IN201 : 6 – Observateur

Solution

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 Contenu

Ce document est un corrigé succint du TP portant sur les observateurs. Il est en particulier plus complet que ce qui vous était demandé. Tous les fichiers source sont disponibles sur le site `http://www.tofgarion.net/lectures/IN201`.

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.

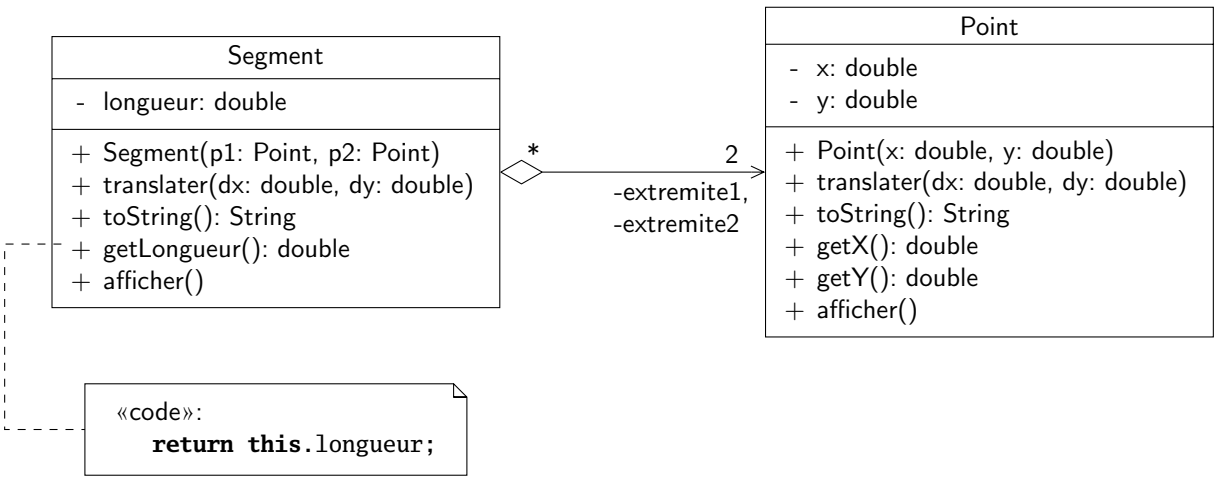

FIGURE 1 – Diagramme de classes initial des classes `Segment` et `Point`

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 $(5, 0)$;
- créer un segment `s` à partir de `p1` et de `p2` ;
- afficher les coordonnées du point `p2` ;

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.

- 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.

   Solution :
   On devrait normalement obtenir la sortie suivante :

```
p2 = (5.0,0.0)

s = [(0.0,0.0);(5.0,0.0)]

longueur de s = 5.0

p2 = (3.0,0.0)

s = [(0.0,0.0);(3.0,0.0)]

longueur de s = 3.0
```

2. dessiner le diagramme de séquence correspondant au scénario.

   Solution :
   Le diagramme de séquence est donné sur la figure 2. Rien de bien particulier, on voit qu'un diagramme de séquence permet de représenter facilement un scénario de test.

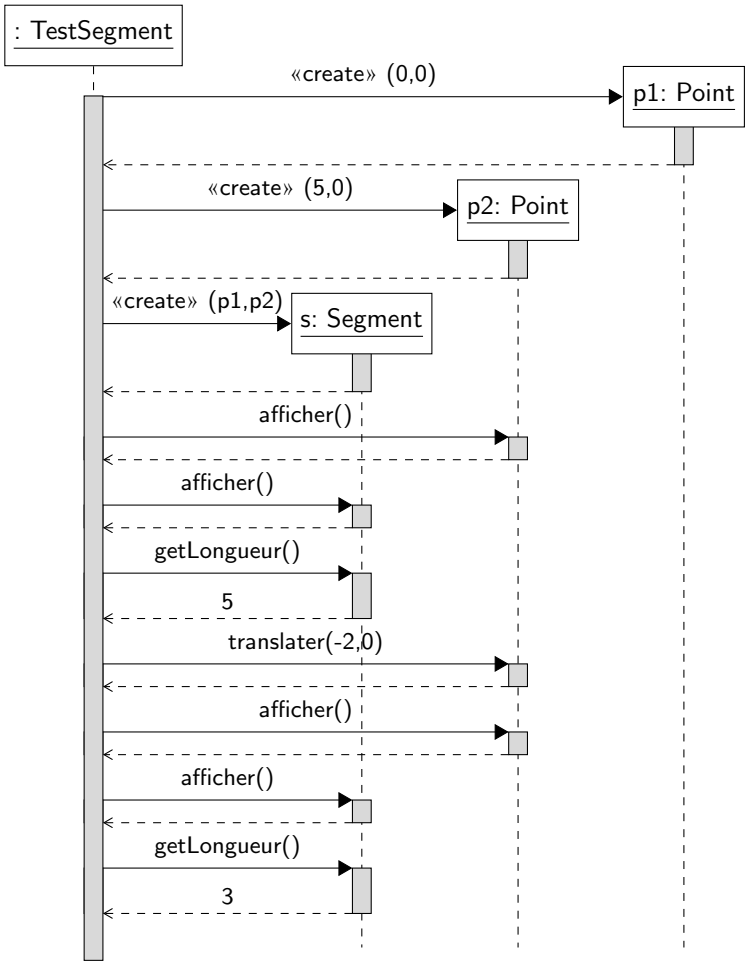

FIGURE 2 – Diagramme de séquence représentant le scénario de validation

4 Première implantation

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

   Solution :
   Le diagramme de séquence est présenté sur la figure 3. Comme on a le détail des classes, on peut préciser quels sont les appels de méthodes à l'intérieur de `afficher` par exemple. Je n'ai pas mis dans ce diagramme tous les résultats des appels à `toString` pour ne pas surcharger le diagramme.

