TD et TP de Compléments en Programmation Orientée Objet n° 5 : Mécanismes et stratégies d'héritage
(Correction)

I) Modélisation géométrique : limites de l'héritage et du sous-typage

Important : Testez vos programmes (dans une méthode `TestExo.main()`) après chaque modification ! Pour ce faire, munissez toutes vos classes d'un `toString()` suffisamment informatif.

Exercice 1 : Rectangles et carrés

Le problème décrit ci-dessous est aussi connu sous le nom du problème "cercle-ellipse". La relation entre un carré et un rectangle est en effet analogue à celle entre le cercle et l'ellipse.

Plus d'informations ici : https://en.wikipedia.org/wiki/Circle-ellipse_problem.

1. Programmez une classe `Rectangle`. Un rectangle est un polygône à 4 côtés avec 4 angles droits. Dans un repère orthonormé, on peut le caractériser, de façon minimale, par
   — les coordonnées de son premier point (ex : pour rectangle $ABCD$, le point A),
   — l'angle de son "premier côté" (ex : l'angle du vecteur $\overrightarrow{AB}$ avec l'axe des abscisses),
   — et la longueur de ses deux premiers côtés (ex : AB et BC).
   Les attributs sont privés, accessibles par "getteurs". N'écrivez pas encore les "setteurs".

Correction :

```
1  public class Rectangle {
2      private double ax, ay, angle, longueur, largeur;
3
4      public Rectangle(double ax, double ay, double angle, double longueur, double largeur)
           {
5          super();
6          this.ax = ax;
7          this.ay = ay;
8          this.angle = angle;
9          this.longueur = longueur;
10         this.largeur = largeur;
11     }
12
13     public double getAx() {
14         return ax;
15     }
16
17     public double getAy() {
18         return ay;
19     }
20
21     public double getAngle() {
22         return angle;
23     }
24
25     public double getLongueur() {
26         return longueur;
27     }
28
29     public double getLargeur() {
30         return largeur;
31     }
32
```

```
33  }
```

2. Tous les livres de géométrie élémentaire disent qu'un carré est un rectangle particulier, dont tous les côtés sont égaux.
   En POO, la relation "est un" se traduit habituellement par de l'héritage.
   Programmez donc la classe `Carre` qui étend `Rectangle` tout en garantissant que l'objet obtenu représente bien un carré.

   Correction :

```
1  public class Carre extends Rectangle {
2      public Carre(double ax, double ay, double angle, double cote) {
3          super(ax, ay, angle, cote, cote);
4      }
5  }
```

3. Vous avez dû trouver une solution sans ajouter d'attributs. Si ce n'est pas le cas, corrigez l'exercice précédent.
4. Premier problème : un carré représenté par une instance de `Carre` contient plus d'attributs que nécessaire. Voyez-vous pourquoi ?

   Correction : Quand on créée un carré défini ainsi, on crée un rectangle, avec tous ses attributs, y compris `largeur` et `longueur`. Or, dans le cas du carré, ces deux valeurs doivent être tout le temps égales, donc une des deux ne sert à rien et on aurait pu s'en passer si on avait déclaré `Carre` directement sans héritage.

   Le problème de la taille en mémoire n'est en fait pas très grave. Ce qui est plus embêtant, c'est que la redondance peut induire des problèmes de cohérence. Voir ci-dessous.
5. Ajoutez maintenant les "setteurs" à la classe `Rectangle` (notamment `setLongueur` et `setLargeur`, modifiant respectivement la longueur et la largeur, sans toucher aux autres propriétés du rectangle).
6. Si dans une méthode on fait :

```
1  Carre c = new Carre(/*ax = */ 0, /*ay = */ 0, /* angle = */ 0, /* cote */ = 3);
2  c.setLongueur(10);
```

   l'objet `c` correspond-il toujours à la modélisation d'un carré ?
   L'objet `c` est-il pourtant encore de type `Carre` ?

   Correction : Non et oui. En effet, on modifie la longueur sans modifier la largeur, on obtient donc un rectangle (géométriquement parlant), or la classe d'un objet n'est pas modifiable. Donc on a, à la fin, une instance de `Carre` représentant en vrai un rectangle non carré !

