Programmation et modélisation orientées objet
(USAL5A & USAL2H)

Protocole client-serveur

Dans ce TP, nous étudions un exemples d'application naturellement concurrente (de type client-serveur) et nous verrons les problèmes généralement rencontrés (accès concurrents aux ressources), leurs conséquences (non-déterminisme et incohérence) et une solution aux problèmes habituels (exclusion mutuelle et synchronisation) basée sur l'utilisation de canaux de communication.

On souhaite implanter un protocole qui permette à un « client » de consulter à distance les données sockées dans une liste sur un « serveur ». Dans un premier temps, le client et le serveur s'exécuteront dans des activités séparées (*Thread*) et la communication se fera grâce à des implantations de `BlockingQueue<E>` dans la JDK (dont nous utiliserons les méthodes *bloquantes*).

Le protocole de départ est le suivant :

- le client envoie un indice pour recevoir la donnée correspondante dans la liste
- le client envoie -2 pour recevoir la taille de la liste
- le client envoie -1 pour mettre fin à la communication

Questions

1. Implanter le protocole ci-dessus, partie serveur et partie client (on supposera d'abord pour simplifier que les données stockées sont de type `Integer`). Pour la partie client, un implantera une API avec deux méthodes `get` et `size` :

   ```
   public int size()
   public Integer get(int index)
   ```

   Peut-on facilement stocker des éléments de type `String` dans la liste au lieu du type `Integer` ?

2. Améliorer le protocole en définissant les types suivants :

   ```
   sealed interface Request {
       record GetSize() implements Request {}
       record GetValue(int index) implements Request {}
       record Stop() implements Request {}
   }

   sealed interface Answer {
       record Size(int size) implements Answer {}
       record Value(int value) implements Answer {}
   }
   ```

3. Modifier le protocole et l'implantation pour une liste qui stocke des `String`.

4. Généraliser le protocole et l'implantation du client et du serveur pour permettre une liste ayant un type d'élément générique.

5. Etendre le protocole pour rendre la liste mutable : on ajoutera pour cela les méthodes `set(int, E)`, `add(int, E)` et `remove(int)`.

6. Modifier la partie client pour implanter un « objet » liste (pas juste une API) qui représente « localement » chez le client la liste « distante » stockée sur le serveur. Ce « mandataire » (*proxy*) sera implanté sous la forme d'une classe `RemoteList`.

7. Compléter la classe `RemoteList` pour implanter complètement l'interface `List<E>` (on pourra pour cela étendre la classe `AbstractList<E>`).

A Instruction *switch*, filtrage et *design pattern* visiteur

A partir de Java 21, il est possible d'utiliser l'instruction *switch* sur des instances d'interfaces scéllées, comme `Request` et `Answer`. La syntaxe est la suivante :

```
switch (request) {
    case GetSize(): { /* case 1 */ }
    case GetValue(int index): { /* case 2 */ }
    case Stop(): { /* case 3 */ }
}
```

Cette instruction peut se traduire en utilisant un visiteur ainsi :

```
RequestVisitor<Answer> v = new RequestVisitor<Answer>() {
    public Answer visitGetSize() { /* case 1 */ }
    public Answer visitGetValue(int index) { /* case 2 */ }
    public Answer visitStop() { /* case 3 */ }
};
return request.accept(v);
```

Cas par défaut

L'instruction `switch` permet aussi de ne pas traiter tous les cas. Les cas restants sont alors traités de manière groupée en utilisant le mot-clé `default` :

```
switch (request) {
    case GetSize(): { /* case 1 */ }
    default: { /* case default */ }
}
```

Pour traduire cela, il faut utiliser la classe abstraite `DefaultRequestVisitor<A>` au lieu de l'interface `RequestVisitor<A>` :

```
RequestVisitor<Answer> v = new DefaultRequestVisitor<Answer>() {
    public Answer visitGetSize() { /* case 1 */ }
    public Answer visitDefault() { /* case default */ }
};
return request.accept(v);
```