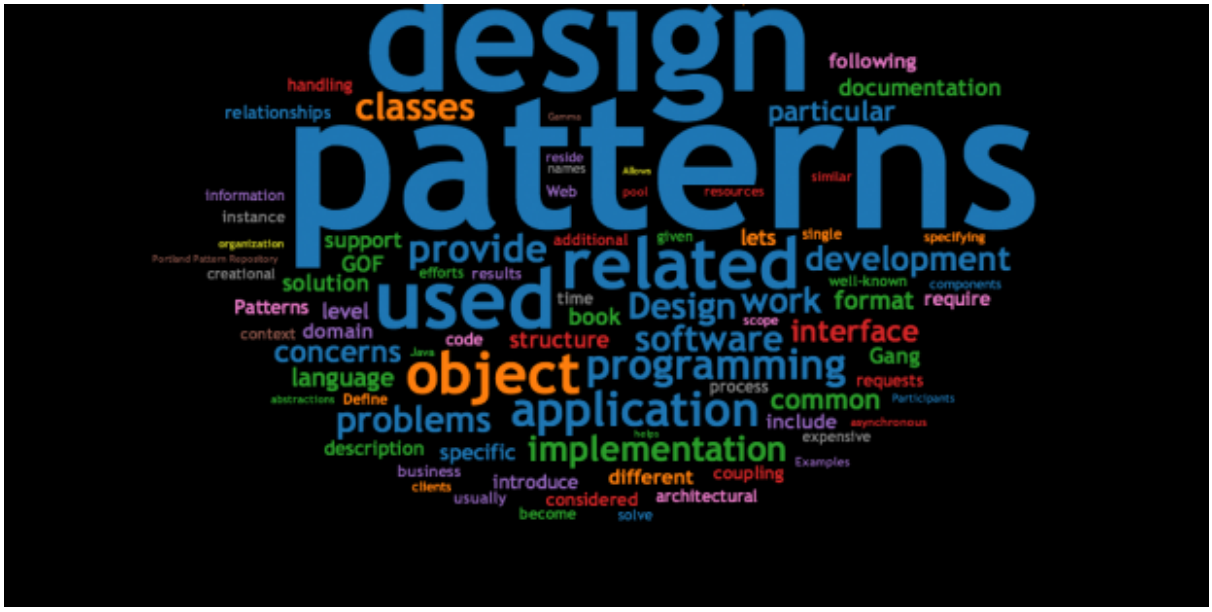
Rapport d'AOC

Mise en œuvre d'un patron de conception *Observer asynchrone* et l'application du PC *Active Object* ...



Année 2017/2018	**Spécialité**
Master 2 Informatique	Ingénierie Logicielle en Alternance

BINÔME	◆ **Guillaume BROSSAULT** ◆ **Abdelghani MERZOUK**
UE	**AOC** **Architectures à objets canoniques** (3 ECTS)

Table des matières :

Introduction **3**

Notre projet **4**

Spécifications 5

Définition des trois algorithmes de diffusion 6

Architecture du projet 7

Les patrons de conception utilisés 8

Diagramme de conception 9

Diagramme de séquence de notre implémentation 10

Résultat 11

Difficultées rencontrées **12**

Conclusion **13**

Lien github des sources du projet **14**

Introduction

Le module d'Architectures à objets canoniques (AOC) a pour but d'offrir aux étudiants la capacité de :

- mettre en œuvre des patrons de conception quelconques, dans un langage objet tel que Java
- appliquer une démarche de construction de logiciels à objets à partir d'un cahier des charges
- appliquer des techniques de conception à objets telles que l'inversion de contrôle et la séparation des préoccupations

La partie pratique (TP) guide à la mise en œuvre d'architectures asynchrones, via l'implémentation du patron de conception Active Object et l'utilisation du langage Java, en outre elle permet d'associer les principaux patrons de conception à une architecture asynchrone.

L'objectif du projet était de réaliser un générateur de valeurs numériques qui seront affichées sur quatre afficheurs différents. La solution à construire devait s'appuyer sur des mécanismes de programmation par « threads », le patron de conception observer ainsi que le patron de conception Active Object. Ce dernier nous a notamment permis de limiter la manipulation de « Threads » Java qui se révèle, le plus souvent, très fastidieuse. Nous avons commencé par analyser les besoins et la faisabilité du projet, puis rédigé les spécifications logicielles avant de réaliser la conception pour terminer par les tests et la validation.

