TELECOM SudParis

Exemple de questions en Programmation

Les seuls documents autorisés sont ceux distribués en cours et en TP, et mis à disposition sur le site Web du module, ainsi que vos notes personnelles.

Notes en préambule :

- les questions qui suivent constituent un entraînement sur la partie « programmation » pour le contrôle final de la deuxième session du module CSC4102. Elles sont extraites d'un ancien sujet. Dans le cadre du module CSC4102, pour couvrir l'ensemble du programme, nous y ajouterions une question sur les « idiomes JAVA » (par exemple sur les méthodes `equals` et `hashCode`) ;
- la durée estimée pour répondre à toutes les questions qui suivent est 1h30 ;
- un barème est donné à titre indicatif ;
- soyez concis et précis, et justifiez vos réponses par des commentaires appropriés ;
- soyez rigoureux dans la syntaxe `JAVA` ;
- **veillez à rendre une copie propre et lisible, avec une marge à gauche.**

1 Sujet

Étude de cas. Le système dont le diagramme de classes est présenté dans la figure 1 permet de communiquer en échangeant des événements de manière asynchrone : les producteurs et les consommateurs d'événements n'ont pas besoin de se connaître. Un événement est modélisé dans la classe Publication contenant un message, c'est-à-dire une chaîne de caractères.

Un consommateur désirant recevoir de l'information, c'est-à-dire des publications, construit un filtre et l'enregistre (avec la méthode subscribe) auprès du courtier (la classe Broker). Un producteur désirant publier de l'information construit une publication et appelle la méthode publish du courtier. Comme spécifié dans la figure 2, lors de l'appel à la méthode publish, le courtier parcourt toutes les souscriptions et appelle la méthode consume de tous les consommateurs pour lesquels la publication « passe » le filtre : la méthode evaluate retourne `true`. Pour votre curiosité, c'est le patron de conception « Publier/Souscrire » (en anglais, « Publish/Subscribe ») qui est mis en œuvre. C'est pour cela que le nom du paquetage dans le diagramme de classes est appelé pubsub.

Les filtres sont soit des filtres simples (les classes FilterStringEndsWith, FilterStringEquals, FilterStringStartsWith), soit des filtres composites. Comme l'indiquent le nom des classes des filtres simples ainsi que le type de l'attribut message de la classe Publication, tous les filtres simples fonctionnent sur des chaînes de caractères. Comme spécifié dans la figure 2, la méthode evaluate du filtre récupère le message de la publication, puis, selon le type de filtre, applique la méthode endsWith ou la méthode equals ou encore la méthode startsWith.

Un filtre composite (classe CompositeFilter) est composé de filtres et la méthode evaluate du filtre composite opère un `ET` logique sur les résultats des appels aux méthodes evaluate des filtres. Pour votre curiosité, nous mettons ici en œuvre le patron de conception « Composite ».

Explications sur la conception Voici quelques éléments supplémentaires :

- toutes les classes font partie du paquetage `pubsub` ;
- classe Broker : « souscrire » (méthode subscribe) consiste à créer une souscription et à l'ajouter dans la collection en vérifiant qu'elle ne l'est pas déjà ;
- classe CompositeFilter : un filtre composite est une conjonction de filtres : dans l'exemple de la figure 3, l'évaluation du filtre f5 retourne true si et seulement si les filtres f3 et f4 retournent tous les deux true ;
- classe Consumer : deux objets de cette classe sont égaux (méthode equals) si et seulement si leurs noms sont égaux ;
- classe ConsumerDisplay : la méthode consume affiche la publication tel que montré dans l'affichage de la figure 4 ;
- classe FilterStringEquals : la méthode evaluate utilise la méthode equals de la classe String pour comparer la chaîne de caractères du filtre avec le message de la publication ;
- classe Subscription : deux objets de cette classe sont égaux (méthode equals) si et seulement s'ils ont les mêmes consommateurs et les mêmes filtres.

2 Questions

Les questions suivantes permettent de réaliser le système. **Jusqu'à la question 7 incluse, vous devez ignorer les cas d'erreur, excepté ceux traités explicitement dans le diagramme de séquence de la figure 2.**

Conseil : pensez à étudier le code de la méthode main dans la figure 3 pour connaître les prototypes des méthodes non montrés dans le diagramme de classes de la figure 1 : par exemple pour les constructeurs.

Question 1 [2 pts] : écrivez tout le code de la classe Publication.

Question 2 [2,5 pts] : écrivez le code de la classe Consumer, excepté les méthodes hashCode et equals.

Question 3 [1,5 pts] : écrivez tout le code de la classe ConsumerDisplay. L'affichage à l'écran utilise la méthode toString de la classe Publication.

Question 4 [2,5 pts] : écrivez le code de la classe Subscription, excepté les méthodes hashCode et equals.

