TP Observer

Dossier de conception

Equipe
Rémi Viotty
Maeva Espana

Année universitaire 2020 - 2021

Sommaire

Sommaire 1

Introduction 2
- Rappel du contexte général du projet 2
- Objectifs du document et plan 2

Description de l'application 3
- Objectifs de l'application 3
- Architecture globale de l'application 3
- Choix techniques 3
- Dépendances du projet 4

Conception 4
- Diagramme de classe 4
- Diagrammes de séquence 5

Validation 6
- Design des test 6
- Couverture des tests 7

I. Introduction

i. Rappel du contexte général du projet

Ce projet est un TP pour le module AOC du M2 Ingénierie Logicielle de l'ISTIC, Université de Rennes 1. Le but à atteindre est de réaliser un service de diffusion de données de capteur. La solution construite doit s'appuyer sur le patron de conception **Active Object**, ainsi que les autres patrons de conception étudiés lors de notre formation. Notre groupe est formé des étudiants alternants Rémi Viotty et Maeva Espana.

ii. Objectifs du document et plan

L'objectif de ce document est de simplifier la mise en œuvre de notre projet en éclaircissant ses points importants et en mettant ces derniers en avant, afin de ne pas oublier sur quoi nous devons nous focaliser. C'est un document particulièrement important dans le cycle de conception/production d'un produit, dont l'utilité est mise en lumière lorsqu'un projet est repris par quelqu'un d'autre, ou lorsque les objectifs commencent à être oubliés. Il permet de montrer comment le produit a été conçu à l'origine, sur quelles bases, et peut également souligner des problèmes de conception, qu'on pourra plus facilement résoudre dans le futur.

Ce dossier de conception est composé des éléments suivants:

- Une première partie qui a pour but de décrire l'application (objectifs, architecture globale, choix techniques, dépendances du projet) ;
- Une seconde partie concentrée sur le côté conception (diagrammes) ;
- Une dernière partie focalisée sur la validation, qui regroupe le design des tests, ainsi que la couverture des tests.

II. Description de l'application

i. Objectifs de l'application

Le service de diffusion élaboré permettra de diffuser un flot de valeurs vers des objets abonnés exécutés dans des threads différents de la source de service. Notre objectif est la mise en œuvre parallèle du patron Observer. Les données diffusées seront une séquence croissante d'entiers. Ce compteur sera incrémenté à intervalle fixe. Enfin, la transmission de l'information vers les abonnés au service emploiera un canal avec un délai de transmission aléatoire.

ii. Architecture globale de l'application

L'architecture globale de l'application doit être composée des éléments suivants:

- Une source active (capteur), dont la valeur évoluera de façon périodique,
- Un ensemble de canaux de transmission avec des délais variables,
- Un ensemble de politiques de diffusion Observer, comprenant :
    - la diffusion atomique (tous les observateurs reçoivent la même valeur, qui est celle du sujet),
    - la diffusion séquentielle (tous les observateurs reçoivent la même valeur, mais la séquence peut être différente de la séquence réelle du sujet),
    - la gestion par époque comme indiquée en cours.

iii. Choix techniques

Nous avons fait certains choix techniques qui nous semblaient nécessaires et justifiés afin de pouvoir implémenter certains points de l'application :

- Les valeurs échangées du capteur ne sont pas du type *Integer* mais du type *StampedValue*, classe que nous avons implémenté et qui contient un *Integer* représentant une valeur et un *long* qui représente la date d'émission de la valeur.

iv. Dépendances du projet

Afin de mener à bien notre projet, nous nous sommes appuyés sur des outils externes. Nous avons opté pour un projet Maven, qui permet une gestion simple des dépendances et une séparation nette et compréhensible entre les package de tests et les packages de notre application.
Ainsi, notre fichier pom.xml contient toutes les dépendances utilisées dans ce projet, qui sont listées ci dessous:

- AssertJ : une bibliothèque open source Java très populaire permettant de rédiger des tests clairs, grâce à un panel d'assertions explicites.
- SLF4F : (ou Simple Logging Facade for Java), est un outil offrant une couche d'abstraction sur les solutions de log de Java
- Logback : qui permet de surveiller les logs générés par SLF4F

