Variations autour de la relation Segment-Point
Corrigé

Exercice 1 : Caractériser la relation entre Segment et Point
L'objectif de cet exercice est de comprendre les notions de composition et d'agrégation en s'appuyant sur les classes Point (listing 2) et Segment (listing 3).
1.1. Que donne l'exécution du programme du listing 1 ? En déduire la nature de la relation entre les classes Point et Segment (agrégation ou composition).

Listing 1 – La classe RelationSegmentPoint

```
/** Programme illustrant la relation entre les classes Segment et Point
  * (agrégation ou composition ?). */
public class RelationSegmentPoint {

   public static void main(String[] args) {
      // Créer trois points
      Point p1 = new Point(3, 2);
      Point p2 = new Point(6, 9);
      Point p3 = new Point(11, 4);

      // Créer deux segments à partir de ces trois points
      // (le point p2 est utilisé pour constuire s12 et s23)
      Segment s12 = new Segment(p1, p2);
      Segment s23 = new Segment(p2, p3);

      // Afficher les deux segments
      System.out.println("s12 = " + s12);
      System.out.println("s23 = " + s23);

      // Translater le point p2 qui a servi à initialiser les segments
      System.out.println("> p2.translater(5, -10);");
      p2.translater(5, -10);

      // Afficher les deux segments
      System.out.println("s12 = " + s12);
      System.out.println("s23 = " + s23);
}  }
```

Solution : Le résultat de l'exécution est le suivant :

```
s12 = [(3.0,2.0)-(6.0,9.0)]
s23 = [(6.0,9.0)-(11.0,4.0)]
> p2.translater(5, -10);
s12 = [(3.0,2.0)-(11.0,-1.0)]
s23 = [(11.0,-1.0)-(11.0,4.0)]
```

On constate que translater le point `p2` modifie les deux segments. Ceci s'explique car les trois points sont directement référencés par les deux segments et, `p2` est extrémité des deux segments.

On a donc une relation d'agrégation entre segment et point. Si on avait eu un relation de composition, les extrémités auraient été propres au segment et translater le point `p2` n'aurait pas eu d'incidence sur les deux semgents.

La relation d'agrégation est notée avec un losange vide du côté du Segment.

1.2. Comment faire pour obtenir l'autre relation ?

Solution :

Reformulation. On veut l'autre relation, c'est-à-dire la relation de composition. La relation de composition signifie que les extrémités du segments sont propres au segment. Ainsi, la translation du point `p2` ne devrait pas provoquer une modification des segments.

La relation de composition se représente par un losange plein au niveau UML, côté Segment.

Idée. Pour que les extrémités du segment lui soient propres, il est nécessaire de créer de nouveaux points. La question est alors de savoir où les construire.

Donner les coordonnées des points en paramètre du segment. Ainsi, c'est dans le segment que l'on est obligé de construire les points.

```
1 public Segment(double x1, double y1, double x2, double y2) {
2    this.extremite1 = new Point(x1, y1);
3    this.extremite2 = new Point(x2, y2);
4 }
```

Cette solution fonctionne mais elle a plusieurs défauts :

1. Il faut reprendre tous les codes qui construisent des segments car la signature du constructeur a changé.
2. On peut facilement mélanger les coordonnées des deux points.
3. Modifier le point, par exemple rajouter une coordonnée en Z, oblige à changer la classe Segment (paramètre supplémentaire) et tous les codes qui construisent des segments.
4. Et si on décide de changer la représentation des points en privilégiant les coordonnées cartésiennes ? Et si on voulait une interface Point et plusieurs réalisations de cette interface ?

Cette solution n'est donc pas acceptable !

Construire les points lors de la création du segment. Une première solution consiste à créer les nouveaux points dans le programme de test.

```
1 // Créer un segment s12 à partir des deux points p1 et p2
2 Segment s12 = new Segment(
3       new Point(p1.getX(), p1.getY(),
4       new Point(p2.getX(), p2.getY()));
```

Cette première solution n'est pas satisfaisante car c'est le programme qui crée les segments qui doit choisir entre agrégation et composition. On n'a donc pas réellement une relation de composition entre les classes Segment et Point.

