IN201 : 2 – Tests unitaires

Solution

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

Résumé

Le but de ce TP est de construire des tests unitaires avec JUnit et de commencer à utiliser Subversion.

1 Contenu

Ce corrigé succinct contient les réponses au TP numéro 2. Vous y trouverez en particulier les diagrammes UML de la partie conception. Le corrigé de la partie implantation en Java est disponible sur http://www.tofgarion.net/lectures/IN201.

2 Présentation informelle

Un segment au sens mathématique peut être défini par ses deux extrémités. Comme toute figure mathématique du plan, on peut le translater. On peut également calculer sa longueur, car ses deux extrémités définissent ses « limites ». Comme pour toute représentation d'un concept, il existe des moyens simples de le représenter sous forme textuelle, par exemple en utilisant ses deux extrémités.

3 Conception de la classe `Segment`

Le sujet vous guidait bien pour concevoir la classe `Segment`. Nous allons revenir sur les questions.

1. quels sont les attributs modélisant l'état d'un objet de type `Segment` ? Quelle est leur visibilité ?

   Solution :

   On pouvait facilement deviner qu'il suffit d'avoir deux point pour déterminer un segment. On va donc choisir de représenter l'état d'un objet de type `Segment` par deux attributs de type `Point`. On manipule ces attributs comme des attributs « normaux ».

   Pour respecter le principe d'encapsulation, on déclare ces deux attributs comme étant privés.

   On aurait pu se demander s'il fallait explicitement stocker la longueur du segment comme un attribut. On peut déduire la longueur des deux autres attributs, donc il n'est pas nécessaire de la stocker explicitement. On parle alors d'attribut *dérivé*. On remarquera dans la question 3 que l'on écrit un accesseur à la longueur du segment. Un utilisateur extérieur ne saura donc pas si la longueur est stockée comme attribut ou calculée : le concepteur de la classe masque l'implantation de la longueur.

2. doit-on écrire un constructeur par défaut pour la classe `Segment` ? Si non, quels sont les paramètres que doit prendre ce constructeur ?

   Solution :

   Là encore, il s'agit d'appliquer un principe : il n'est pas bon d'écrire un constructeur par défaut. L'utilisateur doit toujours penser à l'initialisation de l'objet qu'il est en train de créer.

   Pour définir un segment, on a besoin de deux points. Il est donc logique de prendre comme paramètres du constructeur deux poignées de type `Point`. La signature du constructeur (en UML !) sera donc **`public`** `Segment(p1: Point, p2: Point)`

3. les services rendus par un objet de type `Segment` vont être modélisés par des opérations (ou méthodes) publiques. On souhaite pouvoir effectuer les opérations suivantes sur un segment :
   - le translater ;
   - l'afficher ;
   - renvoyer une chaîne de caractères le représentant ;
   - renvoyer sa longueur.

   Écrire le diagramme UML de la classe `Segment`.

   Solution :

   Là encore, les méthodes demandées apparaissent de façon claire :

- deux accesseurs aux deux attributs privés de `Segment` : `getExtremite1()` et `getExtremite2()` qui renvoient un `Point` ;
- une méthode `getLongueur()` qui renvoie un **double** qui correspond à la longueur (on peut la voir comme un accesseur à la longueur, même si celle-ci n'est pas explicitement stockée comme attribut) ;
- une méthode `translater(x: double, y: double)` qui ne renvoie rien ;
- une méthode `afficher()` qui ne renvoie rien ;
- une méthode `toString()` qui renvoie un objet de type `String`.

Il n'est pas ici nécessaire (à mon avis) d'écrire des modifieurs pour les extrémités du segment.

Toutes ces méthodes sont évidemment publiques : elles représentent des services et doivent donc être accessibles.

Le diagramme UML de la classe `Segment` est présenté sur la figure 1.

