IN201 : 4 – Associations

Solution

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

Résumé

Le but de ce TP est de concevoir une classe en utilisant la notion d'association vue en cours. Cette classe sera également implantée en Java en utilisant une collection.

1 Contenu

Ce corrigé succinct contient les réponses au TP numéro 4. 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 polygone au sens géométrique est un ensemble de points du plan dont le nombre est supérieur à 3 (on considère qu'un polygone « définit » un plan). On doit pouvoir le translater, mais également récupérer son périmètre et son aire.

3 Conception de la classe `Polygone`

1. définir dans un premier temps la relation qui existe entre `Polygone` et `Point`. On pourra utiliser une association simple avec multiplicité, puis une relation d'agrégation ou de composition. Écrire le diagramme UML correspondant.

   Solution :
   Il fallait ici proposer un diagramme UML d'analyse du problème. On considère ici qu'il existe une agrégation entre `Polygone` et `Point` comme représenté sur la figure 1. Je n'ai pas mis les noms de paquetages dans les diagrammes pour ne pas les surcharger.

   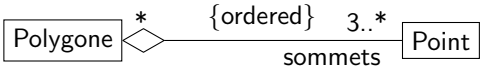

   FIGURE 1 – Association entre `Polygone` et `Point`

   Quelques remarques :
   - la multiplicité pour les objets de type `Point` est supérieure ou égale à 3. C'est bien ce qui était précisé dans la description informelle : un polygone doit au moins avoir trois points ;
   - on a donné un nom de rôle aux objets de type `Point` qui interviennent dans l'agrégation : `sommets`. On verra dans la partie implantation que c'est ce nom qui servira pour l'attribut ;
   - la multiplicité de `Polygone` dans l'association est $*$, ce qui est loin d'être anodin... Cela signifie qu'un point peut entrer en association avec plusieurs objets de type `Polygone`. Une multiplicité de 1 par exemple aurait contraint les points à n'« appartenir » qu'à un seul polygone. L'implantation de `Polygone` dépend donc déjà des choix conceptuels, même à assez haut niveau.
   - j'ai ajouté une *contrainte* UML `{ordered}` sur l'association pour préciser que les points du polygone étaient ordonnés (ce n'est pas simplement un ensemble).

   Notez que dans le cas d'une composition, on a implicitement une multiplicité 1 pour `Polygone`.

2. il faut maintenant réfléchir à une version plus « implantation » de la classe. En particulier, il faut déterminer les attributs de `Polygone`. Pour cela, réfléchir à la construction du polygone : dispose-t-on du nombre exact de points, les connaît-on etc.

   Pour pouvoir gérer l'ensemble de points constituant le polygone, on utilisera une liste de points via une instance de `java.util.ArrayList<Point>`.

   On choisit (arbitrairement !) d'utiliser un constructeur pour `Polygone` qui prend en argument un objet de type `java.util.ArrayList<Point>`.

   Les services pouvant être rendus par le polygone seront représentés par les méthodes suivantes :

– deux méthodes permettant d'ajouter et de retirer un point du polygone à une certaine position ;
– une méthode permettant de récupérer le nombre de sommets du polygone ;
– une méthode `toString` retournant une représentation du polygone sous la forme « `Point1 ... Pointn` » ;
– une méthode de translation ;
– une méthode retournant le périmètre du polygone ;
– une méthode calculant l'aire du polygone.

Proposer un diagramme UML à un niveau plus « implantation » décrivant la classe `Polygone`.

Solution :

On peut tout d'abord proposer un diagramme comportant une navigabilité et une visibilité pour les rôles. Ce diagramme est donné sur la figure 2.