Listing 2 – La classe Point

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

   /** Construire 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 x;
   }

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

   /** Changer l'abscisse du point
     * @param x la nouvelle abscisse */
   public void setX(double x) {
      this.x = x;
   }

   /** Changer l'ordonnée du point
     * @param y la nouvelle ordonnée */
   public void setY(double y) {
      this.y = y;
   }

   @Override public String toString() {
      return "(" + x + "," + y + ")";
   }

   /** Distance par rapport à un autre point */
   public double distance(Point autre) {
      double dx2 = Math.pow(autre.x - x, 2);
      double dy2 = Math.pow(autre.y - y, 2);
      return Math.sqrt(dx2 + dy2);
   }

  /** 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) {
      x += dx;
      y += dy;
}  }
```

Listing 3 – La classe Segment

```
/** Un segment caractérisé par ses deux points extrémités. */
public class Segment {
   private Point extremite1;
   private Point extremite2;

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

  /** 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) {
      extremite1.translater(dx, dy);
      extremite2.translater(dx, dy);
   }

   /** Longueur du segment */
   public double getLongueur() {
      return extremite1.distance(extremite2);
   }

   @Override public String toString() {
      return "[" + extremite1 + "-" + extremite2 + "]";
}  }
```

Une solution pour la composition. Si l'on veut réellement une relation de composition, il faut que **tous les segments** possèdent leurs propres points extrémités. La création des points doit donc être faite à chaque création d'un segment.

Où mettre cette création ? Dans le constructeur, bien sûr !

```
1 public Segment(Point ext1, Point ext2) {
2    extremite1 = new Point(ext1.getX(), ext1.getY());
3    extremite2 = new Point(ext2.getX(), ext2.getY());
4 }
```

L'exécution de `RelationSegmentPoint` donne alors :

```
1 s12 = [(3.0,2.0)-(6.0,9.0)]
2 s23 = [(6.0,9.0)-(11.0,4.0)]
3 > p2.translater(5, -10);
4 s12 = [(3.0,2.0)-(6.0,9.0)]
5 s23 = [(6.0,9.0)-(11.0,4.0)]
```

On constate bien que les deux segments ne changent pas quand le point `p2` est translaté.

Est-ce que cette solution est satisfaisante ?
Le reste du sujet va nous montrer que non !

Exercice 2 : Composition et polymorphisme

Dans l'exercice 1 nous avons proposé une manière de réaliser en Java la relation de composition. Nous allons en voir les limites et proposer une meilleure solution.

2.1. Peut-on créer un segment à partir de points et points nommés (listing 4) ? Pourquoi ?

Solution : Oui, on peut créer un segment à partir de points et points nommés car PointNomme est une spécialisation de Point. Le principe de substitution s'applique donc.

2.2. On modifie le programme du listing 1 pour créer les points p1 et p2 comme des points nommés. La création des trois points devient alors :

```
1 // Créer trois points
2 PointNomme p1 = new PointNomme("p1", 3, 2);
3 PointNomme p2 = new PointNomme("p2", 6, 9);
4 Point p3 = new Point(11, 4);
```

2.2.1. Indiquer ce qui se passe quand on exécute ce programme avec la classe Segment proposée à la question 1.2 (celle identifiée comme réalisant la relation de composition). Expliquer.

Solution : On constate que le nom des points n'apparaît pas.

```
1 s12 = [(3.0,2.0)-(6.0,9.0)]
2 s23 = [(6.0,9.0)-(11.0,4.0)]
3 > p2.translater(5, -10);
4 s12 = [(3.0,2.0)-(6.0,9.0)]
5 s23 = [(6.0,9.0)-(11.0,4.0)]
```

C'est parce que dans le constructeur de Segment, on construit des points à partir des paramètres. On ne tient pas compte du fait que le paramètre peut être une spécialisation des points comme ici un point nommé.

2.2.2. Indiquer ce qui se passe quand on exécute ce programme avec la classe Segment version initiale de la classe Segment (celle identifiée comme réalisant la relation d'agrégation).

Solution : Les noms des points nommés s'affichent grâce à la liaison dynamique mais, bien sûr, on n'a pas la composition !