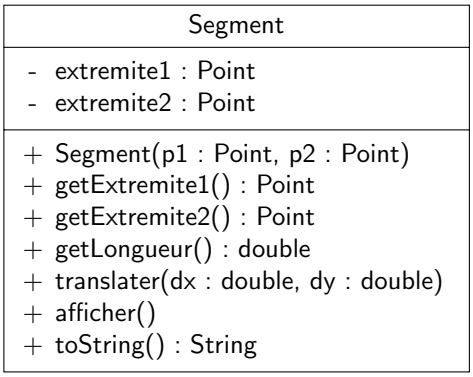

FIGURE 1 – Diagramme UML de la classe `Segment`

4 Implantation de la classe `Segment`

4.1 Travail d'implantation

Solution : Les sources des classes `Point` (nouvelle version), `Segment`, `SegmentTest` (classe de test JUnit) et `TestSegment` (classe applicative) sont disponibles sur le site. On peut faire quelques remarques :

- la classe `SegmentTest` a effectivement été écrite en même temps que la classe `Segment`. Les méthodes ont été testées au fur et à mesure. On remarquera que l'on n'a pas testé les accesseurs et modifieurs, car ils sont très simples ici. Par contre on a testé la méthode `getLongueur` car il n'y a pas d'attribut qui corresponde ;
- le constructeur de `Segment` lie les objets de type `Point` passés en paramètre à ses attributs (car on travaille avec des poignées) :

```
    public Segment(Point p1, Point p2) {
        this.extremite1 = p1;
        this.extremite2 = p2;
    }
```

Un programme écrit de la façon suivante :

```
Point p = new Point(1,2);
Point q = new Point(4,5.5);

Segment s = new Segment(p,q);
```

se « traduira » en mémoire de la façon représentée sur la figure 2.

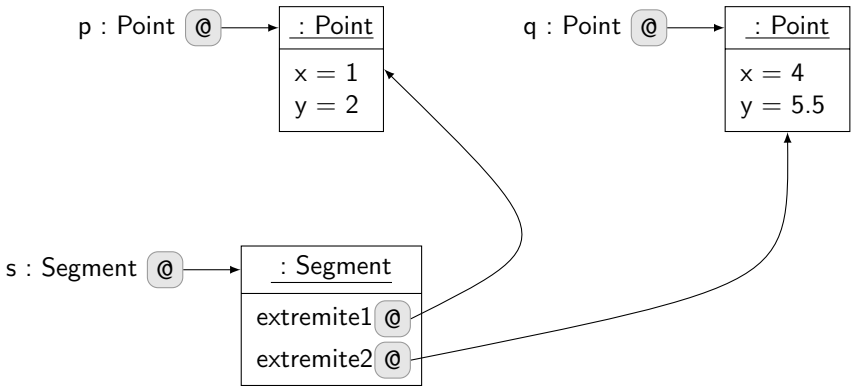

FIGURE 2 – Représentation mémoire dans le cas d'un constructeur liant les points

Une autre solution aurait été d'écrire par exemple :

```
public Segment(Point p1, Point p2) {
    this.extremite1 = new Point(p1.getX(), p1.getY());
    this.extremite2 = new Point(p2.getX(), p2.getY());
}
```

Dans ce cas, on crée bien deux nouveaux objets qui seront différents des points passés en paramètre. La représentation mémoire du programme précédent se traduit alors sur la figure 3.

