Encore les segments et les points !
Corrigé

L'objectif de ces exercices est de programmer en Java le modèle d'analyse présenté à la figure 1 en ayant comme contrainte de réalisation d'avoir un attribut dans la classe Segment pour stocker la longueur du segment.

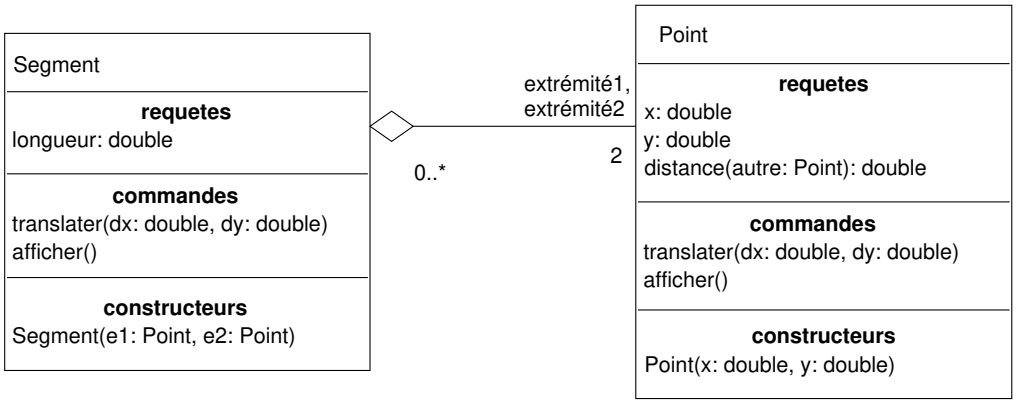


FIGURE 1 – Le diagramme de classe initial des classes Segment et Point

Exercice 1 : Préparer la phase de validation

Avant de commencer la programmation en Java du modèle d'analyse, commençons par spécifier les scénarios de test qui nous permettront de la valider. Chaque scénario permettra d'écrire un programme de test qui sera utilisé pour tester les classes de l'application écrite.

Nous ne décrivons ici que le premier scénario qui consiste à :

- créer un point p1 de coordonnées (0, 0) ;
- créer un point p2 de coordonnées (5, 0) ;
- créer un segment s à partir des points p1 et 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 ;

1.1. Indiquer les résultats qui *devront* être affichés à l'écran par ce scénario de test.

Solution : Commencer par dire qu'un scénario n'a pas d'intérêt si le résultat attendu n'est pas donné. En effet, il faut être capable de valider que le scénario s'est déroulé correctement.

À la fin de l'exécution du scénario de test, les lignes suivantes devraient être affichées à l'écran.

```
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
```

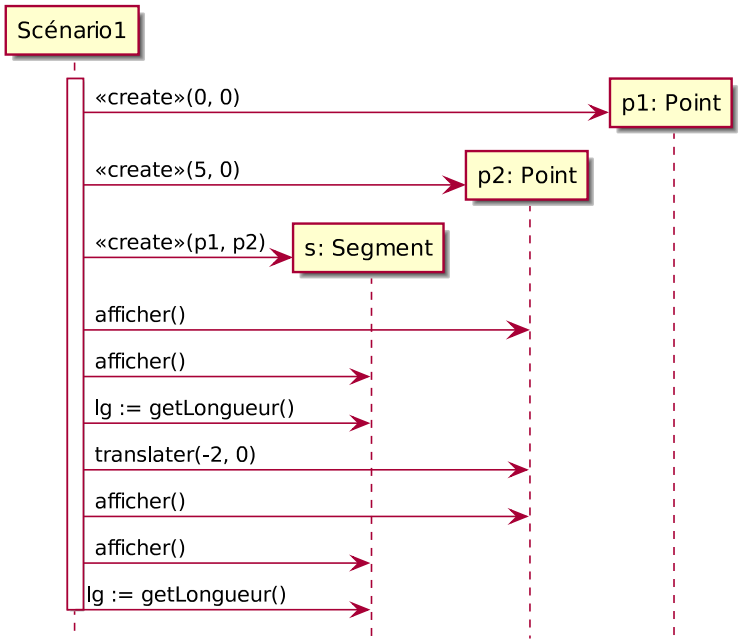
Remarque : Par rapport au scénario, nous avons ajouté le nom des objets affichés. Ceci est utile (et nécessaire) pour pouvoir bien suivre et comprendre l'exécution du programme.

Notons que le cahier des charges initial n'avait pas précisé explicitement l'effet de la translation d'un point sur les segments dont il est extrémité. On aurait pu envisager que celà provoque une translation du segment et non une modification de sa longueur.

Pour la suite, nous supposons que c'est l'exécution décrite qui est le comportement attendu.

1.2. Dessiner le diagramme de séquence qui correspond à ce scénario.

Solution : Le diagramme de séquence a pour objectif de faire apparaître les envois de messages (les appels de méthodes) entre les objets du système. Définir le diagramme de séquence consiste à identifier les méthodes et à les distribuer sur les objets. Ici, le diagramme de classe ayant déjà été défini, il permet simplement de formaliser graphiquement le programme de test.



Exercice 2 : Première implantation

Une première version des classes Point (listing 3) et Segment (listing 4) a été écrite. Le programme de test correspondant au premier scénario de test (exercice 1) est donné au listing 1.

Listing 1 – La classe TestSegment1

```
/** Premier programme de Test de la classe Segment.  */
public class TestSegment1 {
    public static void main(String[] args) {
        Point p1 = new Point(0, 0);
        Point p2 = new Point(5, 0);
        Segment s = new Segment(p1, p2);

        System.out.print("p2 = "); p2.afficher(); System.out.println();
        System.out.print("s = "); s.afficher(); System.out.println();
        System.out.println("longueur de s = " + s.getLongueur());

        p2.translater(-2, 0);

        System.out.print("p2 = "); p2.afficher(); System.out.println();
        System.out.print("s = "); s.afficher(); System.out.println();
        System.out.println("longueur de s = " + s.getLongueur());
    }
}
```

2.1. Dessiner le diagramme de classe d'implantation correspondant à cette première solution.

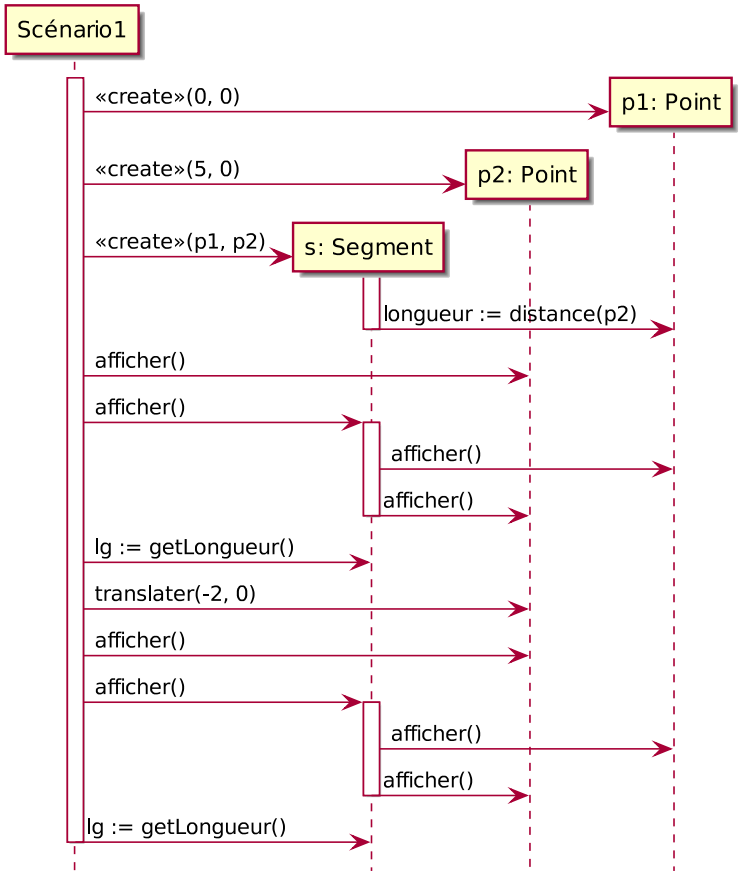
Solution : Essentiellement, l'attribut longueur dans Segment (demandé dans le cahier des charges) et la relation qui est en fait unidirectionnelle.

