Design Patterns - TP1

Partie 1: Les Canards

Nous allons construire un nouveau simulateur de canards en repartant à zéro. Cette fois les canards vont implémenter l'interface Cancaneur :

```
public interface Cancaneur{
      public void cancaner() ;
}
```

Il nous faut tout d'abord de vrais canards : les Colvert et les Mandarin qui font « coincoin ».
Nous allons également créer des CanardEnPlastique qui font « Couic » et des Appeaux qui font « Couincouin ».
On **NE** créera **PAS** de classe Canard super-classe de tous les Canard. Nous nous intéresserons au type Cancaneur.

Question 0 :
Écrire l'interface Cancaneur, les quatre classes de canards (Colvert, Mandarin, CanardEnPlastique et Appeau) et une classe Simulateur qui contient les méthodes suivantes :

```
public void simuler(){
      Cancaneur colvert = new Colvert();
      Cancaneur mandarin = new Mandarin();
      Cancaneur canardEnPlastique = new CanardEnPlastique();
      Cancaneur appeau = new Appeau();

      System.out.println(« Simulateur de Canards »);
      simuler(colvert);
      simuler(mandarin);
      simuler(canardEnPlastique);
      simuler(appeau);
}

public void simuler(Cancaneur c){
      c.cancaner() ;
}
```

Testez !!!!

Question 1 :
Comme tout le monde le sait, un palmipède peut en cacher un autre. Voilà donc une oie qui vient trainer du côté de notre simulateur :

```
public class Oie{
      public void cacarder(){
            System.out.println(« Ouinc ») ;
      }
}
```

Les oies ne cancanent pas, elles cacardent. Pourtant nous souhaitons ajouter cette oie à notre simulateur et pouvoir la simuler sans changer le fonctionnement de notre simulateur (en particulier la méthode simuler).
Utilisez le design pattern **Adaptateur** pour que cela soit possible.
Ensuite, ajoutez une oie au simulateur et testez.

Question 2 :
Les cancanologues étudient les cancans et voudraient pouvoir compter le nombre de cancans produit dans notre simulateur. Nous ne voulons pas modifier nos canards pour autant. Nous allons donc ajouter un nouveau comportement à nos canards en les enveloppant dans un objet **Décorateur** que nous appellerons CompteurDeCancans.

Voici la nouvelle méthode simuler() :

```
public void simuler(){
      Cancaneur colvert = new CompteurDeCancans(new Colvert());
      Cancaneur mandarin = new CompteurDeCancans(new Mandarin());
      Cancaneur canardEnPlastique = new CompteurDeCancans(
                                                new CanardEnPlastique());
      Cancaneur appeau = new CompteurDeCancans(new Appeau());
      Cancaneur oie = new AdapteurDOie(new Oie()) ;

      System.out.println(« Simulateur de Canards »);
      simuler(colvert);
      simuler(mandarin);
      simuler(canardEnPlastique);
      simuler(appeau);
      simuler(oie) ;

      System.out.println(« Nous avons compté » +
                        CompteurDeCancans.getNbCancans() + « cancans ») ;
}
```

Indication : utilisez une variable de classe nbCancans dans la classe CompteurDeCancans afin de compter tous les cancans.
Ajouter un nouveau **Décorateur** Begue, qui fait bégayer les cancaneurs.

Question 3 :

L'inconvénient de ce que nous avons fait dans la question précédente c'est qu'il faut penser à décorer tous les canards. Si nous en oublions un, nous ne compterons pas tous les cancans. La solution va consister à encapsuler la création des canards en utilisant le design pattern **Fabrique Abstraite**.

Vous prévoirez deux fabriques concrètes :

- FabriqueDeCanards qui fournira des instances de canards « classiques »
- FabriqueDeComptage qui fournira des instances de canards enveloppés dans des CompteurDeCancans.

Faites le nécessaire dans le simulateur pour pouvoir tester. Essayez d'interchanger les fabriques concrètes.

Question 4 :

Dans notre simulateur, nous traitons nos cancaneurs (canards et oies) individuellement en appelant la méthode simuler pour chacun d'eux. Nous souhaitons maintenant pouvoir gérer des troupes de cancaneurs et pouvoir demander à toute une troupe de simuler.

Pour cela, utilisez le design pattern **Composite**.

Testez.

Partie 2: Les Tris

On fournit l'interface StrategieTri.

```
import java.util.List;

public interface StrategieTri {
    <T extends Comparable<T>> List<T> trie(List<T> donnees);
}
```

Implémentez trois stratégies de tris : insertion, bulle et java (Collections.sort)
Ecrire la classe BaseDonnees qui contient une liste de chaînes de caractères et fonctionne ainsi :

```
public class Test {

    public static void main(String[] args) {
        BaseDonnees maBase = new BaseDonnees(new TriJava());
        maBase.afficheDonneesTriees();

        BaseDonnees maBase2 = new BaseDonnees(new TriBulles());
        maBase2.afficheDonneesTriees();

        BaseDonnees maBase3 = new BaseDonnees(new TriSelection());
        maBase3.afficheDonneesTriees();
    }
}
```