7. Pour corriger ce problème, redéfinissez (`@Override`) les setteurs dans la classe `Carre` de sorte à préserver l'invariant "cet objet représente un carré". Une possibilité : modifier la longueur modifie aussi la largeur, et vice-versa.

   Correction :

```
1  public class Carre extends Rectangle {
```

```
2       public void setCote(double cote) {
3           super.setLongueur(cote);
4           super.setLargeur(cote);
5       }
6
7       @Override
8       public void setLongueur(double longueur) {
9           setCote(longueur);
10      }
11
12      @Override
13      public void setLargeur(double largeur) {
14          setCote(largeur);
15      }
16
17      public Carre(double ax, double ay, double angle, double cote) {
18          super(ax, ay, angle, cote, cote);
19      }
20
21  }
```

8. Dans la documentation de la classe `Rectangle` (p. ex. : dans la javadoc), il serait raisonnable d'écrire comme spécification pour la méthode `setLongueur`, une phrase comme "modifie la longueur de ce rectangle sans modifier ses autres propriétés" (et une phrase similaire pour `setLargeur`).

   Si une méthode contient les instructions suivante :

```
1  Rectangle r = new Carre(/*ax = */ 0, /*ay = */ 0, /* angle = */ 0, /* cote */ = 3);
2  r.setLongueur(10);
```

   quelle sera alors la largeur de `r` ? La spécification décrite plus haut est-elle alors respectée ?

   Correction : Vu que c'est la méthode de `Carre` qui est appelée (liaison dynamique), la largeur sera aussi modifiée de la même façon que la longueur, donc elle vaudra 10. Ceci contredit la spécification qui dit que, pour le type `Rectangle`, `setLongueur()` ne doit pas avoir d'effet sur la largeur.

9. Dire que `Carre` hérite de `Rectangle` n'a pas l'air de fonctionner bien. Mais peut-on faire le contraire ? Après tout, un rectangle utilise une grandeur en plus, par rapport au carré. En vous inspirant de ce qui précède, montrez que ça ne marche pas non plus.

   Correction : Cet héritage permettrait d'affecter à une variable de type `Carre` un objet instancié comme `Rectangle` avec une longueur et une largeur différente, violant la spécification d'un carré.

Dans cet exercice, on vient de montrer que, bien qu'un carré, en tant qu'entité mathématique figée, soit un rectangle particulier, cette inclusion n'est plus valable quand on parle de carrés et de rectangles en tant qu'objets modifiables préservant leur identité de carré ou de rectangle.

Dans ce cadre, il est donc illusoire de vouloir que `Carre` soit sous-type de (et a fortiori sous-classe de) `Rectangle`. On verra dans la suite qu'on peut obtenir un sous-typage satisfaisant en écrivant des types `Carre` et `Rectangle` immuables.

Exercice 2 : Quadrilatères

Cet exercice est, a priori, indépendant du précédent. Pour un programme propre, il est recommandé de repartir de zéro pour son écriture.

On vient de voir dans l'exercice précédent que la spécification d'un objet modifiable est facilement mise à mal par le sous-typage. Ainsi, dans cet exercice, on n'écrira pas de mutateurs (pour de vraies classes immuables, attendez la suite !).

1. On veut écrire des classes pour les formes suivantes : quadrilatère, trapèze, parallélogramme, losange, rectangle, carré. Vous pouvez consulter `https://fr.wikipedia.org/wiki/Quadrilatère` pour plus de détails.

   Dessinez le graphe de sous-typage idéal. Est-ce qu'il sera possible de le réaliser par uniquement des classes, en matérialisant le sous-typage par de l'héritage ? Pour quelle raison ?

   Quelles solutions peut-on envisager pour régler ce problème ?

   > **Correction :** Idéalement, on veut quadrilatère $<:$ figure, trapèze $<:$ quadrilatère, parallélogramme $<:$ trapèze, losange $<:$ parallélogramme, rectangle $<:$ parallélogramme, carré $<:$ rectangle et carré $<:$ lozange.
   > Le problème, c'est que carré a deux parents. Donc Java interdit que rectangle et lozange soient tous les deux des classes. Plusieurs solutions : ne pas gérer soit les rectangles, soit les losanges, ou bien écrire les catégories de figures en tant qu'interfaces plutôt que classes et les implémenter séparément ensuite.

