Classes internes et patron observateur

Objectifs

— Mettre en pratique le patron de conception « observateur »
— Comprendre les classes internes (d'instance et de classe)
— Comprendre l'implantation du pattern tel que défini dans les API Java

Exercice 1 : Point observable
Le patron de conception « observateur » a été utilisé pour résoudre le problème de dépendance entre les segments et les points. Un objet point gère une liste d'observateurs qu'il avertit (en appelant l'opération maj, mettre à jour) lorsqu'il est translaté.

Un segment dont la longueur est stockée peut alors s'inscrire auprès de ses points extrémités pour que sa longueur soit recalculée chaque fois que l'un de ses points extrémités est translaté.

La figure 1 contient le diagramme de classe correspondant à une solution possible. L'attribut `observateurs` de la classe GroupeObservateurs sera défini du type `Collection`. C'est la manière choisie pour représenter la cardinalité $*$ de GroupeObservateurs avec l'interface Observateur. On utilisera la classe `GroupeObservateurs` pour réaliser la relation entre `Point` et `Observateur`. La classe `Point` aura donc un attribut `observateurs` de type `GroupeObservateurs`.

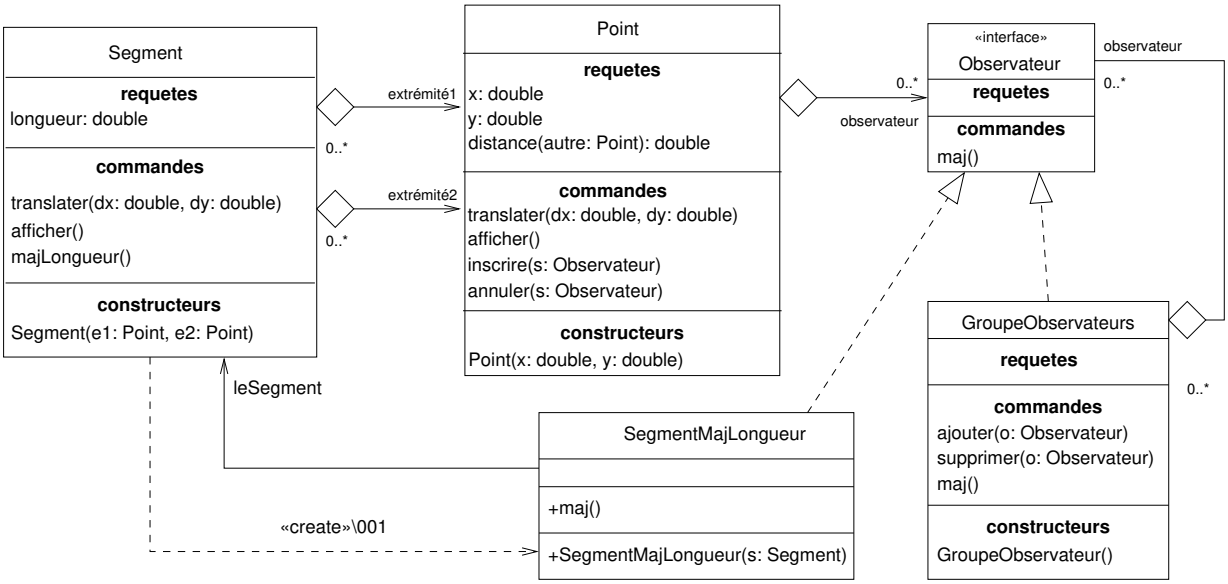

FIGURE 1 – Diagramme de classes des points observables.

1.1 *Compléter le programme.* Les classes de l'application ont été programmées à l'exception de 1) la classe GroupeObservateurs qui gère tous les observateurs d'un point et 2) la classe

SegmentMajLongueur.

Programmer ces deux classes en respectant le diagramme de classe.

1.2 *Observateurs et droits d'accès.* La solution des observateurs permet de rendre la classe Point indépendante des classes qui l'observent (Segment, Polygone, Cercle...). Cependant, cette solution a un défaut puisque la classe Segment est obligée de rendre publique une méthode `majLongueur` qui met à jour sa longueur.

Ce problème peut être résolu en Java en utilisant les classes internes. Les classes internes, comme les attributs ou les méthodes, peuvent être d'instance ou de classe. Nous allons envisager les deux solutions.

1.2.1 Déclarer la méthode `majLongueur` de Segment en privé et constater que l'application ne peut plus être compilée.

1.2.2 Définir la classe SegmentMajLongueur dans la classe Segment (à la fin). On pourra alors la renommer MajLongueur. Il s'agit alors d'une classe interne. On la définira **`static`** ce qui signifie qu'elle est indépendante d'un segment particulier.

Compiler et exécuter l'application.

1.2.3 On peut également définir la classe interne MajLongueur sans le **`static`**. C'est alors une classe interne d'instance qui a accès aux caractéristiques de l'objet à partir duquel elle a été créée. Ainsi, il n'est plus nécessaire de lui donner en paramètre le segment à mettre à jour. Faire les modifications correspondantes.

Exercice 2 : Cercle et Point

Considérons maintenant un Cercle défini par son centre et un point de sa circonférence. On considère que le rayon du cercle est une information stockée et qu'il y a une relation d'agrégation entre le cercle et ses deux points. La translation de l'un de ces deux points a donc un impact sur le cercle :

- une translation du centre du cercle provoque une translation du point de la circonférence ;
- une translation du point de la circonférence change la taille du cercle (le centre reste inchangé) et nécessite donc de mettre à jour le rayon du cercle.

Adapter les classes pour prendre en compte les cercles.

Exercice 3 : Le patron Observateur dans les API Java

Les API Java fournissent déjà une implantation du patron de conception Observateur au travers de :

- l'interface `java.util.Observer`, l'équivalent de notre interface Observateur,
- la classe `java.util.Observable`, classe qui factorise le code qui doit être placé dans les classes observables, Point dans notre cas.

3.1 Lire et comprendre leur documentation.

3.2 Reprendre l'exemple de Point, Segment et Cercle en utilisant l'observateur des API Java à la place des classes et interfaces développées spécifiquement.