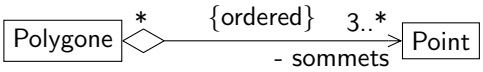

FIGURE 2 – Diagramme « implantation » entre `Polygone` et `Point`

Les méthodes étaient assez clairement données par l'énoncé. Le problème qui se posait était de déterminer les attributs de `Polygone`. D'après l'association décrite sur la figure 1, un polygone est lié à plusieurs points. Une première solution serait donc de les stocker dans un tableau, qui va s'appeler `sommets` d'après le diagramme de classe.

Or, pour construire un tableau, il faut connaître sa taille. Il aurait donc fallu avoir un constructeur qui possède un paramètre étant la taille du tableau : à la création du polygone, on donne la taille maximale que pourra avoir ce dernier. Attention, dans ce cas cette taille n'est pas le nombre effectif de points du polygone !

Cette solution est présentée sur le diagramme de classe est donné sur la figure 3.

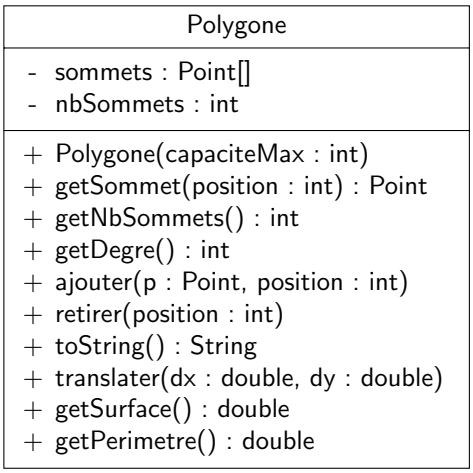

FIGURE 3 – Diagramme de classe détaillé de `Polygone` avec tableau

Nous avons choisi ici d'utiliser un objet de type `java.util.ArrayList<Point>` qui va permettre de gérer l'ensemble de points sans utiliser de tableaux. Le constructeur de `Polygone` prend donc un objet de type `java.util.ArrayList<Point>` en paramètre.

Un diagramme de conception intermédiaire faisant apparaître la classe `java.util.ArrayList<Point>` est donné sur la figure 4. Sur ce diagramme, on voit que chaque objet de type `Polygone` est composé d'un objet de type `ArrayList<Point>`, lui-même lié à au moins trois objets de type `Point`. L'utilisation de `java.util.ArrayList<Point>` permettait de plus de respecter la contrainte `{ordered}`, car une liste est ordonnée.

FIGURE 4 – Diagramme de classe intermédiaire faisant apparaître `ArrayList<Point>`

Si on avait eu une relation de composition entre le polygone et ses points (cf. figure 1), il se serait posé un problème. En effet, l'agrégation entre `ArrayList<Point>` et `Point` nous indique qu'un point peut appartenir à plusieurs ensembles. La contrainte de composition entre `Polygone` et `Point` ne serait donc plus respectée (deux polygones avec deux ensembles distincts auraient pu avoir les mêmes points). Voir la partie implantation pour quelques compléments.
Enfin, le diagramme de classe de plus bas niveau que l'on peut dériver est présenté sur la figure 5.

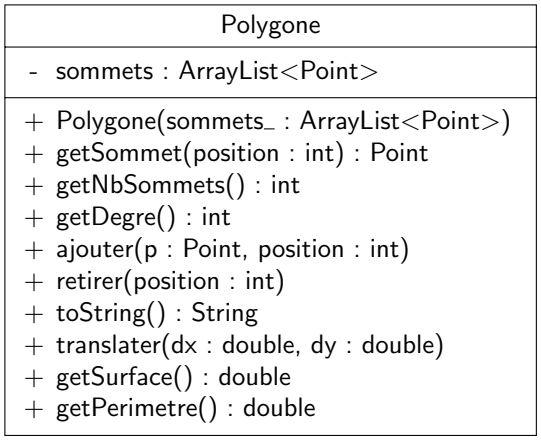

FIGURE 5 – Diagramme de classe détaillé de `Polygone`

On se rend compte qu'un objet de type `Polygone` *délègue* beaucoup de ses services à d'autres classes. Par exemple, pour ajouter un point à un polygone, on ajoute en fait un point à l'ensemble de points agrégé au polygone. Formellement, il faudrait utiliser la notion de *port* et de *connecteur de délégation* en UML 2.0 (voir [1] pour plus de détails). On peut toutefois utiliser les notes comme sur la figure 6 pour représenter cela.

