IN201 : 10 – Observateur

Author : Christophe Garion <garion@isae.fr>
Public : SUPAERO 2A
Date :

Résumé

Le but de ce TP est de revoir les notions vues jusqu'à présent et de découvrir un nouveau *design pattern*, observateur, à travers un petit problème.

1 Objectifs

Les objectifs du TP sont les suivants :
– comprendre et utiliser les diagrammes de séquence ;
– utiliser les interfaces ou les classes abstraites ;
– comprendre un patron de conception.

2 Présentation du problème

La nouvelle modélisation de la classe `Segment` que nous proposons est donnée sur la figure 1. On suppose que l'on utilise `Segment` dans une application qui demande de façon très fréquente la longueur des segments, mais qui les modifie très peu. On a donc introduit un nouvel attribut qui est la longueur du segment. Le code de `getLongueur` est proposé sur le diagramme : la méthode ne fait que renvoyer la valeur de l'attribut `longueur`. Ceci permet d'éviter de calculer effectivement la longueur du segment à chaque appel à `getLongueur`[1].
Vous remarquerez également que certaines méthodes de `Point` ont « disparu » (`setX`, `setY` etc.) pour ne pas alourdir le diagramme. N'oubliez pas que les deux classes héritent de `Figure` et possèdent par exemple une méthode `sfficher`.

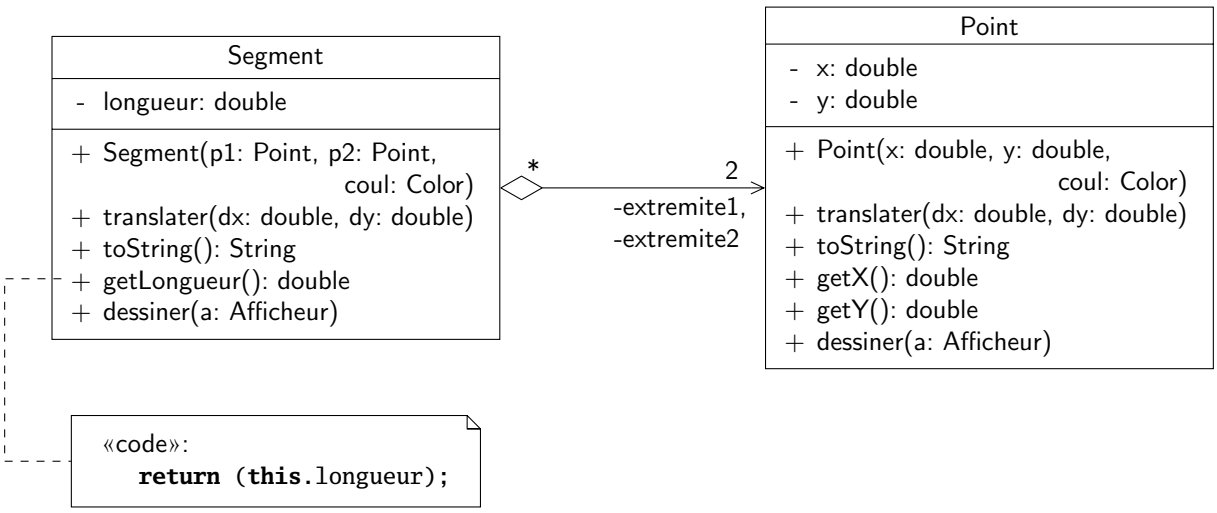

FIGURE 1 – Diagramme de classes initial des classes `Segment` et `Point`

3 Préparer la validation

Avant de nous intéresser à l'implantation en Java du modèle présenté sur la figure 1, nous allons spécifier les scénarios de test permettant de la valider. Chaque scénario devra pouvoir permettre d'écrire un programme de test qui sera utilisé pour tester les classes de l'application.
On ne décrit ici que le premier scénario de test :
– créer un point `p1` de coordonnées $(0, 0)$ et de couleur quelconque ;

1. Ceci est un problème fréquent en informatique. On dispose en effet de deux ressources : un espace de stockage et une unité de calcul. On peut donc choisir pour chaque caractéristique d'une classe soit de la stocker (sous forme d'un attribut par exemple), ce qui consomme de l'espace de stockage, soit de la calculer à chaque fois que l'on souhaite récupérer sa valeur, ce qui consomme du temps de calcul.

– créer un point p2 de coordonnées $(5, 0)$ et de couleur quelconque ;
– créer un segment s à partir de p1 et de p2 ;
– afficher les coordonnées du point p2 ;
– afficher le segment s ;
– afficher la longueur du segment s ;
– translater le point p2 du vecteur $(-2, 0)$;
– afficher les coordonnées du point p2 ;
– afficher le segment s ;
– afficher la longueur du segment s.

1. indiquer les résultats qui devront être affichés à l'écran à l'exécution du programme.

2. dessiner le diagramme de séquence correspondant au scénario.