```
1 s12 = [p1:(3.0,2.0)-p2:(6.0,9.0)]
2 s23 = [p2:(6.0,9.0)-(11.0,4.0)]
3 > p2.translater(5, -10);
4 s12 = [p1:(3.0,2.0)-p2:(11.0,-1.0)]
5 s23 = [p2:(11.0,-1.0)-(11.0,4.0)]
```

2.3. Voyons comment avoir à la fois la notion de composition (les extrémités du segment sont indépendantes des points ayant servi à initialiser le segment) et le polymorphisme (lorsqu'on affiche un segment construit à partir de points nommés, le nom des points nommés apparaît).

2.3.1. Proposer une solution s'appuyant sur la surcharge et la commenter.

Solution : S'appuyer sur la surcharge pourrait consister à définir autant de constructeurs que de combinaisons possibles pour les types des points permettant d'initialiser un segment. Ici on aura 4 constructeurs. Le point créé sera du même type que le paramètre formel correspondant

```
1 public Segment(Point ext1, Point ext2) {
2    extremite1 = new Point(ext1.getX(), ext1.getY());
3    extremite2 = new Point(ext2.getX(), ext2.getY());
4 }
5
6 public Segment(PointNomme ext1, Point ext2) {
7    extremite1 = new PointNomme(ext1.getNom(), ext1.getX(), ext1.getY());
```

```
8       extremite2 = new Point(ext2.getX(), ext2.getY());
9    }
10
11   public Segment(Point ext1, PointNomme ext2) {
12      extremite1 = new Point(ext1.getX(), ext1.getY());
13      extremite2 = new PointNomme(ext2.getNom(), ext2.getX(), ext2.getY());
14   }
15
16   public Segment(PointNomme ext1, PointNomme ext2) {
17      extremite1 = new PointNomme(ext1.getNom(), ext1.getX(), ext1.getY());
18      extremite2 = new PointNomme(ext2.getNom(), ext2.getX(), ext2.getY());
19   }
```

Cette solution a l'air de fonctionner dans certains cas comme par exemple :

```
1  Point p1 = new Point(1, 2);
2  PointNomme pn2 = new PointNomme("pn2", 3, 4);
3  Point p3 = new Point(3, 4);
4
5  Segment s1 = new Segment(p1, pn2);
6  Segment s2 = new Segment(pn2, p3);
```

Pour créer l'objet à attacher à `s1` deux constructeurs sont possibles : `Segment(Point, Point)` qui nécessite d'utiliser une fois le principe de substitution et `Segment(Point, PointNomme)` qui correspond exactement aux paramètres effectifs fournis. C'est donc le deuxième qui est choisi et on crée bien un point pour `extremite1` et un point nommé pour `extremite2`.
Attention : Ici, on a rien démontré ! Ce n'est pas parce que ça marche sur un exemple que c'est une solution qui marche tout le temps !

Cependant, cette solution a plusieurs défauts importants :

1. On constate tout de suite que sa mise en œuvre est très lourde ! Il suffit de compter combien de constructeurs il est nécessaire d'écrire si on définit également des points colorés, des points pondérés, etc.
2. Ajouter un nouveau type de point obligera à modifier la classe Segment et plus généralement toutes les classes qui utilisent des points : Cercle, Polygone, etc.
3. Elle est **FAUSSE** ! Elle ne marche pas ! Pour le montrer il suffit de donner un exemple où on n'obtient pas le résultat attendu. Quel est cet exemple alors ?

   Réfléchissons un peu. Comment marche la surcharge ? Le compilateur s'appuie sur le type des poignées (le type apparent) pour savoir quelle méthode ou constructeur appeler. Pour le tromper, il suffit tout simplement de lui cacher le type réel de l'objet derrière une poignée d'un type plus général, ici créer un segment à partir d'un point nommé accessible depuis une poignée de type Point. Voici un exemple :

   ```
   1  Point q = new PointNomme("N", 1, 2); // un point nommé est attaché à q de type Po
   2  Segment s = new Segment(q, p3); // quel est constructeur appelé ?
   ```

   Le constructeur appelé est `Segment(Point, Point)`. C'est le seul possible puisque le type apparent des deux paramètres est Point (les autres constructeurs attendent au moins un PointNomme) Du coup, le constructeur créera deux points et pas un point nommé pour la première extrémité ! CQFD.

La bonne nouvelle, c'est donc qu'on n'aura pas à multiplier les constructeurs dans Segment et on peut se concentrer sur celui qui prend en paramètre deux points, sachant que derrière les points il pourrait y avoir des points nommés.

2.3.2. Proposer une autre solution.

Solution : Commençons par une **mauvaise solution** : créer systématiquement un point nommé.

On ne doit pas oublier qu'un segment peut quand même être créé avec de simples points. Que se passe-t-il alors dans ce cas ?

Il faudrait définir un nouveau constructeur qui initialise à partir d'un point mais alors comment initialiser le nom ? Avec une valeur par défaut donc non significative. On pourrait décider qu'un nom `null` signifie pas de nom mais on perd alors l'invariant qu'on avait sur la classe `PointNomme` : un point nommé a un nom. On devra modifier la classe `PointNommé` pour tester si le nom est défini ou nom. Cette solution n'est donc pas souhaitable.

De plus, comment, avec cette solution, prendre en compte les points colorés, pondérés, etc. ?

Démarche. Globalement, il y a deux façons d'aborder ce problème :

1. On exprime l'objectif que l'on vise, par exemple en écrivant le premier niveau de raffinage du constructeur de Segment. Si on arrive bien à exprimer cet objectif et non comment il pourrait être atteint, on aura la solution.
2. On écrit une solution puis on prend du recul pour l'analyser, l'améliorer voire l'abandonner et en chercher une autre. C'est ce qu'on a fait dans la question précédente.

Vers une solution. La question précédente a montré qu'il n'est pas possible de s'appuyer sur la surcharge pour résoudre notre problème. Il faut tenir compte du type réel des points passés en paramètre du constructeur.

Une première solution consiste alors à se servir de l'opérateur **`instanceof`**.

Le constructeur de Segment devient alors :

