IN201 : 6 – 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

On propose de construire une classe `Segment` à partir de la classe `Point` comme présenté 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 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.

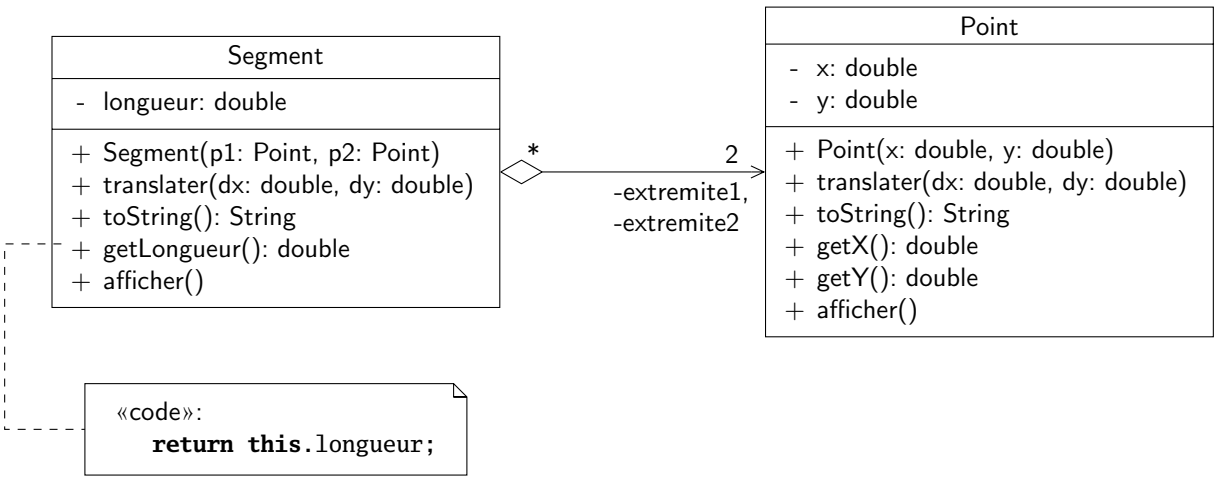


FIGURE 1 – Diagramme de classes initial des classes `Segment` et `Point`

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 $(0, 0)$;
- créer un point p2 de coordonnées $(5, 0)$;

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 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. Le programme de test correspondant au scénario précédent est représenté sur le listing 1.

Listing 1– Programme de test `TestSegment`

```
1  package fr.isae.geometry;
2
3  /**
4   * <code>TestSegment</code> est une classe de test pour la classe
5   * <code>Segment</code>.
6   *
7   * @author <a href="mailto:cregut@enseeiht.fr">Xavier Cregut</a>
8   * @author <a href="mailto:garion@isae.fr">Christophe Garion</a>
9   * @version 1.0
10  */
11 public class TestSegment {
12
13     /**
14      * Programme permettant de "verifier" si la longueur d'un
15      * segment ne change pas lors de la translation d'un des points
16      * ayant servi a le creer.
17      *
18      * @param args non utilise ici
19      */
20     public static void main(String[] args) {
21         Point p1 = new Point(0.0, 0.0);
22         Point p2 = new Point(5.0, 0.0);
23         Segment s = new Segment(p1, p2);
24
25         System.out.print("p2 = ");
26         p2.afficher();
27         System.out.println();
28         System.out.print("s = ");
29         s.afficher();
30         System.out.println();
31         System.out.println("longueur de s = " + s.getLongueur());
32         System.out.println();
33
34         p2.translater(-2.0, 0.0);
35
36         System.out.print("p2 = ");
37         p2.afficher();
38         System.out.println();
39         System.out.print("s = ");
40         s.afficher();
41         System.out.println();
42         System.out.println("longueur de s = " + s.getLongueur());
43     }
44 }
```

Listing 2– Classe `Point`

```
1  package fr.isae.geometry;
2
3  /**
4   * <code>Point</code> definit une classe point mathematique dans un
```

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

```
63       * <code>distance</code> permet de calculer la distance entre deux
64       * points.
65       *
66       * @param p un <code>Point</code> qui est l'autre point pour calculer
67       *          la distance
68       * @return un <code>double</code> qui est la distance entre les deux
69       *         point
70       */
71      public double distance(Point p) {
72          return (Math.sqrt(((this.x - p.x) * (this.x - p.x)) +
73                            ((this.y - p.y) * (this.y - p.y))));
74      }
75  }
```

Listing 3– Classe Segment

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

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

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 ?

