TD 4 : Interfaces et abstractions

Objectifs pédagogiques : interfaces et abstraction.

4.1 Capacités Communes

Un logiciel de géométrie propose des classes de représentation des points avec des coordonnées euclidiennes (x,y) ou des coordonnées polaires (rayon, angle).

Bien que différents, on aimerait expliquer que les objets de ces classes partagent la propriété d'être « mesurables ». La propriété commune concerne le calcul de la distance à l'origine (`getDistance()`).

Quelle solution proposez-vous ?

⇒ **Donnez un diagramme UML** du point de vue du client.

⇒ **Ecrivez le code d'une classe `Polygone`** qui dispose d'une méthode de calcul de distance minimale à l'origine (`double getDistanceMin()`) à partir d'une liste d'objets interne pouvant être soit des points en coordonnées polaires, soit des points en coordonnées euclidiennes ou tout autre implémentation « mesurable ».

⇒ **Ecrivez** dans la classe `Polygone` **une méthode principale `main(...)`** qui crée des objets mesurables, les insère dans un polygone et calcule la distance minimale. Peut-on mélanger les deux types de points au sein d'un polygone ?

La bibliothèque standard Java propose une interface `Comparable<T>` pour le support standard des relations d'ordre entre objets. La définition de cette interface est la suivante :

```
package java.util;
public interface Comparable<T> {
  public int compareTo(T o);
}
```

Une classe C qui implémente l'interface `Comparable<C>` introduit une relation d'ordre totale interne aux objets de la classe C (et sous-classes). La méthode `compareTo(C o)` retourne un entier négatif (-1 par convention) si le receveur (`this`) est plus petit que l'objet o passé en argument. Elle retourne 0 si les deux objets sont égaux et sinon un entier positif (1 par convention).

⇒ **Faites en sorte que les deux types de points deviennent aussi comparables** (entre objets de même type). **Mettez à jour votre diagramme UML.**

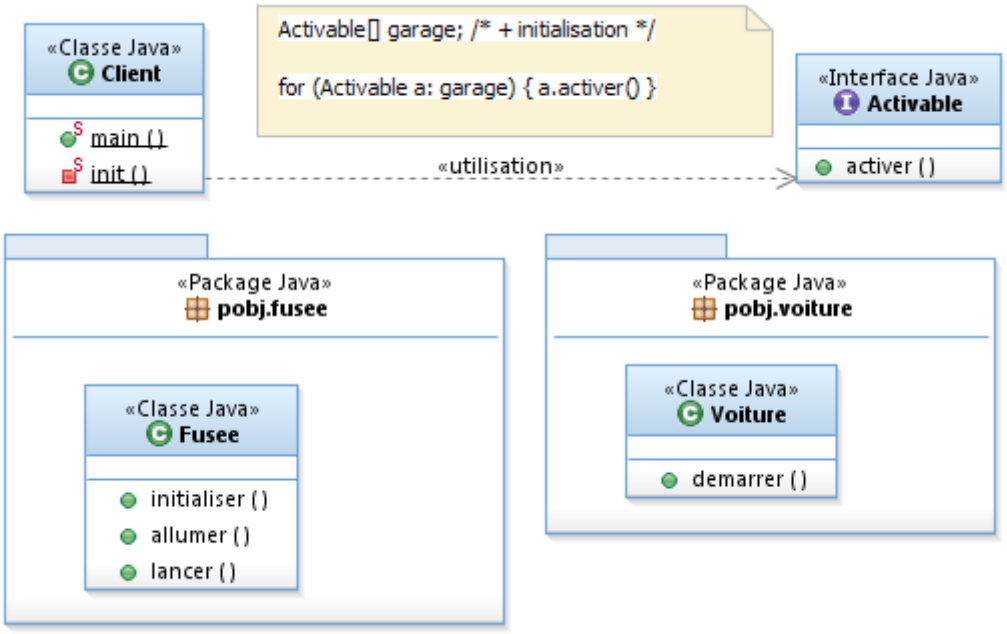
⇒ **Ajoutez** dans votre classe `Polygone` **une méthode qui trie les objets** mesurables par ordre croissant de distance à l'origine. Vous ferez appel à la méthode `sort()` statique fournie par le type `Collections`.

4.2 Hiérarchies séparées

Un rôle important des interfaces est de permettre la liaison entre des hiérarchies séparées de classes, développées par des équipes indépendantes, de façon peu intrusive.

Dans un paquetage `pobj.voiture` une classe `Voiture` est proposée, contenant au moins une méthode `demarrer()` permettant de démarrer la voiture. Dans un paquetage séparé `pobj.fusee` une classe `Fusee` est définie, elle contient des méthodes `initialiser()`, `allumer()`, `lancer()`.

Un logiciel client souhaite pouvoir manipuler de façon interchangeable des voitures ou des fusées, on se concentrera sur la phase de démarrage et la voiture et la séquence de lancement de la fusée. La figure suivante résume la situation.



On peut réunir ces deux notions sous la bannière d'une notion d'objet `Activable`.

⇒ **Proposez un diagramme UML** (client) **et une implémentation minimale de cet exemple** : classes de voiture et de fusée (on utilisera de simples affichages) et code client pouvant activer au choix une voiture, une fusée, etc.

La solution la plus simple consiste à toucher aux classes `Voiture` et `Fusee`. Si vous avez eu à toucher à ce code, imaginez à présent que vous n'avez pas accès à ces classes et proposez une nouvelle solution qui laisse les classes initiales Voiture et Fusee intactes.

⇒ **Vous proposerez un nouveau diagramme UML et le code** qui change par rapport à votre solution précédente.

⇒ Proposez enfin une solution dans le cas où les classes `Voiture` et `Fusee` sont déclarées `final`.