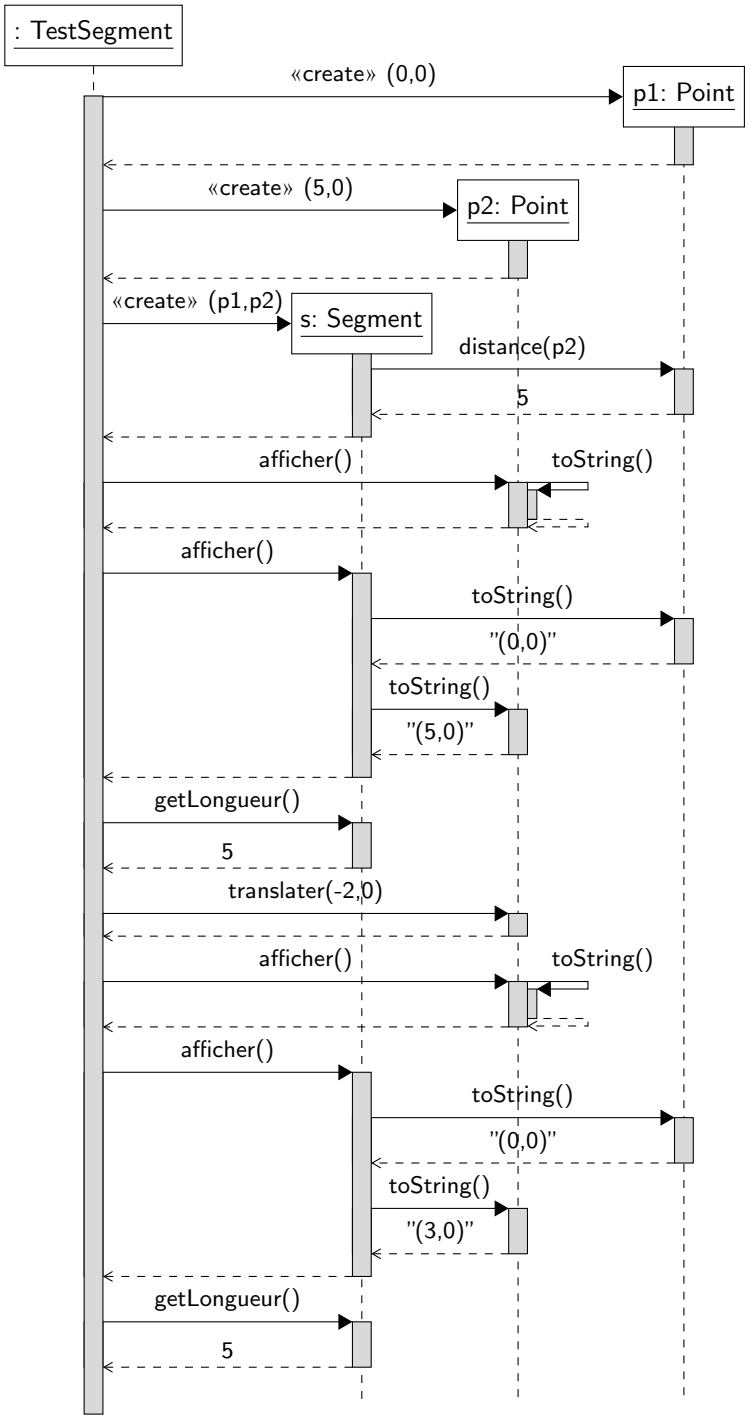

FIGURE 3 – Diagramme de séquence représentant la première exécution de `TestSegment`

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

Solution :

D'après les sources des classes, le résultat affiché à l'écran sera :

```
p2 = (5.0,0.0)

s = [(0.0,0.0);(5.0,0.0)]

longueur de s = 5.0

p2 = (3.0,0.0)
```

```
s = [(0.0,0.0);(3.0,0.0)]

longueur de s = 5.0
```

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

   Solution :
   la longueur du segment n'est pas cohérente. Lorsque le point `p2` est translaté, l'extrémité `extremite2` du segment change également, mais la longueur n'est pas mise à jour.
   Le diagramme de séquence de la figure 3 montre bien que lorsque le point `p2` est translaté, le segment `s` n'a pas connaissance de cette modification.
   On remarquera que le principe d'encapsulation est très mal appliqué ici : les points composant le segment sont accessibles directement depuis l'extérieur du segment ! Il aurait fallu copier les points passés en paramètres du constructeur de `Segment` pour pouvoir encapsuler correctement l'état du segment.

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.

   Solution :
   L'association entre `Segment` et `Point` doit être conservée, ainsi que `longueur` et l'implantation de `getLongueur`. Trois modifications sont à apporter pour pouvoir conserver la cohérence de la longueur :
   - le point doit signaler au segment que ses coordonnées ont été modifiées. Il suffit donc de modifier la méthode `translater` et les modifieurs des coordonnées de façon à recalculer la longueur du segment à chaque changement de l'état du point ;
   - il faut donc de plus qu'il existe une méthode publique `majLongueur` dans `Segment` qui permette au point de mettre à jour la longueur du segment ;
   - enfin, il est naturel que le constructeur du segment inscrive le segment auprès des deux points pour que ceux-ci aient une référence sur laquelle appeler `majLongueur()`.