2.2. Compléter le diagramme de séquence dessiné à la question 1.2.

Solution : Maintenant que le code des classes a été écrit, nous pouvons préciser comment les objets réagissent aux envois de messages qu'ils reçoivent.

Notons que dans une approche classique, on commence par compléter le diagramme de séquence, en faisant des choix supplémentaires (en précisant par exemple la réaction des receveurs) et ensuite on passe à la programmation.

Voici le diagramme de séquence qui correspond à l'implémentation proposée.



2.3. Indiquer ce qu'affiche le programme de test. On pourra éventuellement dessiner l'état de la mémoire à la fin de l'exécution du programme.

Solution : Le programme affiche sur l'écran :

```
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
```

L'état de la mémoire est donné à la figure 2.

2.4. Commenter les résultats obtenus.

Solution : La longueur du segment nous donne 5 au lieu de 3. Elle est donc incorrecte (ou notre scénario de test est faux). Ici, c'est l'application qui est fausse car la longueur du segment `[(0.0,0.0)-(3.0,0.0)]` est bien 3 !

Lorsque le point `p2` est translaté, l'extrémité `extrémité2` du segment `s` change (relation d'agrégation) mais la longueur n'est pas mise à jour.

Le diagramme de séquence de la question 2.2 montre bien que lorsque le point `p2` est translaté, le segment n'est pas modifié et n'a pas connaissance de cette modification de son point extrémité.

Exercice 3 : Correction des classes Segment et Point

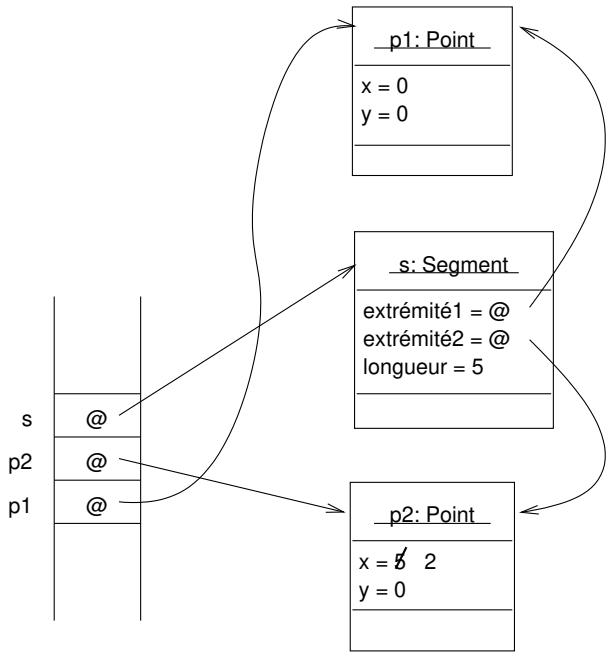


FIGURE 2 – L'état de la mémoire lors de l'exécution de TestSegment1

Nous allons maintenant corriger les classes Segment et Point pour qu'elles respectent le cahier des charges imposé. La solution proposée doit donc respecter les contraintes suivantes :

— la relation entre Segment et Point reste inchangée ;

— l'attribut `longueur` et la méthode `getLongueur()` de Segment restent inchangés.

3.1. Indiquer, sur le diagramme de séquence (question 2.2), les modifications à faire.
Solution : Voici les principales étapes qui ont conduit au diagramme de séquence présenté à la figure 3.

1. L'analyse de l'exécution nous a conduit à constater que la longueur du segment doit être mise à jour. Nous ajoutons donc une méthode `majLongueur()` sur Segment. Cette méthode doit être publique car elle sera utilisée par d'autres classes qui ne seront pas forcément dans le même paquetage.
2. Quand cette méthode doit être appelée ? Deux solutions sont envisageables.
   La première consiste à l'appeler dans TestSegment1, juste après avoir translaté le point p2 (ou en tout cas avant d'utiliser la longueur du segment s). Cette solution n'est pas la bonne pour deux raisons :
   — Il faut penser à appeler cette méthode... et donc on oubliera !
   — La **raison principale** est qu'en procédant ainsi, on modifie le scénario de validation. On ne respecte donc pas le cahier des charges donné !

   La deuxième solution, et donc la bonne solution, est donc de l'appeler dans le translater de Point.

3. Modifier le translater de Point.

   Il faut appeler la méthode majLongueur() de Segment. Il faut donc avoir accès au segment (en l'occurrence le segment s). On doit donc pouvoir naviguer la relation d'agrégation dans les deux sens.

   Attention : Mettre deux relations n'est pas correct. En effet, dans ce cas, on ne garantit pas la cohérence entre les extrémités des deux relations. Ainsi le point p1 pourrait être relié à un segment s2 dont il n'est pas extrémité et ne pas être relié au segment s dont il est extrémité.

   On peut écrire le code de translater.

4. Faut-il faire autre chose ? Il faut créer la liste des segments, donc compléter le constructeur.
5. Faut-il faire autre chose ? Il faut ajouter le segment s dans la liste des segments de p2 (et p1). Quand faut-il le faire ? Quand sait-on qu'un point est extrémité d'un segment ? Quand on crée un segment, donc dans le constructeur de Segment.

   On ajoute une méthode `inscrire(Segment)` sur Point et on complète le constructeur de Segment.

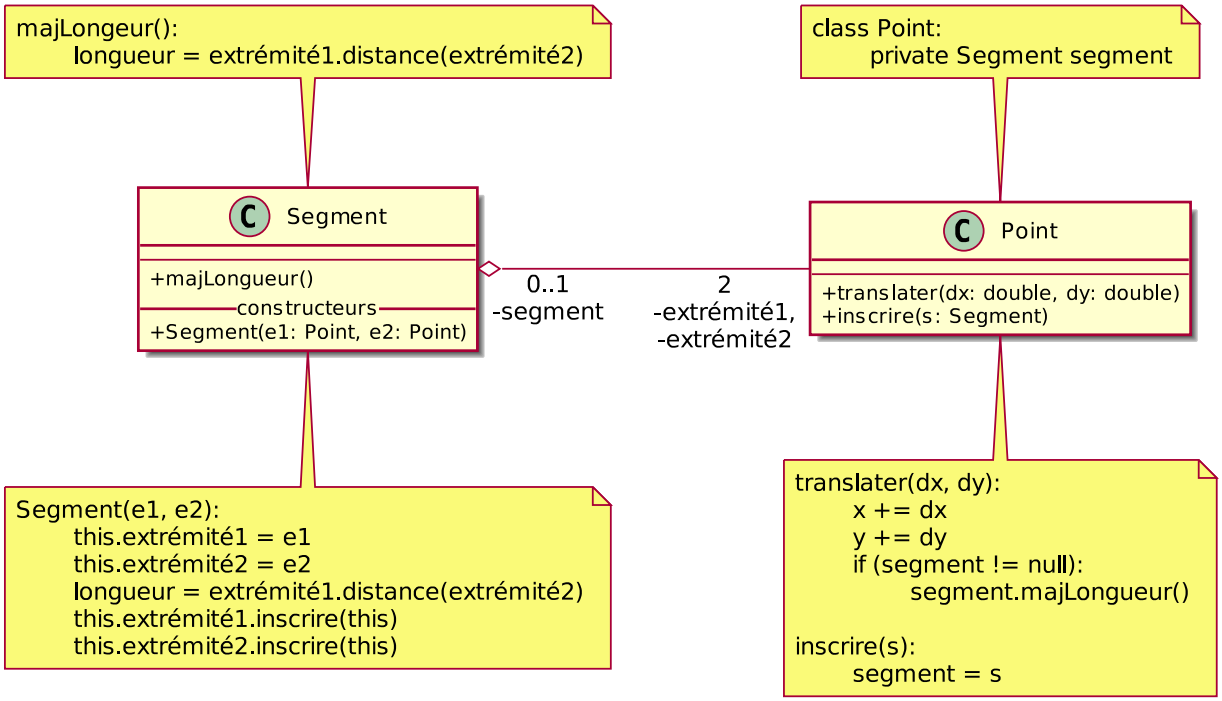
6. Fini ?

