Héritage comme généralisation
Corrigé

Exercice 1 : Formaliser le schéma

Dans la question 3.2 du TP 5, nous avons défini le schéma comme plusieurs objets (points, points nommés et segments) qui sont référencés par des variables différentes. Il serait plus *logique* et *pratique* d'avoir une seule variable qui représente le schéma (on l'appellera naturellement `schema`). Comme un schéma est constitué d'un nombre variable d'éléments, on peut le représenter par un tableau. Si nous appelons X le type des éléments de ce tableau, nous pouvons alors écrire le code du listing 1

1.1. Indiquer à quelles conditions sur X les lignes suivantes compilent.

```
schema[nb++] = s12;
schema[nb++] = barycentre;
schema[i].afficher();
schema[i].translater(4, -3);
```

Solution : La première instruction nous indique que le type de s12 (Segment) doit être un sous-type du type de `schema[nb++]` (X). Donc Segment est un sous-type de X.

Le même raison sur l'instruction suivante nous indique que Point est un sous-type de X.

Les deux dernières instructions nécessitent que le type X spécifie les méthodes afficher() et translater, cette dernière avec deux paramètres qui acceptent des entiers.

1.2. Quel code sera exécuté pour `x.afficher()` et `x.translater(4, -3)` ?

Solution : Ce sera le code de la méthode afficher (ou translater) de la classe de l'objet attaché à x. C'est la **liaison dynamique**.

1.3. Indiquer les autres éléments à définir sur X ? Justifier la réponse.

Solution :

Principe : Comme X généralise Point et Segment, on peut y mettre tout ce qui est commun à Point et Segment.

Éléments à définir : Nous avons déjà identifié les deux méthodes afficher() et translater(double, double) mais il y a aussi `dessiner(Afficheur)` et tout ce qui concerne la gestion de la couleur (l'attribut `couleur`, l'accesseur et le modifieur).

1.4. Donner un nom plus significatif à X.

Solution : Plusieurs noms sont possibles. Nous utiliserons `ObjetGeometrique` dans la suite.

Ne surtout pas prendre Dessin ou Schéma. Ce serait un contre-sens. X représente un objet qui peut être un point, un segment, etc. Pas plusieurs objets !

Exercice 2 : Écrire la classe X et adapter l'application

2.1. Est-ce que l'on sait écrire le code des méthodes `afficher` ou `translater` de X ?

Solution : Non. On ne sait pas comment faire. Éventuellement, afficher pourrait afficher la couleur mais ce ne serait pas l'objectif réel.

Ces méthodes pourront (devront !) être déclarées abstraites.

Listing 1 – La classe `ExempleSchemaTab` (extraits)

```
1
2  public class ExempleSchemaTab {
3
4     public static void main(String[] args)
5     {
6        // Créer les trois segments
7        Point p1 = new PointNomme(3, 2, "A");
8        Point p2 = new PointNomme(6, 9, "S");
9        Point p3 = new Point(11, 4);
10       Segment s12 = new Segment(p1, p2);
11       Segment s23 = new Segment(p2, p3);
12       Segment s31 = new Segment(p3, p1);
13
14       // Créer le barycentre
15       double sx = p1.getX() + p2.getX() + p3.getX();
16       double sy = p1.getY() + p2.getY() + p3.getY();
17       Point barycentre = new PointNomme(sx / 3, sy / 3, "C");
18
19       // Définir le schéma (vide)
20       X schema[] = new X[10]; // le schéma
21          // 10 : capacité suffisante ici, non contrôlée dans la suite.
22       int nb = 0;  // Le nombre d'éléments dans le schéma
23
24       // Peupler le schéma
25       schema[nb++] = s12;
26       schema[nb++] = s23;
27       schema[nb++] = s31;
28       schema[nb++] = barycentre;
29
30       // Afficher le schéma
31       System.out.println("Le␣schéma␣est␣composé␣de␣:␣");
32       for (int i = 0; i < nb; i++) {
33          schema[i].afficher();
34          System.out.println();
35       }
36
37       // Translater le schéma
38       System.out.println("Translater␣le␣schéma␣de␣(4,␣-3)␣:␣");
39       for (int i = 0; i < nb; i++) {
40          schema[i].translater(4, -3);
41       }
42
43       // Afficher le schéma
44       System.out.println("Le␣schéma␣est␣composé␣de␣:␣");
45       for (int i = 0; i < nb; i++) {
46          schema[i].afficher();
47          System.out.println();
48       }
49    }
50
51 }
```

Ceci montre bien qu'ObjetGeometrique est une notion abstraite.

