TP N°3
UML et design patterns

Exercice 1 :

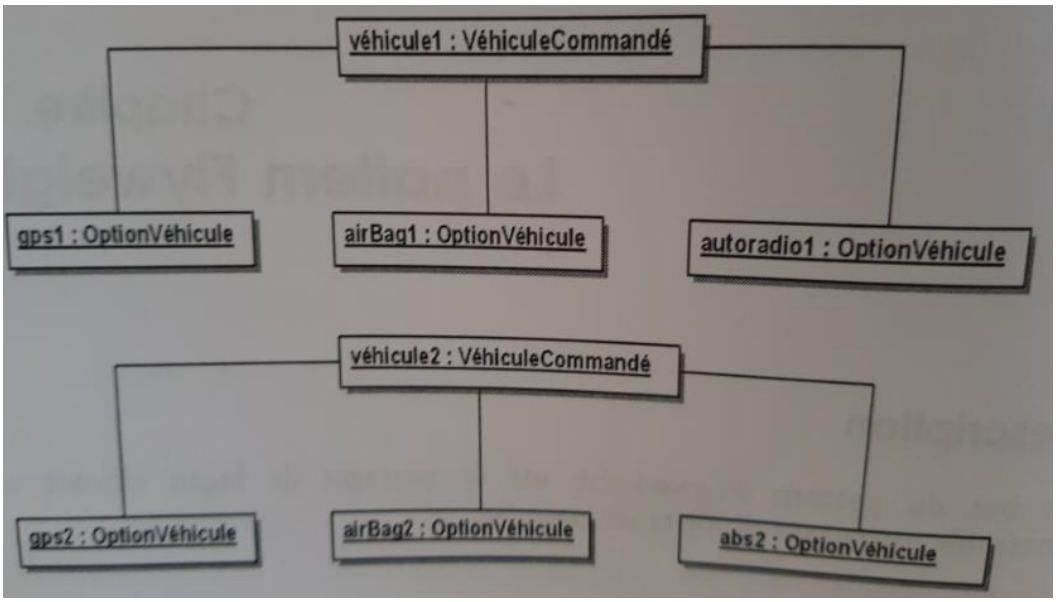
Au sein d'un système de vente de véhicules, nous souhaitons représenter les sociétés clientes, notamment pour connaître le nombre de véhicules dont elles disposent et leur proposer des offres de maintenances de leur parc.
Les sociétés qui possèdent des filiales demandent des offres de maintenance qui prennent en compte le parc de véhicules de leurs filiales.
Une solution immédiate consiste à traiter différemment les sociétés sans filiale et celle possédant des filiales. Cependant, cette différence de traitement entre les deux types de sociétés rend l'application plus complexe et dépendante de la **composition** interne des sociétés clientes.
La classe abstraite société est définie comme suit :

```
Public abstract class Société{
 protected static double coûtUnitVéhicule= 5.0;
 protected int nbrVéhicules;

 public void ajouteVéhicule(){
  nbrVéhicules=nbrVéhicules + 1 ;
 }

 public abstract double calculeCoûtEntretien() ;

 public abstract boolean ajouteFiliale(Société filiale) ;
}
```

1) Quel pattern permet-il de modéliser une solution à cette problématique.
2) Modélisez son utilisation par un diagramme de classe.
3) Programmez ce diagramme de classe en Java. Le programme principal doit pouvoir créer des sociétés et leur différentes filiales et de calculer et d'afficher le coût total d'entretien pour un groupe de sociétés.

Exercice 2 :

Dans un système de vente de véhicules, il est nécessaire de gérer les options que l'acheteur peut choisir lorsqu'il commande un nouveau véhicule.
Ces options sont décrites par la classe *OptionVéhicule* qui contient plusieurs attributs comme le nom, la description, etc.
Pour chaque véhicule commandé, il est possible d'associer une nouvelle instance de cette classe. Cependant, un grand nombre d'options est souvent présent pour chaque véhicule commandé, ce qui oblige le système à gérer **un grand ensemble d'objets de petite taille**. Cette approche présente toutefois l'avantage de **pouvoir stocker au niveau de l'option des informations spécifiques à celle-ci et au véhicule** comme le prix de vente de l'option qui peut différer d'un véhicule commandé à un autre.



1) Quel pattern est le mieux adapté pour concevoir les options d'un véhicule.
2) Modélisez son utilisation par un diagramme de classe.
3) Programmez ce diagramme de classe en Java. La classe *VéhiculeCommandé* gère la liste des options ainsi que la liste des prix de vente. Ces deux listes sont générées en parallèle. Le prix de vente d'une option se trouve au même indice dans la liste *prixDeVenteOptions* que l'option dans la liste *options*. Cette classe doit contenir aussi une méthode *ajouteOptions* et une méthode afficheOptions.
Le programme principal doit pouvoir créer deux véhicules commandés en leur affectant des options communes et d'autres non communes et d'afficher pour chaque véhicule commandé la liste des options choisies.

Exercice 3 :

Dans un système de vente en ligne de véhicules, la classe *VueCatalogue* dessine la liste des véhicules destinés à la vente. Un algorithme de dessein est utilisé pour calculer la mise en page en fonction du navigateur. Il existe **deux versions de cet algorithme** :

- Une première version qui n'affiche qu'un seul véhicule par ligne (un véhicule prend toute la largeur disponible) et qui affiche le maximum d'informations ainsi que quatre photos ;
- Une seconde version qui affiche trois véhicules par ligne mais qui affiche moins d'informations et une seule photo.

L'interface de la classe *VueCatalogue* **ne dépend pas du choix** de l'algorithme de mise en page. Ce choix n'a pas d'impact sur la relation d'une vue d'un catalogue avec ses clients. Il n'y a que la présentation qui est modifiée.
Une première solution consiste à transformer la classe *VueCatalogue* en une interface ou en une classe abstraite et à introduire deux sous-classes d'implantation différant par le choix de l'algorithme. Ceci présente l'inconvénient de complexifier inutilement la hiérarchie des vues de catalogue.
Une autre possibilité est d'implanter les deux algorithmes dans la classe *VueCatalogue* et à l'aide d'instructions conditionnelles d'effectuer les choix. Mais cela consiste à développer une classe relativement lourde et dont le code des méthodes est difficile à appréhender.

1) Quel pattern faut-il mettre en œuvre pour adapter l'interface de la classe *VueCatalogue* aux algorithmes de dessein.
2) Modélisez son utilisation par un diagramme de classe. La classe *VueCatalogue* comporte une méthode *dessine()*.
3) Programmez ce diagramme de classe en Java. Le programme principal consiste à créer deux instances de *VueCatalogue*, la première est paramétrée par le dessein sur trois lignes. La seconde

,instance est paramétrée par le dessein sur une ligne. Après les avoir créés, le programme principal invoque la méthode *dessine* de ses instances. La classe *VueVéhicule* est donnée en annexe.

Annexe :

```
public class VueVéhicule{
protected String description ;

public VueVéhicule (String description)
{
   this.description=description;
}

public void dessine()
{
  System.out.print(description)
}
}
```