   Pour l'instant nous nous limiterons aux classes `Quadrilateral`, `Parallelogram`, `Rectangle` et `Square`. Vérifiez que cela suffit pour éviter le problème soulevé.

   > **Correction :** Là, il n'y a plus qu'une seule lignée d'héritage (plus d'héritage multiple), donc il n'y a plus d'interdiction à tout écrire sous forme de classes directement.

2. Nous allons demander en plus que nos figures implémentent toutes l'interface

```java
interface Shape2D {
    double perimeter();
    double surface();

    /**
     *  Méthode servant juste au test. Ne doit pas servir dans le programme final.
     *  @return true si la figure a bien les propriétés qu'elle prétend avoir
     */
    default boolean checkInvariants() { return true; }
}
```

   Écrivez la classe `Quadrilateral`. Les attributs seront les sommets du quadrilatère. Pour les sommets, vous pouvez utiliser la classe `Point2D.Double` (vous utiliserez sans doute sa méthode `double distance(Point2D pt)`). Pour la surface d'un quadrilatère $ABCD$, vous pouvez utiliser la formule suivante :

$$A = \frac{1}{2}(x_1y_2 - x_2y_1) \text{ où } \vec{AC} = (x_1, y_1) \text{ et } \vec{BD} = (x_2, y_2).$$

   Il s'agit d'une aire algébrique (peut être négative). Vous pouvez prendre la valeur absolue si ça vous arrange (mais la formule devient fausse pour un quadrilatère croisé).

3. Passez les attributs en `private final` et ajoutez les getteurs pour les sommets.

4. Écrivez la classe `Parallelogram` (avec l'héritage qui va bien). Vous devez garantir qu'à la construction, deux vecteurs opposés sont égaux c-à-d, par exemple, vérifier que $\vec{AD} = \vec{BC}$. Une façon de garantir le parallélisme est de ne demander que 3 sommets et de calculer automatiquement la position du 4e.
   N'hésitez pas à créer des méthodes auxiliaires pour éviter d'écrire du code répétitif ! Si vous pensez qu'elles seront utiles dans les sous-classes, celles-ci devront être de visibilité `protected`, sinon `private`.
   Redéfinissez la méthode `checkInvariants()` afin qu'elle renvoie `true` si et seulement si la propriété d'être un parallélogramme est effectivement vérifiée.
5. Les getteurs des sommets de votre classe `Quadrilateral` utilisent-ils, comme type de retour, le type `Point2D.Double` ou directement des valeurs de type `double` ?
   Voyez-vous en quoi utiliser `Point2D.Double` menace-t-il la préservation de l'invariant ?
   Si vous avez le temps, corrigez vos getteurs (sinon, ça n'empêche pas de continuer l'exercice).
   Solutions possibles :
   — retourner des `double`, plutôt qu'une référence vers l'instance de `Point2D.Double` encapsulée dans la figure (il faut alors 2 getteurs par sommet)
   — retourner une copie de l'instance de `Point2D.Double` plutôt qu'une référence vers l'instance encapsulée
   — ne pas utiliser `Point2D.Double` du tout → remplacer par des `double` ou bien par une classe `Point` de votre cru qui serait immuable.
6. Ne voyez-vous pas un problème similaire avec les constructeurs : que se passerait-il si un utilisateur instanciait un `Point2D.Double`, gardait sa référence dans une variable `s`, puis le passait en argument du constructeur d'un `Parallelogram` et enfin, modifiait le point référencé par la variable `s` ?
   Comment corriger ce problème ?