1.Spécifications

Le but de cette application est de proposer à l'utilisateur un programme permettant à l'utilisateur de visualiser l'évolution d'une génération de nombres selon différents algorithme de diffusion de ces valeurs. Nous devions mettre en place un système de transmission des valeurs au sein de l'application basé sur des canaux diffusant les valeurs de leur génération vers leur affichage.

Ces canaux possèdent des délais de propagation aléatoires afin de simuler des latences. Aussi, l'application propose le choix de l'algorithme de diffusion à l'utilisateur.

Nous avons ainsi mis en place plusieurs éléments dans l'interface permettant de répondre à ces spécifications :

- Quatre afficheurs reliés à quatre canaux différents.

- Un bouton Lancer permettant de démarrer la génération.

- Un groupe de radio buttons permettant de sélectionner un algorithme avant de générer des valeurs.

Lorsque la génération a débuté, les quatre afficheurs sont mis à jour avec les nouvelles valeurs et permettent de visualiser les différences de diffusion selon l'algorithme choisis.

Cette interface a été développée en utilisant JavaFX avec un fichier au format fxml.

2. Définition des trois algorithmes de diffusion

Algorithme de diffusion atomique

Avec l'utilisation de la diffusion atomique toutes les valeurs sont envoyées sur tous les afficheurs. En effet, cet algorithme consiste à notifier les canaux chaque fois qu'une valeur est générée et de ne pas générer de nouvelle valeur avant que tous les afficheurs aient été mis à jour. A l'utilisation, on obtient ainsi un affichage de toutes les valeurs pour chaque afficheur et dans le même ordre.

Algorithme de diffusion séquentielle

Le fonctionnement de l'algorithme de diffusion séquentielle est très proche de celui de diffusion séquentielle. Cependant, le générateur n'est pas bloqué lorsque les canaux sont mis à jour. Il continue donc de générer des valeurs pendant que les canaux transmettent une valeur notifiée Cela se traduit par la mise à jour des afficheurs avec une copie de la valeur générée. On obtient ainsi l'affichage des mêmes valeurs pour chaque afficheur et dans le même ordre, mais certaines valeurs n'apparaîtront pas car le générateur aura généré une nouvelle valeur avant que la précédente n'ait été envoyée.

Algorithme de diffusion à époque

La diffusion à époque consiste à associer à chaque valeur une date de production et à les conserver. A chaque fois qu'un afficheur doit être notifié et mis à jour, la valeur utilisée sera la dernière donnée par le générateur, qui peut être récupéré grâce à sa date. Chaque afficheur affichera ainsi la dernière valeur produite par le générateur. De cette manière, les afficheurs n'auront pas toujours la même valeur.

3. Architecture du projet

Le projet se décompose en plusieurs package :

- Le package controller : il contient la classe Main permettant de lancer le programme notamment en chargeant la vue. L'interface du projet étant développée avec JavaFX, la classe Main étend la classe Application et implémente donc sa méthode start qui permet de charger un fichier fxml et d'en afficher l'interface correspondante.

- La package « model » qui se divise en trois sous-package correspondant aux différents design patterns utilisés autour d'Active Object. Nous avons ainsi le sous-package « strategy » qui contient les différents algorithmes utilisés pour la diffusion des valeurs, le sous package « proxy », qui contient les classes faisant partit du patron « proxy » et enfin « observer », qui a été formé en utilisant la même logique que « proxy ».

- La package view qui contient une classe « Controller » étendant la classe « Initializable » permettant de mettre en place une interface avec JavaFX. Cette classe implémente donc la méthode « initialize » appelé au chargement de la vue.

Le projet contient également un dossier « resources » contenant le fichier « afficheurs.fxml » qui est la définition de la vue au format fxml. Nous l'avons généré à l'aide de Scene Builder, qui est un outil permettant de construire une maquette d'interface. Nous avons ensuite connecté les éléments de cette interface pour qu'elle fonctionne avec notre application.