   On exécute le scénario et il fonctionne.

Défauts : Cette solution a des défauts. En particulier, le segment est obligé de publier une méthode `majLongueur`.

3.2. Compléter le diagramme de classe.

Solution :



3.3. Écrire le code des méthodes modifiées et des nouvelles méthodes.

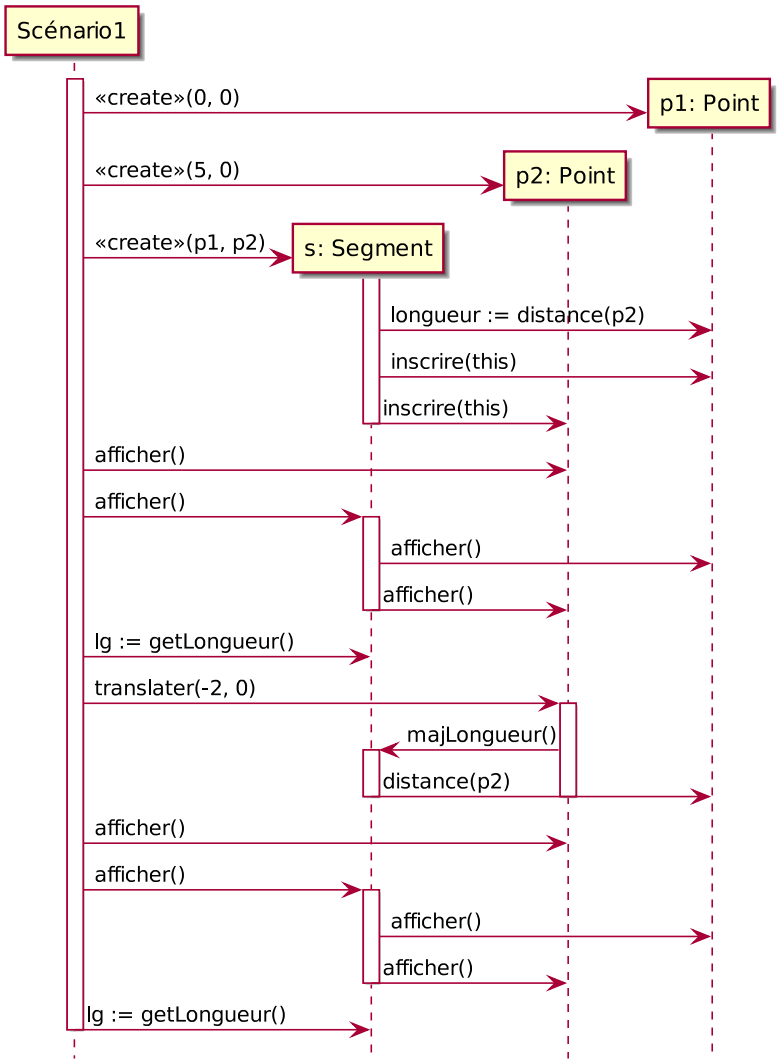


FIGURE 3 – Diagramme de séquence de la solution (version simplifiée)

Solution : Le code des méthodes est donné sous forme d'annotations sur le diagramme de classe. Il est écrit dans une forme à la Python car l'outil utilisé pour produire le diagramme de classe n'autorise pas les accolades fermantes dans les annotations.

Exercice 4 : Gestion de la mémoire

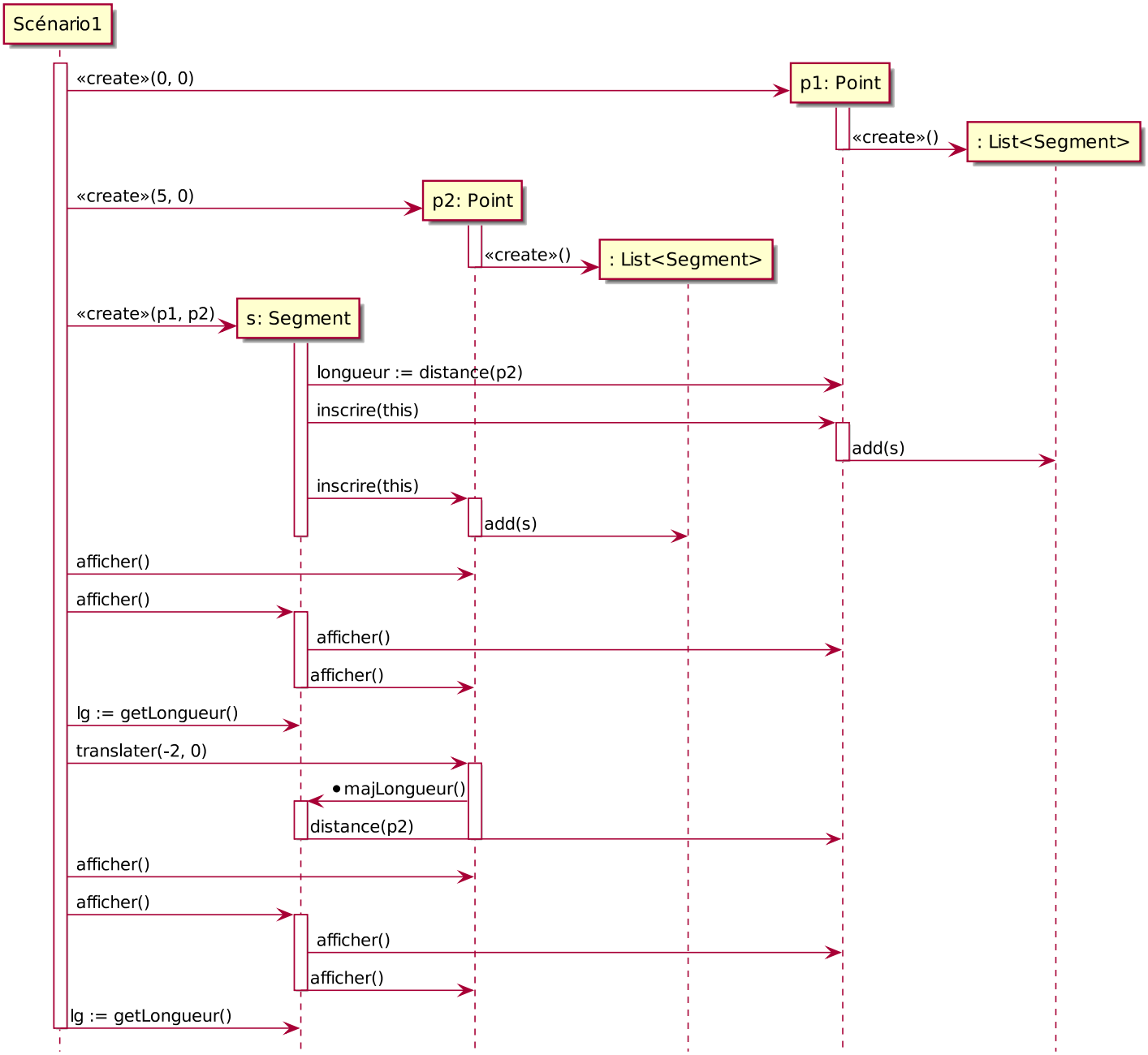
La relation d'agrégation spécifiée entre Segment et Point indique qu'un même point peut être l'extrémité de plusieurs segments.