4 Première implantation

On propose une première implantation des classes `Point` (cf. listing 2) et `Segment` (cf. listing 3). Les sources des classes ne sont pas représentés entièrement sur les listings [2]. Le programme de test correspondant au scénario précédent est représenté sur le listing 1.

Listing 1– Programme de test `TestSegment`

```
1  package fr.supaero.figure;
2
3  import java.awt.Color;
4
5  /**
6   * <code>TestSegment</code> est une classe de test pour la classe
7   * <code>Segment</code>.
8   *
9   * @author Xavier Cregut
10  * @author <a href="mailto:garion@supaero.fr">Christophe Garion</a>
11  * @version 1.0
12  */
13 public class TestSegment {
14
15     public static void main(String[] args) {
16         Point p1 = new Point(0, 0, Color.BLUE);
17         Point p2 = new Point(5, 0, Color.BLUE);
18         Segment s = new Segment(p1, p2, Color.RED);
19
20         System.out.print("p2 = ");
21         p2.afficher();
22         System.out.println();
23         System.out.print("s = ");
24         s.afficher();
25         System.out.println();
26         System.out.println("longueur de s = " + s.getLongueur());
27         System.out.println();
28
29         p2.translater(-2, 0);
30
31         System.out.print("p2 = ");
32         p2.afficher();
33         System.out.println();
34         System.out.print("s = ");
35         s.afficher();
36         System.out.println();
37         System.out.println("longueur de s = " + s.getLongueur());
38     }
39 }
```

Listing 2– Classe `Point`

```
1  package fr.supaero.figure;
2
3  import java.awt.Color;
4  import afficheur.Afficheur;
5
6  /**
7   * <code>Point</code> definit une classe point mathematique dans un
```

2. Les listings des classes ne sont pas complets, en particulier il manque les méthodes de la classe `Figure` comme `dessiner`.

```
 8  * plan qui peut etre considere dans un repere cartesien.<BR>
 9  * Un point peut etre translate. Sa distance par rapport a un autre
10  * point peut etre obtenue.
11  *
12  * @author <a href="mailto:garion@supaero.fr">Christophe Garion</a>
13  * @version 1.0
14  */
15 public class Point extends Figure {
16
17     private double x;
18
19     private double y;
20
21     /**
22      * Cree un nouvelle instance de <code>Point</code>.
23      *
24      * @param x_ un <code>double</code> representant l'abscisse du
25      *           point a creer
26      * @param y_ un <code>double</code> representant l'ordonnee du
27      *           point a creer
28      * @param couleur_ la couleur du point
29      */
30     //@ requires couleur_ != null;
31     public Point(double x_, double y_, Color couleur_) {
32         super(couleur_);
33         this.x = x_;
34         this.y = y_;
35     }
36
37     /**
38      * <code>translater</code> permet de translater le point.
39      *
40      * @param dx un <code>double</code> qui represente l'abscisse du
41      *           vecteur de translation
42      * @param dy un <code>double</code> qui represente l'ordonnee du
43      *           vecteur de translation
44      */
45     //@ also
46     //@ ensures this.getX() == \old(this.getX()) + dx;
47     //@ ensures this.getY() == \old(this.getY()) + dy;
48     @Override public void translater(double dx, double dy) {
49         this.x = this.x + dx;
50         this.y = this.y + dy;
51     }
52
53     /**
54      * <code>toString</code> renvoie un objet de type <code>String</code>
55      * qui represente une chaine de caracteres representant le point.
56      *
57      * @return un objet de type <code>String</code> representant
58      *         le point. Pour un point de coordonnees (2,3), cet objet
59      *         representera la chaine <code>(2,3)</code>.
60      */
61     @Override public String toString() {
62         return("(" + this.x + "," + this.y + ")");
63     }
64
65     /**
```

```
66      * <code>distance</code> permet de calculer la distance entre deux
67      * points.
68      *
69      * @param p un <code>Point</code> qui est l'autre point pour calculer
70      *          la distance
71      * @return un <code>double</code> qui est la distance entre les deux
72      *         point
73      */
74     //@ requires p != null;
75     public double distance(Point p) {
76         return (Math.sqrt((this.x - p.x) * (this.x - p.x) +
77                           (this.y - p.y) * (this.y - p.y)));
78     }
79 }
```

