L2 info S3
UE : Programmation Orientée Objets 1 - Java

TP4 - Modélisation

Objectifs

A la fin de ce TP, vous devez être capables de :

1. Approfondir votre lecture de la documentation API pour savoir reconnaître les classes, placer une classe dans l'arborescence, savoir si elle est abstraite, finale ou non.
2. Programmer en conformité avec la modélisation choisie et les spécifications établies.
3. Avoir un avis critique sur les choix de modélisation (en sachant qu'il n'y a pas de règle absolue).

1 Documentation de l'API de Java

Utiliser la documentation de l'API pour dessiner l'arborescence des classes suivantes en précisant quelles classes sont abstraites, finales :

- `String`
- `Point`
- `Math`
- `Arrays`
- `List`
- `Random`, `SecureRandom`
- `BufferedReader`, `InputStreamReader`

2 Modélisation de l'héritage

2.1 Présentation de la polémique

L'héritage traduit une spécialisation.

Il y a différentes manières d'utiliser l'héritage en POO (B. Meyer en distingue 12). Sans les expliciter toutes, il est intéressant de mettre l'accent sur quelques points importants. Nous distinguerons essentiellement 2 types d'héritage :

- l'héritage de modélisation
- l'héritage d'implantation

Héritage de modélisation

Dans l'optique où les classes forment un modèle du domaine d'application, la relation entre une sous-classe et une classe est un lien "est un" entre les concepts du domaine. Par exemple, un technicien `est un` employé d'une organisation, autrement dit l'ensemble des techniciens est inclus dans l'ensemble des employés. Ainsi, la classe `Technicien` devrait être une sous-classe de la classe `Employe`.
On parlera de généralisation-spécialisation.

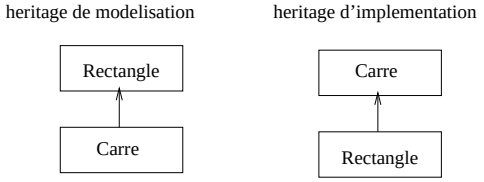
Dans certaines disciplines scientifiques comme la botanique ou la zoologie, il existe déjà des hiérarchies complètes de classification des objets (appelées taxonomies).

Héritage d'implantation

L'héritage d'implantation est celle appliquée en Java (mais pas forcément recommandée). L'intérêt de ce type d'héritage tient à l'`économie d'écriture de code`. On ne réinvente pas ce qui existe et qui nous satisfait déjà partiellement. De plus, on peut compléter ce qui est hérité, voire le redéfinir, de manière à réaliser le comportement voulu. L'extension est donc une technique de réutilisation relativement souple qui se distingue du "tout ou rien".

Modéliser ou implanter ?

Il arrive parfois que la vision taxonomique s'oppose à l'héritage d'implémentation. Prenons un exemple mathématique avec les 2 classes `Carre` et `Rectangle`.



1. D'un point de vue géométrique et donc en héritage de modélisation, un carré est une spécialisation (sous-ensemble) de rectangle :
   — un rectangle a 2 attributs : une longueur et une largeur ;
   — un carré est un cas particulier de rectangle car sa longueur est égale à sa largeur et on utilise alors usuellement le terme de `côté`. Une seule donnée est suffisante.
2. En programmation, avec l'héritage d'implémentation, on utiliserait plus naturellement la hiérarchie inverse : un rectangle est une extension de carré.

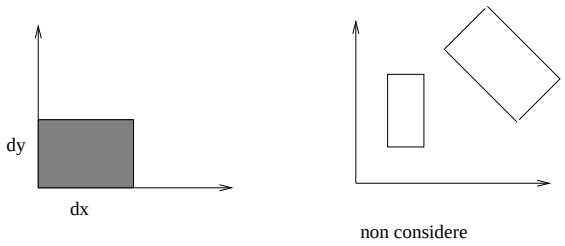
A la question : "en pratique, quel héritage choisir ? " il n'y pas de réponse ferme et définitive.
La consigne est : savoir `négocier` ; il vaut mieux rester le plus proche possible d'une hiérarchie correspondant au lien "est un" sans en faire un dogme absolu.