Question 5 [3,5 pts] : écrivez tout le code de la classe Broker en évitant qu'il y ait deux souscriptions identiques dans la collection.

Question 6 [2 pts] : écrivez le code de la classe CompositeFilter.

Question 7 [2 pts] : écrivez le code de la classe FilterStringEquals.

Question 8 [2 pts] : réécrivez la méthode subscribe de la classe `Broker` pour qu'elle lève l'exception IllegalArgumentException lorsqu'il existe déjà une souscription pour ce consommateur et ce filtre. L'exception IllegalArgumentException est déjà disponible dans les classes de base de Java.

Question 9 [2 pts] : écrivez la classe TestSubscribe qui contient un test écrit avec JUnit qui teste la levée d'exception de la question précédente. Faites en sorte d'avoir trois méthodes pour l'initialisation, le test et le nettoyage.

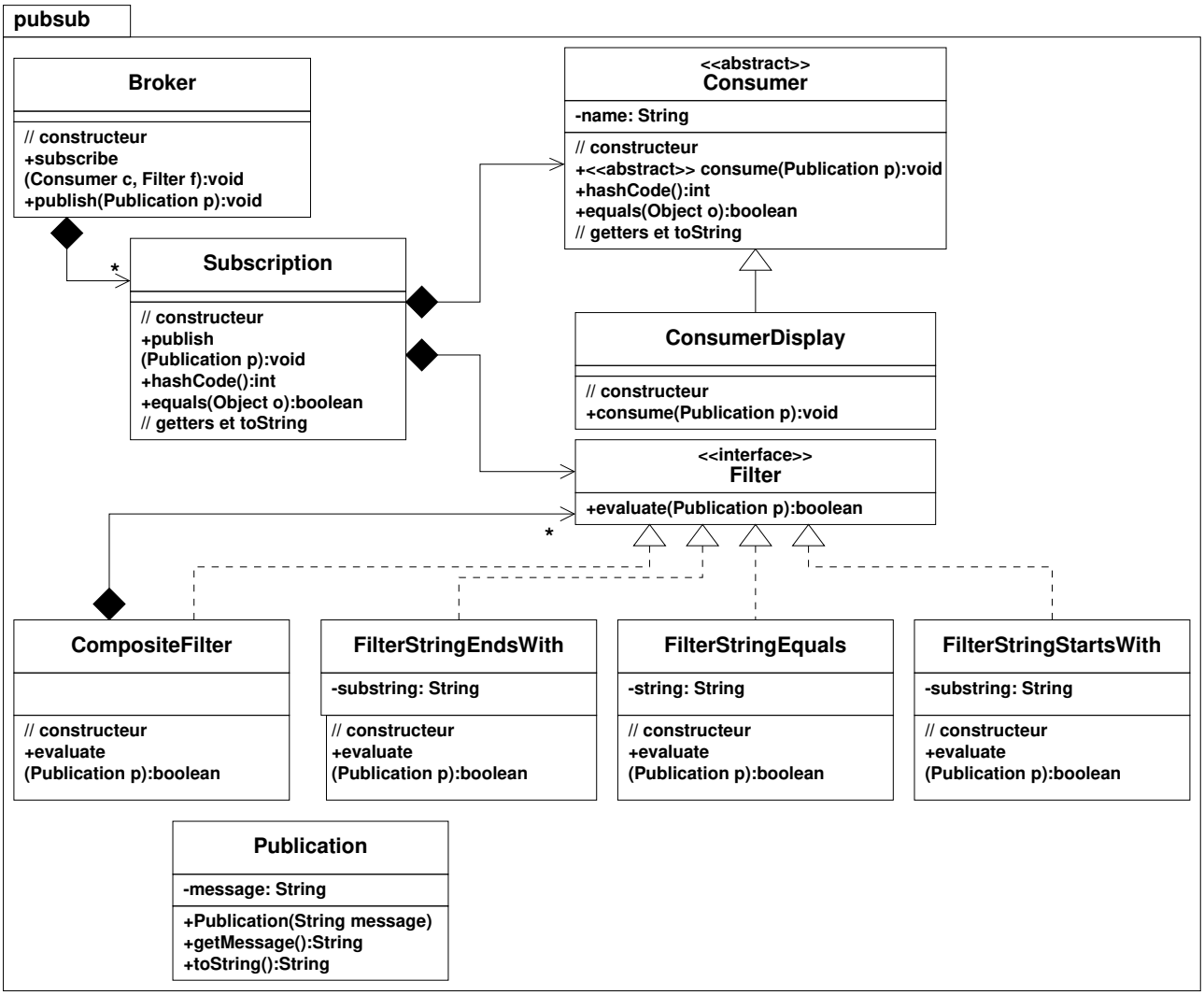

FIGURE 1 – Diagramme de classes

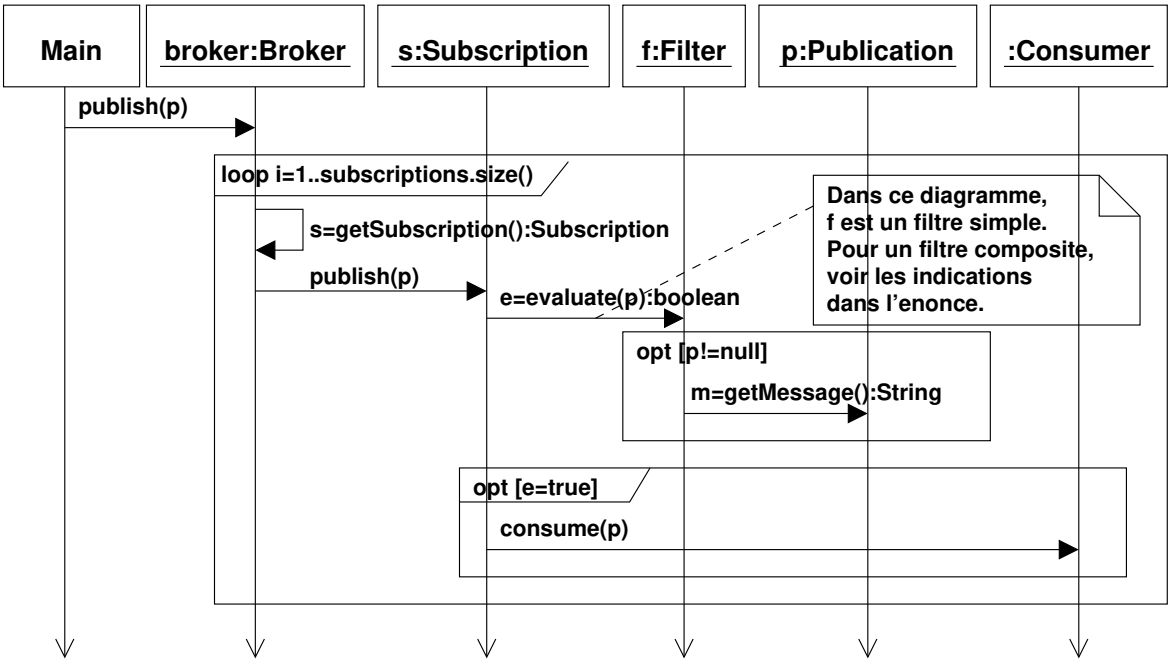

FIGURE 2 – Diagramme de séquence de la publication avec un filtre simple.

```
1  import java.util.List;
2  import java.util.Vector;
3  import pubsub.Broker;
4  import pubsub.CompositeFilter;
5  import pubsub.ConsumerDisplay;
6  import pubsub.Filter;
7  import pubsub.FilterStringEndsWith;
8  import pubsub.FilterStringEquals;
9  import pubsub.FilterStringStartsWith;
10 import pubsub.Publication;
11
12 public class Main {
13         public static void main(final String[] args) {
14                 ConsumerDisplay c1 = new ConsumerDisplay("c1");
15                 ConsumerDisplay c2 = new ConsumerDisplay("c2");