   Le diagramme de séquence modifié est présenté sur la figure 4.

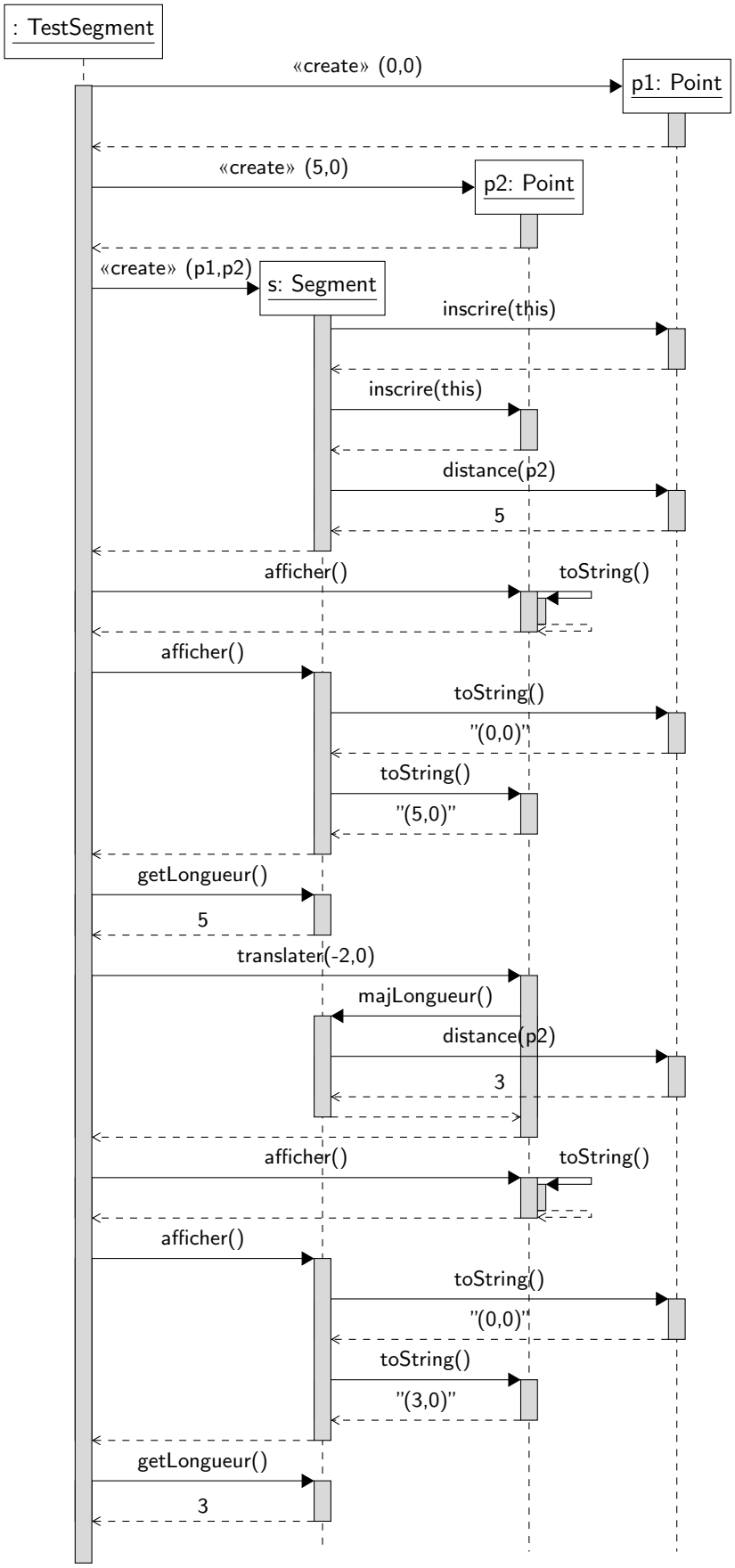

FIGURE 4 – Diagramme de séquence présentant une exécution « correcte » de `TestSegment`

2. compléter le diagramme de classes précédent.

Solution :

Il faut que les instances de la classe `Point` aient accès aux instances de la classe `Segment` dont elles sont extrémités. La relation entre `Segment` et `Point` doit donc être bidirectionnelle. De plus, on doit introduire une méthode `inscrire` dans `Point` pour pouvoir référencer les segments et une méthode publique `majLongueur` dans `Segment` pour recalculer la longueur du segment. J'ai préféré faire apparaître deux associations distinctes pour pouvoir séparer l'agrégation entre le segment et ses extrémités et la relation de « mise à jour » entre un point et un segment. Le diagramme est présenté sur la figure 5.

