TP n°3 - Boulangerie et Boîte aux lettres

CHOISY
Alexis
INF3-FA

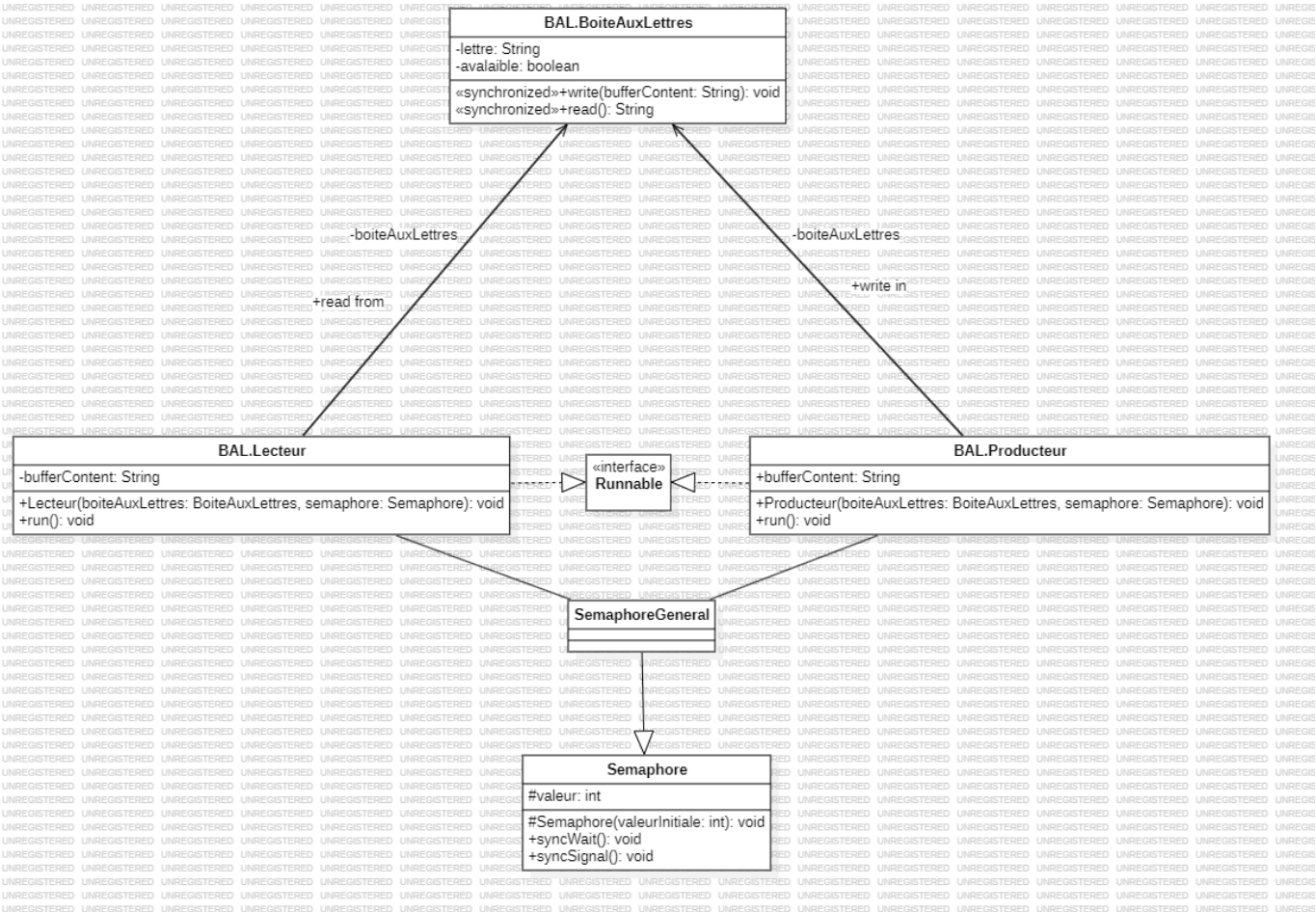
Introduction

Dans ce TP est abordé le design pattern de producteur-consommateur avec comme exemple une boîte aux lettres et une boulangerie

---

BAL

La boîte aux lettres se base sur le design pattern de producteur-consommateur où on a une classe `Producteur.java` qui va mettre à disposition (dans cet exemple) une lettre dans la boîte aux lettres qui est elle représentée par une classe `BoiteAuxLettres.java` et une classe `Lecteur.java` va retirer et lire la lettre de celle-ci.