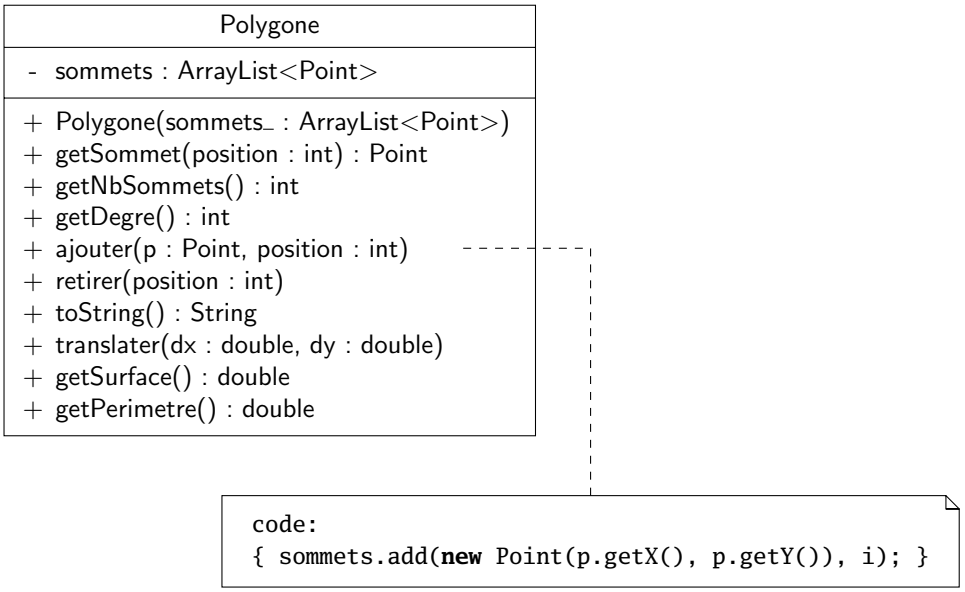

FIGURE 6 – Délégation du service `ajouter` vers `ArrayList<Point>`

Quelques remarques supplémentaires :
– toutes les méthodes qui renvoient quelque chose (i.e. dont le type de retour n'est pas **`void`**) commencent par `get` : elles permettent d'accéder à une caractéristique de la classe (qui n'est pas forcément stockée sous forme d'attribut). On parle alors de *requête* ;

– les méthodes qui ne renvoient rien permettent quant à elles de modifier l'état d'un objet : on parle alors de *commandes*. En particulier, ces méthodes ne peuvent pas être déclarées comme « pures » pour JML.

3. écrire un diagramme de séquence représentant le scénario suivant :
   – le programme de test crée un point de coordonnées (0,0)
   – le programme de test crée un point de coordonnées (2,2)
   – le programme de test crée un point de coordonnées (4,5)
   – le programme de test crée une instance de `ArrayList` à partir de ces points (on ne représentera pas les ajouts des points dans la liste pour ne pas alourdir le diagramme)
   – le programme de test crée un polygone à partir de la liste
   – le programme de test translate le polygone avec un vecteur $(-2, 3)$
   – le programme de test affiche le périmètre du polygone

   Solution :
   Rien de bien difficile ici. La construction du diagramme permettait de voir qu'il fallait appeler la méthode `translater` sur chacun des points (même si cela était évident ici). Le diagramme est présenté sur la figure 7.