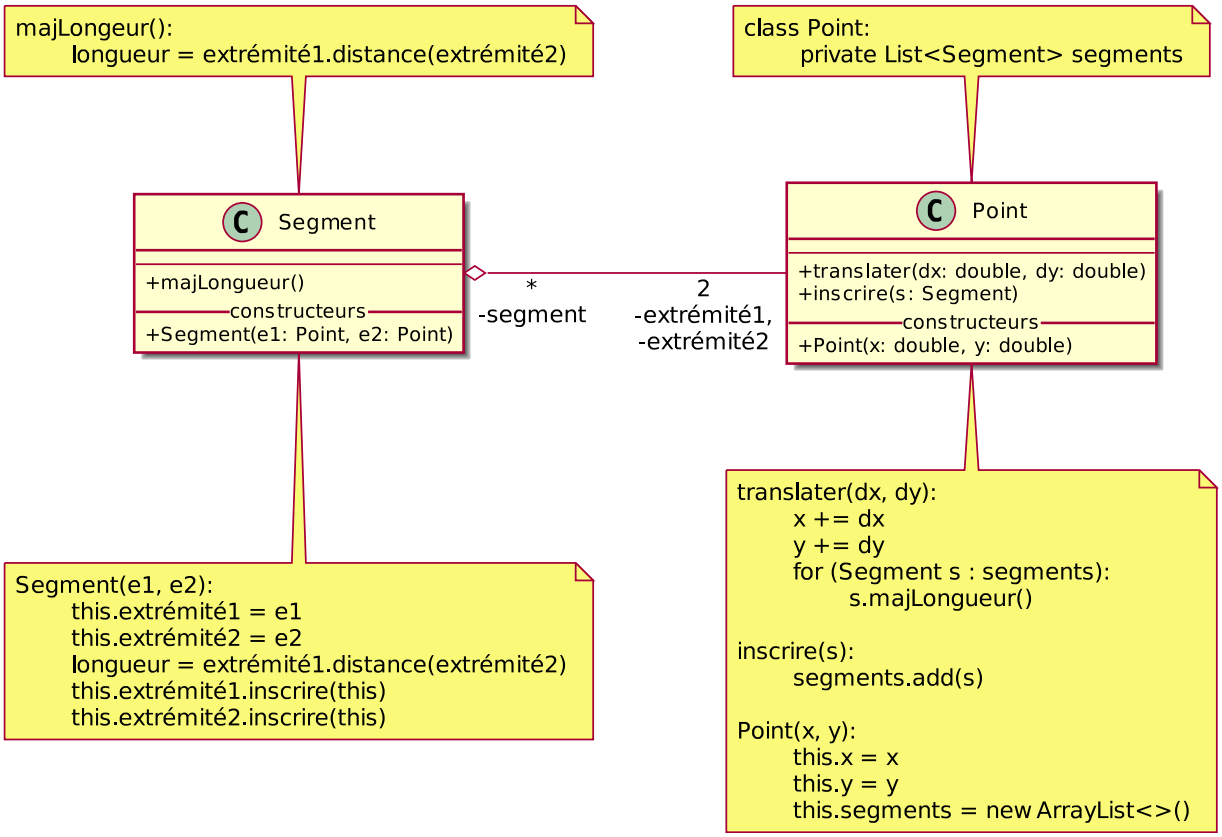
4.1. Cette propriété remet-elle en cause la solution proposée ? Donner les éventuelles modifications à apporter.

Solution : Un point peut effectivement être utilisé pour construire plusieurs segments. C'est d'ailleurs le principe de la relation d'agrégation. Ainsi, il n'y a pas, en général, un seul segment dont le point est extrémité mais un nombre quelconque. Il faut donc remplacer la multiplicité `0..1` par une multiplicité $*$. Cette multiplicité peut se traduire par un attribut de type `List<Segment>` qui permet au point de conserver la *liste* des segments dont il est extrémité pour pouvoir tous les mettre à jour lorsqu'il est modifié. Concrètement, on peut utiliser une ArrayList ou une LinkedList.

On préfèrera utiliser une List (et plus généralement une structure de données de l'API collections) plutôt qu'un tableau car elle offre des opérations de haut niveau (y compris le redimensionnement). De plus, depuis Java 5 et la généricité, on conserve le contrôle de type.

Remarque : On notera que le scénario envisagé ne couvre pas le cas d'un même point extrémité de plusieurs segments. Ceci montre qu'il est important d'identifier tous les aspects importants d'un cahier des charges et de décrire un ou plusieurs scénarios permettant de les valider. Dans le cas contraire, on risque de passer à côté d'aspects importants de l'application.


Scénario1
«create»(0, 0)
p1: Point
«create»()
: List
«create»(5, 0)
p2: Point
«create»()
: List
«create»(p1, p2)
s: Segment
longueur := distance(p2)
inscrire(this)
add(s)
inscrire(this)
add(s)
afficher()
afficher()
afficher()
afficher()
lg := getLongueur()
translater(-2, 0)
majLongueur()
distance(p2)
afficher()
afficher()
afficher()
afficher()
lg := getLongueur()



Problème : Cette solution fonctionne mais pose un problème.

La méthode majLongueur() est publique. Elle apparaîtra donc dans la documentation de Segment alors qu'elle n'a jamais à être appelée par un utilisateur de Segment (à l'exception de ses points extrémité).

4.2. On considère le programme du listing 2. Indiquer combien de segments sont récupérés par le ramasse-miettes. Expliquer pourquoi et proposer une amélioration à la solution proposée.

Listing 2 – La classe TestSegment3

```
/** Programme de Test de la classe Segment. */
public class TestSegment3 {
   public static void main(String[] args) {
      Point p1 = new Point(1, 2);

      for (int i = 0; i < 100; i++) {
         Segment s = new Segment(new Point(i, i), p1);
      }

      System.out.println("Translation␣!");
      p1.translater(10, 0);
      System.out.println("Fin␣!");
} }
```

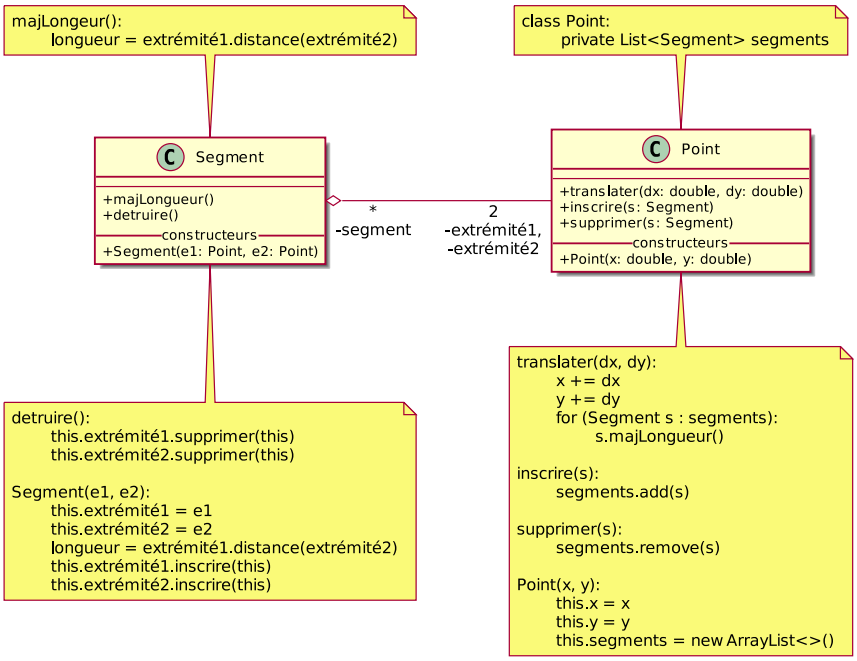
Solution : On constate qu'aucun segment n'est récupéré par le ramasse-miettes car le point p1 du programme principal garde un lien sur chaque segment (au travers de l'attribut `sesSegments`). On peut également le constater en affichant un message de trace dans `majLongueur`.