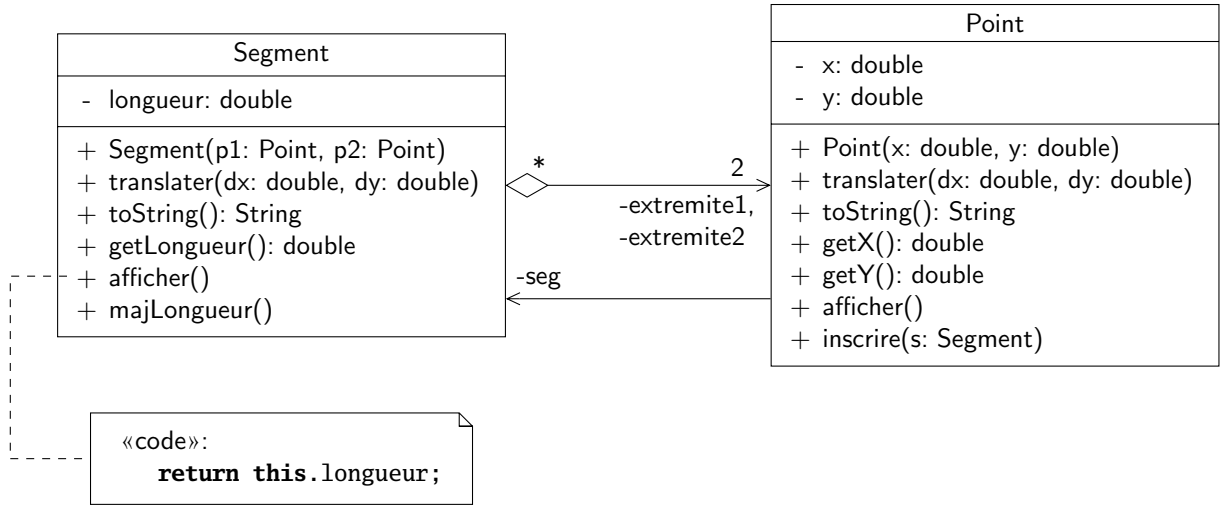

FIGURE 5 – Diagramme de classe présentant les deux classes `Point` et `Segment` modifiées

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 ?

Solution :

Rien n'empêche qu'un point soit extrémité de plusieurs segments (le diagramme de classe nous le précise). Il faut donc stocker l'ensemble des segments dont le point est extrémité dans la classe `Point`. Pour cela, on peut utiliser une liste d'objets de type `Segment` via la classe `ArrayList`.

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.

Solution :

Le problème est donc de généraliser ce qui se passe entre la classe `Point` et la classe `Segment`. La difficulté provient également des différentes mises-à-jour possibles qui ont une sémantique différente (recalculer la longueur du segment, recalculer le rayon du cercle, translater le point appartentant à la circonférence du cercle, recalculer le perimètre du polygone, etc.).

Ce problème, qui consiste à effectuer un certain nombre d'actions lorsque le point change, a été résolu par un *design pattern* (patron de conception) appelé *Observateur*. Les *design patterns* [1, 2]sont des solutions de conception réutilisables à desproblèmes génériques en informatique.

Le diagramme de classe proposé par le patron *Observateur* est représenté sur la figure 6.

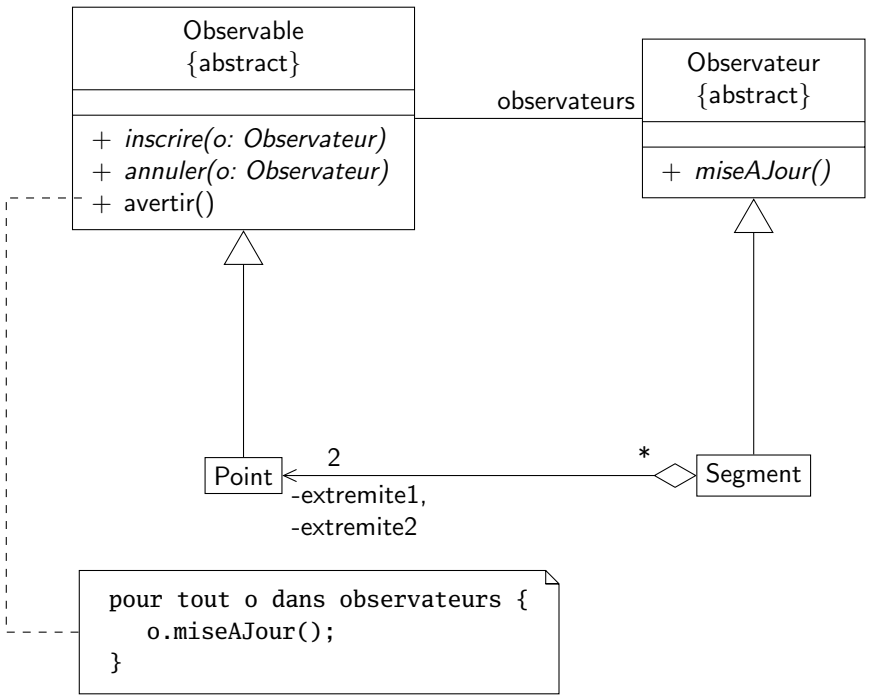

FIGURE 6 – Patron de conception *Observateur* adapté au problème étudié

3. aurait-on pu utiliser des interfaces à la place de classes abstraites pour `Observable` et `Observateur` ?

   Solution :

   On aurait très bien pu utiliser des interfaces pour `Observable` et `Observateur`. En particulier, cela est nécessaire si par exemple `Point` ou `Segment` héritent déjà d'une autre classe avec Java. Mais on perd le bénéfice de pouvoir implanter des méthodes (cf. question suivante).

4. la classe `Observable` est-elle nécessairement abstraite ? Si non, pourquoi ?