```
1  public Segment(Point ext1, Point ext2) {
2     // initialiser extremite1 (commentaire incomplet !)
3     if (ext1 instanceof PointNomme) {
4        PointNomme pn = (PointNomme) ext1;
5        extremite1 = new PointNomme(pn.getNom(), ext1.getX(), ext1.getY());
6     } else { // ext1 est un Point
7        extremite1 = new Point(ext1.getX(), ext1.getY());
8     }
9
10    // initialiser extremite2 (commentaire incomplet !)
11    if (ext2 instanceof PointNomme) {
12       PointNomme pn = (PointNomme) ext2;
13       extremite2 = new PointNomme(pn.getNom(), ext2.getX(), ext2.getY());
14    } else { // ext2 est un Point
15       extremite2 = new Point(ext2.getX(), ext2.getY());
16 }  }
```

Il faut faire attention :

— à l'ordre dans lequel les **`instanceof`** sont testés. En effet, il faut toujours commencer par le type le plus précis, c'est-à-dire le plus bas dans la relation d'héritage.

— à la nécessité de faire un transtypage pour pouvoir accéder à toutes les caractéristiques du point (cas du nom du point nommé).

Analysons cette solution. Est-ce qu'elle fonctionne ? OUI.

Est-ce qu'elle est simple à mettre en œuvre ? Elle n'est pas très compliquée. Il suffit de faire attention à l'ordre dans lequel les `instances` sont traitées et ne pas oublier de tester un cas. Comme le dit si bien la loi de Murphy, s'il y a risque de se planter alors... on se plantera !

Est-ce qu'elle prend en compte facilement les évolutions ? Les évolutions peuvent consister d'une part à définir de nouveaux types de points (points colorés, pondérés...) et d'autre part à avoir d'autres classes qui nécessitent une relation de composition avec les points (le cercle et son centre, les polygones et leurs sommets...).

Il faut alors ajouter plein de tests sur les nouveaux types (`PointColoré`, `PointPondéré`) à **tous** les endroits où on souhaite la composition (`Segment`, `Cercle`, `Polygone`...). On constate que l'on fait le même traitement dans plein d'endroits. Il doit donc y avoir une meilleure solution !

Conclusion. Cette solution fonctionne mais elle n'est pas élégante et ne permet pas d'étendre notre application facilement.

Essayons donc de trouver une autre solution !

Comment faire ? On peut se poser la question de ce que l'on fait dans chacune des instructions qui suit une conditionnelle. Et ce qu'on fait c'est... initialiser une extrémité du segment avec une copie du point passé en paramètre. La copie dépend du type réel, donc de la classe dont le point en paramètre est instance. La formulation « avec une copie du point » soit faire penser à définir une méthode « copie » sur la classe `Point`. Ceci doit devenir un réflexe. Il faut donc penser à faire une méthode polymorphe [1]. C'est généralement ce que l'on fera pour éviter un traitement par cas du le type d'un objet.

On définit une méthode copie sur la classe Point.