Il est donc souhaitable, lorsqu'on sait que l'on n'aura plus besoin d'un segment, de penser à

le désinscrire de ses points extrémités. Ceci évite de garder un lien qui déclenchera des calculs inutiles et empêchera le ramasse-miettes de récupérer la mémoire correspondant au segment. On appelle `détruire` cette méthode car elle correspond à la notion de destructeur. Le destructeur est le pendant des constructeurs. C'est une « méthode » appelée par le système juste avant que la mémoire utilisée par un système soit libérée. En Java, la méthode `finalize` d'`Object` est l'équivalent du destructeur. Cependant, contrairement a un véritable destructeur, on n'a pas de garantie qu'elle soit réellement appelée. De toute façon dans notre cas, elle ne pourra pas être appelée car tous les segments sont encore accessibles par le point p1, le ramasse-miettes ne récupèrera donc pas les segments et n'exécutera donc pas `finalize`.

L'inconvénient de la méthode `détruire` par rapport à un vrai destructeur est qu'il nous faut penser à l'appeler dès qu'un objet n'est « logiquement » plus accessible de notre application. Nous retrouvons les problèmes de gestion de la mémoire ou, plus généralement, à la gestion de ressources...

Définir la méthode `détruire` sur les segments, suppose que l'on définisse une méthode pour `annuler` une inscription sur les points.



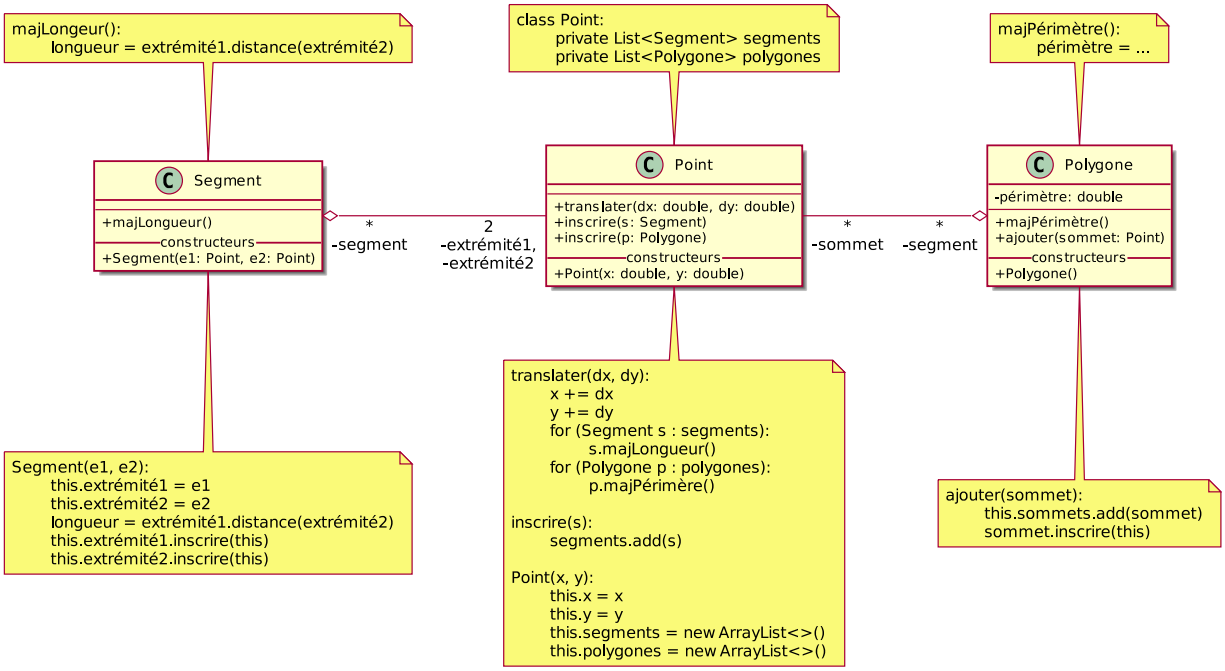
Dans la suite, nous ne reprendrons pas ces méthodes (`annuler` et `detruire`).

Exercice 5 : Polygone et Point

La solution fonctionne correctement pour les classes Point et Segment. Considérons maintenant une classe Polygone. Un polygone est défini comme un ensemble de sommets. Comme dans le cas du segment, on décide de stocker le périmètre du polygone. Il doit donc être recalculé dès qu'un de ses sommets est translaté.

Adapter la classe Point pour prendre en compte les segments mais aussi les polygones.

Solution : Une solution naïve consiste à appliquer la même démarche que pour la relation Segment-Point. On définit donc une classe Polygone pour gérer l'inscription des polygones auprès de leurs sommets, on ajoute une méthode majPérimètre sur la classe Polygone, etc.



Cette solution fonctionne mais elle a plusieurs défauts :

1. elle est lourde et fastidieuse car on refait pour Polygone ce qui a déjà été fait pour Segment (et il faudra le faire pour les autres classes qui dépendent des modifications des coordonnées d'un point).
2. il faut pouvoir modifier la classe Point pour réaliser ces modifications.

Il faut voir les classes Segment, Polygone, etc. comme autant d'applications différentes qui dépendent toutes de Point. Quand on ajoute une nouvelle application (par exemple Cercle dans l'exercice suivant), on ne devrait pas avoir à modifier Point (et Point ne devrait pas connaître toutes ces applications). Il faudrait pouvoir fermer la classe Point et ne avoir à la réouvrir pour les différentes applications qui l'utiliseront.

L'idée est alors de casser la dépendance entre Point et les classes Segment, Polygone, etc. qui l'utilisent. Aussi, il faut abstraire ce que Point a à connaître de ces classes. On peut le formaliser par une interface.

Qu'est-ce que Point a à connaître de ces classes ? En fait seulement les méthodes `majLongueur`, `majPérimètre`, etc. Même si elles ont des noms différents, elles représentent la même intention, on peut donc unifier [1] leurs noms en `maj` (mettre à jour).

Remarque : On peut arriver au même résultat en se posant les questions "comment faire pour n'avoir qu'une liste plutôt que plusieurs ?", "comment avoir une seule méthodes pour inscrire un segment, un polygone, etc. ?". Attention, cette approche peut conduire à une mauvaise abstraction car ici, on ne veut pas factoriser translater, afficher, etc mais bien abstraire ce Point utilise de ces classes. En particulier, on ne doit pas arriver à une généralisation qui serait ObjetGéométrique !

1. Si on ne veut pas changer leurs noms, on ajoute une nouvelle méthode `maj` qui appelle la méthode existante. Les méthodes `majLongueur`, `majPérimètre`, etc. peuvent alors être définies privées mais `maj` devra être déclarée publique.