   Solution :

   On peut effectivement si `Observable` doit être abstraite. En effet :
   - l'inscription et l'annulation d'un observateur peut se faire à travers un attribut de type `ArrayList<Observateur>` défini dans `Observable`. On peut alors écrire `inscrire` et `annuler` (qui délèguent leur service vers `ArrayList`).
   - si on stocke les observateurs dans une liste dans `Observable`, on sait écrire `avertir` : il suffit d'utiliser la boucle **`for`** sur l'instance de `ArrayList` pour demander la mise-à-jour de tous les observateurs.

   On peut donc choisir ici d'avoir une classe concrète avec toutes ses méthodes implantées. On remarquera que l'on aurait même pu rendre toutes les méthodes finales, i.e. empêcher leur redéfinition.

   Mais est-il judicieux de pouvoir créer directement un objet de type `Observable` ? Je ne pense pas, car la méthode `avertir` doit être appelée par un objet réellement observable et pas par un utilisateur extérieur par exemple. On peut laisser `Observable` abstraite, *même si toutes ses méthodes sont concrètes* : Java impose qu'une classe ayant une méthode abstraite soit abstraite, mais on peut déclarer une classe abstraite même si elle n'a aucune méthode abstraite. On ne pourra pas l'instancier.

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`.

   Solution :

   `inscrire` et `annuler` sont des méthodes qui pourront être appelées par des classes à l'extérieur du paquetage `fr.isae.observer` (comme `Segment` par exemple). Il faut donc qu'elles soient publiques. Par contre, `avertir` ne devrait pas l'être : un utilisateur extérieur ne devrait pas pouvoir forcer un objet à avertir tous ses observateurs, c'est à l'objet lui-même de décider quand le faire. On va donc rendre `avertir` *protégée* afin qu'elle soit accessible dans la classe `Point` qui sera une sous-classe de `Observable` par exemple.

   Le diagramme de classe final à utiliser est présenté sur la figure 7.

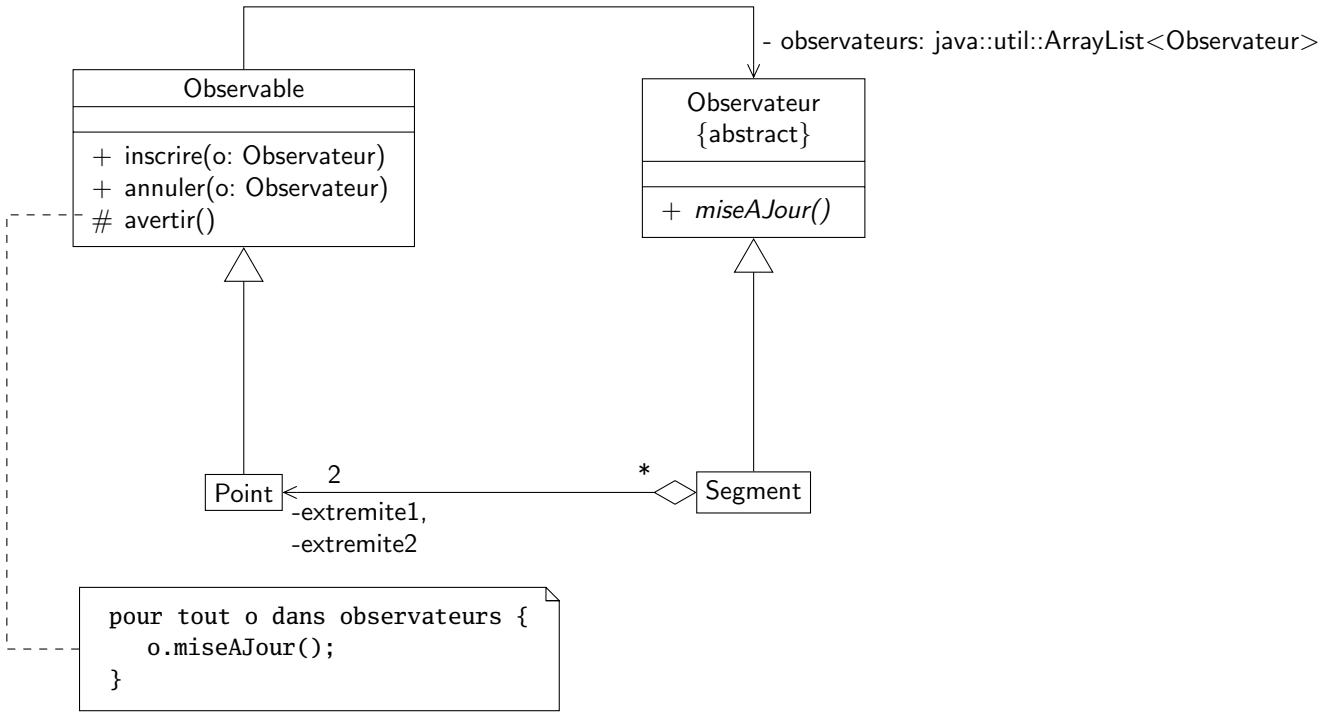

FIGURE 7 – Patron de conception *observateur* à implanter

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 ?