Ici dans mon implémentation de la BAL j'ai une lettre représentée par un `String`, les deux `Producteur.java` et `Lecteur.java` on un buffer pour stocker respectivement la lettre à déposer et la lettre à lire stocké sur le moniteur `BoiteAuxLettres` qui a un champs pour la lettre stockée et l'autre pour savoir si celle-ci est en ce moment accessible. J'ai aussi fait appel à une Semaphore afin de pouvoir écrire des entrées par clavier avec `Scanner`, concrètement le Thread qui correspond au lecteur va dormir lorsque le producteur va écrire et le Thread qui correspond au producteur va dormir quand le lecteur lit

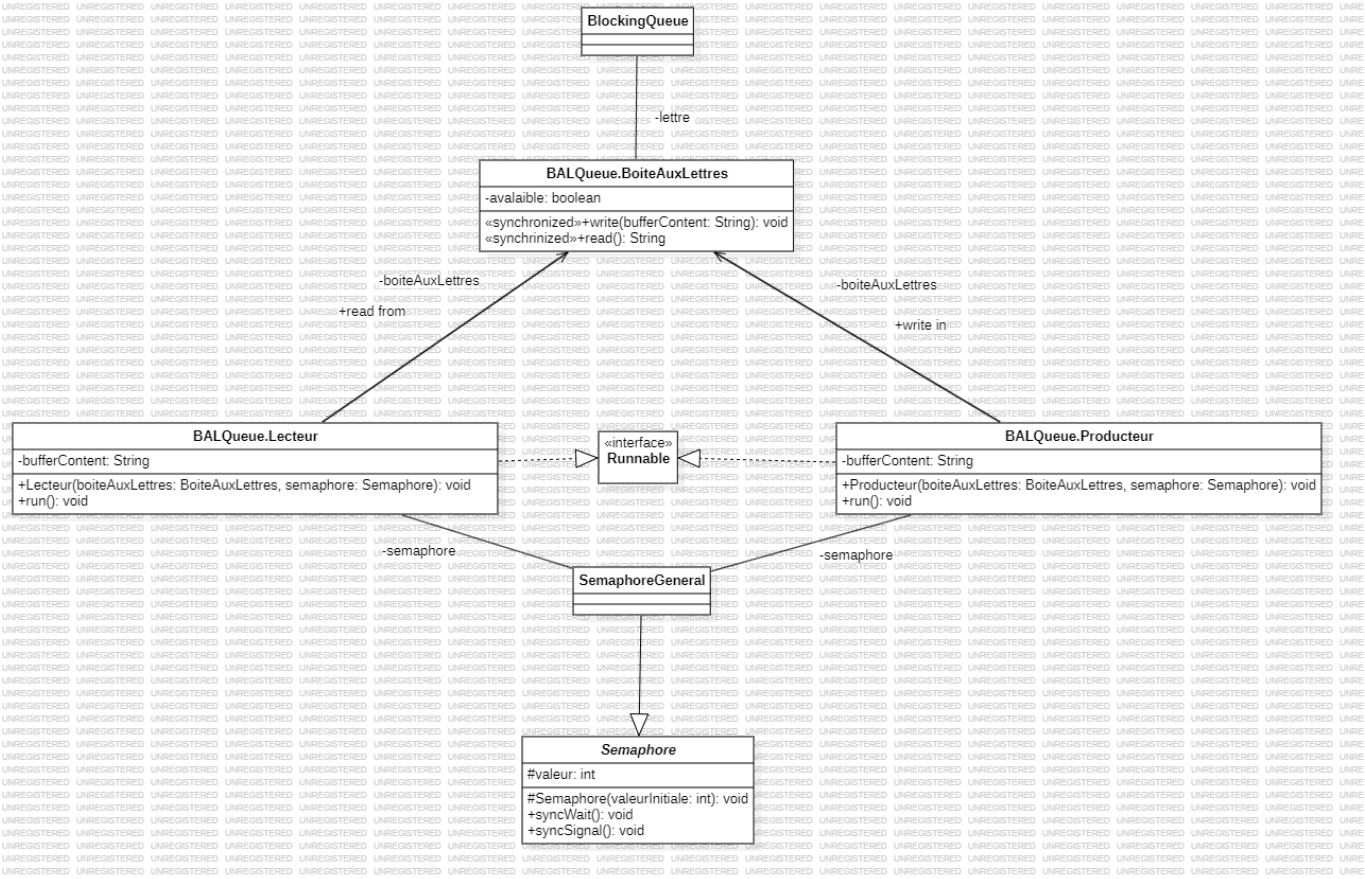
```
    @Override
    public void run() {

        while (true)
        {
            try
            {
                Thread.sleep(1000);
                semaphore.syncWait();
                Scanner scanner = new Scanner(System.in);

                System.out.print("Entrez une lettre : ");
                String bufferContent = scanner.next();
                boiteAuxLettres.write(bufferContent);
                if (!Objects.equals(boiteAuxLettres.read(), "")) {
                    System.out.println("Producteur: J'écris '" +
bufferContent + "' dans la boîte aux lettres");
                }
                else
                {
                    System.out.println("Producteur: La boîte aux lettres
est pleines");
                }
                semaphore.syncSignal();
            }
            catch (InterruptedException e)
            {
                throw new RuntimeException(e);
            }

        }
    }
```

```
    @Override
    public void run() {
        while (true) {
            try {
                Thread.sleep(1000);
                semaphore.syncWait();
                bufferContent = boiteAuxLettres.read();
                if (Objects.equals(bufferContent.toLowerCase(), "q"))
                {
                    System.out.println("Lecteur: Ok, je m'arrête");
                    exit(0);
                }
                else if (!bufferContent.equals(""))
                {
                    System.out.println("Lecteur: Je lis '" + bufferContent
+ "'");
                }
```

```
                else
                {
                    System.out.println("Lecteur: Il n'y a rien dans la
boîte aux lettres");
                }
                semaphore.syncSignal();
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
    }
```