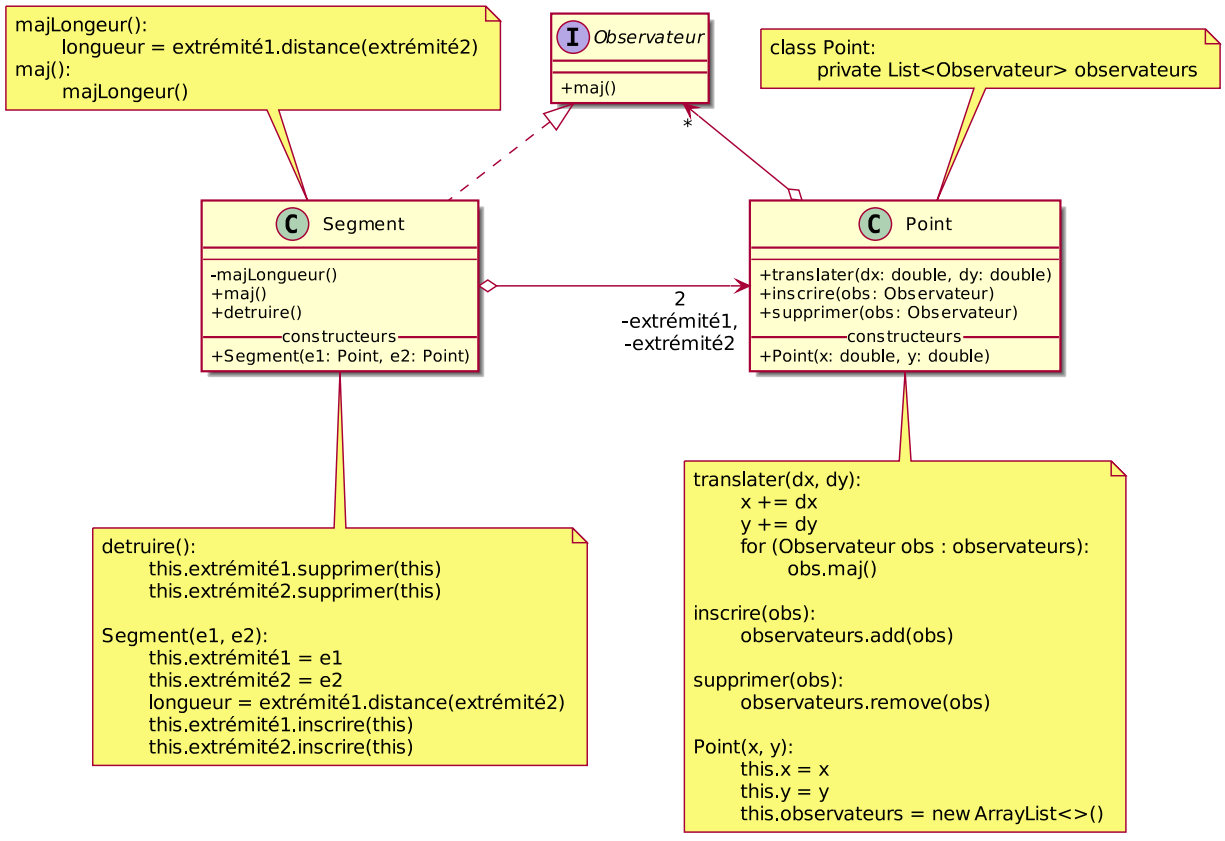
On appelle Observateur l'interface ainsi mise en évidence. Le segment, le polygone et, plus généralement, tout observateur souhaite être averti de toute translation sur un point. Ceci lui permet par exemple de mettre à jour sa longueur ou son périmètre ou de faire tout traitement adapté. On définit donc une méthode `maj` sur `Observateur`, cette méthode est retardée (son code dépend d'un observateur particulier). La méthode `maj()` généralise les méthodes `majLongueur` de `Segment` et `majPérimètre` de `Polygone`. Techniquement, il faut faire le lien entre l'une et l'autre.

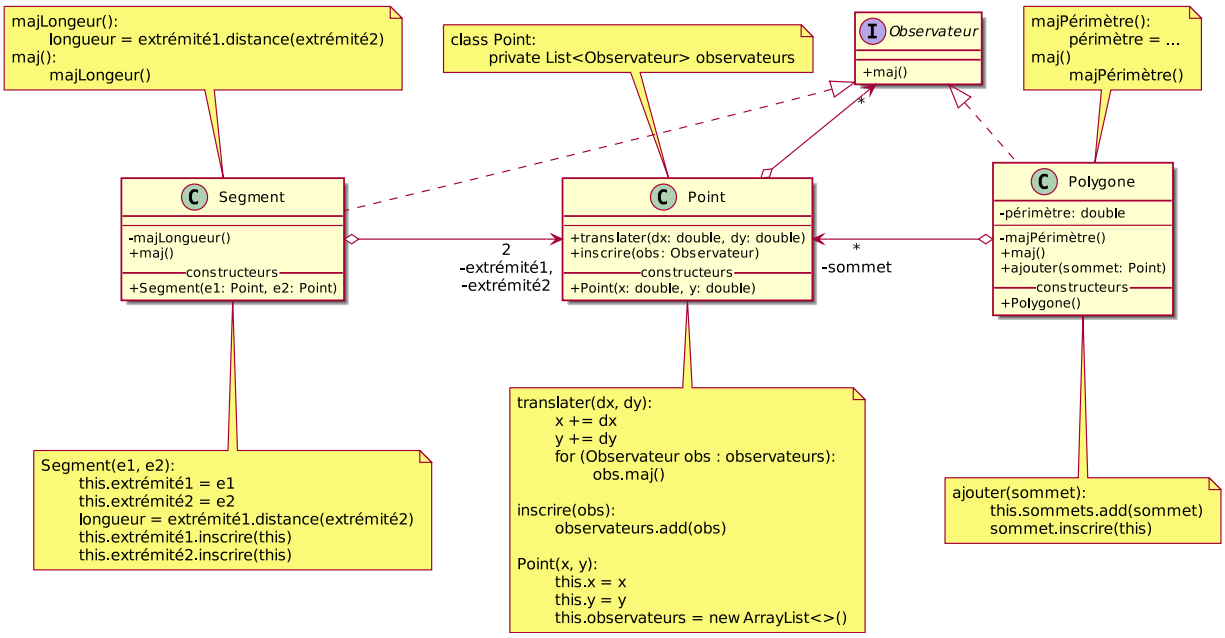
Le point appelle la méthode `maj()` de l'observateur pour lui signifier sa translation. Libre à l'observateur de faire ce qu'il juge utile.

La notion Observateur est donc définie en Java comme une interface (plutôt qu'une classe abstraite).

En résumé, le point doit donner la possibilité d'inscrire des observateurs et d'en supprimer (annuler). Les observateurs sont stockés dans l'attribut `observateurs` de type `List<Observateur>`. Lorsque le point est translaté, il appelle la méthode `maj` de chaque observateur inscrit. La méthode effectivement appelée (liaison dynamique) est spécifique de l'observateur. Par exemple, le segment recalcule sa longueur.

Dans ce cas, `Segment` hérite de `Observateur` et définit `maj` pour appeler `majLongueur`.