Solution :

C'est la partie la plus compliquée à prendre en compte (elle ne vous était pas demandée). On peut remarquer également que le traitement des mises à jour est fait explicitement dans la classe `Segment`. Or, il se peut très bien qu'un segment soit un observateur d'objets autre que les deux points. Comme il existe une seule méthode `miseAJour()` dans `Segment`, cette méthode doit faire *toutes les mises à jour possibles*, ce qui peut prendre énormement de temps. On pourrait également prendre l'exemple d'un cercle qui est caractérisé par deux points : son centre et un point de sa périphérie. Le comportement des deux points du cercle lors d'une translation pourrait ne pas être le même : si l'on translate le centre du cercle, on veut translater le cercle et si l'on translate le point périphérie du cercle, on conserve l'état du centre du cercle.

Il est donc intéressant de déléguer le traitement d'une mise à jour spécifique à un objet particulier de type `Observateur`.

La figure 8 présente une solution permettant de résoudre ce problème. On peut finalement remarquer qu'en appliquant les connaissances dont nous disposons, cette solution est problématique : il faut que l'instance de `MajLongueurSegment` associée à un segment ait accès à l'attribut `longueur` du segment (via par exemple une méthode publique `calculLongueur`). Il serait plus judicieux d'utiliser une *classe interne* dans `Segment`, notion qui abordée plus tard dans le cours. La recherche d'une solution la plus générique possible peut parfois conduire à une conception qui n'est pas très élégante...

Pour reprendre l'exemple du cercle, on pourrait alors avoir deux observateurs particuliers associés au cercle : `TranslationCercle` qui serait utilisé par le centre du cercle, et `MajRayon` qui serait utilisé par le point périphérie du cercle.

On aurait également pu passer en paramètre de `miseAJour` l'objet appelant la mise-à-jour pour pouvoir ensuite différencier les traitements à effectuer. C'est la solution retenue dans l'implantation du patron fournie par l'API de Java, cf. section 8.

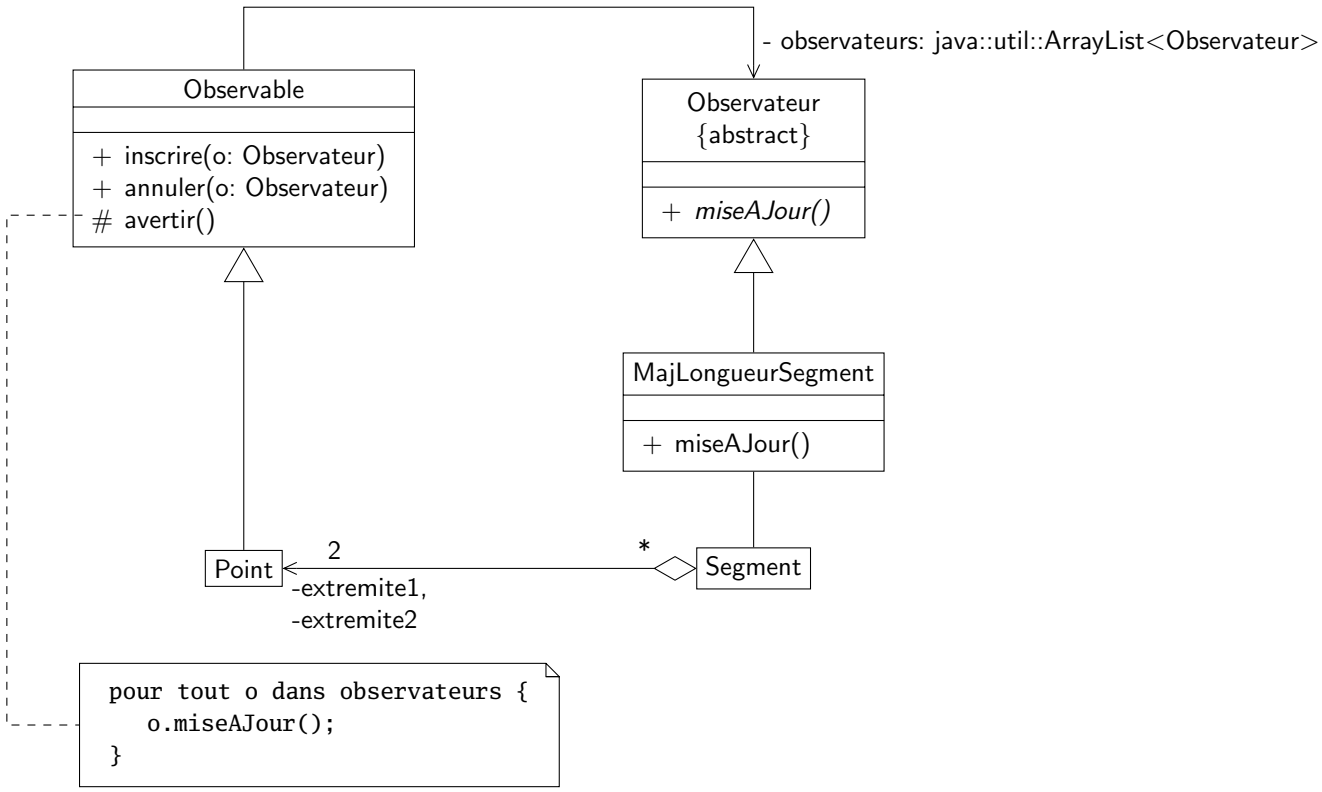

FIGURE 8 – Diagramme de classes présentant la solution la plus « générale » possible

7 Implantation des classes

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.
3. exécuter `TestSegment`.

   Solution :

   Toutes les classes sont disponibles sur le site. La solution proposée ne fait pas apparaître la classe `MajLongueurSegment`.

