IN201 : 8 – Classes abstraites

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

Solution

Résumé

Le but de ce TP est de construire par généralisation des classes précédemment développées une classe `Figure`. Il permet également de réviser les notions aperçues lors de la séance sur l'héritage.

1 Contenu

Ce document contient le corrigé de la partie « conception » du TP. Les sources Java des classes sont disponibles sur le site http://www.tofgarion.net/lectures/IN201.

2 Problématique

Le but de ce TP est d'utiliser la relation de généralisation pour construire une hiérarchie de classes à partir des classes existantes (`Point`, `Polygone`, `Segment` et `PointNomme`). Contrairement à la dernière séance, on ne cherchera pas à spécialiser les classes, mais à les généraliser en une classe appelée `Figure`.

3 Conception de la classe `Figure`

La classe `Figure` doit être une généralisation des classes déjà disponibles, i.e. `Point`, `Polygone`, `Segment` et `PointNomme`.

1. trouver les attributs et les méthodes de la classe `Figure` à partir de ces classes. Ces méthodes sont-elles abstraites ?

   Solution :
   Il n'y a pas d'attributs commun à toutes les classes (mis à part la couleur que nous introduisons explicitement ici). On aurait pu par exemple considérer un point de référence pour chaque figure (barycentre du polygone, milieu du segment etc).
   Les méthodes communes que l'on peut généraliser sont `translater` et `afficher`. La méthode `translater` est bien sûr abstraite, car on ne peut lui donner une implantation au niveau de la classe `Figure`. Ce sont les classes qui spécialisent `Figure` qui devront le faire. Par contre, la méthode `afficher` est concrète, car on sait écrire son code puisque l'on a une méthode `toString` dans toutes les classes (il s'agit de `System.out.println(this)`). Mais dans ce cas, je choisis de placer une méthode `toString` abstraite dans `Figure`, ce qui obligera les sous-classes de `Figure` à la redéfinir (car `afficher` utiliser implicitement `toString`).
   Ceci nous permet d'assurer que tout objet instance d'une classe étendant `Figure` possède une méthode `translater` et une méthode `toString` définie dans la classe (sinon la classe dont l'objet est issu serait abstraite et n'aurait donc pas d'instance).

2. ajouter ensuite un attribut `couleur` à la classe `Figure` (il devra donc être du type `java.awt.Color`) ;
3. on souhaite enfin que toutes les classes héritant de `Figure` aient une méthode publique `dessiner(afficheur.Afficheur afficheur)` qui permette de dessiner la figure sur un afficheur donné. Modifier les classes en ce sens et fournir le diagramme UML de la hiérarchie ainsi construite.

   Solution :
   Le diagramme UML est fourni sur la figure 1.
   On remarque que l'on peut utiliser l'interface `Afficheur` comme type du paramètre de la méthode `dessiner`. On choisira ensuite à l'exécution une instance d'une classe réalisant `Afficheur`, comme `Ecran` par exemple.

4 Implantation de la classe `Figure`

1. implanter et documenter la classe `Figure` en Java ;