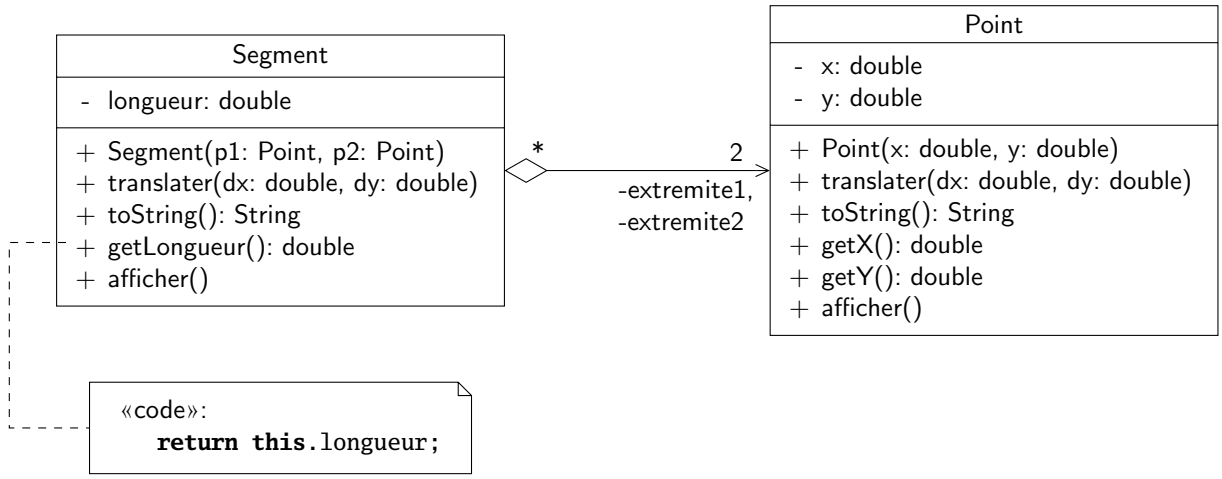
5 Correction des classes

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. aurait-on pu utiliser des interfaces à la place de classes abstraites pour `Observable` et `Observateur` ?
4. la classe `Observable` est-elle nécessairement abstraite ? Si non, pourquoi ?
5. quelle doit-être la visibilité des méthodes de `Observable` ? On supposera que `Point` et `Segment` sont dans le paquetage `fr.isae.geometry` et `Observable` dans le paquetage `fr.isae.observer`.
6. (facultatif) est-ce que la méthode `miseAJour` est bien définie en terme de signature ? Que se passe-t-il si par exemple `Segment` est observateur d'autres objets que ses extrêmités ?

7 Implantation des classes

Pour réaliser le TP, vous allez devoir créer un projet Java sous Eclipse en utilisant votre dépôt Subversion. Si vous avez configuré votre dépôt pour qu'il soit disponible dans la vue *SVN Repository Exploring* d'Eclipse, vous créérez votre projet en faisant un *checkout* **à partir du répertoire TP6** de votre dépôt en cliquant droit dessus. N'oubliez pas de configurer votre *build path* pour pouvoir utiliser les éventuelles archives JAR placées dans le répertoire `lib`.

Plusieurs classes sont fournies sur le site :
- les classes `Segment` et `Point` ;
- la classe `TestSegment` qui devra fonctionner correctement à la fin du TP.

Vous trouverez également dans l'archive `lab6.jar` les classes de test `SegmentTest` et `PointTest` pour vous permettre de vérifier que votre implantation ne modifie pas le comportement initial des classes.
Le travail demandé est donc le suivant :

1. écrire les classes `Observateur` et `Observable`. Elles devront appartenir au paquetage `fr.isae.observer`.
   Pour pouvoir stocker plusieurs observateurs dans un `Observable`, on utilisera la classe `java.util.ArrayList`.
   On pourra réfléchir aux tests éventuels de `Observable` et `Observateur`.
2. modifier les classes `Segment` et `Point` pour qu'elles correspondent à la solution de conception donnée.

Attention, il existe déjà une classe `Observable` dans `java.util`, utilisez votre classe !

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 ?

Références

[1] Oracle. *The Java Tutorials – Trail : Learning the Java Language – Lesson : Nested classes*. 2014. URL : http://docs.oracle.com/javase/tutorial/java/javaOO/nested.html.

License CC BY-NC-SA 3.0