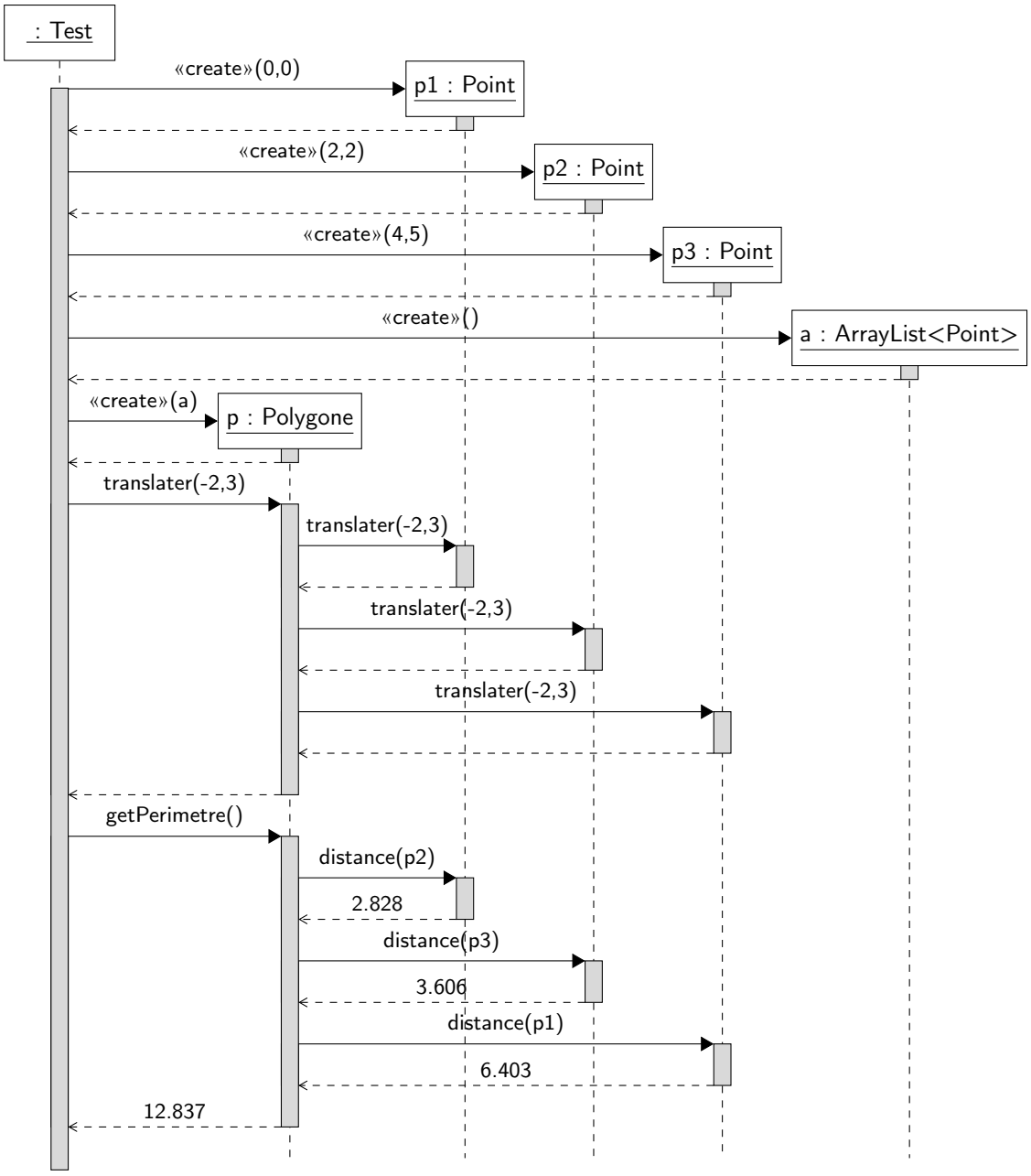

FIGURE 7 – Diagramme de séquence représentant le scénario

4 Implantation de la classe `Polygone`

Les sources corrigées sont disponibles sur le site web, avec les classes de test. Le développement des méthodes était grandement facilité si vous preniez le soin de les tester une par une avec JUnit par exemple et si vous utilisiez JML.
J'ai choisi de spécifier entièrement le polygone en utilisant JML. Les assertions (préconditions, postconditions et invariants) sont suffisamment claires à mon avis. J'ai également choisi dans le constructeur de copier *entièrement* l'objet de type `ArrayList<Point>` passé en paramètre, ceci pour éviter de pouvoir manipuler les points du polygone sans passer par les méthodes de la classe `Polygone`, ce qui violerait le principe d'encapsulation. Cela permet également d'éviter que l'objet de type `ArrayList<Point>` soit partagé entre plusieurs polygones, ce que vous deviez faire si vous aviez choisi une composition entre `Polygone` et `Point`.
On peut remarquer que la principale difficulté provient de la relation d'agrégation proposée dans la section 3. La solution d'implantation est plutôt une composition, car les points ne peuvent pas être partagés. Mais cela ne pose pas de problème, car l'agrégation nous indique que les points *peuvent* être partagés (ce n'est pas obligatoire).
Il existe une deuxième version de la classe applicative : `ScenarioPolygoneAfficheur`. Celle-ci utilise l'afficheur graphique disponible sous forme de fichier JAR sous l'onglet « Ressources ».
Quelques remarques sur la classe `ArrayList` qui est utilisée :

– la méthode `remove` de la classe `ArrayList` qui est utilisée renvoie l'objet que l'on veut enlever (cf. javadoc). Cela n'est pas gênant, même si l'on ne fait rien avec cet objet :

```
  public void retirer(int position) {
      this.sommets.remove(position - 1);
  }
```

  L'objet renvoyé par `remove` est « posé » sur la pile de la méthode `retirer` et « disparaît » sitôt la méthode finie.