```
1  public class Point {
2     /** Obtenir une copie de ce point.
3      * @return une copie de ce point */
4     public Point copie() {
5        return new Point(x, y);
6     }
```

Dans les classes dérivées, par exemple PointNomme, on redéfinit cette méthode.

```
1  public class PointNomme extends Point {
2     /** Obtenir une copie de ce point nommé.
3      * @return une copie de ce point nommé */
4     @Override public PointNomme copie() {
5        return new PointNomme(nom, getX(), getY());
6     }
```

Remarque : En Java, il est possible depuis Java5 de remplacer le type de retour d'une méthode par un sous-type lors de sa redéfinition dans la sous-classe. Ceci est cohérent avec le principe

1. Le terme méthode polymorphe est certainement abusif puisqu'en Java toutes les méthodes sont potentiellement polymorphes si elles ne sont pas déclarées **`final`**. Ce que je veux dire c'est que cette méthode sera redéfinie dans (presque) toutes les classes dérivées.

de substitution. L'appelant s'attend à avoir un objet du type déclaré comme retour de la méthode dans la superclasse, il aura en fait un objet d'un type plus précis.

Enfin, le constructeur de Segment s'écrit alors tout simplement :

```
1 public Segment(Point ext1, Point ext2) {
2    extremite1 = ext1.copie();
3    extremite2 = ext2.copie();
4 }
```

Remarque : Il serait aussi possible de définir une méthode pour copier des segments :

```
1 public class Segment {
2    /** Obtenir une copie de ce segment.
3     * @return une copie de ce segment */
4    public Segment copie() {
5       return new Segment(extremite1.copie(), extremite2.copie());
6    }
```

Le diagramme de classe de la figure 1 décrit cette architecture.

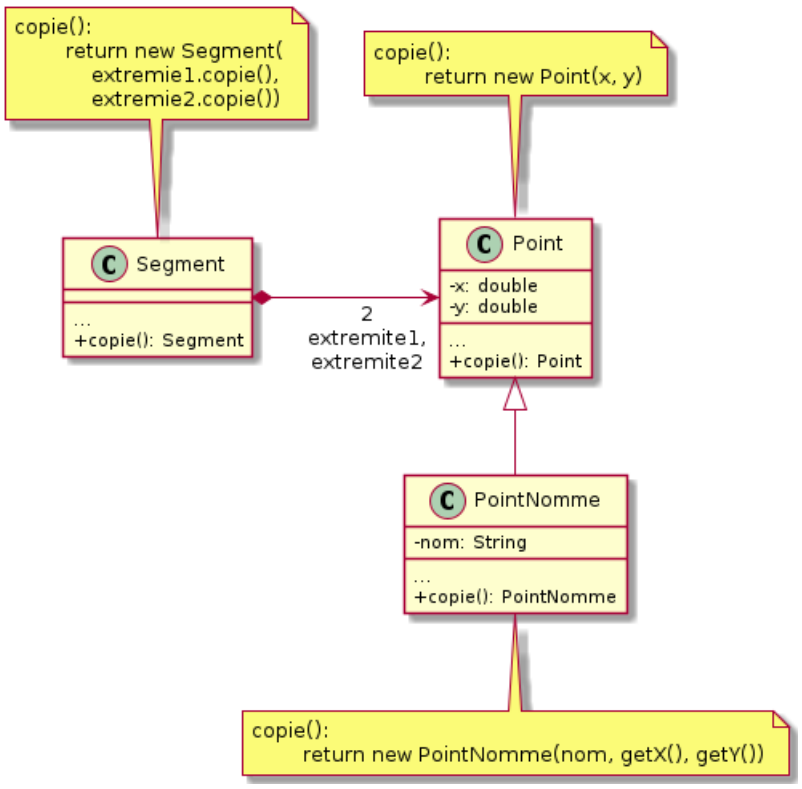

FIGURE 1 – Diagramme de classe faisant apparaître les méthodes `copie`

Et avec l'API Java. La class `Object` de l'API Java propose une méthode `clone()` qui a pour objectif de réaliser une copie de l'objet.

Cette méthode est déclarée **protected** car par défaut, elle se contente de faire une copie de la zone mémoire occupée par l'objet ce qui peut ne pas être suffisant pour réaliser une vraie copie. Ainsi, une classe ne fournit pas cette méthode `clone` à ses utilisateurs.

Si une classe veut fournir cette méthode `clone` à ses utilisateurs, elle doit augmenter son droit d'accès. Pour cela, il faut la redéfinir. Dans le code, on peut appeler **super**`.clone()`. Cependant, cette méthode lèvera une exception `CloneNotSupportedException` si la classe ne réalise pas l'interface `Cloneable`. Cette interface de marquage (elle ne définit aucune méthode), permet de spécifier que faire une copie superficielle de l'objet est suffisant pour le cloner. Plus généralement, ceci signifie que les objets de cette classe sont clonables... et les objets de ses sous-classes aussi ! Cette propriété ne pourra plus être perdue !

Voici la définition de `clone` dans `Point` :

