TD5 : Proxy, Flyweight

Exercice 1 - Proxy

Voici quelques classes permettant de gérer un album de photo : photo0.zip (http://igm.univ-mlv.fr/ens/IR/IR2/2012-2013/ProgOO/photo0.zip). On cherche pour déboguer à afficher un message lorsque l'on entre ou sort d'une méthode particulière d'un objet.

1. Créez un objet `java.util.logging.Logger` et affichez un message avec le niveau LOG sur la sortie standard (un peu de javadoc à lire !).
   - afficher un message de niveau FINEST. Que faut-il faire pour le voir ?
   - ajouter un FileHandler pour que les messages de log soient aussi sauvés dans un fichier "proxy.log"

   - conseil : lisez la javadoc et faites un schéma pour résumer le fonctionnerment des loggers et handlers ; et résumer le rôle des Level pour le filtrage des messages au niveau Handler et au niveau Logger. Expliquez ce qu'est un LogRecord.
   - conseil : appeler setUseParentHandlers(false) sur votre logger

2. Rappelez le principe du design pattern `Proxy`.
   Écrivez la méthode `createPhotoProxy(Photo photo,Logger logger)` dans la classe `PhotoFactory` qui prend un objet de type `Photo` et un logger et qui renvoie un objet de type `Photo`.
   Lorsque des méthodes seront appelées sur l'objet Photo renvoyé, elles devront afficher sur le logger :
   – lors de l'entrée dans une méthode le message `enter method` suivi du nom de la méthode,
   – lors de la sortie de la méthode le message `exit method` suivi du nom de la méthode.

   Pour tester le proxy, changez le code de la méthode `createPhoto(File file)` de sorte que, si l'on crée la factory avec un logger, on utilise le proxy.

   Pour les pro ! Ou chez vous :
3. On souhaite écrire un proxy générique qui affiche les messages d'entrée et de sortie quel que soit l'objet.
   Pour cela nous allons utiliser la classe `java.lang.reflect.Proxy`.
   Écrivez une méthode `createProxy()` générique et correctement typée créant un proxy générique.
4. Modifiez votre code pour que ne soient affichées que les méthodes ayant l'annotation `Log`.

Exercice 2 - Flyweight

Le centre ville va être refait. Le logiciel de conception doit pouvoir représenter les rues et les différents pavés qui vont être posés. 3 types de pavés vont être posés :

- le standard, décrit par ses dimensions, sa couleur, l'endroit où il va être posé (supposons une position terrestre donné par une paire de Double), son poids, etc.
- les pavés de bordures, un peu plus jolis, qui seront mis en bordure des routes. Leur description est quasi identique à celle des pavés standards.
- les pavés fantaisies, avec des initiales ou des dessins et qui seront disséminés dans la ville. Les pavés fantaisie sont décrits comme les pavés standards avec, en plus, une indication

d'initiale ou un numéro de dessin.

Question : rappeler le principe du design pattern Flyweight

Implémentation : on souhaite implémenter, en utilisant le pattern Flyweight, le cœur de ce logiciel de conception.
Dans notre conception, un Pave a ses différentes caractéristiques (tel que décrit pour les 3 types de pavés) et a aussi une position terrestre (l'endroit où il sera placé).
Avant d'implémenter le DP Flyweight, l'interface de base était :

```
interface Pavement {
      // inclut les dimensions, la position
      //et les éventuelles caractéristiques particulière (dessin, ...)
      String getDescription() ;
}
```

- La position des pavés sera traitée comme la *donnée extrinsèque* de nos objets Pave
  - modifiez l'interface en conséquence
- Implémentez les 3 classes de Pavés
- Implémentez la Factory qui gérera les Pavés standards comme des Flyweights (partagés)
- Implémentez la classe qui gérera les données extrinsèques
- Codez votre solution et un petit programme de tests.