Listing 3– Classe Segment

```
 1 package fr.supaero.figure;
 2
 3 import java.awt.Color;
 4 import afficheur.Afficheur;
 5
 6 /**
 7  * <code>Segment</code> est une classe permettant de modeliser un
 8  * segment geometrique. Ce segment est compose de deux points et on
 9  * peut recuperer sa longueur et le translater.
10  *
11  * @author <a href="mailto:garion@supaero.fr">Christophe Garion</a>
12  * @version 1.0
13  */
14 public class Segment extends Figure {
15
16     private Point extremite1;
17     private Point extremite2;
18     private double longueur;
19
20     /**
21      * Cree une nouvelle instance de <code>Segment</code>. Attention,
22      * les points passes en parametre sont affectes directement aux
23      * attributs de l'objet a creer.
24      *
25      * <p> On aurait pu egalement creer de nouveaux points a partir des
26      * points passes en parametre.
27      *
28      * @param p1 un <code>Point</code> representant la premiere extremite
29      *           du segment
30      * @param p2 un <code>Point</code> representant la seconde extremite
31      *           du segment
32      * @param couleur_ la couleur du segment
33      */
34     //@ requires couleur_ != null;
35     //@ requires p1 != null;
36     //@ requires p2 != null;
37     //@ requires (p1.getX() == p2.getX()) ==> (p1.getY() != p2.getY());
38     //@ requires (p1.getY() == p2.getY()) ==> (p1.getX() != p2.getX());
39     public Segment(Point p1, Point p2, Color couleur_) {
40         super(couleur_);
41         this.extremite1 = p1;
```

```
42          this.extremite2 = p2;
43          this.longueur = p1.distance(p2);
44      }
45
46      /**
47       * <code>getLongueur</code> renvoie la longueur du segment.
48       *
49       * @return un <code>double</code> qui est la longueur du segment
50       */
51      public double getLongueur() {
52          return this.longueur;
53      }
54
55      /**
56       * <code>translater</code> permet de translater le segment.
57       *
58       * @param dx l'abscisse du vecteur de translation
59       * @param dy l'ordonnee du vecteur de translation
60       */
61      @Override public void translater(double dx, double dy) {
62          this.extremite1.translater(dx, dy);
63          this.extremite2.translater(dx, dy);
64      }
65
66      /**
67       * <code>toString</code> renvoie une chaine de caracteres (un
68       * objet de type <code>String</code>) representant le segment.
69       *
70       * @return un objet de type <code>String</code> representant
71       *          le segment. Pour un segment compose des deux points
72       *          <code>(1,0)</code> et <code>(2,3)</code>, cet objet
73       *          representera la chaine <code>[(1,0);(2,3)]</code>
74       */
75      @Override public String toString() {
76          return ("[" + this.extremite1 + ";" + this.extremite2 + "]");
77      }
78  }
```

1. compléter le diagramme de séquence dessiné à la section 3.

2. indiquer les resultats affichés à l'écran après l'exécution du programme.

3. commenter ces résultats. La longueur du segment est-elle cohérente ?

5 Correction des classes

Nous allons maintenant corriger les deux classes pour qu'elles respectent le cahier des charges. La solution proposée devra respecter les contraintes suivantes :
– la relation entre `Segment` et `Point` reste inchangée ;
– l'attribut `longueur` et la méthode `getLongueur()` de `Segment` restent inchangés.

1. indiquer les modifications à apporter en complétant le diagramme de séquence de la section 4.

2. compléter le diagramme de classes précédent.

6 Dernières mises au point et utilisation d'un *design pattern*

1. un point peut-il être extrémité de plusieurs segments ? Comment cela se traduit-il sur la solution ?
2. la solution précédente n'est pas satisfaisante. En effet, un point doit connaître `Segment` et plus généralement toutes les classes qui dépendent de ses changements (ex. de l'appel à `majLongueur`). On pourrait ainsi imaginer définir un cercle comme étant défini par l'agrégation de deux points, son centre et un point lui appartenant et par son rayon. Dans ce cas, si l'on translate un des points, il faut modifier le rayon du cercle si c'est le point appartenant à sa circonférence ou translater les deux points si le point est le centre du cercle. On pourrait également introduire un attribut représentant le périmètre d'un polygone qu'il faudrait modifier de la même façon.

   Proposer une solution générale et le diagramme de classes correspondant pour que la classe `Point` n'ait pas à connaître ces classes.

3. la solution proposée en cours sous forme d'un patron de conception est-elle réalisable dans notre cas ? Vous réfléchirez sur ce problème vis-à-vis de deux points :
   (a) l'existant (en particulier le fait que `Segment` et `Point` héritent de `Figure`) ;
   (b) (facultatif) la réutilisabilité de la solution.

7 Implantation des classes

Plusieurs classes sont fournies sur le site :
– les classes `Figure`, `Segment` et `Point` ;
– les classes de test `FigureTest`, `SegmentTest` et `PointTest` ;
– la classe `TestSegment` qui devra fonctionner correctement à la fin du TP.
Le travail demandé est donc le suivant :

1. écrire les interfaces `Observateur` et `Observable`.
2. modifier les classes `Segment` et `Point` pour qu'elles correspondent à la solution de conception donnée.
   Pour pouvoir stocker plusieurs observateurs dans un Point, on utilisera la classe `java.util.ArrayList`.
3. exécuter `TestSegment`.
4. (facultatif) le problème de cette solution est que les points extrémités d'un segment ont une référence sur ce segment. Si, dans une application, on ne se sert plus d'un segment prélablement inscrit (par le biais d'un objet de type `majLongueurSegment`) dans un `Point`, il ne sera pas récupéré par le *garbage collector* (ramasse-miettes). Comment éviter cela ?