```
1  public class Point implements Cloneable {
2     /** Obtenir une copie de ce point.
3      * @return une copie de ce point */
4     @Override public Point clone() { // droit d'accès augmenté
5        try {
6           return (Point) super.clone();
7        } catch (CloneNotSupportedException e) {
8           // Ne devrait pas se produire !
9           throw new InternalError(e);
10    }  }
```

— La méthode a été déclarée publique (**public**) alors qu'elle été **protected** dans `Object`.

— Elle retourne un `Point` (et non un `Object`).

— On doit transtyper en `Point` le retour de de **super**`.clone()` qui est du type `Object`.

— Si on a bien réalisé l'interface `Cloneable`, l'exception `CloneNotSupportedException` ne peut pas se produire. Comme elle est vérifiée, nous la récupérons et levons une nouvelle exception `InternalError` qui normalement ne devrait pas se produire.

Notons que si l'on redéfinit la méthode `clone` de `Point` correctement (comme ci-dessus), on pourrait ne pas redéfinir celle de PointNomme (sauf à vouloir préciser son type de retour en remplaçant `Point` par `PointNomme`). C'est une différence avec la méthode `copie` qui doit absolument être redéfinie dans `PointNomme`.

```
1  public class PointNomme extends Point {
2     /** Obtenir une copie de ce point.
3      * @return une copie de ce point */
4     @Override public PointNomme clone() {
5        return (PointNomme) super.clone();
6     }
```

Remarque : On pourrait aussi fournir la méthode `clone` sur les segments. Cependant, dans ce cas la copie de premier niveau n'est pas suffisante : les deux segments partageraient les mêmes points extrémités. Il faut donc en plus faire une copie explicite des extrémités du segment. Voici le code correspondant :

```
1  public class Segment implements Cloneable {
2     /** Obtenir une copie de ce segment.
3      * @return une copie de ce segment */
4     @Override public Segment clone() { // droit d'accès augmenté
```