Remarque : On aurait pu écrire translater, si par exemple un ObjetGéométrique possédait un point d'ancrage. Il suffirait alors de translater ce point d'ancrage. Les méthodes afficher et dessiner resteraient abstraites.

2.2. Peut-on créer des instances de X ?

Solution : Non, car c'est une notion abstraite. D'autant plus ici, car elle contient des méthodes abstraites.

2.3. Quels constructeurs définir sur X ?

Solution : Un constructeur qui prend en paramètre la couleur.

Ceci peut paraître étrange puisqu'on vient de dire qu'on ne peut pas créer un objet à partir de la classe ObjetGéométrique !

2.4. Quand ces constructeurs seront-ils appelés ?

Solution : Quand on créera une sous-classe : le constructeur d'une sous-classe doit nécessairement commencer par appeler un constructeur de sa super-classe.

Ceci garantira que pour tout objet géométrique aa couleur sera bien initialisée.

2.5. Écrire le code de la classe X.

Solution :

```
1  import java.awt.Color;
2
3  /** La classe ObjetGeometrique factorise les caractéristiques communes aux
4    * différents éléments qui composent un schéma mathématique.
5    *
6    * <strong>Remarque :</strong> Nous définissons une classe et non une interface
7    * car dans les caractéristiques communes il y a la couleur qui et un attribut
8    * (et les deux méthodes associées).  Ce ne peut donc pas être une interface.
9    *
10   * @author  Xavier Crégut
11   * @version 1.4
12   */
13 public abstract class ObjetGeometrique {
14
15     private Color couleur;    // couleur de l'objet géométrique
16
17     /** Constructeur de ObjetGeometrique à partir de sa couleur.
18       * @param saCouleur la couleur de l'objet
19       */
20     public ObjetGeometrique(Color saCouleur) {
21         this.setCouleur(saCouleur);
22     }
23
24     /** Obtenir la couleur de l'objet.
25      * @param la couleur de l'objet
26      */
27     public Color getCouleur() {
28         return this.couleur;
29     }
30
31     /** Changer la couleur de l'objet.
32       * @param nouvelleCouleur nouvelle couleur
```

```
33          */
34        public void setCouleur(Color nouvelleCouleur) {
35            this.couleur = nouvelleCouleur;
36        }
37
38        /**  Afficher sur le terminal les caractéristiques de l'objet.  */
39        public abstract void afficher();
40
41        /** Translater l'objet géométrique.
42         *  @param dx déplacement suivant l'axe des X
43         *  @param dy déplacement suivant l'axe des Y
44         */
45        public abstract void translater(double dx, double dy);
46
47       /** Faire pivoter le point.
48        * @param pivot point servant de pivot
49        * @param angle angle de la rotation (en radian)
50        */
51        abstract public void pivoter(Point pivot, double angle);
52
53        /** Dessiner l'objet géométrique sur l'afficheur.
54          * @param afficheur l'afficheur à utiliser
55          */
56        public abstract void dessiner(afficheur.Afficheur afficheur);
57
58    }
```

2.6. Lister et effectuer les modifications à apporter aux autres classes de l'application.
Solution : Il suffit :

- d'ajouter une clause `extends ObjetGéométrique` au niveau des classes Point et Segment. Il n'est pas nécessaire de modifier la classe PointNommé puisqu'elle hérite de Point et donc, transitivement, de ObjetGéométrique.
- de supprimer la gestion de la couleur dans les classes Point, Segment, Cercle, etc. Elle n'est pas présente dans PointNommé.

  Question : Que se passe-t-il si ce n'est pas fait ? On se retrouve avec deux attributs couleur, l'un au niveau de ObjetGéométrique, l'autre au niveau de Point (et autres objets géométrique).
- Dans les constructeurs, remplacer « this.couleur = Color.GREEN » par « super(Color.GREEN) » qui doit être la première ligne du constructeur.

Remarque : Revenons sur le problème de la couleur. Toutes les classes dérivées considèrent la couleur comme étant vert par défaut. Il suffirait donc de définir, en plus du constructeur explicite pour la couleur, un constructeur sans paramètre qui l'initialise à vert. Il suffit alors de supprimer l'affectation de l'attribut couleur dans les classes dérivées.

L'inconvénient est que, dans un autre contexte, on peut oublier d'initialiser la couleur de l'objet géométrique avec la bonne valeur !

Exercice 3 : Construire le schéma en utilisant les listes
Au lieu d'utiliser un tableau comme dans l'exercice 2, on veut utiliser l'interface `List` et sa

réalisation `ArrayList` du paquetage `java.util` (en particulier la méthode `add` et la structure de contrôle `foreach`).