III. Conception

i. Diagramme de classe

Le diagramme étant trop grand pour être affiché sur ce document, nous l'avons mis à disposition sous forme de fichier *.drawio* dans notre répertoire GIT.

ii. Diagrammes de séquence

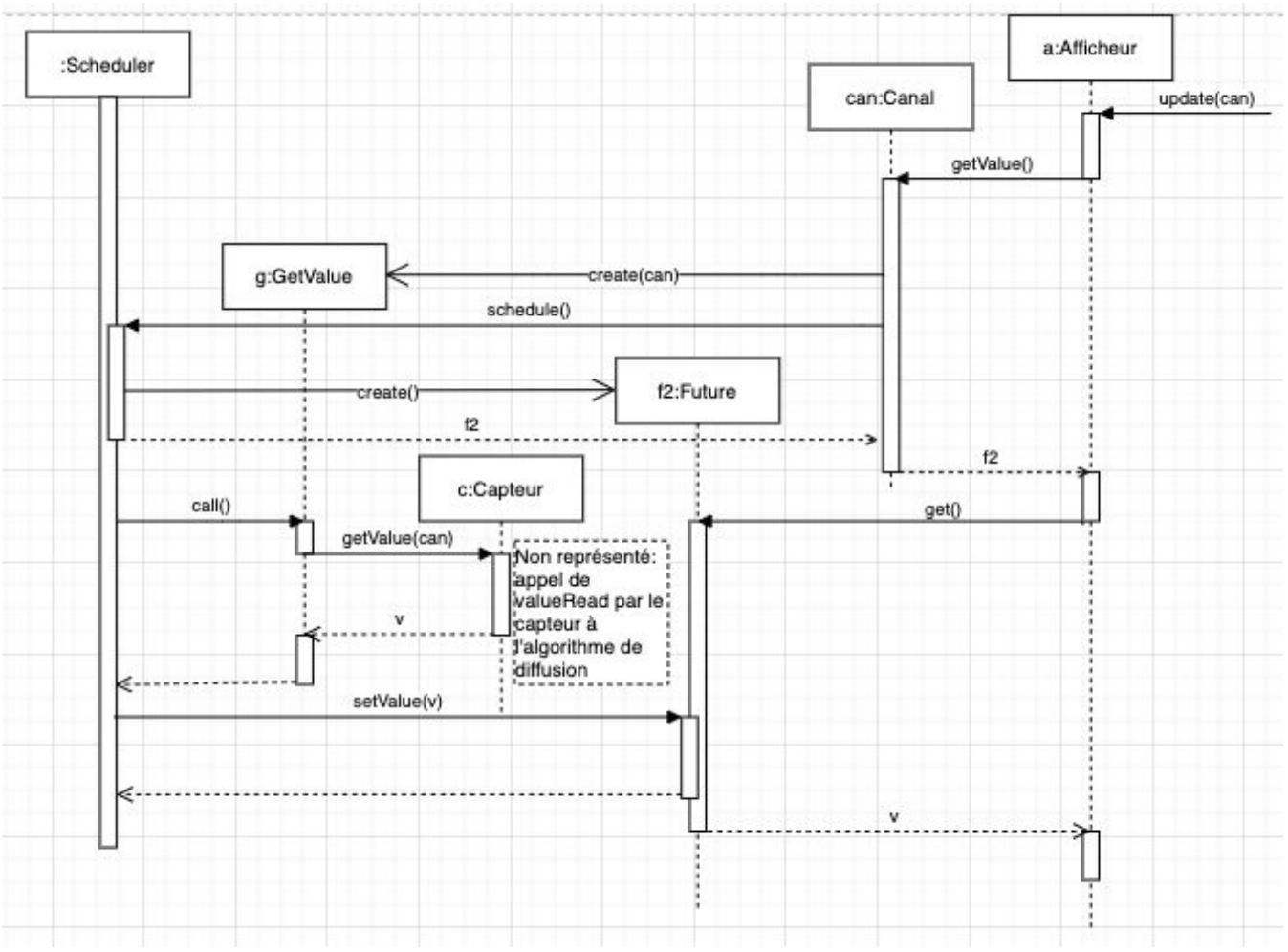

Diagramme de séquence représentant l'appel de update sur l'afficheur

IV. Validation

i. Design des test

Nous avons une classe de tests comprenant un test pour chaque algorithme de diffusion. Comme spécifié dans la sous-partie portant sur les dépendances du projet, nos Afficheurs enregistrent leurs message dans un logger SLF4F, ainsi nous avons choisi de “surveiller” les logs grâce à Logback pour tester si les suites de valeurs renvoyées par les afficheurs étaient conformes aux différentes logiques de diffusions.
Pour mettre en place les tests, nous utilisons deux annotations JUnit :