   Il fallait bien évidemment utiliser le diagramme de séquence fait en salle pour vérifier que l'on implantait correctement la solution. L'utilisation du diagramme permettait par exemple d'éviter les erreurs classiques suivantes :
   - oubli de l'inscription du segment dans le constructeur de `Segment` ;
   - oubli de l'appel à `avertir` dans `translater`.

   La question des tests éventuels de `Observable` et `Observateur` était intéressante. En ce qui concerne `Observateur`, il n'y a pas de tests à effectuer, car toutes les méthodes de la classe sont abstraites. Par contre, `Observable` a des méthodes concrètes (toutes ses méthodes en fait !) et on devrait les tester. Par contre, comme la classe est abstraite, on ne peut pas l'instancier et de plus on a besoin d'observateurs pour vérifier le fonctionnement...

   Une solution classique est de créer des sous-classes instantiables qui ne servent que pour les tests (on parle de *dummy objects*). J'ai donc créé une sous-classe `DummyObservateur` de `Observateur` qui possède un booléen que l'on passe à **`true`** lorsque l'objet est mis-à-jour et une sous-classe `DummyObservable` de `Observable` qui sert juste à appeler `avertir`. Les tests correspondants sont contenus dans la classe `fr.isae.observer.ObservableTest`. Afin d'éviter la création directe de ces classes dans le paquetage `fr.isae.observer` alors qu'elles ne servent que pour les tests,

j'ai défini `DummyObserver` et `DummyObservable` comme des classes *internes* à `ObserverTest`. Une classe interne est encapsulée dans une autre classe (et a accessoirement accès aux caractéristiques de la classe englobante même si elles sont privées). Comme ce sont des caractéristiques de `ObservableTest` au même titre qu'un attribut ou une méthode, je les ai rendues privées, afin d'éviter leur instantiation en dehors de `ObservableTest`. Vous trouverez plus de détails sur les classes internes sur [3] et nous en reparlerons lors du cours sur les interfaces graphiques.

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 ?

   Solution :

   Pour comprendre le problème posé par la destruction d'un segment, il faut revenir sur la gestion de l'espace mémoire dans une machine virtuelle Java. On crée des instances d'une classe en utilisant l'opérateur **`new`**, qui alloue de la mémoire dans le tas pour stocker l'objet que l'on veut créer. Par contre, il n'y a pas d'opérateur `delete` qui permette de libérer la mémoire utilisée par un objet dont on n'a plus besoin (correspondance `malloc`/`free` que vous avez vue en C).

   Quand on décide qu'un objet n'est plus « utile », il suffit de ne plus le référencer (par exemple, en affectant la référence qui pointait dessus à **`null`**, en affectant un nouvel objet à la référence ou en n'utilisant plus la référence tout simplement). Un tel objet est appelé *garbage* en Java. Il existe un « processus » qui est exécuté en même temps que votre application dans la JVM et qui est appelé *garbage collector* (ramasse-miettes). Le ramasse-miettes a pour but de libérer l'espace utilisé par les objets qui ne sont plus référencés.

   Dans notre cas, supposons que nous ayons un objet de type `Segment` référencé par `s` et qui soit inscrit en tant qu'observateur auprès d'un point `p1`. Même si on ne référence plus le segment par `s`, il existe toujours une référence vers l'objet de type `Segment` référencé par `s`. En effet, `p1` a comme attribut une instance de `ArrayList` (via `Observable`) qui possède elle même une référence vers le segment considéré. Le segment n'est donc pas candidat pour le ramasse-miettes à une libération de l'espace mémoire. Ceci est gênant : on ne se sert plus de l'objet, mais on ne peut pas libérer l'espace mémoire qu'il occupe.

   Il existe une méthode dans la classe `Object` dont la signature est la suivante : **`protected void`** `finalize()` **`throws`** `Throwable`. Cette méthode est exécutée avant que la mémoire occupée par l'objet correspondant ne soit réclamée. Par contre, il n'y a aucune garantie que cette méthode soit appelée à un moment précis. Il se peut même très bien que la méthode ne soit jamais appelée. De plus, cette méthode n'est appelée que si l'objet correspondant est candidat à une libération d'espace mémoire. Or, dans notre cas, le segment ne sera jamais candidat à une libération, puisqu'il existe une référence dessus (via le point `p1`).

   Il faut donc écrire une méthode spécifique dans la classe `Segment` qui libère les références restantes (en utilisant par exemple `p1.annuler(this)`) et que l'on appelle dans le programme de test lorsque l'on est sûr de ne plus se servir du segment en question.

Ce genre de question est très technique et n'apparaîtra évidemment pas dans l'examen.

Vous remarquerez également que dans le paquetage `fr.isae.geometry.more`, j'ai implanté la solution utilisant comme présenté en section 6 : une classe `MajLongueurSegment` qui sert d'observateur « réel » pour le segment. J'ai implanté `MajLongueurSegment` comme une classe privée interne à la classe `Segment`. Elle a donc accès directement aux attributs de la classe `Segment`. Encore une fois, nous reviendrons sur la notion de classe interne dans le cours sur les interfaces graphiques et vous trouverez plus de détails sur le sujet dans sur [3].

