**UE Conception Orientée Objet**

# TD Simulation d'afficheurs lumineux

Le but de cet exercice est de simuler en Java les afficheurs lumineux qu'on voit un peu partout et qui font circuler un texte en boucle. On s'intéressera d'abord à une structure de données plus générale.

**Exercice 1 : File (FIFO).**

On modélise ici une structure pouvant stocker une suite d'exactement $L$ objets d'un type donné avec $L > 0$. $L$ est constant et est appelé *largeur* de la file. Initialement, tous les objets sont à `null`.

Cette suite se comporte comme une file FIFO (*first in, first out*) d'objets. Considérons que dans la file les objets sont rangés de la gauche (le premier) vers la droite (le dernier). Cela signifie alors qu'il est possible d'ajouter des objets à la file et que cet ajout se fait par la droite. L'objet ajouté se place en fin de la file à la position la plus à droite. Tous les éléments qui le précèdent sont alors décalés d'une position vers la gauche. L'élément qui se trouvait initialement en première position est alors sorti de la file[1].

Les quatre fonctionnalités de ces files sont :

- `getLargeur` renvoie la largeur de la file,
- `raz` force tous les éléments à la valeur `v` passée en paramètre,
- `ajoute` ajoute un élément, passé en paramètre, à la file. La méthode renvoie l'élément qui a été sorti de la file.
- `toString` renvoie sous forme de `String` une copie du contenu de la suite d'éléments de la file (on concatènera de gauche à droite les `toString` de chacun des éléments).

**Q 1 .** Définir le type `FileFIFO` : donnez un diagramme UML.

On définira un constructeur avec lequel on fixe la largeur de la file ainsi qu'une valeur par défaut pour tous ses éléments.

Donnez un code pour ce type. On peut imaginer deux implémentations de ce type, l'une gérant la file à l'aide d'une liste, la seconde à l'aide d'un tableau (dans les deux cas, on cherchera à mettre en place une structure "circulaire" afin d'éviter les manipulations de copie par décalage de valeurs).

**Exercice 2 : Les afficheurs lumineux**

On s'intéresse maintenant aux afficheurs lumineux.

Ces objets sont en charge de l'affichage d'un message d'une longueur $l$ non nulle. Ce message déroule en boucle sur un écran de largeur $L(\neq 0)$ qui peut être plus petit ou plus grand que $l$.

Un afficheur est donc initialement défini par la largeur $L$ de son écran.

Le message défile dans l'afficheur (en fait sur l'écran) en se décalant d'une position vers la gauche à chaque top d'une "horloge". Le message défile ainsi en boucle, quand le dernier caractère du message vient d'entrer dans la partie affichée, au prochain top horloge on reprend le message au début et c'est le premier caractère du message qui y entre à son tour, et ainsi de suite.

Le code de la classe `Afficheur` pourrait ressembler à :

```
public class Afficheur {
   ...
   // fixe un nouveau message a afficher
   public void setMessage(String message) {...}

   // appele a chaque top d'horloge, decale le message vers la gauche de l'ecran
   public void decale() {...}

   // renvoie ce qui apparait a l'ecran (ce qui est affiche)
   public String toString(){...}
}
```

[1] parmi tous les élements c'est lui qui était entré (*in*) le premier dans la file, il en sort (*out*) donc le premier.

Pour fixer les idées, soit la classe ci-dessous, l'invocation `new Horloge(new Afficheur(5)).tester(8) ;` produit alors la trace de droite.

```
public class Horloge {
   private Afficheur afficheur ;
   public Horloge(Afficheur a) { this.afficheur = a ; }
   public void tester(int nbTop) {
      String message = "abcd" ;
      this.afficheur.setMessage(message) ;
      for (int i=0 ; i<nbTop ; i++) {
         this.afficheur.decale() ;
         System.out.println("<<"+this.afficheur+">>") ;
      }
   }
}
```

```
<<    a>>
<<   ab>>
<<  abc>>
<< abcd>>
<<abcda>>
<<bcdab>>
<<cdabc>>
<<dabcd>>
```

**Q 1 .** Où est le lien avec l'exercice 1 ?

**Q 2 .** Complétez le code de la classe `Afficheur`.

# 1 Les afficheurs avec latence

On remarque que pour les afficheurs de l'exercice précédent il est difficile de voir où se termine le message. Pour éviter ce problème, on veut une nouvelle classe d'afficheurs pour lesquels on pourra spécifier lors de leur création un "*temps de latence*" entre l'entrée du dernier et celle du premier caractère. Ce temps de latence sera exprimé par un nombre positif ou nul d'espaces à insérer entre ces deux caractères. Appelons `Latence` cette nouvelle classe d'afficheurs.

Avec l'invocation `new Horloge(new Latence(5,3)).tester(8) ;` on crée un afficheur avec une latence de trois espaces et on le teste, ce qui donne :

```
<<    a>>
<<   ab>>
<<  abc>>
<< abcd>>
<<abcd >>
<<bcd  >>
<<cd   >>
<<d   a>>
```

**Q 3 .** Définissez la classe `Latence`.

# 2 Les afficheurs avec latence et vitesse paramétrable

A chaque top d'horloge, les afficheurs précédents font un seul décalage. On voudrait une nouvelle sorte d'afficheur dont on pourrait fixer le nombre de décalages effectués à chaque top. Ce nombre sera un entier positif ou nul. Appelons `Vitesse` cette nouvelle classe d'afficheurs.

**Q 4 .** Définissez la classe `Vitesse` et testez là.