   Correction : Pour l'instant, le sujet a juste demandé de "bétonner" les getteurs. Pourtant, ce n'est pas suffisant. En effet, il est encore possible de modifier les positions des sommets sans passer par ces accesseurs : il suffit de récupérer une référence vers un des points et de modifier directement sans passer par les méthodes de `Quadrilateral` ou une de ses sous-classes.
   Pour obtenir une telle référence, il y a actuellement 3 façons :
   — utiliser une des méthodes `getX()`, qui retourne un point
   — instancier un point, garder sa référence, et la passer à `setX()`
   — instancier plusieurs points, garder les références et les passer au constructeur de la figure.
   Pour éviter cela, une technique est d'exclure `Point2D` de l'API du quadrilatère (aucun attribut public ni type de retour ou de paramètre de méthode ne doit utiliser ce type) : on peut utiliser `double` à la place, avec une méthode pour chaque coordonnée (mais c'est très lourd, et ça rend difficile la vérification de contraintes portant sur les deux coordonnées ensemble), ou bien définir une classe point immutable (et penser à la rendre `final`, sinon on peut encore créer une sous-classe non-immutable et en envoyer des instances aux constructeurs et setteurs...).
   Une autre technique est d'effectuer des copies de sécurité : dès qu'une méthode accepte un point, il ne faut pas l'utiliser directement, mais plutôt immédiatement créer une copie de l'objet qui sera utilisée à sa place. Dès qu'une méthode retourne un point, il ne faut pas retourner un point utilisé dans l'état de l'objet (attribut),

mais une nouvelle instance dont on ne se servira plus dans la classe.
Le corrigé fourni sur Moodle définit des classes `Point2D` et `Vector2D` immutables, avec les opérations affines et vectorielles qui vont bien.

7. Écrivez les classes `Rectangle` et `Square`, en respectant la même hygiène que pour `Parallelogram`. Attention : le rectangle a un invariant de plus que le parallélogramme : les angles doivent rester droits ; et le carré a encore un invariant supplémentaire : tous les côtés ont la même longueur.

   Notez qu'à ce point, on ne doit plus avoir directement accès aux attributs et que la seule façon d'accéder aux sommets, c'est via les accesseurs hérités (`super.getA()`, `super.setB(...)`, ...) et les éventuelles méthodes auxiliaires `protected`.

Correction : Voir le fichier `.zip` fourni sur Moodle pour le code source.
Remarque : les classe de ce zip implémentent les mutateurs (un setteur pour chaque sommet) qui ne sont pas demandés dans cet exercice.
Regardez, au passage, comment faire en sorte que les redéfinitions des setteurs préservent les invariants du type "cette figure est un... carré/rectangle/...".
Remarquez aussi que, par conséquent, ce ne sont plus juste des setteurs (ils modifient plus d'un seul sommet) et que ce n'est donc pas une solution idéale (cf. exercice précédent).

Arrivé à la fin de cet exercice, on a normalement des classes pour représenter différentes figures, avec les garanties suivantes :

- *toute instance directe des classes programmées représente bien à tout moment une figure du type donné par le nom de la classe (ex : une intance de `Carre` représente un carré)*
- *les méthodes `perimeter()` et `surface()` retournent bien respectivement le périmètre et l'aire de la figure.*

Mais ces garanties ne valent que pour les instances directes de ces classes. Il est encore possible de tout "casser" en créant des mauvaises sous-classes. Cela pourra être réglé en ajoutant le mot-clé `final` devant la déclaration de la classe qui ne doit pas être extensible, mais ce n'est pas si simple : dans cet exercice on eu besoin de l'héritage, par exemple pour passer de `Parallelogram` à `Rectangle`. Il faudra donc "ruser" (à suivre...).

On commence d'ailleurs à distinguer ce qui est nécessaire pour écrire une classe immuable :

- *ne pas permettre la modification des attributs ;*
- *si certains attributs référencent des objets mutables, faire en sorte qu'aucune référence vers ces objets ne puisse exister à l'extérieur de la classe (faire des copies défensives) ;*
- *empêcher de créer des sous-classes.*

Exercice 3 : Cylindres

Nous voulons maintenant construire des cylindres (solides constitués de deux bases parallèles superposables et d'une "surface cylindrique" constituée de lignes droites parallèles joignant les deux bases). Ainsi, en ce qui nous concerne, un cylindre se caractérise par sa base (une forme 2D telle que décrite dans l'exercice précédent) et sa hauteur.