4. Les patrons de conception utilisés

Observer

Le patron de conception Observer est utilisé en programmation pour envoyer un signal à des modules qui jouent le rôle d'observateurs. En cas de notification, les observateurs effectuent alors l'action adéquate en fonction des informations qui parviennent depuis les modules qu'ils observent. Dans notre cas, les classes qui définissent les algorithmes utilisent le générateur. Ce sont ces modules qui notifient les observateurs.

Proxy

Un proxy est une classe se substituant à une autre. Par convention et simplicité, le proxy implémente la même interface que la classe à laquelle il se substitue. L'utilisation de proxy ajoute une indirection à l'utilisation de la classe à substituer.

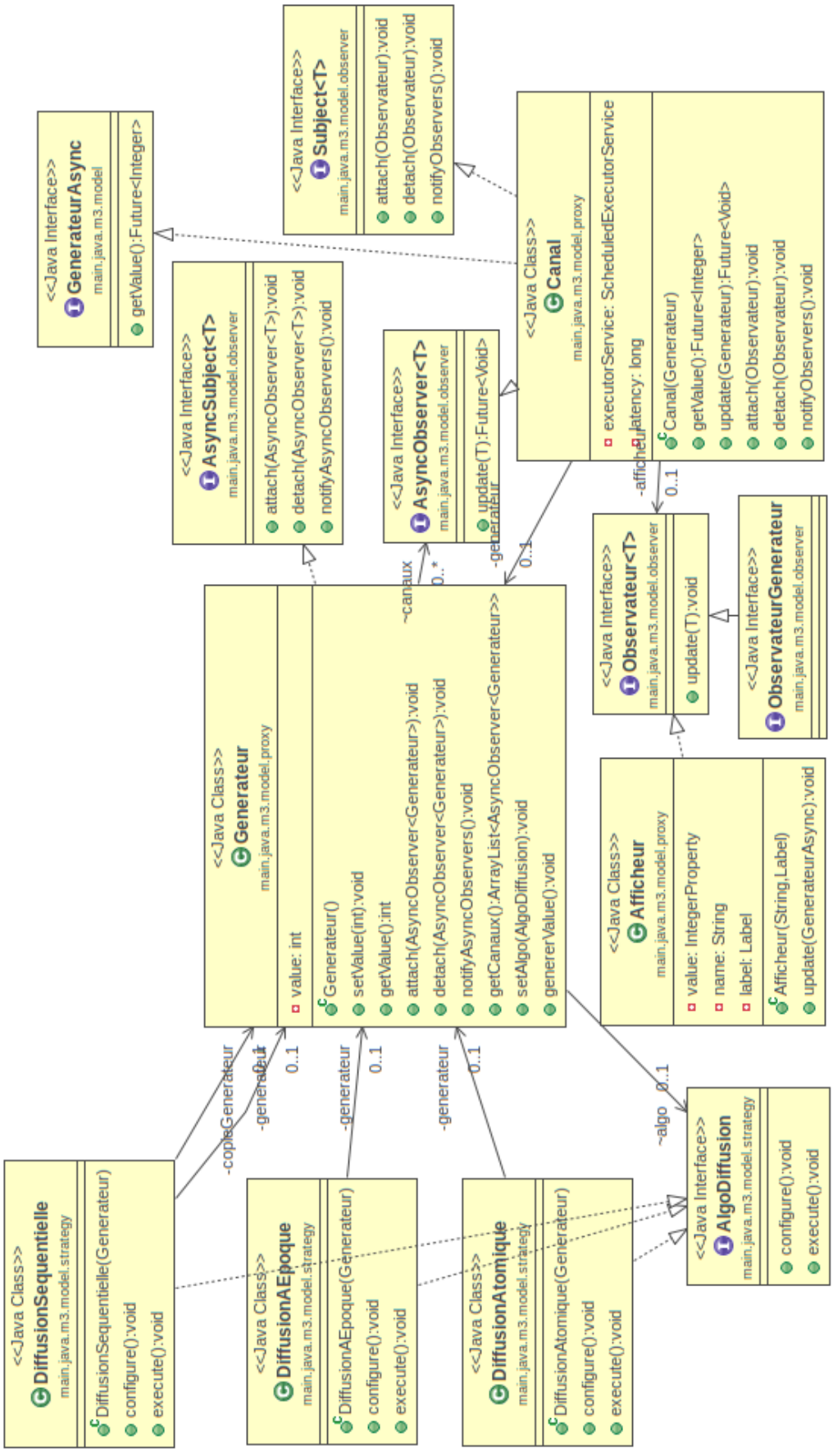
Strategy

Le patron stratégie est un patron de conception de type comportementale grâce auquel des algorithmes peuvent être sélectionnés à la volée au cours de l'exécution selon certaines conditions. Il est particulièrement utile pour des situations où il est nécessaire de permuter dynamiquement les algorithmes utilisés dans une application. Dans notre cas, nous utilisons 3 stratégies qui correspondent aux 3 algorithmes que nous souhaitons utiliser pour diffuser les valeurs du générateur.

