runnable" ou un "Callable" soit ajouté à l'Activation Queue, traite le message puis attend de nouveau.

Activation Queue : deux propriétés essentielles : (1) un thread-safe "push" : multiple threads peuvent y accéder et (2) l'appel à "pop" est bloquant jusqu'à ce que des données soient disponibles dans la queue .

Scheduler : un thread séparé des threads servants (dans le cas où on utilise pas de Active Object dédié). Lorsque une method request devient runnable et suivant un algorithme adapté, le scheduler l'enlève de la queue le transmet au servant.

Proxy : une interface publique que le client peut invoquer, et que l'Active Object réalise, en travaillant avec Servant de façon pc Command. C'est à dire que nous déplaçons tout le code à exécuter sur une variable vers le Servant. (C'est lors de la création du Servant que la variable à traiter sera passée en paramètre).

2.Appel de méthode, attente de résultat (côté client)

Method Request: Il s'agit d'une abstraction du contexte d'une méthode. L'Active Object crée un message contenant le nom de méthode et ses arguments et le place dans une queue. Cette étape s'appelle Methode Request. L'information est passée au Scheduler puis est placée dans l'Activation Queue.

Le thread client est bloqué en attente d'une valeur de retour("get" sur le Future), et sera réveillé par le Servant. Le client reçoit le résultat sous forme de Future (une sorte de callback) lors de l'appel d'une méthode avec retour. Ce sera la manière pour le client de recevoir le résultat de méthodes invoquées grâce à l'AO.

Future : permet au client d'obtenir le résultat d'une invocation de méthode après la fin d'exécution du Servant. Le client, récupère une Future lors de l'invocation et à le choix

d'attendre la fin d'exécution (Synchronous waiting) ou de tester périodiquement la fin de l'exécution (polling).

```
public interface Future<V> {
    boolean cancel(boolean mayInterruptIfRunning);
    boolean isCancelled();
    boolean isDone();
    V get() throws InterruptedException, ExecutionException, CancellationException;
    V get(long timeout, TimeUnit unit) throws InterruptedException, ExecutionException, CancellationException, TimeoutException;
```

PARTIE 3 : IMPLÉMENTATION DU PC

C'est sur le thread client que l'invocation de méthode est faite (l'appel de la méthode). L'active object déclare la méthode en implémentant l'interface "proxy" (d'où le nom de "objet actif), sans que ce dernier soit bloqué jusqu'à la fin de l'exécution de la méthode.

Lors de l'invocation de méthode, l'Active Object crée un message contenant le nom de méthode et ses arguments et le message est placé dans une queue. Le message est traité par le scheduler qui crée un thread appelé Servant, pour chaque exécution de méthode.

Le client reçoit le résultat sous forme de Future (une sorte de callback).

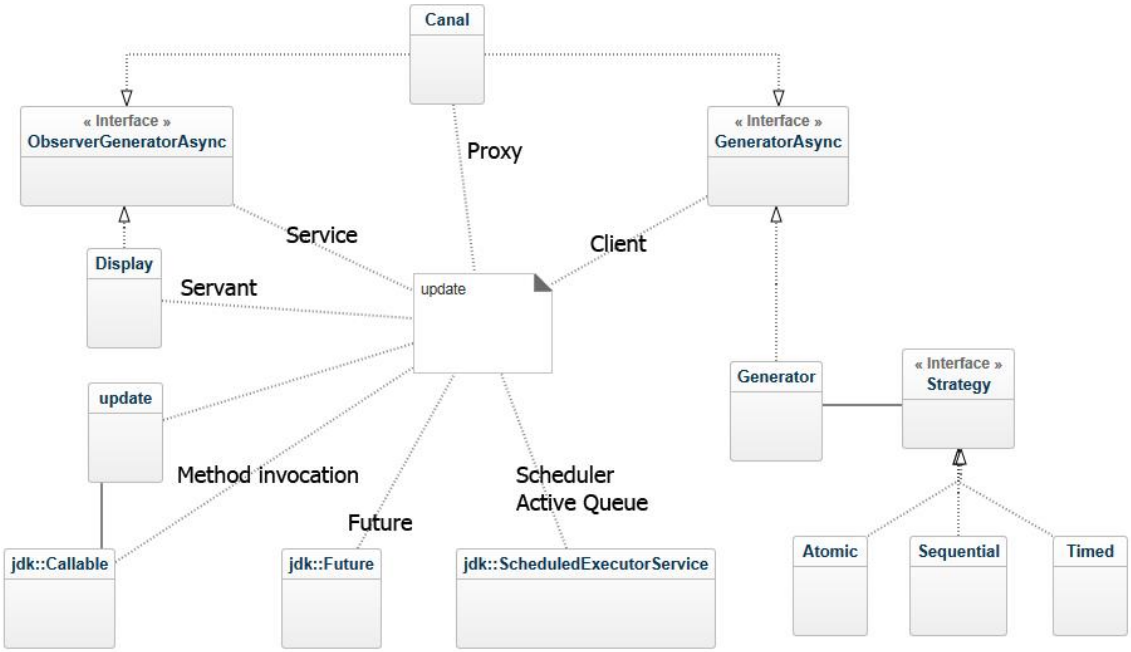
1.Stratégies de gestion de la cohérence :