- *@Before* afin de construire les objets communs et nécessaires à l’exécution des algorithmes de diffusion, nous y construisons donc 1 pool d’exécution, 1 capteur, 4 canaux et 4 afficheurs mais aussi un objet *MemoryAppender* qui étend de *ListAppender* et permet de consulter la liste des messages diffusés par les afficheurs.
- *@After* afin de fermer le pool d’exécution et d’attendre 3 secondes que tous les threads aient eu le temps de finir de s’exécuter, et puissent pas perturber les logs générés et analysés par les tests suivants.

Les assertions de nos tests s’appuient sur l’outil AssertJ.

ii. Couverture des tests

Afin de déterminer la couverture de nos tests, nous avons utilisé l'outil *Code coverage* de IntelliJ. Ci-dessous les résultats obtenus :

Overall Coverage Summary

Package	Class, %	Method, %	Line, %
all classes	100% (11/ 11)	97,5% (39/ 40)	94,7% (107/ 113)

Coverage Breakdown

Package	Class, %	Method, %	Line, %
com.rviottymespana	100% (5/ 5)	95,2% (20/ 21)	89,5% (51/ 57)
com.rviottymespana.algosdiffusion	100% (3/ 3)	100% (12/ 12)	100% (39/ 39)
com.rviottymespana.methodinvocations	100% (2/ 2)	100% (4/ 4)	100% (11/ 11)
com.rviottymespana.models	100% (1/ 1)	100% (3/ 3)	100% (6/ 6)

Code coverage global du projet

Coverage Summary for Package: com.rviottymespana

Package	Class, %	Method, %	Line, %
com.rviottymespana	100% (5/ 5)	95,2% (20/ 21)	89,5% (51/ 57)

Class	Class, %	Method, %	Line, %
Afficheur	100% (1/ 1)	100% (4/ 4)	85,7% (12/ 14)
Canal	100% (1/ 1)	100% (3/ 3)	100% (9/ 9)
CapteurImpl	100% (2/ 2)	100% (11/ 11)	90,3% (28/ 31)
CapteurUtils	100% (1/ 1)	66,7% (2/ 3)	66,7% (2/ 3)

Détails du code coverage du package n'ayant pas une couverture de 100%

On constate que la quasi-totalité de notre code est couvert par nos tests, en y regardant de plus près, on remarque que les lignes qui ne sont pas couvertes par les tests sont les blocs permettant de gérer les exceptions, certaines d'entre elles ne pouvant pas être produites dans notre cas.

```
 9  public class Afficheur implements ObserverDeCapteur {
10
11      private static final Logger logger = LoggerFactory.getLogger(Afficheur.class);
12
13      private final Integer indexAfficheur;
14
15      private long lastTimeStamp;
16
17      public Afficheur(Integer indexAfficheur) {
18          this.indexAfficheur = indexAfficheur;
19          this.lastTimeStamp = 0;
20      }
21
22      @Override
23      public void update(CapteurAsync subject) {
24          try {
25              StampedValue stampedValue = subject.getValue().get();
26              // vérification utile pour l'algorithme de gestion par époque
27              if (stampedValue.getTimestamp() > lastTimeStamp) {
28                  lastTimeStamp = stampedValue.getTimestamp();
29                  logger.info("Afficheur {} : Valeur reçue {}", indexAfficheur, stampedValue.getValue());
30              }
31          } catch (InterruptedException | ExecutionException e) {
32              e.printStackTrace();
33          }
34      }
35
36
37      public Integer getIndexAfficheur() {
38          return indexAfficheur;
39      }
40  }
```

Code de la classe Afficheur affichant que les lignes non couvertes (en rouge) par nos tests sont les exceptions

Pour plus de détails, nous avons intégré le rapport du coverage dans le dossier *coverage_report_AOC* à la racine de notre repository GIT.