BAL - Queue

C'est sensiblement la même chose à part que la BAL sauf qu'on utilise une `BlockingQueue` qui est elle même déjà l'implémentation d'un moniteur.



On a la methode `write()` qui fait appel à la methode `put()` de `BlockingQueue` qui est elle même une implémentation d'un write dans le moniteur `BlockingQueue` et c'est la même chose pour la methode `take()` qui est appelé par la methode `read()`.

```
public class BoiteAuxLettres {
    private BlockingQueue<String> lettre;
    private boolean avalaible;

    public BoiteAuxLettres() {
        lettre =  new ArrayBlockingQueue<>(1);
        this.avalaible = true;
```

```
    }

    synchronized public void write(String bufferContent) throws
InterruptedException {
        if (avalaible) {
            avalaible = false;
            lettre.put(bufferContent);
            avalaible = true;
        }
    }

    synchronized public String read() throws InterruptedException {
        if (lettre.size() == 0)
        {
            return "";
        }
        return lettre.take();
    }
}
```

Ici, à la place d'avoir un `String`, on a une `BlockingQueue`, ce qui est plus approprié au contexte.

Boulangerie

La boulangerie utilise le même design pattern que la BAL sauf que l'exemple est différent, dans cet exemple on a des `Client` (les consommateurs) et des `Boulanger` (les producteurs), les boulangers mettent à disposition du pain dans la `Boulangerie` dans cet exemple la boulangerie a pour champ une `ArrayBlockingQueue` d'une taille spécifié dans le constructeur (20 dans l'instanciation de la boulangerie dans le main). L'exemple de la boulangerie implémente le principe de pilule empoisonnée représenté par le champ final `PAIN_EMPOISONNE`, c'est un pain qui, si il est mangé par un thread, l'interompt. Une interruption est une indication à un thread qu'il devrait arrêter ce qu'il fait et faire autre chose à la place, sauf que dans notre cas il ne fait rien donc il meurt.