3.1. Indiquer les avantages et inconvénients des listes par rapport aux tableaux.

Solution : Les avantages sont :

— essentiellement, le fait que le ArrayList est redimmensionnable. On s'affranchit donc des problèmes de capacité, redimensionnement, etc.

— la classe `ArrayList` contient des opérations de plus haut niveau qu'un tableau (appartenance d'un élément, insertion...).

Depuis Java 1.5 et la généricité, il n'y a pas réellement d'inconvénient à utiliser ArrayList !

Conclusion : Il est préférable d'utiliser ArrayList et plus généralement l'API des collections Java plutôt que les tableaux de base.

3.2. Construire le schéma en utilisant une liste.

Exercice 4 : Définir un groupe

Dans un éditeur de schémas mathématiques, il serait pratique de pouvoir grouper plusieurs X pour les manipuler comme un seul et leur appliquer à tous, en une seule fois, la même opération (translater, afficher, etc.).

4.1. Sachant que la classe `X` est abstraite, la classe `Groupe` est-elle abstraite ou concrète ?

Solution : Elle est concrète. Deux façons de le voir. La première est que, d'après l'énoncé, on souhaite pouvoir grouper plusieurs objets (donc créer un groupe) pour manipuler plusieurs objets comme un seul. « Créer un groupe », c'est donc que la notion de groupe est concrète.

La deuxième façon est de savoir s'il y a des méthodes abstraites sur la classe Groupe. Il n'y en a pas. Par exemple, translater un groupe, c'est translater tous les objets géométriques du groupe. On s'appuie sur une méthode abstraite, celle d'ObjetGeometrique, mais on sait écrire la méthode translater de Groupe. C'est d'ailleurs ce qu'on a fait pour translater un schéma ! Toutes les méthodes sont concrètes. On n'a pas de raison ici d'en faire une classe abstraite.

4.2. Écrire la classe Groupe et l'utiliser (`ExempleSchemaGroupe`).

Solution : Voir les fichiers fournis dans la solution proposée.

4.3. On souhaite pouvoir mettre un groupe dans un groupe. Par exemple, on souhaite grouper les trois segments, puis ce groupe et le barycentre. Comment faire ?

Solution : Voici un exemple de code qu'on souhaiterait pouvoir écrire :

```
1 Groupe triangle = ...;
2 Groupe schema = ...;
3 schema.ajouter(triangle);
```

Ceci ne marche pas actuellement (erreur de compilation) car la méthode `afficher` de `Groupe` attend un `ObjetGeometrique`.

Deux solutions pour faire que ça marche.

Solution 1 : Surcharger ajouter. Dans la classe Groupe, on surcharge la méthode `ajouter`, cette deuxième méthode prend en paramètre un groupe et stocke le groupe dans une liste de groupes. Le diagramme de classe correspondant est donné à la figure 1.

Ceci fonctionne mais est relativement lourd car pour les méthodes translater, afficher, dessiner... il faudra écrire deux répétitions, l'une sur les objets géométriques, l'autre sur les groupes.

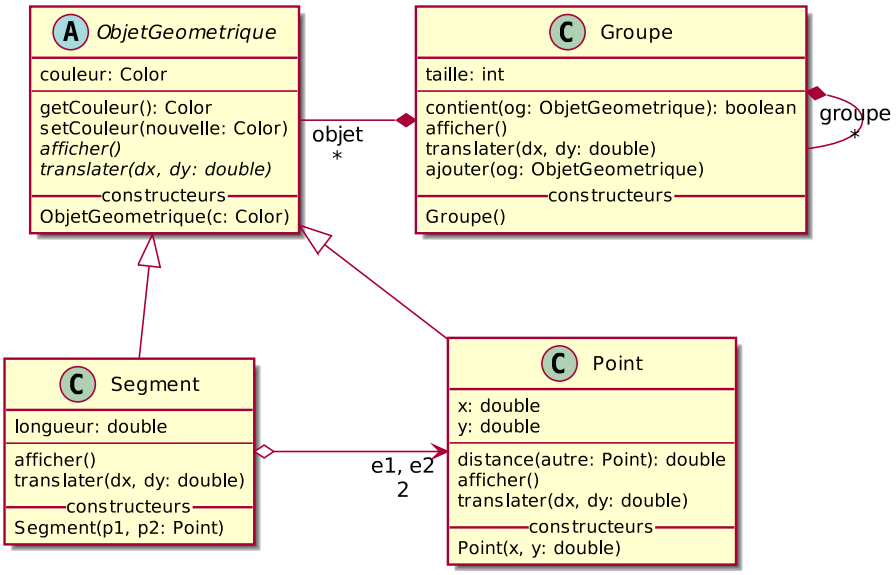

FIGURE 1 – Diagramme de la classe : Groupe contient des ObjetGeometrique et des Groupe

Solution 2 : Groupe devient un ObjetGeometrique. La deuxième solution consiste à donner à Groupe le type attendu par la méthode `ajouter` : ObjetGeometrique. On fait donc hériter `Groupe` de `ObjetGeometrique`. Le diagramme de classe correspondant est donné à la figure 2.