Exercice 6 : 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 :

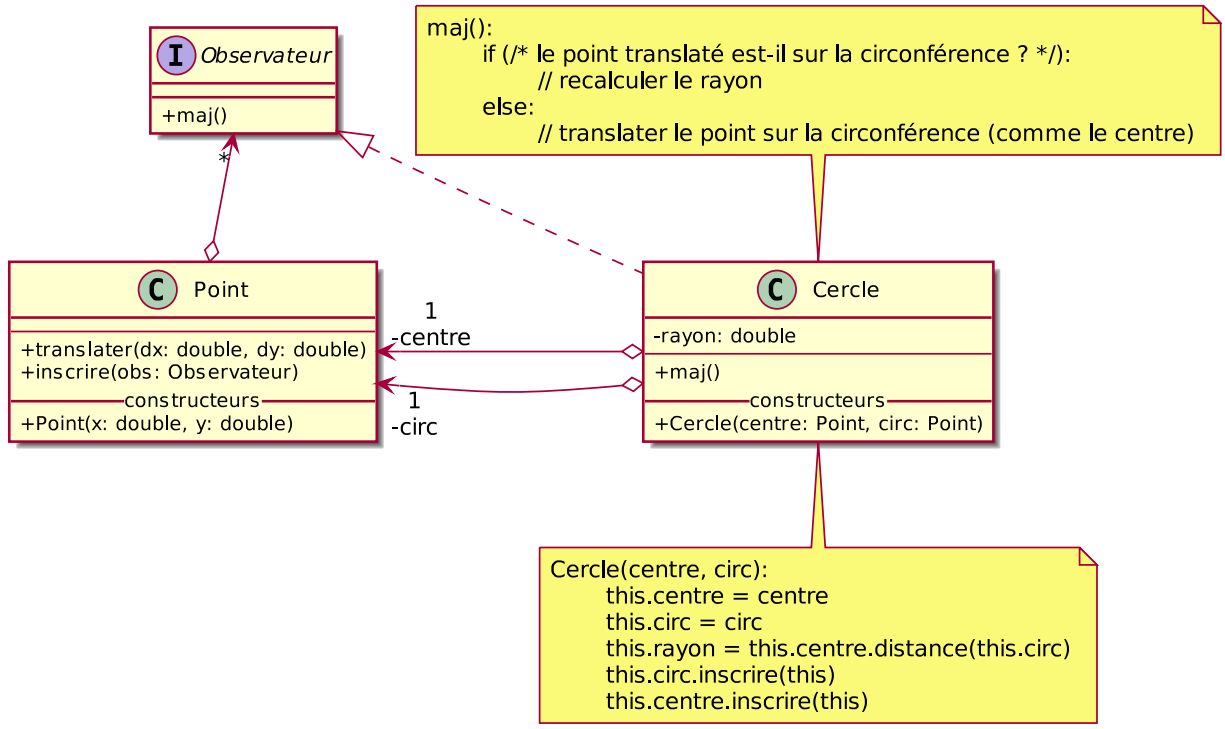
— une translation du centre du cercle provoque une translation du point de la circonférence ;

— 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.

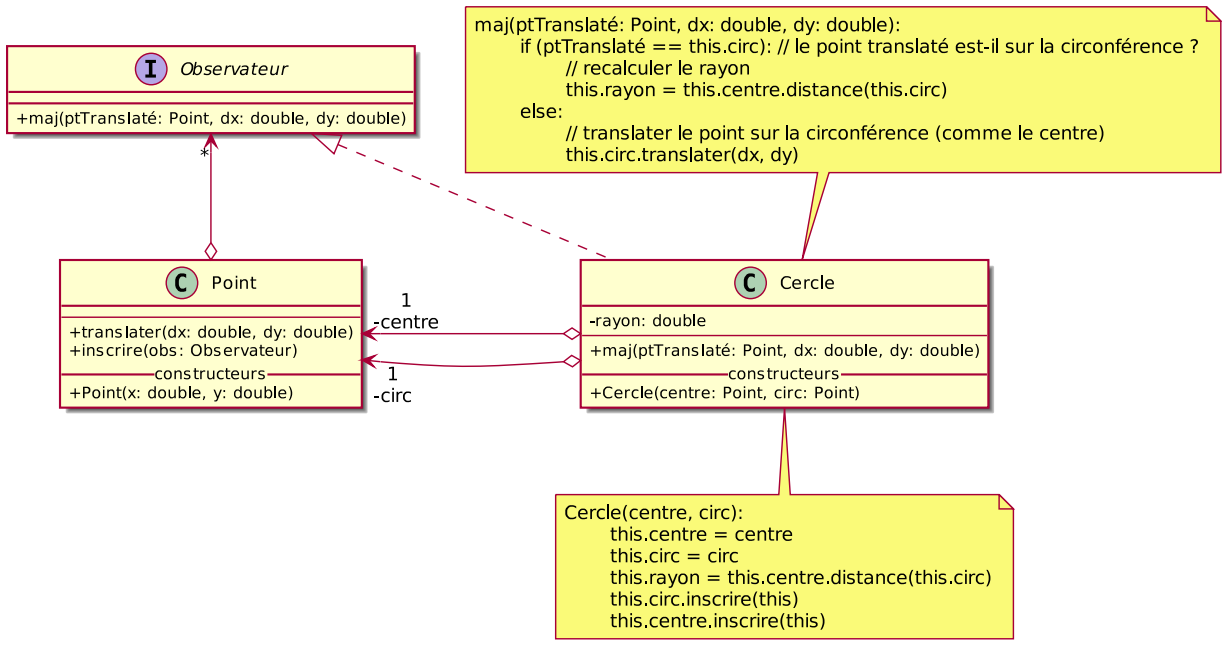
Solution :

On peut appliquer le même principe que pour Segment et Polygone. La classe Cercle réalise donc l'interface Observateur. Dans le constructeur de Cercle, on inscrit le cercle auprès des deux points : point du centre et point de la circonférence. Cependant, l'action à exécuter dépend du point qui a été translaté.



Le plus simple pour connaître le point translaté, le plus simple est de l'avoir en paramètre de maj. Ainsi, le point, quand il appelle la méthode maj des observateurs, s'ajoute comme premier paramètre.

De plus, quand on déplace le point du centre, il faut déplacer le point de la circonférence du même dx et dy qui étaient en paramètre du point du centre. On peut aussi les mettre en paramètre de maj.



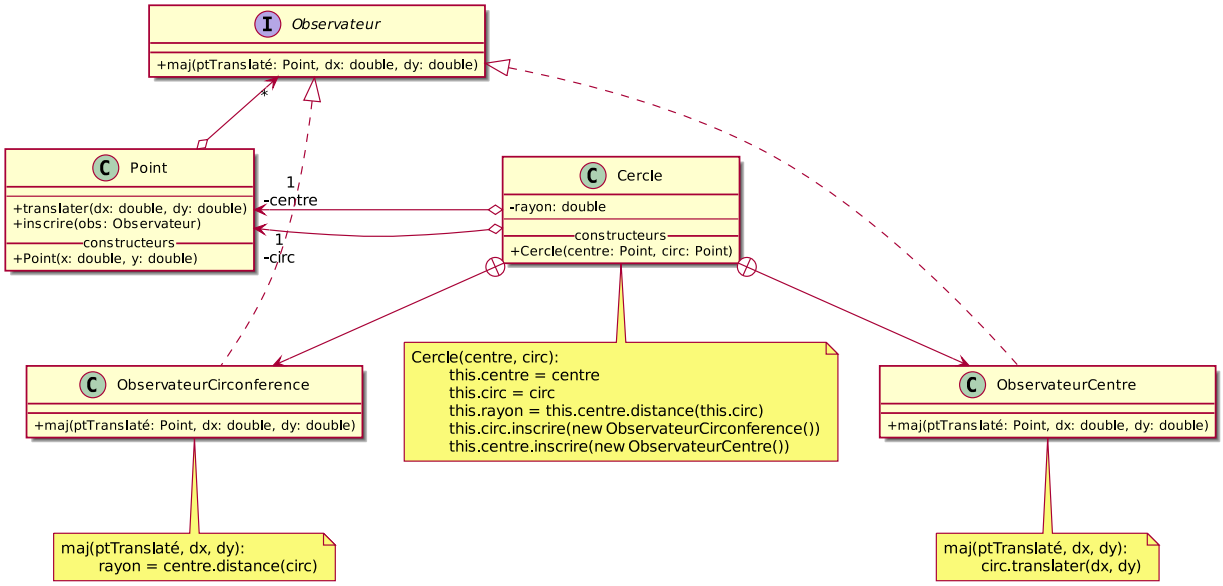
Cette solution fonctionne mais pose deux problèmes :

1. la méthode maj est toujours publique sur Cercle (Segment ou Polygone...),
2. on fait un test dans la méthode maj. Or dans une approche objet, on cherche souvent à limiter les tests.