Atomique : L'ordre d'arrivée est pris en compte et doit être dans l'ordre des numéros des afficheurs. L'envoi se fait une fois que le précédent est terminé.

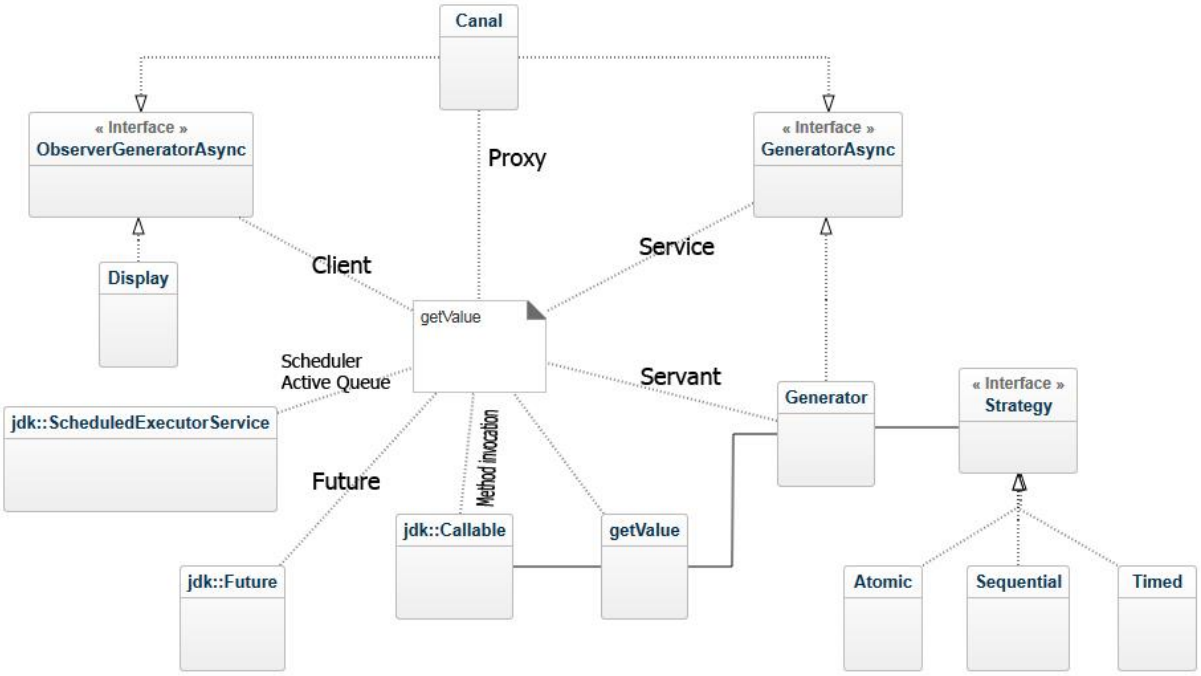
Séquentiel : L'ordre d'arrivée n'est pas pris en compte.

Par époque (timed) : Non implémenté.