Sur de tels cylindres, nous voudrions en outre calculer la surface (= 2 fois la surface de la base + hauteur * périmètre de la base) et le volume (= surface de la base * hauteur).

1. Si nous écrivions une classe `QuadriCylinder`, pour un cylindre à base quadrilatère, qui serait sous-classe de `Quadrilateral` (avec attribut ajouté `height`, ainsi que les méthodes ajoutées `surface()` et `volume()`), est-ce qu'il serait possible d'obtenir une

classe `SquareCylinder` pour un cylindre à base carrée qui serait sous-type de `Square` et de `QuadriCylinder` ?

Est-ce que cette classe `QuadriCylinder` était une bonne idée de toute façon ? (argumentez)

2. Nous allons nous y prendre autrement et utiliser la composition : un cylindre n'est pas une forme 2D "améliorée", mais une forme 3D qui possède une base, qui est une forme 2D, ainsi qu'une hauteur.

   Écrivez une telle classe, avec les méthodes surface et volume demandées.

3. Peut-on maintenant écrire des sous-classes de cylindres, en fonction de la forme de leur base ? Si oui, faites-le.

   Faites attention à ce qu'il soit impossible de changer le type de base pendant la vie d'un tel objet (on ne peut pas modifier la base d'un cylindre à base carrée de telle sorte à ce qu'il devienne un cylindre à base circulaire, par exemple).

II) Pause documentation

Lisez le complément de cours sur l'immuabilité (cf. Moodle) avant de passer à la suite.

III) Modélisation géométrique (suite)

Exercice 4 : Rectangles et carrés, une solution ?

Comme le problème de l'exercice 1 est qu'on ne peut pas concilier à la fois l'invariant "cette figure est un XXX" et la spécification des mutateurs héritée du supertype, une version immuable des figures devrait être plus robuste.

Évidemment, les mutateurs fournissaient une fonctionnalité utile. Pour les remplacer, il est possible d'écrire des méthodes retournant un nouvel objet identique à `this`, sauf pour la propriété qu'on souhaite "modifier". Exemple : `rect.withLongueur(15)` retourne un rectangle identique à `rect` mais dont la longueur est 15 ; en revanche, `carre.withLongueur(15)`, pour préserver l'indépendance des propriétés largeur et longueur, retourne un rectangle et non un carré.

À faire :

1. Écrivez les classes immuables `RectangleImmuable` et `CarreImmuable` sous-classes de `Rectangle` (classe abstraite fournissant les fonctionnalités communes, sans mutateur, mais avec les opérations `withXXX()` en tant que méthodes abstraites). Notez que les méthodes de "modification" de `CarreImmuable` respectent automatiquement l'invariant du carré car `this` n'est pas modifié.

Correction :

```java
package immuable;

public class RectangleImmuable {
    // Note : toutes les méthodes sont final, à défaut de pouvoir marquer la classe comme final.
    // Ceci garantit au moins l'immutabilité des propriétés qui font d'un rectangle un rectangle.
    private final double ax, ay, angle, longueur, largeur;

    public final RectangleImmuable withAx(double ax) {
        return new RectangleImmuable(ax, ay, angle, longueur, largeur);
    }
```

```
11
12      public final RectangleImmuable withAy(double ay) {
13          return new RectangleImmuable(ax, ay, angle, longueur, largeur);
14      }
15
16      public final RectangleImmuable withAngle(double angle) {
17          return new RectangleImmuable(ax, ay, angle, longueur, largeur);
18      }
19
20      public final RectangleImmuable withLongueur(double longueur) {
21          return new RectangleImmuable(ax, ay, angle, longueur, largeur);
22      }
23
24      public final RectangleImmuable withLargeur(double largeur) {
25          return new RectangleImmuable(ax, ay, angle, longueur, largeur);
26      }
27
28      RectangleImmuable(double ax, double ay, double angle, double longueur, double
            largeur) {
29          super();
30          this.ax = ax;
31          this.ay = ay;
32          this.angle = angle;
33          this.longueur = longueur;
34          this.largeur = largeur;
35      }
36
37
38      public final double getAx() {
39          return ax;
40      }
41
42      public final double getAy() {
43          return ay;
44      }
45
46      public final double getAngle() {
47          return angle;
48      }
49
50      public final double getLongueur() {
51          return longueur;
52      }
53
54      public final double getLargeur() {
55          return largeur;
56      }
57
58  }
```

et

```
1   package immuable;
2
3   public final class CarreImmuable extends RectangleImmuable {
4       public CarreImmuable withCote(double cote) {
5           return new CarreImmuable(getAx(), getAy(), getAngle(), cote);
6       }
7
8       public CarreImmuable(double ax, double ay, double angle, double cote) {
9           super(ax, ay, angle, cote, cote);
10      }
11
12  }
```

