

Compléments en Programmation Orientée Objet

TD et TP n° 5 : Mécanismes et stratégies d'héritage

1 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”.
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.
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 ?
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` ?

7. Pour corriger ce problème, redéfinissez (avec `@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.
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 ?

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.

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 va proposer des noms de classes en anglais pour marquer cette séparation.

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 ?

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

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

```

1 interface Shape2D {
2     double perimeter();
3     double surface();
4
5     /**
6      * Méthode servant juste au test. Ne doit pas servir dans le programme final.
7      * @return true si la figure a bien les propriétés qu'elle prétend avoir
8      */
9     default boolean checkInvariants() { return true; }
10 }

```

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 :

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

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 instanciat 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 ?

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`.

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 instance 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.

2 Modélisation géométrique (suite)

Exercice 3 : 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.
2. Ci-dessus, pour empêcher la création de sous-types mutables, il faudrait que `RectangleImmuable` ne soit pas extensible ; mais dans ce cas, `CarreImmuable` ne peut pas en être un sous-type. 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`).

Exercice 4 : 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.