2.Diagramme de classes :

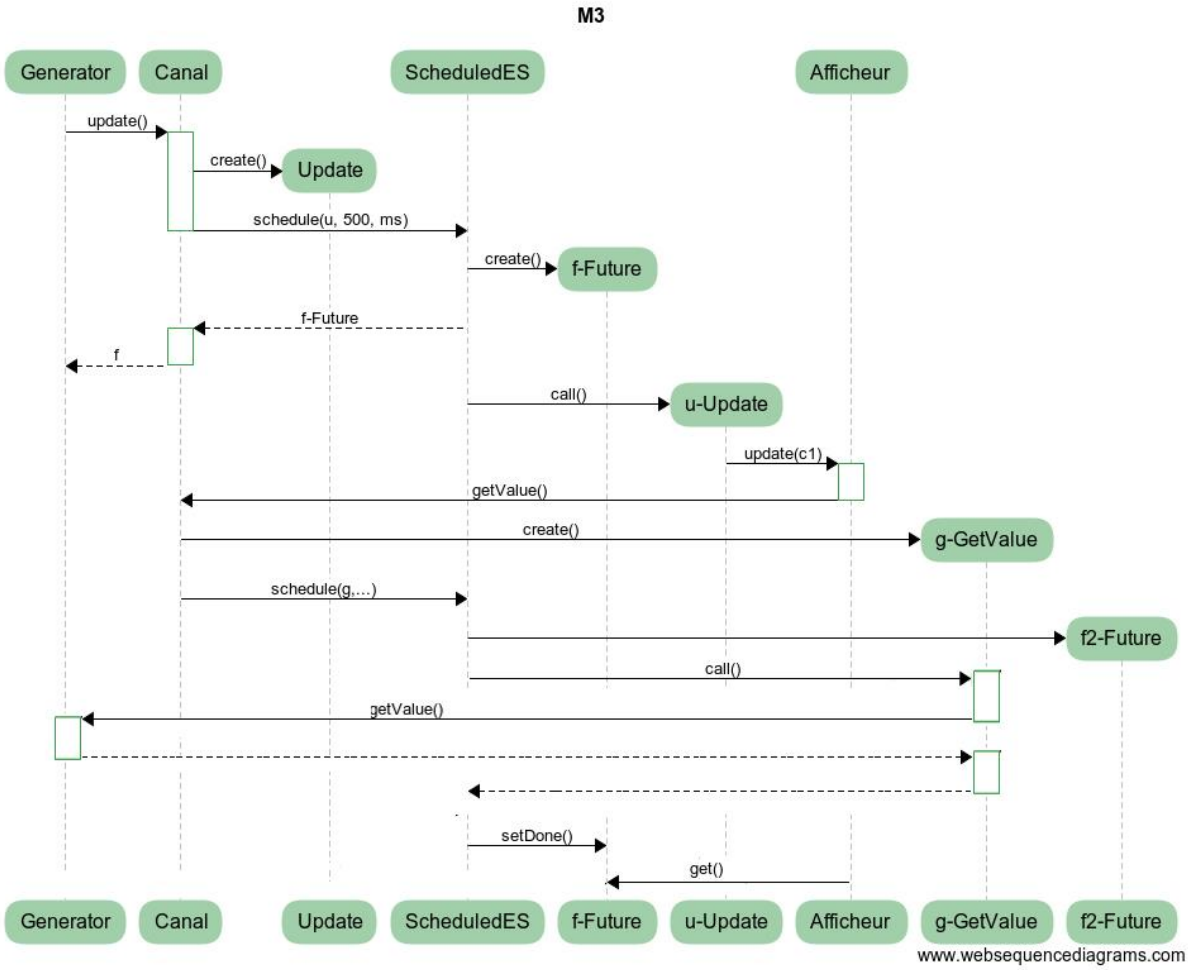


Update



GetValue

3.Diagramme de séquence



CONCLUSION

Le cœur du PC AO est une Activation Queue et un simple thread dispatcher. En utilisant ce patron de conception nous avons pu passer du déroulement du patron observer synchrone à un model asynchrone en deux temps à savoir update et getValue .

De plus le PC AO offre aux threads d'exécution la possibilité de n'apparaître que tels un objet unique et simple mais de surcroît, l'application peut traiter multiple demandes clients en mode parallèle (optimisé).