Est-ce légitime ? En d'autres termes est-ce Groupe est un sous-type de ObjetGéométrique ? Oui, car d'après l'énoncé grouper des objets, c'est pour les considérer comme un objet géométrique et pouvoir ainsi le translater, afficher, etc.

Dans ce cas, il faut définir les autres méthodes de ObjetGéométrique : en particulier setCouleur doit être redéfinie car elle consiste à changer la couleur de tous les objets du groupe. La méthode getCouleur est plus délicate. On peut lui donner le sens suivant : elle retourne la couleur commune à tous les objets du groupe, null s'il n'y a pas de couleur commune.

Il reste juste un petit problème : l'attribut couleur a-t-il un sens pour un Groupe ? En fait, non. On peut toujours l'ignorer (on a redéfini les méthodes getCouleur et setCouleur qui ne l'utilisent plus) mais ce n'est pas très satisfaisant. Le problème ne vient pas de Groupe mais de ObjetGeometrique. On constate que d'en faire une classe abstraite n'était pas une bonne idée car tous les objets géométrique n'ont pas une couleur propre. Il aurait été préférable d'en faire une interface (sans l'attribut, sans le constructeur et sans le code des méthodes getCouleur et setCouleur) et de définir une classe abstraite ObjetGeometriqueAbstrait qui factorise la gestion de la couleur et définit le constructeur. Point et Segment hériteraient alors de ObjetGeometriqueAbstrait et Groupe réaliserait ObjetGeometrique.

Attention, dans les deux cas à ne pas créer un cycle !

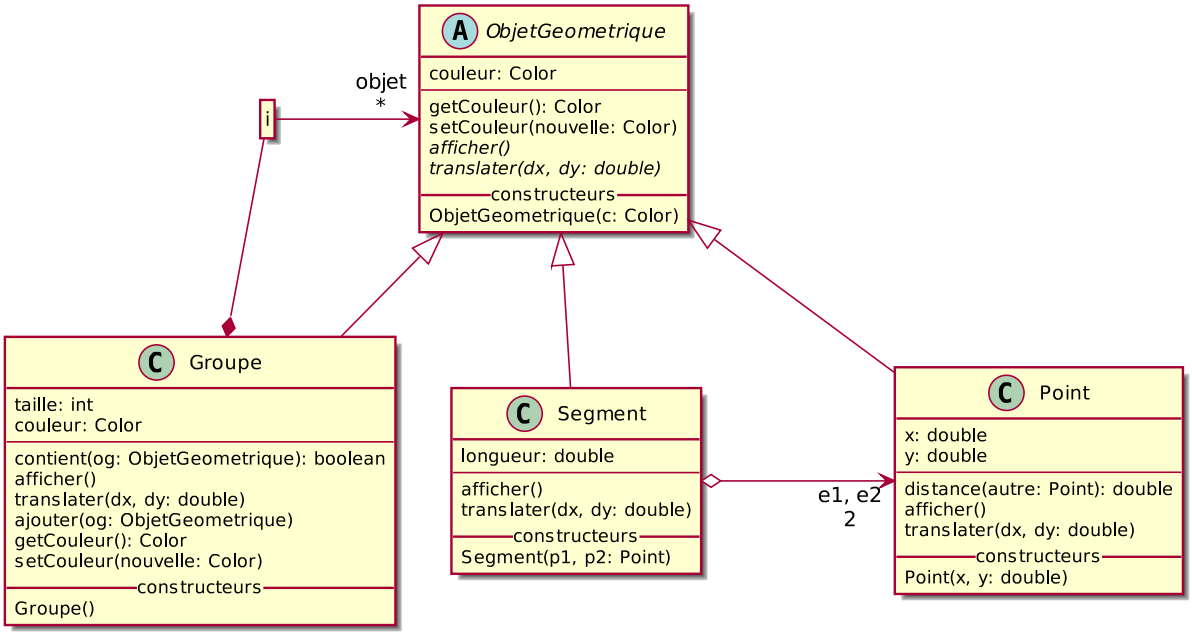

FIGURE 2 – Diagramme de la classe : Groupe est un ObjetGeometrique. La boîte [i] est une astuce pour représenter la relation de composition entre Groupe et ObjetGeometrique avec un point de contrôle intermédiaire.