Comment faire ?

Il ne faut plus que Cercle réalise l'interface Observateur si l'on ne veut pas que cette classe possède la méthode publique maj. Si ce n'est pas Cercle qui réalise cette interface, c'est donc une autre classe et même ici deux classes que l'on pourrait appeler ObservateurCentre et ObservateurCirconférence. Les instances de ces classes sont inscrites auprès du point concerné. C'est ce qui permet de supprimer le test. Chaque maj fait le traitement correspondant au point observé.

Le problème qui se pose alors est que dans les classes ObservateurCentre et ObservateurCirconférence nous n'avons pas accès aux attributs privés de Cercle pour faire les opérations attendues. La solution est de les définir comme classes internes d'instances !



La figure 4 présente le diagramme de classe correspondant à cette solution appliquée aux classes Segment et Cercle.

Remarque : Les paramètres dx et dy pourraient être regroupés dans une classe appelée Deplacement. Le principal intérêt est de limiter le nombre de paramètre de la méthode maj. Ceci permettrait d'avoir des informations supplémentaires liées au déplacement qui pourraient être ajoutée facilement.

On peut conserver le paramètre pointTranslé qui identifie le point qui a été translaté mais au aurait pu aussi le définir comme un attribut de Deplacement.

Conclusion : Est-ce que ceci était clairement spécifier dans le cahier des charges ? Est-ce contradictoire ?

Ce n'était pas spécifié explicitement dans le cahier des charges : la spécification n'était que implicite, en particulier au travers du scénario fourni.

Ce n'est pas contradictoire. On voit bien ici le principe d'un patron de conception qui va compléter un diagramme d'analyse avec de nouvelles classes qui apparaissent en phase de conception

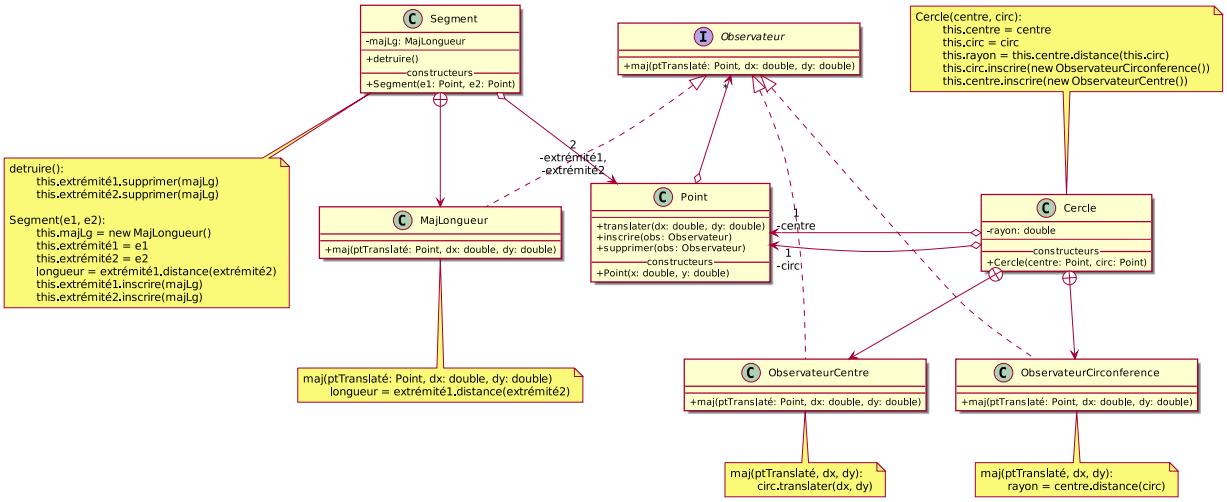


FIGURE 4 – Diagramme de classe avec observateur et classes internes

pour rendre la solution plus réutilisable, extensible... D'où le terme « conception » dans « patron de conception ».

Et si d'autres classes doivent être aussi observées ? Pour les nouvelles classes « observables », il faudra aussi gérer une liste d'observateurs, inscrire un observateur, annuler une inscription et avertir les observateurs. Cette dernière méthodes est une méthode protégée que nous avons pas explicitée dans la solution précédente (boucle for dans translater de Point).

Si on veut faciliter l'écriture de nouvelles classes « observables », il faudrait factoriser ces attributs et méthodes dans une classe Observable. Elle serait abstraite même si toutes les méthodes sont définies car, si la méthode avertir est définie, elle n'est appelée nulle part. La sous-classe de Observable devra appeler explicitement la méthode avertir quand il changer (seule modification à faire dans cette classe !).

On doit aussi rendre plus générale la méthode maj() de Observateur. Le premier paramètre n'est plus Point mais Observable. Le second ne peut plus être Deplacement car il dépend de l'objet observé et de ce qui est observé. On peut alors autorisé n'importe quel objet (Object) ou utilisé la généricité pour définir le type de l'information en deuxième paramètre de maj.

Les classes Observable et Observer sont présentes dans la bibliothèque Java dans le paquetage java.util. Depuis Java9, elles sont devenues obsolètes. Les JavaBeans (paquetage java.beans) fournissent un mécanisme d'observateur mieux conçu, plus riche et plus robuste.

Listing 3 – La classe Point

```
/** Définition d'un point. */
public class Point {
   private double x;  // abscisse
   private double y;  // ordonnée

   /** Construction d'un point à partir de son abscisse et de son ordonnée.
     * @param x abscisse
     * @param y ordonnée */
   public Point(double x, double y) {
      this.x = x;
      this.y = y;
   }

   /** Abscisse du point. */
   public double getX() {
      return this.x;
   }

   /** Ordonnée du point. */
   public double getY() {
      return this.y;
   }

   /** Distance par rapport à un autre point. */
   public double distance(Point autre) {
      return Math.sqrt(Math.pow(autre.x - this.x, 2)
               + Math.pow(autre.y - this.y, 2));
   }

  /** Translater le point.
    * @param dx déplacement suivant l'axe des X
    * @param dy déplacement suivant l'axe des Y */
   public void translater(double dx, double dy) {
      this.x += dx;
      this.y += dy;
   }

   /** Afficher le point. */
   public void afficher() {
      System.out.print("(" + this.x + "," + this.y + ")");
   }
}
```

Listing 4 – La classe Segment

```
/** Définition d'un segment. */
public class Segment {
   private Point extremite1;
   private Point extremite2;
   private double longueur;

   /** Construire un Segment à partir de ses deux points extremité.
     * @param ext1 le premier point extrémité
     * @param ext2 le deuxième point extrémité */
   public Segment(Point ext1, Point ext2) {
      this.extremite1 = ext1;
      this.extremite2 = ext2;
      this.longueur = extremite1.distance(extremite2);
   }

   /** Longueur du segment. */
```

```
    public double getLongueur() {
       return this.longueur;
    }
    /** Translater le segment.
      * @param dx déplacement suivant l'axe des X
      * @param dy déplacement suivant l'axe des Y */
    public void translater(double dx, double dy) {
       this.extremite1.translater(dx, dy);
       this.extremite2.translater(dx, dy);
    }
    /** Afficher le segment. */
    public void afficher() {
       System.out.print("[");
       this.extremite1.afficher();
       System.out.print(";");
       this.extremite2.afficher();
       System.out.print("]");
    }
}
```