2. Ci-dessus, pour empêcher la création de sous-types mutables, `RectangleImmuable` n'est

pas extensible ; pour cette raison, `CarreImmuable` n'en est pas un sous-type.

Cependant la technique des classes scellées permet, pour une classe donnée, de définir une liste fermée de sous-types et donc de garantir l'immuabilité du supertype. Utilisez cette technique pour définir des types immuables `Rectangle` et `Carre` avec `Carre` sous-type de `Rectangle` (imbriquez vos déclarations dans une classe-bibliothèque `FormesImmuables`).

Correction : Principe : on profite du fait que la portée de `private` est en réalité la classe englobante.
Ainsi, on peut utiliser l'architecture suivante :

```
1  public final class FormesImmuables { // notez le pluriel
2      private FormesImmuables() {} // on empêche d'étendre la bibliothèque
3
4       // pas une interface : il faut pouvoir restreindre le sous-typage (constructeur privé)
5      public abstract class Rectangle {
6          // pas d'attributs longueur et largeur
7          private Rectangle() { ... } // pas de sous-classes hors de FormesImmuables
8          ...
9      }
10
11     // classe privée, car l'API n'expose que le type Rectangle, qui ici est super-type
12     de Carre (RectangleImpl ne l'est pas)
13     private static class RectangleImpl extends Rectangle {
14         private final double longueur, largeur;
15         private RectangleImpl(...) { ... }
16         ...
17     }
18
19     public static class Carre extends Rectangle {
20         private final double cote;
21         private Carre(...) { ... }
22         ...
23     }
24
25     // fabriques statiques (Indispensable pour Rectangle car RectangleImpl est privée.
           Sinon on aurait pu s'en passer et mettre les constructeurs en public et les
           classes en final.)
26
27     public static Rectangle newRectangle(...) {
28         return new RectangleImpl(...);
29     }
30
31     public static Carre newCarre(...) {
32         return new Carre(...);
33     }
34 }
```

On pourrait faire plus simple : avec une classe `Rectangle` directement instanciable (contenant attributs largeur et hauteur) et une classe `Carre` qui l'étend, mais `Carre` aurait des attributs redondants, ce qui est une mauvaise pratique.

Exercice 5 : Quadrilatères

1. Modifiez les classes de l'exercice 2 pour les rendre immuables.
2. Ajoutez-y des méthodes pour "déplacer" les différents sommets. Comme les figures ne sont pas modifiables, ces méthodes retournent donc une nouvelle figure.
3. Ajoutez des méthodes de conversion d'une figure à une autre. Par exemple : `public Rectangle toRectangle()`.

Ces méthodes peuvent être déclarées en tant que méthodes abstraites très haut dans l'arbre d'héritage (classe `Figure`).

Évidemment, certaines conversions n'ont pas de sens (p. ex : convertir un parallélogramme quelconque en carré). Ce qu'on peut faire, c'est vérifier que la figure `this` représente effectivement une figure du type vers lequel on veut la convertir (pour le carré : vérifier angle droit et 4 côtés égaux), et retourner `null` [1] quand la vérification échoue.

Remarque : à cause de l'arithmétique flottante, la précision n'est pas absolue. Il faut donc se permettre une marge d'erreur, sinon on ne pourrait presque jamais convertir un rectangle en carré. Cette marge peut être paramétrée par un attribut statique de `Figure` ou bien être passée en paramètre des méthodes de conversion.

1. On peut explorer d'autres solutions plus "propres" : la classe `Optional`, ou bien lancer des exceptions.