```
5         try {
6            Segment nouveau = (Segment) super.clone(); // copie au premier niveau
7            nouveau.extremite1 = extremite1.clone();
8            nouveau.extremite2 = extremite2.clone();
9            return nouveau;
10        } catch (CloneNotSupportedException e) {
11           // Ne devrait pas se produire !
12           throw new InternalError(e);
13    }  }
```

Le diagramme de classe de la figure 2 donne l'architecture de cette solution. Elle est détaillée ci-après.

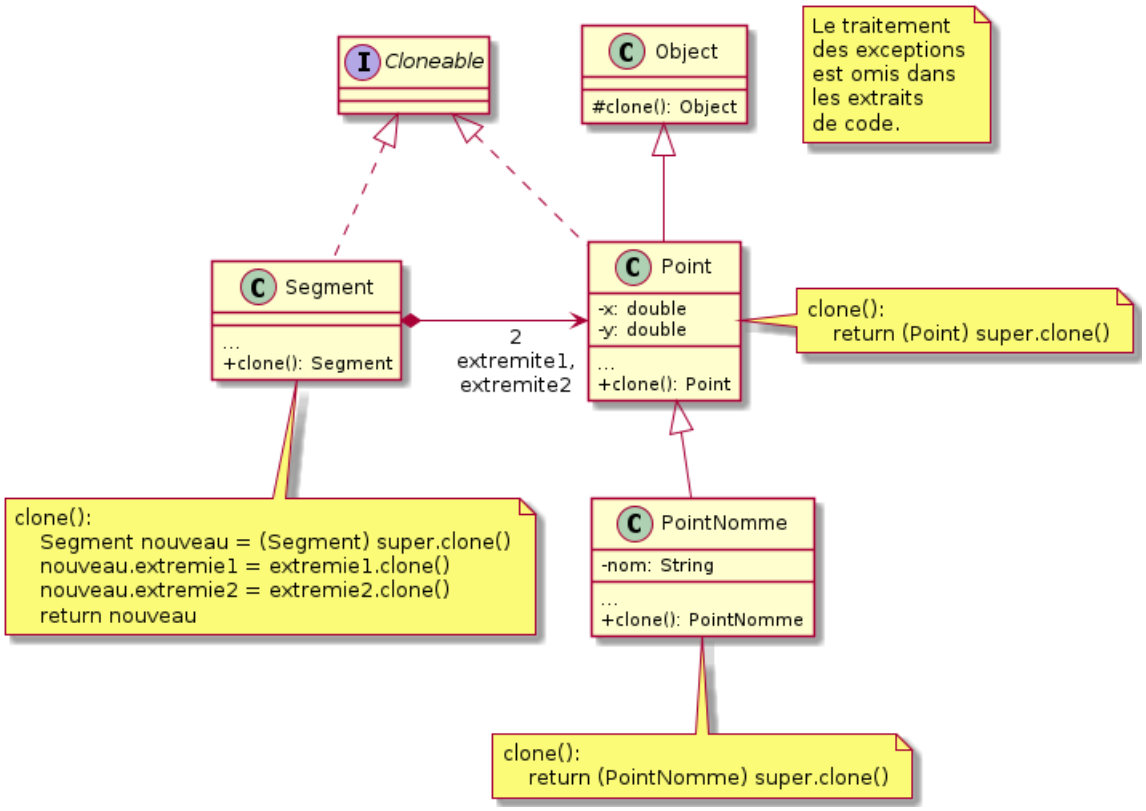

FIGURE 2 – Diagramme de classe faisant apparaître les méthodes `clone`

Comment réaliser une relation de composition en Java ? Plusieurs stratégies pour avoir une relation de composition entre C (client) et F (fournisseur) :

1. **Type élémentaires.** Si F correspond à un type élémentaire (int, boolean, double, etc.), on représente normalement cette relation par un attribut car elle correspond implicitement à de la composition car les types élémentaires sont traités par valeur.
2. **Classe immuable.** Si F est une classe dont les objets sont immuables (ils ne peuvent pas être modifiés après leur création, par exemple String, Integer, Double, etc.), il y a implici-

tement une relation de composition. Les objets ne pouvant pas être modifiés, ils peuvent être partagés et chacun a l'impression d'avoir sa propre version de l'objet.

3. **Classe modifiable.** Si F est une classe dont les objets sont modifiables, on peut faire une copie, par exemple en définissant correctement et utilisant la méthode `clone` de `Object`.
4. **Compteur de référence.** L'inconvénient de l'approche précédente avec copie systématique est que l'on peut créer des copies inutiles. Par exemple, dans le listing 1 pourquoi copier les points `p1` et `p3` qui ne sont pas modifiés au cours du programme ?

   Une solution consiste à ajouter un proxy qui va compter le nombre de référence sur l'objet et qui fera des copies de l'objet que quand c'est nécessaire : si l'objet est partagé et que l'un de ses possesseurs appelle une de ses méthodes pour le modifier. On appelle `copy-on-write` ce proxy. Voir l'examen du 17 juin 2008 pour illustration de ce principe.

Listing 4 – La classe PointNomme

```
1  /** Un point nommé est un point avec un nom. */
2  public class PointNomme extends Point {
3     private String nom;
4
5     /** Construire un point nommé. */
6     public PointNomme(String nom, double x, double y) {
7        super(x, y);
8        this.nom = nom;
9     }
10
11    /** Nom du point nommé */
12    public String getNom() {
13       return nom;
14    }
15
16    /** Changer le nom du point nommé
17     * @param nom le nouveau nom */
18    public void setNom(String nom) {
19       this.nom = nom;
20    }
21
22    @Override public String toString() {
23       return nom + ":" + super.toString();
24 }  }
```