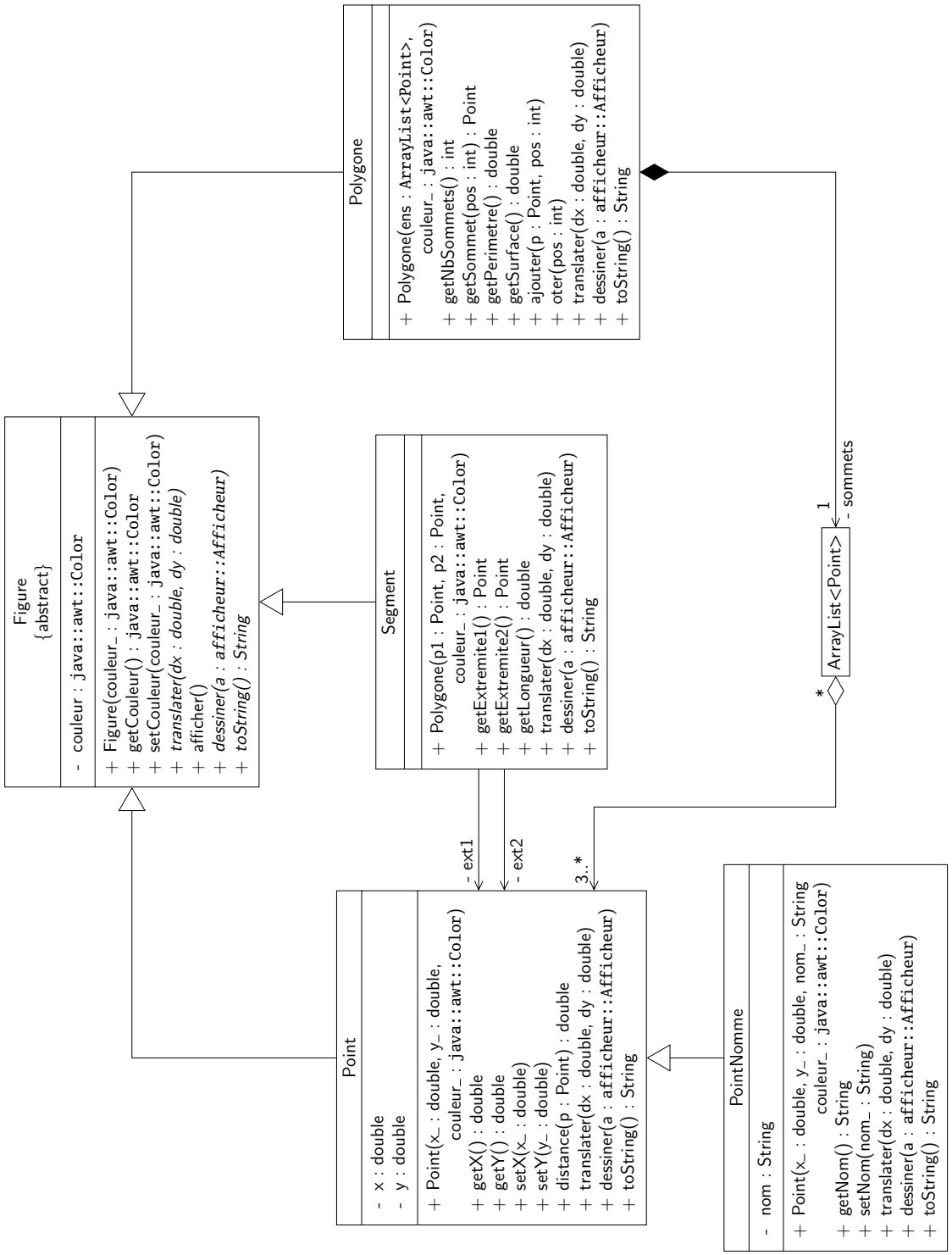

FIGURE 1 – Diagramme de classe UML récapitulatif

Solution :

Le code source de la classe est disponible sur le site.

2. modifier les classes existantes pour respecter la conception donnée en 3. On pourra utiliser la commande « Change method signature » du menu contextuel « Refactor ». Celle-ci permet en particulier d'ajouter un argument à une méthode et de répercuter ce changement dans les classes utilisant la méthode en question.

   Que se passe-t-il si une des classes n'implante pas la méthode `dessiner` demandée ?

   Solution :

   Le fait de généraliser les classes nous imposait de modifier le code source des classes disponibles. En particulier, il ne fallait pas oublier de modifier les constructeurs, car la classe `Figure` ne dispose pas d'un constructeur par défaut.

   On peut remarquer également que les spécifications JML écrites dans les classes et qui concernent la méthode `translater` par exemple doivent commencer par le mot-clé **`also`**, car elles redéfinissent le contrat d'une méthode.

   Si l'on oublie d'implanter une méthode abstraite dans une des classes, on obtient le message d'erreur suivant à la compilation :

```
Point.java:14: fr.supaero.figure.Point is not abstract and does not override
 abstract method dessiner(afficheur.Afficheur) in fr.supaero.figure.Figure
public class Point extends Figure {
       ^
1 error
```

   Le processus de généralisation est donc très coûteux !

3. comment faire pour tester avec JUnit la classe `Figure` ? S'inspirer de la classe `PointNommeTest` fournie.

   Solution :

   Les classes de test modifiées sont disponibles sur le site. Comme `Figure` est une classe abstraite, on ne peut pas initialiser un objet de type `Figure`. Par contre, on peut écrire les méthodes de test pour `getCouleur` et `setCouleur`[1]. On utilise donc la technique utilisée dans la classe `PointNommeTest` : pour vérifier que `PointNomme` passe les tests unitaires de `Point` (principe de Liskov), on déclare tout d'abord `PointNommeTest` comme classe fille de `PointTest`. Comme les méthodes de test de `PointTest` sont publiques, elles seront utilisées dans `PointNommeTest`. On déclare une *factory method* protégée qui renvoie un objet de type `Point` dans `PointTest` et on la redéfinit dans `PointNommeTest`. On appelle ensuite la méthode `setUp` de `PointTest` dans la méthode `setUp` de `PointNommeTest` grâce à **`super.`**`setUp()`.

   La classe `FigureTest` est elle-même abstraite, car on ne peut pas instancier un test à partir de cette classe : l'acteur du test n'est pas initialisé correctement. La *factory method* est elle-même abstraite, car on ne sait pas l'écrire. On sera donc obligé de redéfinir cette *factory method* dans toutes les sous-classes de `FigureTest`.

   La classe `AllFiguresTest` vous présente comment faire simplement une classe générant une suite de tests. Ici, tous les tests contenus dans les classes `PointTest`, `PointNommeTest`, `PolygoneTest` et `SegmentTest` seront lancés par cette classe.

   Remarque : on peut remarquer que ce processus de généralisation est *très* coûteux. On est par exemple obligé de réécrire la classe de test de la classe `Polygone` car la signature du constructeur de `Point` a changé.