Vous trouverez cette problématique exprimée/utilisée de façon explicite ou implicite, oralement ou dans les écrits comme dans l'information en ligne (cf par exemple : http ://gfx.developpez.com/tutoriel/java/composition/).

2.2 Programmation des deux formes d'héritage

Ecrire les 2 formes d'héritage sur l'exemple des carrés et des rectangles afin de programmer en conformité avec la modélisation choisie et de vous rendre compte des avantages/inconvénients de chaque approche.

Pour simplifier l'exercice et nous concentrer sur l'essentiel, rectangles et carrés
- n'auront pas de point (coin inférieur gauche) pour les situer dans le plan (ils seront placés à l'origine)
- seront parallèles aux axes (pas d'angle)
- auront une taille selon l'axe des `x` (respectivement `y`) notée `dx` (respectivement `dy`) pour ne pas utiliser la terminologie `largeur` et `longueur` avec la contrainte que cela implique : $largeur \leq longueur$.
- `dx` et `dy` seront strictement positifs et seront transmis correctement par les utilisateurs (pas de gestion d'erreur demandée, pas d'appel à `Math.abs` pour calculer le périmètre).



2.2.1 Héritage de modélisation : Carre sous-classe de Rectangle

1. Compléter le corps des méthodes de la classe `Rectangle`

```
public class Rectangle
{
    protected double dx ;
    protected double dy ;
    public Rectangle(double dx, double dy) {...}
    public double getdx() {...}         // accesseur
    public double getdy() {...}         // accesseur
    public void setdx(double dx) {...}  // modificateur
    public void setdy(double dy) {...}  // modificateur
    public double perimetre() {...}
    public double surface() {...}
    public String toString()
    {
      return "Je suis un rectangle. " ;
    }
}
```

2. Ecrire la classe `Carre`

```
public class Carre extends Rectangle
{
      // dx=dy
      public Carre(double c) {...}
      public double getCote() {...}       // accesseur
      public void setCote(double c) {...} //modificateur
      ...
}
```

sachant que
— On ne peut supprimer d'attributs dans la classe dérivée mais seulement en ajouter.
— Les méthodes sont nommées `getCote` et `setCote` pour adopter la terminologie de l'utilisateur et masquer la structure interne.

et après avoir répondu aux questions suivantes :

(a) Gestion des attributs :
Pour rester proche de la terminologie des utilisateurs, un carre a un côté.
Pour gérer un `cote` ou 2 attributs `dx` et `dy`, plusieurs solutions sont envisageables :

a1) les 2 attributs hérités sont "oubliés" et on ajoute un attribut spécifique `cote` dans la classe `Carre`.

a2) des 2 attributs `dx` et `dy` hérités, on choisit "d'oublier" `dy` et d'utiliser uniquement `dx`.

a3) les 2 attributs hérités sont gérés de telle sorte que quelle que soit la modification `dx=dy`.

Donner brièvement votre avis sur chacune de ces propositions.
Vous programmerez la 3ème solution.

(b) Méthodes :
Considérer les 4 méthodes `getdx`, `getdy`, `setdx`, `setdy` héritées de `Rectangle`.

b1) Dire si on peut interdire leur utilisation dans `Carre`.

b2) Dire s'il est souhaitable de les rendre `private` ou `protected` dans Rectangle.

b3) Dire si elles peuvent être redéfinies comme `private` dans `Carre`.

b4) Donner les méthodes qui devront être redéfinies dans `Carre` (inclure `toString` dans la discussion).

