TD4

Adaptateur

Pour pouvoir accéder au contenu d'une collection, Java fournit des itérateurs. Un itérateur est un objet implantant l'interface `java.util.Iterator` et permettant d'accéder de manière uniforme aux objets d'une collection, sans s'occuper de la façon dont ces objets sont stockés. L'interface `Iterator` possède trois méthodes :

- `hasNext()` renvoie `true` si la collection possède encore des éléments non parcourus,
- `next()` renvoie le prochain élément de la collection (sous la forme d'une instance d'`Object`) et avance d'un élément dans la collection,
- `remove()` supprime l'élément courant.

1. Écrivez une méthode statique qui prend un itérateur d'`Object` en paramètre et affiche toutes les chaînes de caractères se trouvant dans la collection correspondante.
2. Les anciennes classes représentant les collections en Java (*e.g.*, `Vector`) implémentent une méthode `elements()` qui renvoie un objet de type `java.util.Enumeration`. L'interface `Enumeration` permet également de parcourir ces collections, mais pas d'enlever un élément de la collection. Elle possède deux méthodes :
   - `hasMoreElements()` renvoie `true` s'il y a encore des éléments à parcourir dans la collection,
   - `nextElement()` renvoie le prochain élément (sous la forme d'une instance d'`Object`).

   Est-il possible d'utiliser la méthode statique écrite précédemment avec un objet de type `Enumeration` (sur une instance de `Vector` par exemple) sans modifier la méthode ni les interfaces `Enumeration` et `Iterator` ? Pourquoi ?
3. Proposez un diagramme de classes qui modélise et résout le problème en utilisant le pattern *Adaptateur*.
4. Écrivez une implémentation de l'adaptateur. Que faire pour la méthode `remove` de l'interface `Iterator` ?
5. Supplément :
   (a) Comment adapter une liste doublement chaînée en une pile ?

```
public interface Stack {
  void push(Object o);
  Object pop();
  Object top();
}
```

   (b) Comment adapter un ensemble d'entiers en une file de priorité ?

```
public interface PriorityQueue {
  void add(Object o);
  int size();
  Object removeSmallest();
}
```

Procuration

Soit une interface `Provider` fournissant quatre services :

```
public interface Provider {
  void service1();
  void service2();
  void service3();
  void service4();
}
```

Nous souhaitons facturer l'utilisateur en fonction des services qu'il utilise. Le coût d'un service est le produit de son coût unitaire par le nombre de fois où le service a été utilisé. Le coût total est la somme des coûts de chaque service. Cette gestion des coûts doit être totalement transparente à la classe qui rend des services.

1. Proposez un diagramme de classes qui modélise et résout le problème en utilisant le pattern *Procuration*.
2. Écrivez une implémentation des classes du diagramme, ainsi qu'un programme de test affichant le coût total de l'utilisation des services.
3. Nous proposons d'ajouter une nouvelle règle de gestion ne permettant l'accès aux services 3 et 4 qu'aux utilisateurs authentifiés par mot de passe. Cette règle d'accès doit également être gérée sans modifier la classe qui rend des services. Implémentez un *proxy* qui demande à l'utilisateur de s'authentifier chaque fois qu'il souhaite utiliser les services 3 et 4. Si l'utilisateur entre le bon mot de passe, le *proxy* exécute le service demandé.
4. De quel type de procuration s'agit-il ?