– `ArrayList` possède une méthode `toString`. Si on consulte la documentation javadoc de `ArrayList`, `toString` apparaît dans « `Methods inherited from class java.util.AbstractList` » (il s'agit d'une méthode *héritée*, nous en reparlerons dans quelques cours). Si l'on consulte la documentation de cette méthode, on s'aperçoit que la méthode `toString` va utiliser la représentation sous forme de chaîne de caractères des éléments de la liste et séparer ces chaînes par des virgules. On pouvait donc utiliser directement la liste attribut de `Polygone` dans la méthode `toString` de `Polygone` [1]

Remarque avancée : vous pourrez remarquer que j'ai créé un deuxième constructeur pour la classe Polygone :

```
  public Polygone(Point... sommets_) {
      this.sommets = new ArrayList<Point>();
      for (int i = sommets_.length - 1; i >= 0; i--) {
        this.sommets.add(0, new Point(sommets_[i].getX(), sommets_[i].getY()));
      } // end of for (int i = sommets_.length; i >= 1; i--)
  }
```

Ce constructeur prend un argument de type `Point...` (notez les « `...` »). On appelle ce type d'argument un *argument de taille variable*. Cela permet de préciser dans ce constructeur que l'on peut passer en paramètre un nombre *indéfini* de points. On les récupère ensuite sous la forme d'un tableau d'instance de `Point` appelé `sommets_` (cf. corps du constructeur qui manipule le tableau).
On peut voir l'utilisation de ce constructeur dans la classe `ScenarioPolygoneAfficheur` :

```
      Polygone p1 = new Polygone(new Point (1, 1),
                                 new Point (2, 1),
                                 new Point (3, 4));
```

Si vous voulez utiliser un argument de taille variable comme paramètre d'une méthode, il doit apparaître comme dernier paramètre de la méthode.

Références

[1] G. Booch, J. Rumbaugh et I. Jacobson. *The Unified Modeling Language reference manual*. Addison-Wesley, 2004.

1. Je ne l'ai pas fait et ai écrit « directement » la méthode `toString` de `Polygone`.