AOC report

Nicolas Fortun et Julien Laurent

January 2018

Contents

1 Introduction

Le but de ce projet de conception objet est de mettre en place le patron de conception *Active Object* afin de permettre des appels asynchrones sur un modèle synchrone. Le développement s'est effectué en trois étapes : conception du modèle synchrone, implémentation du patron *Active Object*, utilisation du JDK Oracle.

2 Choix techniques

JavaFX : Technologie Oracle permettant la création d'interfaces graphiques en Java. Elle permet une meilleure séparation du code métier/IHM que Swing, grâce à l'injection de dépendances et de l'utilisation d'un MVC. C'est pour cela que nous avons implémenté notre IHM en utilisant JavaFX.

Gluon : Gluon est une application embarquant un SceneBuilder, simplifiant l'écriture du fichier FXML pour JavaFX.

Java 8 : Nous avons développé notre application sous Java 8. Elle dispose des derniers ajouts, notamment des lambdas expressions. Cela permet d'avoir un code plus clair et facile à maintenir.

3 Modèle M1

Le modèle M1 de ce projet est l'implémentation d'observer synchrones.

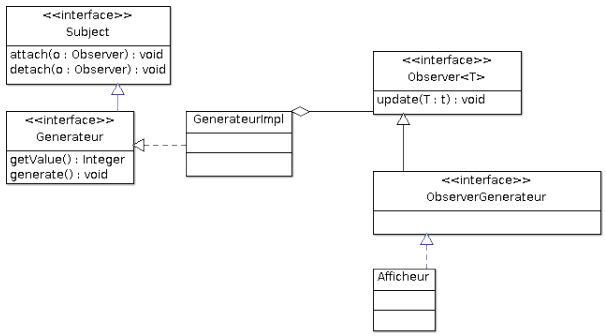

Figure 1: Diagramme de classe du modèle M1

Ce modèle est extrêmement simple : Lorsque le générateur génère un nombre, il notifie son observer (ici Afficheur) qui va juste afficher ce nombre dans la console.

4 Modèle M3

Ce modèle est l'implémentation du patron *Active Object*, s'appuyant sur la bibliothèque standard Oracle.

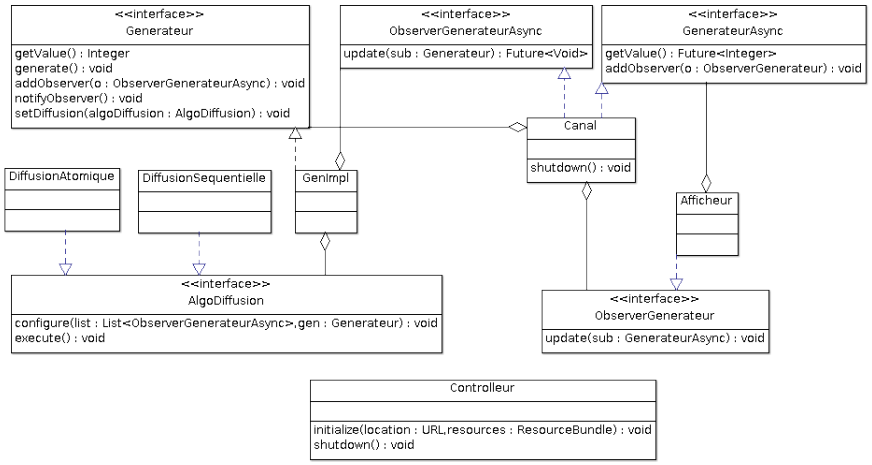

Figure 2: Diagramme de classe du modèle M3

4.1 Stratégie de diffusion

Ce modèle ajoute deux algorithmes de diffusion : séquentielle et atomique. Ces algorithmes ont pour but de définir la réaction du générateur par rapport à l'asynchronisme du programme. Ainsi, la diffusion atomique va notifier un observer et attendre que ce dernier ait fini, là où la diffusion séquentielle n'attend pas. Dans le deuxième cas, comme il n'y a pas d'attente, lorsque l'observer notifié demandera la valeur générée, elle aura potentiellement changé depuis la notification. Cet algorithme a l'avantage d'être rapide, puisqu'il n'y a pas d'attente. Cependant s'il y a une logique dans la génération de la valeur, elle sera potentiellement perdue.

La mise en place du pattern stratégie permet de changer d'algorithme de diffusion de manière efficace en limitant la duplication de code. De plus, l'ajout d'un algorithme de diffusion est plus simple.