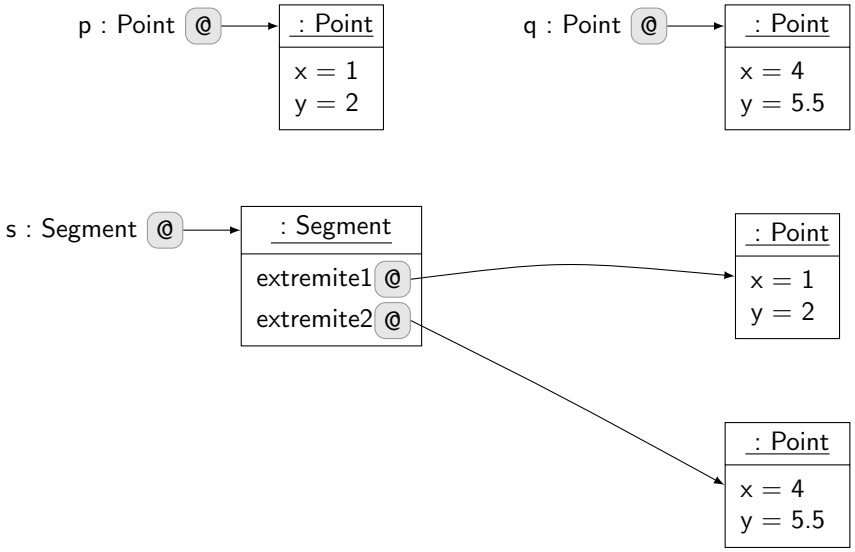

FIGURE 3 – Représentation mémoire dans le cas d'un constructeur créant effectivement de nouveaux points

– les accesseurs sur `extremite1` et `extremite2` renvoient une référence sur chacun des points. Par exemple, `getExtremite1` s'écrit :

```
    public Point getExtremite1() {
        return this.extremite1;
    }
```

Dans ce cas, un appel à `getExtremite1` sur un segment `s` nous renvoie une référence vers l'instance de `Point` représentant l'extrémité de `s`. On peut donc manipuler *directement* l'extrémité de `s`, ce qui ne devrait pas être possible d'après le principe d'encapsulation (c'est pour cela que l'on a mis `extremite1` comme attribut privé dans `Segment`). Le principe

d'encapsulation serait-il violé ? Oui, si on le prend au sens strict. Par contre, comme aucune contrainte ne s'applique sur les points extrémités du segment, on peut autoriser cette manipulation.

– la classe `Console` s'utilise facilement à travers ses méthodes statiques. Elle permettait de manipuler de telles méthodes. Par exemple, **`double`** `d = Console.readDouble("Entrez un reel: ");` utilise la méthode statique `readDouble` de la classe `Console`. La phrase « `Entrez un reel:` » s'affiche dans la console et le **`double`** entré est stocké dans la variable `d`.
On peut ainsi construire un point de la façon suivante :

```
Point p1 = new Point(Console.readDouble("Abscisse du premier point : "),
                     Console.readDouble("Ordonnee du premier point : "));
```

La documentation `javadoc` fournie sur le site vous permettait d'utiliser facilement `Console`.

5 Pourquoi une méthode `toString` ?

Dans le sujet, il était demandé d'avoir une méthode retournant une chaîne de caractères représentant l'objet `Segment` manipulé. Par exemple, pour un segment constitué des points de coordonnées $(1, 0)$ et $(2, 3)$, cette méthode devait renvoyer `[(1,0);(2,3)]`. Cette méthode s'appelle ici `toString()` et est publique (cf. la documentation de la classe `Point`).
Lorsque l'opérateur + est utilisé après un objet de type `String` (représentant une chaîne de caractères), il est considéré comme un opérateur de *concaténation* de chaînes. Le deuxième opérande doit donc lui aussi être un objet de type `String`. Lorsque Java rencontre par exemple l'opération `"La valeur de x est "+ 3`, il doit transformer `3` en objet de type `String`, ce qu'il sait faire. Le résultat de `"La valeur de x est "+ 3` est donc un objet de type `String` ayant pour valeur `"La valeur de x est 3"`. Mais si par exemple `s` est une poignée sur un objet instance de `Segment`, que se passe-t-il lorsque l'on essaye d'effectuer `"Le segment est "+ s` ?
Dans ce cas, le compilateur essaye de trouver une méthode `toString()` dans la classe `Segment` qui renvoie un objet de type `String`, i.e. qui permette de représenter un objet de type `Segment` par une chaîne de caractères. Si le compilateur ne trouve pas cette méthode dans la classe concernée, il utilise le *hashcode* de l'objet comme représentation (nous reverrons cela en détail dans le cours sur l'héritage).