16                 ConsumerDisplay c3 = new ConsumerDisplay("c3");
17                 ConsumerDisplay c4 = new ConsumerDisplay("c4");
18                 Broker broker = new Broker();
19                 Filter f1 = new FilterStringEquals("hello");
20                 broker.subscribe(c1, f1);
21                 Filter f2 = new FilterStringStartsWith("hello");
22                 broker.subscribe(c2, f2);
23                 List<Filter> fs1 = new Vector<Filter>();
24                 Filter f3 = new FilterStringStartsWith("hello");
25                 fs1.add(f3);
26                 Filter f4 = new FilterStringEndsWith("world");
27                 fs1.add(f4);
28                 Filter f5 = new CompositeFilter(fs1);
29                 broker.subscribe(c3, f5);
30                 List<Filter> fs2 = new Vector<Filter>();
31                 fs2.add(new FilterStringStartsWith("hello"));
32                 fs2.add(new FilterStringEndsWith("word"));
33                 Filter f6 = new CompositeFilter(fs2);
34                 broker.subscribe(c4, f6);
35                 broker.publish(new Publication("hello world"));
36         }
37 }
```

FIGURE 3 – Exemple de méthode main utilisant le système.

```
c2 is notified: Publication [message=hello world]
c3 is notified: Publication [message=hello world]
```

FIGURE 4 – Affichage correspondant à l'exemple : le consommateur `c1` n'est pas notifié car « `!"hello".equals("hello world")` » et le consommateur `c4` n'est pas notifié car « `!"hello world".endsWith("word")` ».