3. Compléter le programme de test suivant qui utilise ces 2 classes et le polymorphisme

```
public class TestHeritageModelisation {
  public static void main (String[] args)
  {
     ...[] tableau;                  // type a definir
     tableau = new ...;              // a completer
     tableau[0]= new Rectangle(5,2.25);
     tableau[1]= new Carre(10);
     tableau[2]= new Carre(8.5);
     tableau[3]= new Rectangle(2.5,6);
     tableau[4]= new Rectangle(3.3,7.1);

     for (int i=0; i<5 ; i++)
     {
       System.out.println(...);    // a completer
       System.out.println();
     }
   }
}
```

permettant d'afficher

```
Je suis un rectangle. J'ai pour abscisse et ordonnee : 5.0, 2.25
Mon perimetre et ma surface sont respectivement : 14.5, 11.25

Je suis un carre. J'ai pour abscisse et ordonnee : 10.0, 10.0
Mon perimetre et ma surface sont respectivement : 40.0, 100.0

Je suis un rectangle. J'ai pour abscisse et ordonnee : 3.3, 7.1
Mon perimetre et ma surface sont respectivement : 20.799999999999997, 23.429999999999996

...
```

Le test explicite (`getClass`, `instanceof`) sur le type effectif d'un objet ne doit se faire que si cela s'avère indispensable.

2.2.2 Héritage d'implantation :

1. Compléter le corps de la classe `Carre`

```
public class Carre
{
   protected double dx ;
   public Carre(double c) { ... }
   public double getCote() {...}
   public void setCote(double c) {...}
   public double perimetre() {...}
   public double surface() {...}
   public String toString()
   {
     return "Je suis un carre. " ;
   }
}
```

   L'attribut a pour nom `dx` et non `cote` en vue de l'extension de la classe et ne pas poser de problème de vocabulaire.

2. Ecrire la classe `Rectangle`.

```
public class Rectangle extends Carre
{
   // attributs ??
    public Rectangle(double dx, double dy) {...}
    public double getdx() {...}
    public double getdy() {...}
    public void setdx(double dx) {...}
    public void setdy(double dy) {...}
    ...
}
```

   Comme pour la modélisation précédente, préciser avant de programmer
   — quels seront les attributs de la classe Rectangle. Préciser ceux qui sont hérités, ajoutés.
   — les méthodes à redéfinir.

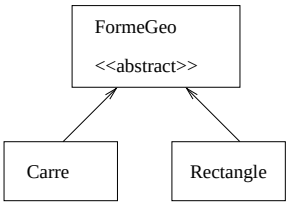
3. Ecrire le programme de test (fichier `TestHeritageImplantation`).
   Pour cela, reprendre `TestHeritageModelisation`. Modifier le code. L'affichage ne devra pas changer.

4. Dire si vous avez pu vous dispenser des tests explicites (`getClass`, `instanceof`) sur le type effectif d'un objet. Si votre réponse est "non", dire comment améliorer votre code pour les supprimer.

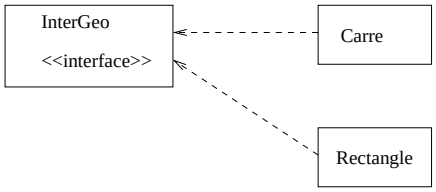
2.2.3 Exercice complémentaire

Pour bénéficier du polymorphisme, Carre et Rectangle doivent être liés d'une façon quelconque.
Vous pourriez programmer différentes formes possibles :

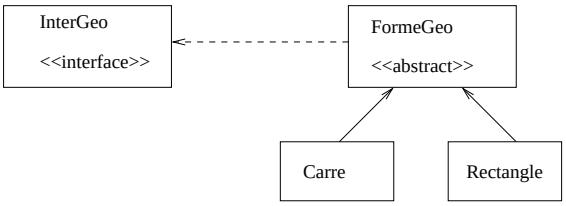
1. Avec une classe abstraite



2. Avec une interface (s'il n'y a pas d'attribut commun)



3. Avec une classe abstraite et une interface



Pour chaque proposition

1. Déterminer les attributs et méthodes des classes, méthodes de l'interface pour les propositions concernées.
2. Programmer les classes.
3. Ecrire un programme de test.