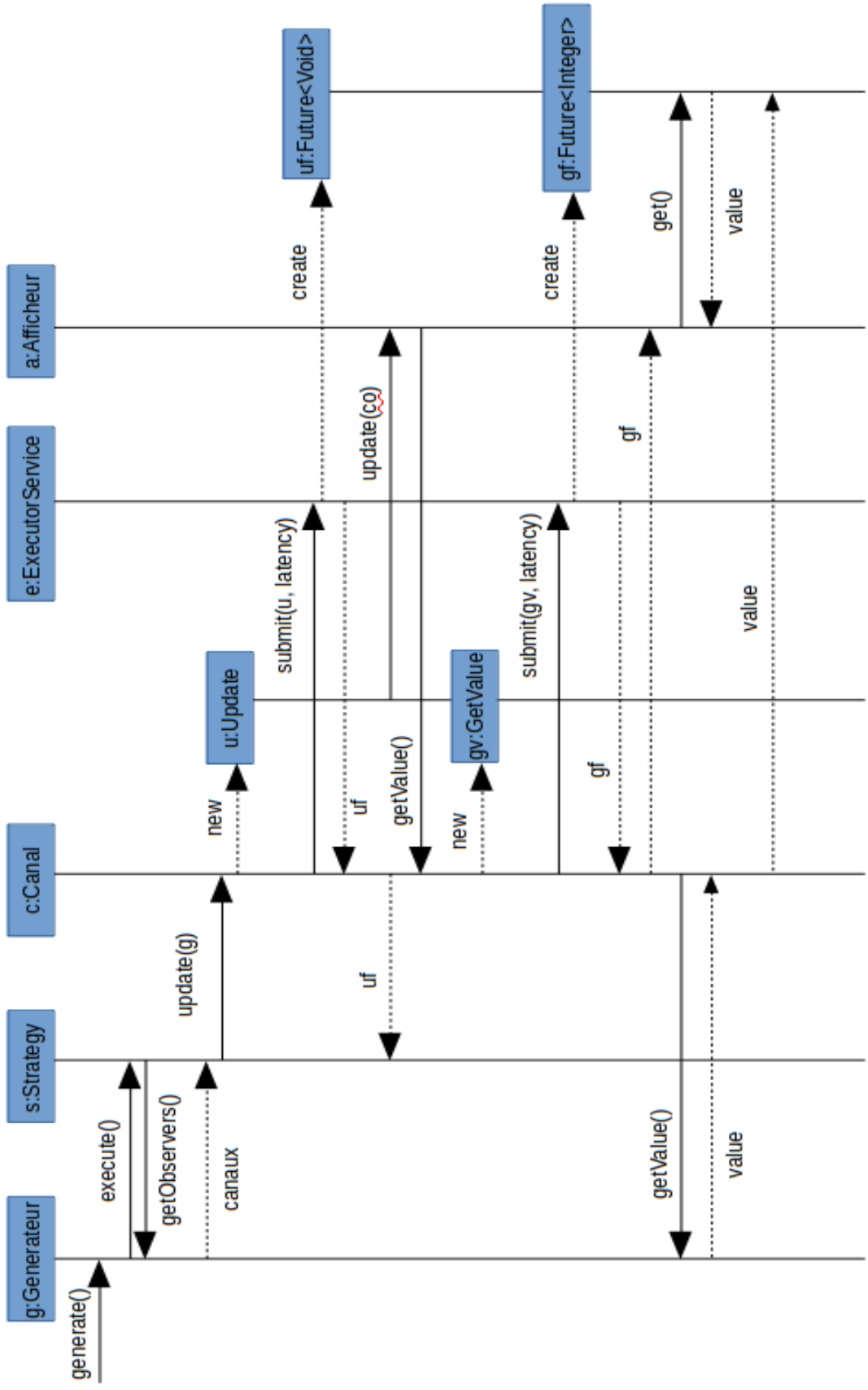
Active Object

Le patron de conception Active Object permet de découpler les méthodes d'exécution des méthodes d’invocation pour améliorer la concurrence et simplifier l'accès synchronisé d’un objet qui réside dans son propre thread de contrôle. Le principe est de pouvoir faire communiquer les objets entre eux de manières à ce que les messages échangés soient asynchrones. Le point fort de ce patron de conception est de permettre au développeur de limiter la programmation parallèle classique, avec les «Threads» Java par exemple. En effet, il est capable de gérer de manière automatique les actions asynchrones sans que le développeur n’ait à se soucier des éventuels problèmes de synchronisation et d’accès concurrent sur une même variable. L’ordre d'exécution de méthodes peut alors différer de leur ordre d’invocation. C’est ce que nous souhaitons visualiser dans ce projet à l’aide des afficheurs.

5. Diagramme de conception



6. Diagramme de séquence de notre implémentation



7. Résultat

Nous avons donc implémenté la vue avec les éléments que nous avons définis, c'est-à-dire les radio buttons, le bouton « Lancer » et les quatre afficheurs. Lorsque l'utilisateur appuie sur le bouton « Lancer », l'algorithme sélectionné est pris en compte (sauf l'algorithme « A époque » qui n'a pas été implémenté) et les valeurs sont générées. L'afficheur permet de visualiser l'évolution de ses valeurs. L'exécution s'arrête lorsque le nombre 20 est atteint.

Afin de pouvoir mettre à jour la vue au fur et à mesure de la génération des valeurs, nous avons utilisé les Service et Task JavaFX qui permettent de réaliser la génération de valeurs en arrière-plan pour de ne pas bloquer la vue pendant l'exécution.

Difficultés rencontrées