4.2 Exécution du programme

Le diagramme de séquence suivant explique la génération d'un nombre aléatoire. Le générateur génère une valeur aléatoire et notifie ensuite le canal. Celui-ci crée une *method invocation* qui est ensuite ajouté au *scheduler* et qui sera exécutée de manière asynchrone.

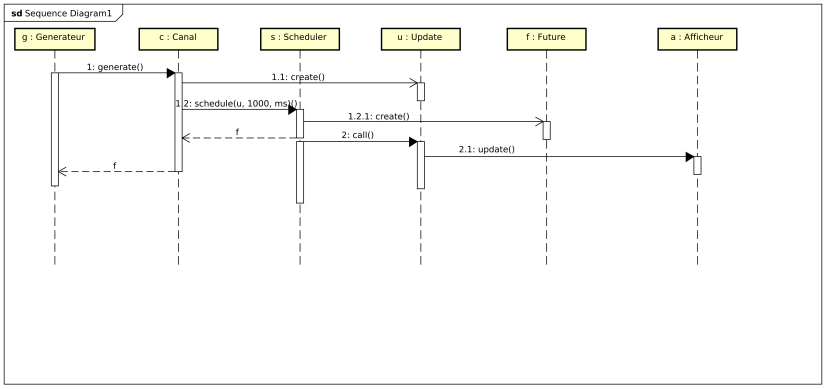

Figure 3: Génération d'un nombre aléatoire par le modèle M3

Le diagramme suivant explique la récupération d'une valeur générée aléatoirement.

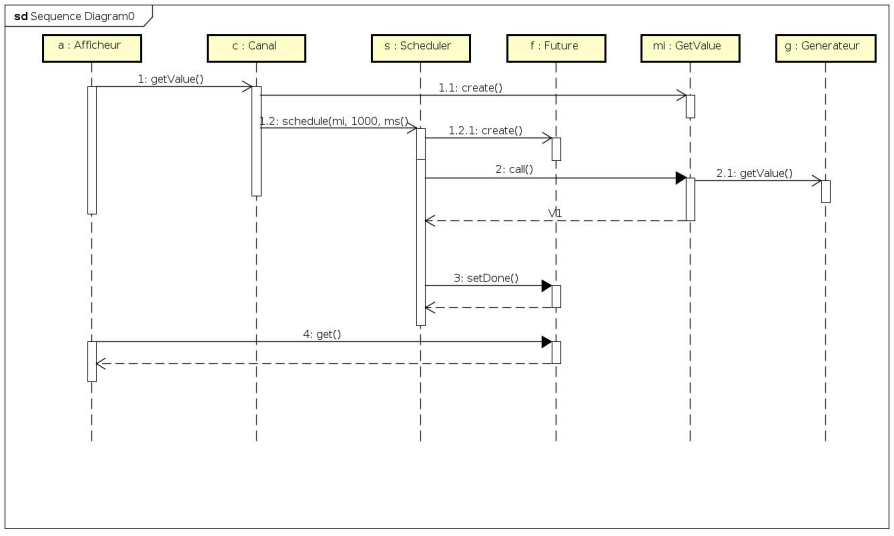

Figure 4: Récupération du nombre généré par le modèle M3

5 Tests

5.1 Modèle M1

La simplicité de ce modèle ne nous permet pas de faire de tests unitaires, car de lourdes modifications étaient prévues pour inclure l'asynchronisme ainsi que

l'interface graphique. De plus l'utilisateur n'a absolument aucun impact sur son exécution. Par conséquent lancer le programme et vérifier que le résultat obtenu est celui attendu est suffisant.

5.2 Modèle M3

La testabilité de l'application a été difficile, étant donné que le patron de conception *Active Object* est une mise en place d'appels asynchrones. Nous avons donc procédé à des tests ponctuels, via l'ajout de logs dans notre code. Notre application possédant aussi une IHM, nous avons effectué divers tests graphiques.

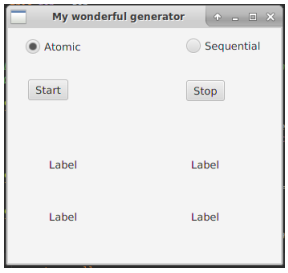

Figure 5: IHM JavaFX du projet AOC

Lors de la sortie du programme, l'exécution de la fonction *ExecutorService::shutdownNow* peut entraîner une *InterruptedException*. Il nous a été recommandé d'utiliser cette fonction car gère elle-même l'arrêt des pools de threads. De plus, étant donné que nous ne sauvegardons rien et que l'appel ne se fait qu'à la fermeture du programme, l'exception n'est pas gênante.