8 La classe `Observable` et l'interface `Observer` de l'API

L'API Java fournit une classe, `Observable`, et une interface, `Observer`, toutes deux contenues dans le paquetage `java.util` et qui implantent le patron de conception Observateur.
La classe `Observer` est représentée sur la figure 9. Elle possède des méthodes pour ajouter ou retirer un observateur sur l'objet observable. Elle possède des méthodes pour avertir tous les observateurs ou un observateur particulier que l'état de l'objet a changé. Attention, pour que ces méthodes préviennent effectivement les observateurs, il faut avoir auparavant appelé la méthode `setChanged` pour signifier que l'état de l'objet observable a changé. Vous pouvez vous référer à la javadoc de `Observable` pour en savoir plus.

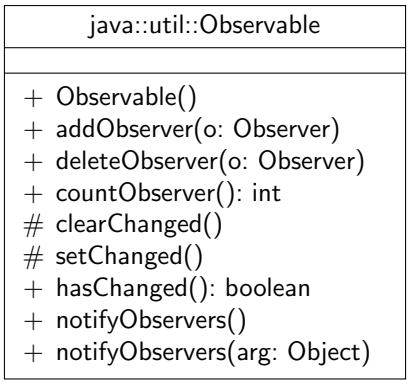

FIGURE 9 – La classe `java.util.Observable`

L'interface `Observer` est représentée sur la figure 10. Elle ne possède qu'une seule méthode, `update`, qui sert de méthode de mise à jour. L'argument `arg` de cette méthode permet de passer un objet en paramètre de la mise-à-jour, comme par example des paramètres nécessaires à la mise-à-jour. L'argument `o` de type `Observable` est instancié lors de l'appel à `notifyObservers` depuis l'objet observable. On peut donc s'en servir pour connaître l'objet qui a appelé la mise à jour.

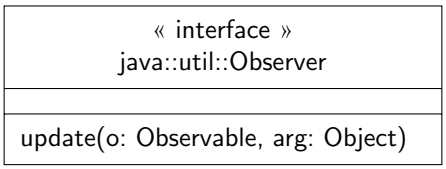

FIGURE 10 – L'interface `java.util.Observer`

J'ai utilisé cette classe et cette interface dans le paquetage `fr.isae.geometry.api` proposé dans la solution. Dans ce paquetage, j'ai implanté les classes `Segment`, `Cercle` et `Point` de la façon suivante :

- la classe `Cercle` a pour attributs deux points (un représentant le centre du cercle et un autre un point de sa périphérie) et un réel représentant le rayon du cercle.
- les classes `Segment` et `Cercle` réalisent l'interface `Observer` et implanteront donc la méthode `update`.
- la classe `Point` devrait normalement étendre la classe `Observable`. Or `Point` étend déjà `Figure`, donc ce n'est pas possible. J'ai donc choisi la seule solution qui restait, i.e. encapsuler dans `Point` un objet de type `Observable`. C'est sur cet objet que l'on va inscrire les segments et les cercles via une méthode `getObservable` permettant d'y avoir accès (cf. constructeurs de `Segment` et de `Cercle`). Reste toutefois un problème : lorsque l'on veut prévenir les observateurs d'un point que celui-ci a changé d'état via une des méthodes `notifyObservers`, il faut appeler la méthode `setChanged` sur l'observable associé (cf. javadoc de `Observable`). Or la méthode `setChanged` est protégée : on ne peut pas l'appeler depuis `Point` [2] ! J'ai donc créé une classe interne `PointObservable` qui étend `Observable` et redéfinit `setChanged`. La redéfinition de `setChanged` ne fait qu'appeler **`super.setChanged()`**, mais on peut l'appeler cette fois-ci depuis les méthodes `setX`, `setY` et `translater` de `Point`.
- le dernier problème à régler était les changements de coordonnées du point représentant le centre du cercle. En effet, dans ce cas on considère que le cercle va être translaté. Il faut donc translater également le point « périphérie » du cercle. Mais pour cela, il faut disposer des coordonnées du vecteur de translation du point centre. J'ai donc utilisé l'argument de type `Object` de la méthode `notifyObservers` pour passer en paramètre de cette méthode un tableau de deux réels représentant les coordonnées du vecteur de translation :

```
public void translater(double dx, double dy) {
    this.x = this.x + dx;
    this.y = this.y + dy;
    double[] coordTranslation = {dx, dy};
    this.setChanged();
    this.notifyObservers(coordTranslation);
}
```

On récupère ensuite ce tableau [3] dans la méthode `update` de `Cercle` :

2. `Point` n'appartient pas au paquetage `java.util` et n'est pas une sous-classe de `Observable`.
3. Après transtypage, car le type du paramètre est `Object`.

```
        if (o == this.centre) {
            double[] coordTranslation = (double[]) arg;
            this.periph.translater(coordTranslation[0],
                                   coordTranslation[1]);
        } else if (o == this.periph) {
            this.rayon = this.centre.distance(this.periph);
        }
    }
}
```

– enfin, la méthode `update` de `Cercle` était appelée dans deux cas : changement de coordonnées du centre ou du point « périphérie ». Pour distinguer ces deux cas, j'utilise l'argument de type `Observable` pour savoir si le point demandant la mise à jour est le centre ou le point « périphérie ». Attention, il faut bien utiliser l'égalité en mémoire avec == pour cela.

Je n'ai pas écrit de classes de test JUnit mais une petite application `TestFigures` pour vérifier que tout fonctionnait.

Références

[1] E. Gamma et al. *Design Patterns - Elements of reusable object-oriented software*. Addison-Wesley professional computing series. Addison-Wesley, 1994.

[2] E. Freeman et al. *Head first design patterns*. O' Reilly, 2005.

[3] 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