La phase d'analyse est une étape importante d'un projet informatique car elle permet une bonne compréhension du sujet et est nécessaire à la conception d'un programme efficace et fiable. La première difficulté que nous avons eue a donc été l'analyse du sujet et la compréhension de la mise en place des différents patron de conception autour d'Active Object. Il a également été important d'avoir une bonne compréhension des algorithmes de diffusion afin de trouver la meilleure façon de les formaliser en Java.

Une autre difficulté a été la mise en place de Active Object, avec l'utilisation de la bibliothèque standard Java pour les classes Future et Scheduler. Il a fallu nous documenter et analyser son fonctionnement à l'aide du diagramme de classe le représentant, mais aussi en utilisant des diagrammes de séquences qui nous ont permis d'assimiler son exécution.

Enfin, nous avons dû faire quelques recherches sur JavaFX utilisé avec un fichier fxml, afin d'utiliser au mieux les éléments de l'interface que nous avons créé, notamment les groupes de RadioButton, la détection d'actions et l'exécution en arrière-plan de la mise à jour des Label.

Conclusion

La réalisation de ce projet, avec la mise en place du patron de conception Active Object, nous a permis de renforcer notre compréhension des patrons de conception et de leurs implémentations.

Il y a en effet dans cette application plusieurs patrons de conception qui fonctionnent ensemble, dont Active Object, qui semblait plus complexe à mettre en place que les patrons que nous avions déjà vus, de son part le grand nombre de rôles qu'il définit. L'analyse et la construction de ce projet ont également amélioré notre appréhension des diagrammes de classes et de séquences, et la production de code Java à partir de ces éléments.

La mise en place d'Active Object nous a aussi permis de découvrir un moyen de mettre en place du parallélisme dans une application sans utiliser les outils que nous avions vu l'an dernier, notamment les Threads.

Lien Github des sources du projet

https://github.com/gbrossault/AOC_project