Feuille 6 - Limites de l'héritage

NB : Le problème décrite ci-dessous est aussi connu comme le "Circle-Ellipse Problem" `https://en.wikipedia.org/wiki/Circle-ellipse_problem`.

1 La classe Rectangle

Donnez une classe qui représente un rectangle $ABCD$ en utilisant
— Deux doubles pour les coordonnées du point A.
— Deux doubles pour les dimensions (longueur, largeur).
Donnez aussi un constructeur, des méthodes getteurs, et une méthode qui calcule le périmètre d'un rectangle.

```
1  public class Rectangle {
2          private double ax, ay, length, width;
3          public Rectangle(double ax, double ay, double length, double width){
4                  this.ax = ax;
5                  this.ay = ay;
6                  this.length = length;
7                  this.width = width;
8          }
9          public double getAx() {
10                 return ax;
11         }
12         public double getAy() {
13                 return ay;
14         }
15         public double getLength() {
16                 return length;
17         }
18         public double getWidth() {
19                 return width;
20         }
21         public double perim() {
22                 return 2*length+2*width;
23         }
24 }
```

2 La classe Carré

Selon les principes de la géométrie, un carré **est un cas spéciale** d'un rectangle. En fait, un carré est un rectangle dont les côtés sont égaux. Donnez donc une implémentation d'une classe qui représente les carrés, en héritant de la classe Rectangle.

```
1 class Square extends Rectangle {
2         public Square(double ax, double ay, double side){
3                 super(ax,ay,side,side);
4         }
5         public double perim() {
6                 return 4*getLength();
7         }
8 }
```

2.1 Reflection

Considérer le programme suivant. Va-t-il compiler ?

```
Rectangle r = new Square(2,3,4);
System.out.println(r.perim());
```

L'avantage principale de l'héritage est qu'on a le droit d'utiliser le code ci-dessus. On peut utiliser un objet de type Square où on peut utiliser un objet de type Rectangle, parce que le type Square est un sous-type du type Rectangle. On appelle ce principe Liskov substitution principle `https://en.wikipedia.org/wiki/Liskov_substitution_principle`.

Cependant, cette implémentation presente quelques désavantages.

— Peut-on imaginer une implémentation de la class Square qui est plus efficace (en termes de mémoire, temps d'exécution,...) que l'impémentation héritée ?

3 Classe Modifiable

Ajouter de méthodes setteurs dans la classe Rectangle. Considérer le programme suivant :

```
Square s = new Square(2,3,4);
s.setWidth(5);
System.out.println(s.perim());
```

— Va-t-il compiler ?

— Quel est le problème ?

Le problème ici est que la méthode setteur peut violer la modélisation d'un carré. Le résultat de l'opération `setWidth` est un objet qui n'est plus un carré (en termes mathématiques). Cependant, l'objet reste un carré pour le système des types de Java !

4 Solutions

4.1 Redéfinition des Setteurs

Donnez une version de la class Square où les deux setteurs `setLength,setWidth` ont été redéfinis de sorte que le résultat de leur exécution est garanti d'être un carré.

Considérer le code suivant :

```
Rectangle r = new Square(2,3,4);
r.setWidth(5);
System.out.println(r.perim());
```

Que va-t-il afficher ?

Le problème avec cette approche est que la documentation (le contrat) de la classe Rectangle dit concernant la méthode `setWidth` très probablement quelque chose comme : "Méthode qui modifie l'attribut `width` *et rien d'autre*". On a donc violé le contrat de la classe Rectangle, et quelques programmes qui l'utilisent courent le risque de ne plus fonctionner si on passe un objet de type Square.

4.2 Héritage Inverse

On peut dire que la source de nos problèmes est qu'en termes d'implémentation, la classe `Rectangle` devrait être la sous-classe de `Square`. Ça peut paraître raisonable : un rectangle est un carré avec quelques informations en plus. Cependant, cette solution est une mauvaise idée.

Considérer le programme suivant :

```
Square s = new Rectangle(1,2,3,4);
```

Si on définit `Rectangle` comme sous-classe de `Square`, ce programme compile !

4.3 Classes Immuables

Une solution pour éviter les problèmes ci-dessus serait de définir la classe `Rectangle` comme une classe d'objets immuables, c'est-à-dire, d'objets dont les attributs sont `final`. Dans ce cas on a les avantages suivants :

- Un carré immuable est toujours garanti de rester un carré.
- Les méthodes qui modifient un carré peuvent retourner un nouveau objet de type `Rectangle`
- On peut donner des versions modifiables des deux classes. Cependant, on n'est pas obligé d'hériter pour les classes modifiables. Alors, on garde les avantages de l'héritage (les deux versions immuables ont une relation de sous-type), et on évite les problèmes.

```
1  class RectangleIm {
2          private final double ax, ay, length, width;
3          public RectangleIm(double ax, double ay, double length, double width){
4                  this.ax = ax;
5                  this.ay = ay;
6                  this.length = length;
7                  this.width = width;
8          }
9          public RectangleIm setLength(double l){
10                 return new RectangleIm(ax,ay,l,width);
11         }
12         public double getAx() {
13                 return ax;
14         }
15         public double getAy() {
16                 return ay;
17         }
18         public double getLength() {
```

```
19                  return length;
20          }
21          public double getWidth() {
22                  return width;
23          }
24          public double perim() {
25                  return 2*ax+2*ay;
26          }
27  }
28
29  class SquareIm extends RectangleIm {
30          public SquareIm(double ax, double ay, double side){
31                  super(ax,ay,side,side);
32          }
33          public SquareIm setSide(double s){
34                  return new SquareIm(getAx(),getAy(),s);
35          }
36          public double perim() {
37                  return 4*getLength();
38          }
39  }
```